Un document Latex

Victor Hohlov

2018-05-05

Contents

1 Course: Introduction

Introduction to Programming Paradigms

section[In how many different ways can we reverse a sequence of integers?] In how many different ways can we reverse a sequence of integers?

We start the lecture by looking at three different Java programs which reverse a sequence of integers.
The *C-programmer* style:

```
public class Rev {
        public static void show (Integer[] v){
                for (Integer i:v)
                        System.out.println(i);
        }
        public static void main (String[] args) {
                Integer[] v = new Integer[] {1,2,3,4,5,6,7,8,9};
                int i=0;
                while (i < v.length/2){
                        int t = v[i];
                        v[i] = v[v.length-1-i];
                        v[v.length-1-i] = t;
                        i++;
                }
                show(v);
        }
}
```

The first approach stores the sequence as an array, and performs reversal in-place, by swapping the first half of the array with the second. The code is **compact** and fairly **efficient** in terms of computational complexity.
The *functional* style:

```
class List {
        Integer val;
        List next;
        public List(Integer val, List next){
                this.val = val;
                this.next = next;
        }
        @Override
        public String toString(){
                if (next == null)
                        return val.toString();
                return val.toString()+" "+next.toString();
        }
}
public class FuncRev {

        private static List rev(List x, List y){
                if (x == null)
                        return y;
                return rev(x.next,new List(x.val,y));
        }
        public static List reverse (List l){
                return rev(l,null);
        }
        public static void main (String[] args) {
                List v = new List(1, new List(2, new List(3, new List(4, new List (5,null)))));
                System.out.println(reverse(v));
        }
}
```

Unlike the former approach, here the code focuses more on **input-output**: reversal takes a list (instead of an array), and produces another list. The strategy relies on an auxiliary function `rev` which uses an accumulator to reverse the list. Both `rev` and the display function are recursive. However, `rev` is **tail-recursive** hence quite efficient.
The *object-oriented* style:

```
import java.util.Iterator;

class RVList<T> implements Iterable<T> {
        private T[] array;

        public RVList(T[] array){
                this.array = array;
        }

        @Override
        public Iterator<T> iterator() {
                return new Iterator<T>(){
                        private int crtIndex = array.length - 1;

                        @Override
                        public boolean hasNext(){
                                return crtIndex >= 0;
                        }
```

```java

                        @Override
                        public T next(){
                                return array[crtIndex--];
                        }

                        @Override
                        public void remove () {}
                };
        }
}

public class OORev {

        public static void main (String[] args) {
                String[] s = new String[]{"1", "2", "3", "4", "5", "6"};

                Iterator<String> r = (new RVList<String>(s)).iterator();
                while (r.hasNext()){
                        System.out.println(r.next());
                }
        }
}
```

The final alternative relies on the Java concept of **iterator** of collections. Informally, a collection of elements has the property that its elements can be *enumerated* or *iterated.* Any such collection is an `Iterable` (extends the Java interface Iterable). Here, `RVList` is such an iterable collection, which takes its elements at construction, from an array. `RVList` is **generic**, hence its elements can be of any type `T`.
The fundamental trait of an `Iterable` object is that it implements the method `iterator`, which returns just that. An **iterator** is an object which allows enumerating all elements of the collection at hand, via the methods `hasNext` and `next`. Sometimes (as in the case of a `Set`) the order is unimportant. Here, the order matters - elements are explored in their reverse order. Technically, the iterator object is an **anonymous class**.
This alternative focuses on **list traversal**, and on the **generality** of the approach. Basically, any Java collection (for which order makes sense) can be transformed to an array (via the `toArray` method) and subsequently - an `RVList` which can be traversed in reverse order via its iterator.

section[Why so many reversals?] Why so many reversals?

The reason for choosing reversal as a running example is that the task is algorithmically trivial: take the sequence of elements of the collection at hand, be it array of list (or anything else), in their **reverse** order.
However, there are **many different ways** for implementing reversal, and each makes perfect sense in some setting. Moreover, some of these ways are: **subtle**, **conceptually challenging**, and require **higher skill/knowledge** in operating the programming language at hand.
Our point, and one of the major objectives of this lecture, is to emphasise that, apart

from developing fast and correct algorithms, a task of equal challenge and importance is **writing down the code**, in the **suitable programming language**.
We have clear metrics for choosing algorithms. Do we also have metrics for **code writing**? For instance:

- legibility
- compactness (number of code lines)
- ease of use (can other programmers use the code easily)
- extensibility (can other programmers add modifications to the code easily)
- good documentation

is a plausible list. While some of these criteria overlap and are difficult to assess objectively, programmers more often than not agree that some programs are **well-written while others are poor** (w.r.t. some criteria).

section[How many other ways?] How many other ways?

There are many possible variations to our three examples, but a few elements do stand out:

- the functional style for **representing a list** in the second example - very akin to Abstract Datatypes
- the functional style for reversal - relying on recursion, in the same example
- the Object Oriented style for **traversing a list** in the third example - which although is tightly linked to Java Collections, can be migrated to any other object-oriented language.

These **elements of style** are frequently called **design patterns** - generic ways of writing code, which can be deployed for different implementations.
Some elements of style may have some common ground, or rely on certain traits of the programming language. These latter are called programming **paradigms**. For instance, writing programs as recursive functions as well as representing data as ADTs (recall that constructors *are* functions) are both styles of the **functional** paradigm.
In this lecture, we shall go over the following paradigms:

- imperative
- object oriented
- **functional**
- **logical** (or logic programming)
- **associative** (or rule-based)

Other questions

- Is there a **right** programming language? How to people choose programming languages?
- Who invented paradigms? (A history between Lisp vs C)
- Relationship between paradigms and programs (one-to-one, one-to-many, many-to-many)
- How **extensible** can programs be?

2 Functional Programming

subsection[] Side-effects
Programming with side-effects is one defining feature of imperative programming. A procedure produces a side-effect if its call produces a change in the state of a running program (the memory), apart from the returned value.
We (re-)consider the following code discussed in the previous lecture:

```
int main (){

        int* v = malloc(sizeof(int)*7);
        v[0] = 1; v[1] = 9; v[2]=4; v[3]= 15; v[4] = 2; v[5] = 6; v[6]
    = 0;

        show(v);
        insertion_sort(v,7);
        show(v);
}
```

The two identical calls on a (hypothetical) `show` functions will produce different effects. The state of `v` has changed after the `insertion_sort` call.

Programming with side-effects is perhaps one of the most wide-spread styles, and it may seem difficult/unnatural to refrain from using it. However, there are situations where using side-effects may be produce a code prone to bugs.
Consider the following implementation of a depth-first traversal of an oriented graph:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NODE_NO 3
// graph represented as adjacency matrix
typedef struct Graph {
        int** m;
        int nodes;
}* Graph;

Graph make_graph (int** m, int n){
        struct Graph* g = (struct Graph*)malloc(sizeof(struct Graph));
        g->m = m;
        g->nodes = n;
        return g;
}

int visited[NODE_NO] = {0,0,0};

void visit(Graph g, int from){
        visited[from] = 1;
        printf(" Visited %i ",from);
        int i;
        for (i = 0; i<g->nodes; i++){
                if (g->m[from][i] && !visited[i])
                        visit(g,i);
        }
}

int main (){

        int** a = (int**)malloc(NODE_NO*sizeof(int*));
        a[0] = (int*)malloc(NODE_NO*sizeof(int));
        a[0][0] = 0; a[0][1] = 1; a[0][2] = 0;
        a[1] = (int*)malloc(NODE_NO*sizeof(int));
        a[1][0] = 0; a[1][1] = 0; a[1][2] = 1;
        a[2] = (int*)malloc(NODE_NO*sizeof(int));
        a[2][0] = 0; a[2][1] = 0; a[2][2] = 0;

        Graph g = make_graph((int**)a,3);
        visit(g,0);

        //visit(g,1);

}
```

- A graph is represented as an adjacency matrix; nodes are identified by integers.
- The code relies on the global array `visited` which marks those nodes which have been visited during the traversal.
- The procedure `visit` initiates the traversal: it marks the initial node (`from`) as visited and recursively visits all adjacent nodes.

subsubsection[] Side-effects may easily introduce bugs
The problem with this implementation is that `visited` is modified as a side-effect. As it is, two subsequent calls on `visit` will produce different results:

```
int main () {
     visit(g,0);  // produces a correct traversal
     visit(g,1);  // produces an invalid traversal
}
```

This happens because `visited` was not initialised before the second use of `visit`, and it still holds the visited nodes from the previous traversal.

This bug can be easily solved in different ways:

- make sure `visited` flags all nodes as not visited before each traversal;
- use a `visited` array for each traversal;

This example is important because:

- side-effects introduce a bug which may be difficult to identify at compile-time (the program will compile) as well as at run-time (the problem only occurs during subsequent traversals, and does not produce a crash).
- the programmer needs to enforce a discipline regarding the scope of a side-effect (e.g. what variables outside of `visit` are allowed to be modified by `visit`). Such a discipline may naturally lead to an object-oriented programming style (relying on encapsulation) or to a functional style (limiting or completely forbidding side-effects).

subsubsection[] Side-effects may be unpredictable

Consider the following code:

```
int fx (){ printf("Hello x !\n"); return 1;}
int fy (){ printf("Hello y !\n"); return 2;}

void test_call(int x, int y){}

```

```
int main(){
     test_call(fx(),fy());
}
```

Both functions `fx` and `fy` produce a side-effect, since they display a message, apart from the returned value. Programmers would generally assume that `test_call` will display:

```
Hello x !
Hello y !
```

However this is not generally the case. In the C standard, there is no specification on the order in which parameters are evaluated. As it happens, on some compilers, `test_call` will produce the above output, while on others (e.g. `here` [1]), the messages are in a different order.
This is another situation where side-effects may produce hard-to-find bugs.
subsection[] Recursion
Recursion is a natural way of expressing algorithms, for instance the computation of the nth Fibbonacci number:

```
int fib (int n){
        if (n<2)
                return n;
        return fib(n-1)+fib(n-2);
}
```

subsubsection[] Recursion may be inefficient

The above implementation is inefficient since it triggers an exponential number of `fib` calls. For instance, the call `fib(4)` will trigger the following sequence of stack modifications.
In the diagram below, each line represents the state of the call stack. The first function call corresponds to the top of the stack.

```
f 4
f 3 | f 4
f 2 | f 3 | f 4
f 1 | f 2 | f 3 | f 4  //f(1) returns 1;
f 0 | f 2 | f 3 | f 4  //f(0) returns 0;
f 1 | f 3 | f 4        //f(2) returns 1+0;
f 2 | f 4              //f(1) returns 1; f(3) returns 1+1
f 1 | f 2 | f 4
f 0 | f 2 | f 4        //f(1) returns 1;
[]                     //f(2) returns 1+0(again), f(4) returns 2 + 1
```

By solving the recurrence `T(n) = T(n-1) + T(n-2) + O(1)` we can determine the number of function calls. Conceptually, the problem with this implementation is that it re-computes previously-computed values (e.g. `f(2)` is computed twice).
subsubsection[] Solving the call explosion problem
We can easily rewrite the implementation of `fifo` by introducing an accumulator:

```
int fib_aux (int f1, int f2, int i, int n){
        if (i == n)
                return f2;
        return fib_aux(f2,f1+f2,i+1,n);
}

int fib2 (int n){
        return fib_aux(0,1,0,n);
}
```

The call `fib2(4)` produces the following sequence of calls:

```
fib2 4
fib_aux 0 1 0 4
fib_aux 1 1 1 4
fib_aux 1 2 2 4
fib_aux 2 3 3 4
fib_aux 3 5 4 4
```

which is linear in the value of `n`, and does not-recompute values. However, in an imperative language such as C, the call stack can quickly become full for larger `n`. To solve this problem, many programming languages (e.g. Scala and Haskell, but not C) implement tail-call optimisation. Conceptually, tail-call optimisation relies on the following reasoning.
Consider the following table, in which `call no` represents the *address* of the function call.

```
Call no.   Stack contents            Return address
---------- ------------------------- ------------------------------
1          fib 4
2          fib_aux 0 1 0 4           return to call from 1  value 8
3          fib_aux 1 1 1 4           return to call from 2  value 8
4          fib_aux 1 2 2 4           return to call from 3  value 8
5          fib_aux 2 3 3 4           return to call from 4  value 8
6          fib_aux 3 5 4 4           return to call from 5  value 8
```

For instance, the call `fib_aux 1 2 2 4` has address `5` and after executing it must return to the address `4`. An intelligent compiler may realise the following reasoning:

- the call `fib_aux 1 2 2 4` only returns a value to the subsequent call.
- hence, it makes no sense to keep this call on the stack. Instead, it may directly replace the call `fib_aux 1 2 2 4` by `fib_aux 3 5 4 4`.

Following this reasoning, after each recursive call, the stack will contain only one call of `fib_aux`, as shown below:

```
fib 4 | fib_aux 0 1 0 4            // after first recursive call
fib 4 | fib_aux 1 1 1 4            // after second recursive call
fib 4 | fib_aux 1 2 2 4            // after 3rd recursive call
fib 4 | fib_aux 2 3 3 4            // after 4th recursive call
fib 4 | fib_aux 3 5 4 4            // after 5th recursive call
fib 4 | 5
5
```

Note that the same reasoning does not hold for the implementation of `fib`, since the recursive calls: `fib(n-1) + fib(n-2)` require an additional computation (addition) before returning the value.
For tail-end optimisation to take place: a recursive function must be tail-end. A recursive function is tail-end if it returns a call *to itself* without any additional computation.
Tail-call optimisation in C may be implemented (for `fib_aux`) as follows:

```
fib_aux (int f1, int f2, int i, int n){
start:
        if (i == n)
                return f2;
        int nf1 = f2;
        int nf2 = f1+f2;
        int ni = i+1;
        int nn = n;       //this instruction may be later eliminated by
    subsequent optimisation stages.
        f1 = nf1;
        f2 = nf2;
        i = ni;
        nn = n;
        goto start;
}
```

subsection[] Higher-order functions subsubsection[] Overview Higher-order functions are a programming style which supports modularisation, by replacing recurring programming patterns by a common procedure. Using higher-order functions is not possible in any programming language, and requires being able to pass arbitrary functions as parameter.
Suppose we have a list implementation in C which is *concealed* by the following constructors and observers:

```
int head (List l);
List tail (List l);
List cons (int e, List t);
void show (List l);
int size (List l);
```

as well as the implementation of the addition and product functions over list elements:

```
int sum (List l){
    if (l == Nil)
        return 0;
    return head(l)+sum(tail(l));
}
```

```

int product (List l){
    if (l == Nil)
        return 1;
    return head(l)*product(tail(l));
}
```

Let us ignore the fact that both recursive functions are not tail end (thus inefficient).
Both implementations follow a common pattern, which only differs in two aspects:

- the value which is returned when the list is empty
- the operation which is performed between the first member of the list, and the recursive call

We could express this common pattern by the following implementation:

```
int fold (int init, int(*op)(int,int), List l){
    if (isEmpty(l))
        return init;
    return op(head(l),fold(init,op,tail(l)));
}
```

The function `fold` receives as parameter exactly the initial value, and the operation to be performed. Next, addition and product can be rewritten as:

```
int op_sum (int x, int y){return x + y;}
int op_product (int x, int y){return x * y;}
int op_size (int x, int y){return 1 + y;}

int sum (List l) {return fold(0,op_sum, l);}
int product (List l) {return fold(1,op_product, l);}
```

which is a more modular code. `fold` is called a higher-order function, because its behaviour is defined with respect to another function (here op). We shall review and improve this definition later on.

One possible criticism to `fold` is that it is not tail-recursive, however it can be rewritten as:

```
int tail_fold (int init, int(*op)(int,int), List l){
    if (l == Nil)
        return init;
    return tail_fold(op(head(l),init),op,tail(l));
}
```

However, `fold` and `tail_fold` are not different only with respect to efficiency. They also have different behaviours. Consider the call:

```
fold(i, op, [a,b,c])
op (a, fold(i,op, [b,c]))
op (a, op (b, fold (i, op, [c])))
op (a, op (b, op (c, fold (i, op, []))))
op (a, op (b, op (c, i)))
a op (b op (c op i))
```

versus:

```
tail_fold(i, op, [a,b,c])
tail_fold(op(a,i),op, [b,c])
tail_fold(op(b,op(a,i)),op, [c])
tail_fold(op(c,op(b,op(a,i))),op,[])
c op (b op (a op i))
```

The two fold implementations are only the same if 'op' is commutative (addition and product happen to be commutative).
subsubsection[] Question How can `sum` be implemented using `fold` ?
subsection[] Programming with higher-order functions is difficult in C
One criticism to the `fold/tail_fold` implementations is that they only work for operations over integers. There are many situations where folding may be useful of operations of other types. Consider the following implementation for list concatenation, which cannot be supported by our fold implementations:

```
List append (List l1, List l2) {return fold(l2,append_op,l1);}

List append_op(int e, List l2) {return cons(e,l2);}
```

For instance, `append([1,2,3], [4,5,6])` will produce:

```
fold([4,5,6], append_op, [1,2,3])
cons(1, fold([4,5,6], append_op, [2,3]))
cons(1, cons(2, fold([4,5,6], append_op, [3])))
cons(1, cons(2, cons(3, fold([4,5,6], append_op, []))))
cons(1, cons(2, cons(3, [4,5,6])))
[1,2,3,4,5,6]
```

The above implementation of append cannot rely on our fold, since the type of op supported by `fold` is: `int(*op)(int,int)`, while `append_op` has type `List(*op)(int,List)`. There are possible fixes to this issue (e.g. implementing another fold to support such operation types, or using void pointers) however none are clear/-natural enough to actually be used in real code. subsection[] Introduction to Haskell
Haskell is a purely-functional programming language. In short the term pure refers to the fact that side-effects are not permitted by design. Thus, expressions of the form:

```
x = expr
```

do not produce side-effects. They create a variable `x` which is permanently bound to the expression `expr` for the entire program execution. The variable `x` is immutable: it cannot be modified.
Haskell also supports higher-order functions. The implementation of `fold` is as follows:

```
fold op acc l = if empty(l) then acc else op x (fold op acc xs)
```

However, function definitions in Haskell allow specifying *conditional behaviour* depending on how objects are constructed (i.e. base constructors). This is called pattern-matching and deserves a more elaborate discussion (which will be done in a future lecture). Here, we merely state that `[]` and `:` (cons) are the base constructors for lists. The code using pattern matching becomes:

```haskell
foldr op acc [] = acc
foldr op acc (x:xs) = op x (foldr op acc xs)
```

The name (foldr) suggests the order in which the operation `op` is applied (to the right). We also note that `foldr`, although is not tail-recursive, is actually efficient in Haskell. This is related to the evaluation order, which will be discussed in detail in a future lecture.
In Haskell, `tail_fold` is called `foldl` (to the left), and is implemented slightly different:

```haskell
foldl op acc [] = acc
foldl op acc (x:xs) = foldl op (op x acc) xs
```

The Haskell implementation calls `(op x acc)` instead of `(op acc x)` and there is no particular reason for (or against) this choice. Remember that is only holds in Haskell. Other fold functions implemented in other languages may work differently.

The functions `foldr` and `foldl` are implemented by-default in Haskell, and can be directly used to define addition, product, or the size of a list:

```
sum = foldr (+) 0
prod = foldr (*) 1
size = foldr (1+) 0
```

as well as other functions on lists:

```
reverse l = foldl (:) [] l
append l1 l2 = foldr (:) l2 l1
```

subsection[Lab: Introduction to Haskell] Lab: Introduction to Haskell section[] Haskell: Introducere
Scopul acestui laborator este a de familiariza studenții cu limbajul Haskell.
subsection[] Introducere
Haskell este un limbaj de programare `pur-funcțional` [2]. În limbajele imperative, programele sunt secvențe de instrucțiuni pe care calculatorul le execută ținând cont de stare, care se poate modifica în execuție. În limbajele funcționale nu există o stare, iar funcțiile nu au efecte secundare (e.g. nu pot modifica o variabilă globală), garantând că rezultatul depinde doar de argumentele date: aceeași funcție apelată de două ori cu aceleași argumente, întoarce același rezultat. Astfel, demonstrarea corectitudinii unui program este mai facilă.
subsection[] Platforma Haskell

Pentru început, avem nevoie de un mod de a compila cod. Puteți descărca platforma Haskell de `aici` [3] (Windows/OS X/Linux) sau puteți instala pachetul "ghc".
Vom lucra în modul interactiv, care ne permite să apelăm funcții din consolă și să vedem rezultatul lor. Putem să încărcăm și fișiere cu fragmente de cod din care să apelăm funcții definite de noi.
Pentru modul interactiv, rulați "ghci". Ar trebui să vedeți ceva asemănător:

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

^Comenzi folositoare în ghci: ^^^Comandă ^Efect ^|:l *filename* |încarcă un fișier din directorul din care e rulat ghci ||:r |reîncarcă ultimului fișier încărcat ||:t *expresie* |afișează tipul expresiei ||:? |afișează meniul de help cu comenzile disponibile ||:q |închide ghci |
subsection[] Tipuri de date în Haskell
Haskell este un limbaj tipat static (`statically typed` [4]) care poate face inferență de tip (`type inference` [5]). Asta înseamnă că tipul expresiilor este cunoscut la *compilare*; dacă nu este explicit, compilatorul îl *deduce*. Deasemenea, Haskell este tare tipat (`strongly typed` [6]), ceea ce înseamnă că trebuie să existe conversii explicite între diferite tipuri de date.

subsubsection[] Tipuri de bază
Haskell are tipuri asemănătoare celor din alte limbaje studiate până acum, ca C, Java etc: Int, Integer (întregi de dimensiune arbitrară), Float, Double, Char (cu apostrofuri, e.g. 'a'), Bool.

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t True
True :: Bool
Prelude> :t 0 == 1
0 == 1 :: Bool
Prelude> :t 42
42 :: Num a => a
```

"::" se citește ca "are tipul". Deci, de exemplu, "True are tipul Bool".

Tipul lui `42` nu pare să fie ceva intuitiv, ca `Int`. Deocamdată e suficient să menționăm faptul că, în Haskell, constantele numerice pot să se comporte ca diferite tipuri, în funcție de context. `42` poate să fie considerat întreg, în expresii ca `42 + 1`, sau numărul în virgulă mobilă `42.0` în expresii ca `42 * 3.14`.
subsubsection[] Liste
Listele sunt structuri de date omogene (i.e. pot conține doar elemente de același tip). O listă este delimitată de paranteze pătrate, cu elementele sale separate prin virgulă:

```
[1, 2, 3, 4]
```

Lista vidă este `[]`.

Tipul unei liste este dat de tipul elementelor sale; o listă de întregi are tipul `[Int]`. Putem avea liste de liste de liste de liste ... atâta timp cât respectăm condiția de a avea același tip.

```
[['H', 'a'], ['s'], ['k', 'e', 'l', 'l']] -- corect, are tipul [[Char]]
[['N', 'u'], [True, False], ['a', 's', 'a']] -- greșit, conține și
    elemente de tipul [Bool] și de tipul [Char]
```

Pentru a reprezenta șiruri de caractere, putem folosi tipul `String` care este un alias peste `[Char]`.
Astfel, `"String"` este doar un mod mai lizibil de a scrie
`['S', 't', 'r', 'i', 'n', 'g']`, care, la rândul său, este un mod mai lizibil de a scrie
`'S':'t':'r':'i':'n':'g':[]`

subsubsection[] Tupluri
Tuplurile sunt structuri de date eterogene (i.e. pot conține elemente de diferite tipuri). Un tuplu este delimitat de paranteze rotunde, cu elementele sale separate prin virgulă:

```
(1, 'a', True)
```

Tipul unui tuplu este dat de numărul, ordinea și tipul elementelor sale:

```
Prelude> :t (True, 'a')
(True, 'a') :: (Bool, Char)
Prelude> :t ('a', True)
('a', True) :: (Char, Bool)
```

```
Prelude> :t ("sir", True, False)
("sir", True, False) :: ([Char], Bool, Bool)
```

subsection[] Funcții în Haskell
subsubsection[] Apelare
În Haskell, funcțiile pot fi prefixate sau infixate. Cele prefix sunt mai comune, au un nume format din caractere alfanumerice și sunt apelate prin numele lor și lista de argumente, toate separate prin spații (fără paranteze sau virgule):

```
<Prelude> max 1 2
2
```

Funcțiile infixate sunt cele cu nume formate din caractere speciale, de forma *operand1 operator operand2*

```
Prelude> 1 + 2
3
```

În anumite situații ne-am dori să schimbăm modul de afixare al funcțiilor (e.g. din motive de claritate).
Pentru a prefixa o funcție infixată, folosim operatorul `()`. Astfel, următoarele expresii sunt echivalente:

```
3 * 23
(*) 3 23
```

Dacă o funcție are două argumente, putem să o infixăm cu operatorul ‘‘ (*backticks* - dacă nu aveți un layout exotic, ar trebui să fie pe tasta de dinainte de 1). Astfel, următoarele două expresii sunt echivalente:

```
mod 10 3
10 `mod` 3
```

subsubsection[] Definirea unei funcții
Creați, în editorul de text preferat, un fișier cu extensia `.hs` în care vom defini prima funcție:

```
-- intoarce modulul dublului celui mai mare dintre cele doua argumente
myFunc x y = abs (2 * max x y)
```

Rolul parantezelor aici este de a forța ordinea dorită de a efectua operațiilor. Altfel funcția ar înmulți 2 (modulul lui 2) cu cel mai mare dintre numere, pentru că aplicarea funcției are precedență mai mare.

Puteți testa funcția din ghci:

```
Prelude> :l first.hs
[1 of 1] Compiling Main             ( first.hs, interpreted )
Ok, modules loaded: Main.
*Main> myFunc 10 11
22
*Main> myFunc (-10) (-11)
20
*Main> myFunc 3.14 2.71
6.28
```

Dacă folosiți o versiune mai veche de ghci, aveți nevoie de cuvântul cheie "let" pentru a defini o funcție direct în interpretor:

```
Prelude>let myFunc x y = abs (2 * max x y)
```

Observăm că funcția noastră merge și pentru numere întregi și pentru numere în virgulă mobilă. Am precizat mai devreme că orice expresie trebuie să aibă un tip, cunoscut la compilare. Cum noi nu am precizat tipul funcției noastre, compilatorul l-a dedus ca fiind:

```
Prelude> :t myFunc
myFunc :: (Num a, Ord a) => a -> a -> a
```

Fără a intra în detalii, funcția noastră ia ca argumente două numere de același tip care pot fi ordonate și întoarce un rezultat de același tip. Metoda prin care se realizează această deducție va fi studiată în continuare la PP.

Deși nu e întotdeauna necesar, specificarea manuală a tipului unei funcții e considerată "good practice". Editați fișierul de mai devreme astfel încât să arate așa:

```
-- intoarce modulul dublului celui mai mare dintre cele doua
   argumente
```

```
myFunc :: Int -> Int -> Int
myFunc x y = abs (2 * max x y)
```

Observați că, acum, tipul arătat de `:t` este cel specificat și primiți o eroare dacă încercați să pasați ca argumente numere în virgulă mobilă.

subsubsection[] Pattern matching

Vom scrie acum o altă funcție prin care ne propunem să calculăm, în mod recursiv, suma tuturor elementelor dintr-o listă. Pentru o listă vidă aceasta este 0, altfel este capul listei plus suma celorlalte elemente.

```
{-
 - if este o expresie; cum toate expresiile
 - trebuie sa intoarca o valoare, ramura else
 - este necesara
 -
 - exista deja o functie "sum" predefinita si
 - vrem sa evitam ambiguitatea apelului, motiv
 - pentru care folosim un nume nou.
 - (e interesat de mentionat ca, in Haskell,
 - apostroful este un caracter valid in numele
 - unei functii, i.e. am putea defini o functie
 - sum'; din pacate, aici, pe wiki, aces lucru
 - nu este suportat).
 -}
mySum l = if length l /= 0
```

```
           then head l + mySum (tail l)
           else 0
```

Deși funcționează, implementarea de mai sus nu este foarte elegantă. Putem să ne folosim de pattern matching (ca la TDA-uri).

```
mySum2 [] = 0
mySum2 (x:xs) = x + mySum2 xs
```

Ordinea din fișier este importantă. Patternurile sunt evaluate de sus în jos și ar trebui scrise de la cel mai particular la cel mai general (ultimul fiind un "catch-all").

Dacă lista dată ca argument e vidă, atunci se face match pe primul pattern și rezultatul este 0. Altfel, se trece la pattern-ul următor; aici se fac două legări - capul listei este legat de numele `x`, iar restul de numele `xs` (parantezele din jurul `x:xs` sunt necesare) și este apelată funcția în mod recursiv.
subsubsection[] Tail recursion
Pentru a știi unde să întoarcă execuția după apelul unei funcții, un program ține adresele apelanților pe o stivă. Astfel, după ce execuția funcției se termină, programul se întoarce la apelant pentru a continua secvența de instrucțiuni. În exemplul nostru, apelul `mySum2 [1, 2, 3, 4, 5]` se va evalua în felul următor:

```
mySum2 [1, 2, 3, 4, 5]
1 + mySum2 [2, 3, 4, 5]
1 + (2 + mySum2 [3, 4, 5])
1 + (2 + (3 + mySum2 [4, 5]))
1 + (2 + (3 + (4 + mySum2 [5])))
1 + (2 + (3 + (4 + (5 + mySum2 []))))
1 + (2 + (3 + (4 + (5 + 0))))
15
```

Astfel, pentru a aduna capul listei la suma cozii, trebuie mai întâi calculată această sumă, iar apoi să se revină în funcția apelantă.
Putem folosi un acumulator transmis ca parametru la funcția apelată, eliminând nevoia de a mai ține pe stivă informați despre apelant.

Folosirea unui acumulator în acest scop este un tipar des întâlnit, util pentru că poate permite reducerea spațiului de stivă necesar de la O(n) la O(1). Unele limbaje (e.g. C) nu garantează această optimizare, care depinde de compilator.
Modul în care Haskell asigură tail-recursion o să fie mai clar când vom discuta despre modul de evaluare al funcțiilor. Tot atunci vom vedea și capcanele acestuia.

```
-- Folosim sintaxa "let ... in" pentru a ascunde functia
-- auxiliara folosita si a pastra forma functiei sum (un singur
-- argument)
mySum3 l = let
             sumAux [] acc = acc
             sumAux (x:xs) acc = sumAux xs (x + acc)
             in
             sumAux l 0
```

subsection[] Exerciții
1. Scrieți o funcție cu care să determinați factorialul unui număr. Scrieți o implementare straight-forward, fără restricții, apoi încercați să o faceți tail-recursive.
2. Scrieți o funcție care primește un număr `n` și întoarce al `n`-lea număr din șirul lui Fibonacci (`1, 1, 2, 3, 5...` unde primul `1` are indexul `0`). Scrieți o implementare straight-forward, fără restricții, apoi încercați să o faceți tail-recursive.
3. a) Scrieți o funcție `cat` care primește două liste de același tip și returnează concatenarea lor. Funcția ar trebui să funcționeze exact ca operatorul ++ deja existent în Haskell.

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> [1, 2, 3] ++ [4, 5, 6]
[1,2,3,4,5,6]
```

b) Scrieți o funcție `inv` care primește o listă și întoarce inversul ei. Funcția ar trebui să funcționeze exact ca funcția `reverse` deja existentă în Haskell.

```
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> reverse [1, 2, 3]
[3,2,1]
```

4. Implementați o funcție care primește o listă și o sortează folosind algoritmul:

a) `merge sort` [7]. Ajutați-vă, în implementare, de metodele `take`, `drop` și `div`.
b) `insert sort` [8].
c) `quick sort` [9].
5. Scrieți o funcție care întoarce numărul de inversiuni dintr-o listă.

Într-o listă `L` cu `n` elemente, o inversiune este o pereche de elemente `(L[i], L[j])` astfel încât `L[i] >L[j]` ∧ `i <j; i, j = [0, 1, ... n-1]`

subsubsection[] Linkuri utile

- `Learn you a Haskell for Great Good` [10]
- `Why Functional Programming Matters` [11]
- `Why Haskell Matters` [12]
- `Real World Haskell` [13]
- `Haskell in industry` [14]
- `Tutorials and more` [15]

section[Course: Higher-order functions] Course: Higher-order functions
subsection[] Higher-order functions
In our previous examples, we have seen functions such as (+) or (∗) which are passed as parameter to other functions, e.g. `foldr`. ***A function which expects another function as parameter is called a higher-order function.***

One distinctive functional programming trait relies on defining and using higher-order functions. Haskell makes this programming style very natural.
Functions can be defined in several ways, e.g.

```haskell
f x = x + 1
```

as anonymous functions:

```haskell
f = \x -> x + 1
```

(which is read *"lambda x ..."*) or using pattern matching:

```haskell
f 0 = 1
f 1 = 2
f 2 = 3
```

subsubsection[] Currying
Consider the following function definitions:

```haskell
f1 x y = x + y
f2 x = \y -> x + y
f3 = \x -> \y -> x + y
f4 = \x y -> x + y
```

Read syntactically:

- the first function expects two parameters and returns their additions
- the second function returns a function which adds y to x. For instance `(f2 4)` is a function which adds `4` to an integer
- the third function is similar to the previous one, but defined as an anonymous function

- the fourth function is an anonymous function definition with two parameters

There is no conceptual or functional difference between *any* of the above definitions. From a ***Haskell compiler's view*** all function definitions look exactly as the third one. For instance, `(f1 1)` is a correct function call which will return a function adding `1` to its parameter. `(f1 1)` is called a functional closure, because it creates (closes) a context in which the first parameter is equal to the value `1`.
At the same time, the function call `((f1 1) 2)` is correct, returns `3`, and is equivalent to `(f1 1 2)`.
More generally, functions defined as `f2` or `f3` are called curried (or curry) functions (in honor of Haskell Curry, a mathematician which contributed to functional programming) — they receive parameters *in turn*. The other definitions are called uncurried.
We again stress that, in Haskell, there is no difference between curried and uncurried functions. This is not the case in other functional languages, e.g. Lisp, Scheme.
subsubsection[] Other higher-order functions

Imagine we want to do a certain computation on each member of a list, individually. Let's say, double each element. If we think functionally, we want to take a list `[a,b,c, ...]`, a function `f`, and build the list:

```
[f a, f b, f c, ...]
```

It turns out we can write such a function relying on fold:

```
pp_map f = foldr (\x y->(f x):y) []
```

There is also a shorter way to write `pp_map`:

```
pp_map f = foldr ((:).f) []
```

In the expression `(:).f`, `.` (the dot) stands for functional composition. Consider the anonymous function:

```
\x->x+1
```

which takes a parameter `x` and returns its natural successor. We can compose it as follows:

```
(\x -> 2*x).(\x->x+1)
```

to obtain the function `\x ->2*(x+1)`. In general, the function call `(f.g) x` is equivalent to `f (g x)`.
To understand `(:).f` let us write the cons `(:)` as an anonymous function:

```haskell
\h t -> h:t
```

Hence, the following expressions are equivalent:

```haskell
((:).f)
((\h t -> h:t).f)
((\h->\t->h:t).f)
(\t->(f h):t)
```

A natural mistake is to consider `((:).f)` as invalid, since it receives two parameters, and we cannot compose such a function with one receiving only one parameter. However, every function in Haskell receives one parameter and returns a value (which can be a function or a primitive value).
There are other examples of useful higher-order functions. We illustrate them using examples:

```haskell
filter (>3) [1,2,3,4,5,2]
zipWith (+) [1,2,3] [3,2,1]
```

Guess what each function does, by testing it on several lists.
subsubsection[] Haskell implementation of foldl
In the previous lecture, we have defined the fold left procedure as:

```haskell
foldl op acc []     = acc
foldl op acc (h:t) = foldl op (op h acc) t
```

Instead, in Haskell, `foldl` is implemented as follows:

```
foldl op acc []     = acc
foldl op acc (h:t) = foldl op (op acc h) t
```

Note the call of the `op` function.
subsubsection[] An exercise in modular programming with higher-order functions
Let us consider the task of matrix multiplication in Haskell. Example:

```
1  2  3         1  0  0        1+3  2  2+3      4   2   5
4  5  6    x    0  1  1    =   4+6  5  5+6  =   10  5   11
7  8  9         1  0  1        7+9  8  8+9      16  8   17
```

paragraph[Matrix representation in Haskell] Matrix representation in Haskell

The basic representation building-block in Haskell is the list. We can represent matrices in Haskell as a list where each element is a row (hence a list of elements). Example:

```
m = [[1,2,3],[4,5,6],[7,8,9]]
```

A good warmup exercise is to write a nice display function for matrices. We transform each element of a line into a string:

```
displayline l = map (\e->(show e)++" ") l
```

The code can be improved for legibility:

```
displayline = map ((++" ").show)
```

Next, we fold the list into a string with a newline character:

```haskell
displayline l = foldr (++) "\n" (map ((++" ").show) l)
```

and simplify again, by expressing the pipeline function calls as functional composition:

```haskell
displayline :: Show a => [a] -> [Char]
displayline = (foldr (++) "\n") .  (map ( (++ " ") . show ) )
```

Note that we need to explicitly state the type of display. Sometimes, in Haskell, an explicit type declaration is required. For now, we omit details.
Next, we apply the above process on all matrix lines:

```haskell
display :: Show a => [[a]] -> [[Char]]
display =
    let bind = foldr (++) "\n"
    in bind.(map (bind . (map ((++" ") . show ) ) ) )
```

Notice that we have separated the binding process, because we reuse it.
Finally, we make all matrices displayable:

```haskell
instance (Show a) => Show [[a]] where
    show =
        let bind = foldr (++) "\n"
        in bind.(map (bind.(map ((++" ").show ) ) ) )
```

More details about this implementation (e.g. instances, classes) will be given in future lectures.
paragraph[Matrix multiplication] Matrix multiplication

subparagraph[Step 1: Transposition] Step 1: Transposition

Matrix multiplication operates on the lines of the first matrix and columns of the second. We transpose the second matrix, so that we now operate on lines on both matrices. The following code extracts the first line from a matrix `m`:

```
map head m
```

A matrix `m` without its first column is:

```
map tail m
```

Finally transposition is given by:

```
transpose ([]:_) = []
transpose m = (map head m) : transpose (map tail m)
```

Notice that the *basis case* corresponds to a list containing empty lists.
subparagraph[Step 2: Computing multiplication] Step 2: Computing multiplication

To compute the `i,j`th element of the multiplication matrix, we need to multiply per element the `i`th line by the `j`th column:

```
zipWith (*) li cj
```

and then add-up the values:

```
foldr (+) 0 (zipWith (*) li cj)
```

To obtain the `i`th line of the multiplication matrix, we need to repeat the above process for each column of the second matrix, in other words, for each line of its transposition:

```
map (\col -> foldr (+) 0 (zipWith (*) li col) ) (transpose m2)
```

Finally, to obtain the multiplication matrix, we need to compute all its lines, hence:

```
mult m1 m2 =
  map (\line -> map (\col -> foldr (+) 0 (zipWith (*) line col) )
    (transpose m2) ) m1
```

paragraph[Matrices as images] Matrices as images

A matrix can be used to represent a rasterized image (a collection of pixels). In this example, we consider that pixels can have values: ' ' (white), '.' (grey) and '*' (black).
Higher-order functions can be naturally used to represent image-transformations, for instance, flipping:

```
flipH = map reverse
flipV = reverse
```

Rotations:

```
rotate90left = flipV.transpose
rotate90right = flipH.transpose
```

The *negative* of an image:

```
invert = map (map (\x->if x=='*' then ' ' else '*') )
```

Scaling an image horizontally:

```
scalex = foldr (\h t->h:h:t) []
```

Scaling vertically:

```
scaley = map scalex
```

Balanced scale:

```
scale = scalex . scaley
```

Or just a random sequence of operations:

```
rand = foldr (.) id [rotate90left, invert, scale]
```

subsection[Lab: Higher-order functions] Lab: Higher-order functions section[] Funcții
Scopul laboratorului:

- Semnificația termenului *aplicație*
- Definirea funcțiilor anonime în Haskell
- Curry vs uncurry (și de ce nu contează în Haskell)
- Combinatori vs Închideri funcționale

subsection[] Funcții de ordin superior
Funcțiile de ordin superior sunt funcții care lucrează cu alte funcții: le primesc ca parametrii sau le returnează.

Pentru a înțelege importanța lor, vom da următorul exemplu: ne propunem să scriem două funcții care primesc o listă și returnează, într-o nouă listă

- toate elementele pare
- toate elementele mai mari decât 10

```
evenElements [] = []
evenElements (x:xs) = if even x
                      then x : evenElements xs
                      else evenElements xs

greaterThan10 [] = []
greaterThan10 (x:xs) = if x > 10
                       then x : greaterThan10 xs
                       else greaterThan10 xs
```

În loc de `if`, puteți folosi următoarea sintaxă:

```
myFunction x
        | x < 10    = "One digit"
        | x < 100   = "Two digits"
        | x < 1000  = "Three digits"
        | otherwise = "More than four digits"
```

Testăm funcțiile scrise:

```
*Main> evenElements [1..20]
[2,4,6,8,10,12,14,16,18,20]
*Main> greaterThan10 [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
```

Observăm că funcțiile definite mai sus sunt foarte asemănătoare. De fapt, doar condiția verificată în `if` diferă. Scriem, deci, o funcție generală care primește o funcție pentru testarea elementelor:

```haskell
-- In primul pattern, nu folosim functia de testare, deci nu ne
    intereseaza ca aceasta
-- sa fie legata la un nume, lucru marcat prin "_"
myFilter _ [] = []
myFilter test (x:xs) = if test x
                       then x : myFilter test xs
                       else myFilter test xs
```

Acum putem rescrie funcțiile noastre, într-un mod mai elegant, utilizând funcția de filtrare:

```haskell
evenElements = myFilter even

greaterThan10 = myFilter (> 10)
```

subsection[] Currying vs. uncurrying
Currying (numit după `tizul Haskell-ului` [16]) este procesul prin care, dintr-o funcție care ia mai multe argumente, se obține o secvență de funcții care iau un singur argument.

De exemplu, dintr-o funcție de două argumente `f : X × Y → Z` se obține o funcție `curry(f) : X → (Y → Z)`. Noua funcție `curry(f)` primește un argument de tipul `X` și întoarce o funcție de tipul `Y → Z` (adică o funcție care primește un argument de tipul `Y` și întoarce un rezultat de tip `Z`). Considerând operatorul → asociativ la dreapta, putem omite parantezele, i.e. `curry(f) : X → Y → Z`.
Operația inversă se numește "uncurry": `f : X → Y → Z`, `uncurry(f): X × Y → Z`.
Întorcându-ne la funcția de filtrare definită mai devreme, care este tipul ei?

```
*Main> :t myFilter
myFilter :: (a -> Bool) -> [a] -> [a]
```

Și în `laboratorul trecut` [17], am observat că nu există o separare între domeniu și codomeniu, de genul
`(a ->Bool) x [a] ->[a]`.

Acest lucru se datorează faptului că, în Haskell, toate funcțiile iau un singur argument. Alternativ, putem spune despre ele că sunt *curried.*

Tipul funcției noastre trebuie interpretat astfel:
primește ca argument o funcție de tipul `a ->Bool` (ia un argument și întoarce o booleană) și întoarce o funcție de tipul `[a] ->[a]` (ia o listă și întoarce o listă de același tip).
De aceea, în exemplul de mai sus am putut definit `evenElements` (o funcție care ia o listă și returnează o listă) ca fiind `myFilter even` care are exact acest tip, i.e. `[a] ->[a]`.

Haskell pune la dispoziție funcțiile `curry` și `uncurry`.

```
*Main> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
*Main> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Funcțiile primite, respectiv returnate de `curry` și `uncurry` iau tot un singur argument, numai că acesta este un tuplu.

```
*Main> let evenElements = (uncurry myFilter) even

<interactive>:12:39:
    Couldn't match expected type '(a0 -> Bool, [a0])'
                with actual type 'a1 -> Bool'
    In the second argument of 'uncurry', namely 'even'
    In the expression: (uncurry myFilter) even
    In an equation for 'evenElements':
        evenElements = (uncurry myFilter) even
*Main> let evenElements l = (uncurry myFilter) (even, l)
*Main>
```

subsection[] Funcții anonime
Ne propunem să scriem o funcție care primește o listă și întoarce toate elementele ei divizibile cu 5. Având deja o funcție de filtrare, putem scrie:

```haskell
testDiv5 x = mod x 5 == 0
multiplesOf5 = myFilter testDiv5
```

Această abordare funcționează, însă am poluat spațiul de nume cu o funcție pe care nu o folosim decât o singură dată - `testDiv5`.
Funcțiile anonime (cunoscute și ca expresii lambda) sunt funcții fără nume, folosite des în lucrul cu funcții de ordin superior.
În Haskell, se folosește sintaxa: `\x ->x + 5`. Funcția definită ia un parametru și returnează suma dintre acesta și 5. Caracterul \ este folosit pentru că seamănă cu λ (lambda).
Rescriind funcția noastră, obținem:

```haskell
multiplesOf5 = myFilter (\x -> mod x 5 == 0)
```

Următoarele expresii sunt echivalente:

```haskell
f x y = x + y
f x = \y -> x + y
f = \x y -> x + y
```

Nu există vreo diferență între `\x y ->x + y` și `\x ->\y ->x + y`.

subsection[] Combinatori și închideri funcționale

Un combinator este o funcție care nu se folosește de *variabile libere*. O variabilă este liberă în definiția unei funcții dacă nu se *referă* la niciunul dintre parametrii formali ai acesteia.

```
-- Exemple de combinatori
f1 = \a -> a
        -- prin f1 îi dăm un nume funcției anonime de după egal
        -- această funcție anonimă are un parametru formal, "a"
        -- definiția ei conține doar evaluarea lui a
        -- este un combinator deoarece nu folosește nicio altă
   variabilă în afară de a

f2 = \a -> \b -> a
f3 = \f -> \a -> \b -> f b a

-- Exemple de "ne-combinatori"
f1_no = \a -> a + x
  where x = some_other_expr
  -- x este o variabilă liberă în contextul definiției funcției anonime
   de după egal

-- Pentru mai multe exemeple, continuă să citești
```

Închiderile funcționale (closures) sunt opusul combinatorilor - se folosesc de variabile libere în definiția lor. Cu alte cuvinte, sunt funcții care, pe lângă definiție, conțin și un environment de care se folosesc; acest environment conține variabilele libere despre care vorbeam. Denumirea provine din faptul că environment-ul nu este menționat explicit, el este determinat implicit; spunem că funcția se închide peste variabilele (*libere*) a, b, etc.
Diferența dintre combinatori și închideri funcționale este de multe ori subtilă. Să aruncăm o privire asupra unuia dintre exemplele de mai sus.

```
flip = \f -> \a -> \b -> f b a -- was f3
```

După cum am spus, acesta este un combinator. Ținând cont de faptul că în Haskell currying-ul funcțiilor (gândiți-vă la *parantezarea tipului*) nu contează, putem rescrie această funcție astfel:

```
flip f = \a -> \b -> f b a
       -- ^^^^^^^^^^^^+^^^^
       --     f este o variabilă liberă în contextul funcției anonime de
    după egal
```

Ce ne returnează funcția flip, atunci când îi dăm un singur argument? O *închidere funcțională* peste f. Prin urmare, funcția mod_flipped de mai jos este o închidere funcțională care atunci când primește 2 parametrii va folosi funcția mod (stocată - într-un mod neobservabil – în contextul său) și va returna restul împărțirii celui de-al doilea la primul.

```
mod_flipped = flip mod
```

Alte exemple de închideri funcționale:

```
mod3 x = mod x 3 -- închidere funcțională peste funcția mod din Prelude
    și constanta 3
plus5 x = x + 5  -- închidere funcțională peste funcția (+) și
    constanta 5
(+5)             -- aceeași ca mai sus;
                 --     (+) este o funcție care primește 2 argumente și
    se evaluează la suma lor
                 --     (+5) este închiderea funcțională rezultată în
    urma "hardcodării" unuia dintre argumente
```

Mai multe detalii teoretice despre închideri funcționale, variabile legate/nelegate și combinatori se vor discuta în cadrul cursului.

Puteți găsi și alte exemple de închideri funcționale care se găsesc în acest suport de laborator?

subsection[] Exerciții updated 1. Scrieți o funcție ce primește un număr x și o listă de numere de forma $[a_1, a_2, ...a_n]$ ca parametrii și calculează $x - a_1 - a_2 - ... - a_n$.
2. Inversați o listă folosind foldl.
3. Scrieți o închidere funcțională care elimină caracterul 'a' dintr-un string primit ca parametru. Scrieți o implementare ce utilizează o funcție de tip fold și alta care se folosește funcția filter.
4. Implementați o funcție ce primește ca parametru o lista de string-uri și returnează concatenarea lor folosind fold.
5. Se dau două cuvinte de aceeași lungime. Să se numere câte caractere sunt diferite în cele doua șiruri.
6. Implementați o funcție ce primește 2 parametri: un șir de cifre de forma "11122211133312" și un număr K. Această funcție trebuie să facă următorii pași de K ori:

1. se grupează cifrele identice în felul următor "111222111333112" ->["111", "222", "111", "333", "11", "2"]
2. pentru fiecare element din lista rezultată se generează un element de forma (numărul de apariții al cifrei din șir, cifra din șir) ex: ["111", "222", "111", "333", "11", "2"] ->[("3", "1"), ("3", "2"), ("3", "1"), ("3", "3"), ("2", "1"), ("1", "2")]
3. se face flatten pe lista rezultată la pasul anterior ex: [("3", "1"), ("3", "2"), ("3", "1"), ("3", "3"), ("2", "1"), ("1", "2")] ->"313231332112"

subsection[] Exerciții 1. Definiți o închidere funcțională care prefixează [1,2,3] la o listă primită ca parametru.
2. Definiți o funcție de ordin superior care primește o funcție și un număr, și aplică de două ori funcția pe numărul respectiv.
3. Definiți o funcție care primește un operator binar, și întoarce același operator în care ordinea parametrilor a fost inversată *(e.g. 1/3 ->3/1)*
4. Implementați și testați funcțiile:

1. foldl
2. foldr
3. map
4. filter

5. zipWith
6. compunerea de funcții (operatorul `.` din Haskell)

Dacă nu cunoașteți vreuna dintre funcții, o puteți căuta pe Hoogle pentru a vedea tipul și scopul ei.

5. Implementați, folosind foldl sau foldr:

1. map
2. filter

subsection[] Linkuri utile

- `Hoogle - motor de căutare pentru funcții Haskell`[18]
- `Higher order functions - haskell.org`[19]
- `Higher order functions - learnyouahaskell.com`[20]

subsection[Lab: Even more higher-order functions] Lab: Even more higher-order functions section[] Aplicații cu funcții de ordin superior
Scopul laboratorului este de a exersa lucrul cu funcții de ordin superior.
subsection[] Recapitulare
O funcție de ordin superior este *o funcție care primește ca argument sau returnează o funcție.*

```haskell
applyTwice f x = f (f x)
adder x = (+x)
```

subsubsection[] Folduri Două funcții de ordin superior foarte importante sunt foldurile pe liste: `foldl` și `foldr`. Aceasta combină, în mod recursiv, elementele unei liste cu o valoare default, pentru a obține un rezultat.

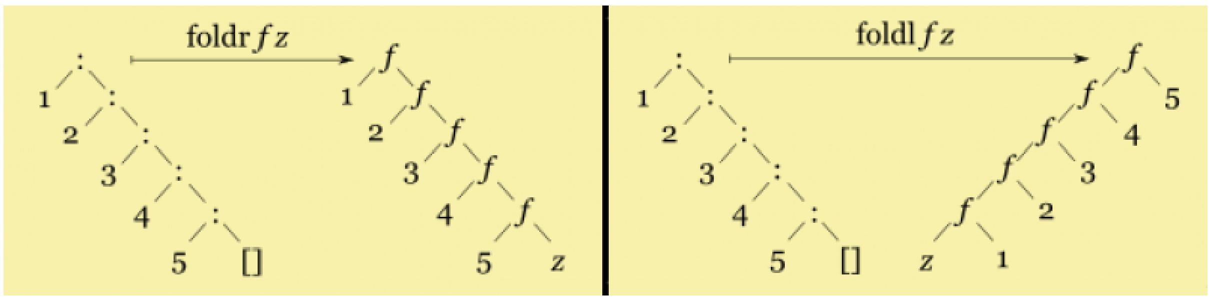
```haskell
foldr (:) [] l          -- intoarce o lista identica cu l
foldl (flip (:)) [] l   -- intoarce inversul listei l
```

O variantă importantă a lui `foldl` este `foldl'`, care se găsește în modulul `Data.List`.

```
Prelude> import Data.List
Prelude Data.List> :t foldl'
foldl' :: (a -> b -> a) -> a -> [b] -> a
```

Puteți citi `aici` [21] despre diferențele dintre aceste trei folduri.

O vizualizare utilă



:
Tipurile acestor funcții sunt:

```
Prelude> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
Prelude> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Nu trebie să rețineți pe de rost tipurile. Încercați să înțelegeți ce exprimă și de ce sunt așa.

Alte funcții de ordin superior des întâlnite: `map`, `filter`, `zipWith`, `flip`.
subsection[] Exerciții
1. Fie două matrici reprezentate ca liste de liste. În rezolvarea exercițiilor de mai jos, puteți folosi doar funcții de ordin superior (împreună cu `take` și `drop`).
Implementați funcții care să returneze:

- linia `i` dintr-o matrice
- elementul `(i, j)` dintr-o matrice
- suma a două matrici
- transpusa unei matrici
- produsul a două matrici

2. O imagine poate fi reprezentată ca o matrice de caractere (numiți, în continuare, "pixeli"). Considerăm că avem trei tipuri de pixeli: `'.'`, `'*'`, `' '`
Implementați următoarele funcții:

- flip orizontal, flip vertical, rotație de 90 în sens trigonometric și invers trigonometric
- negativul ('*' si '.' devin ' ', iar ' ' devine '*')
- scalarea unei imagini cu x unități
- alipirea a două imagini (cu aceeași înălțime) pe orizontală
- alipirea a două imagini (cu aceeași lungime) pe verticală
- crop orizontal de la de la coloana x la coloana y
- crop vertical de la linia x la linia y
- Implementați suprapunerea unei imagini peste o alta (având aceeași dimensiune)

subsection[] Resurse
Puteți folosi următoarele pentru a vă testa funcțiile:

```
l0="        ***** **            ***** **    "
l1="     ******  ****        ******  ****   "
l2="    **   *  *  ***      **   *  *  ***  "
l3="   *    *  *    ***    *    *  *    *** "
l4="       *  *      **        *  *      ** "
l5="      ** **      **       ** **      ** "
l6="      ** **      **       ** **      ** "
l7="    **** **      *      **** **      *  "
l8="   * *** **     *      * *** **     *   "
l9="      ** *******          ** *******    "
l10="      ** ******           ** ******     "
l11="      ** **               ** **         "
l12="      ** **               ** **         "
```

```
l13="      ** **               ** **         "
l14=" **   ** **          **   ** **         "
l15="***   *  *          ***   *  *          "
l16=" ***    *            ***    *           "
l17="  ******              ******            "
l18="    ***                 ***             "

img =
    [l0,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10,l11,l12,l13,l14,l15,l16,l17,l18]

m1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
m2 = [[1, 0, 0], [0, 1, 1], [1, 0, 1]]

summ1m2 = [[2,2,3],[4,6,7],[8,8,10]]
prodm1m2 = [[4,2,5],[10,5,11],[16,8,17]]

-- functii care printeaza intr-un mod human-readable o matrice sau o
    imagine
display :: (Show a) => ([a] -> String) -> [[a]] -> IO ()
display displayLine = putStr . foldr (++) "" . map displayLine

-- folositi pentru a afisa o matrice de numere (ex. 1)
displayMat :: (Show a) => [[a]] -> IO ()
displayMat = display (foldr (\x acc -> show x ++ "   " ++ acc) "\n")

-- folositi pentru a afisa o "imagine" - matrice de siruri (ex. 2)
displayImg :: [String] -> IO ()
displayImg = display (++ "\n")
```

section[Course: Types in functional programming] Course: Types in functional programming section[] Types in functional programming

subsection[] Typing in programming languages subsubsection[] Strong vs weak typing Consider the following example from Javascript, where the operator + is applied on arguments of different types. The result is given below:

```
[] + []  =  ""
[] + {}  = "[object]"
{} + []  = 0
{} + {}  = null
```

where [] is the empty array and {} is the empty object. The result is quite surprising, and unpredictable. To explain it, we need to look at the implementation of the Javascript interpreter. In Javascript, there is a distinction between primitive and non-primitive values. Arrays (like []) and objects (like {}) are considered non-primitive. Since the operation + is performed on primitive values, Javascript attempts to convert the operands to primitive. A pseudocode for this is shown below:

```
     Convert(value) {
        if (value.valueOf() is a primitive) return it
   //conversion to integer works?
        else if (value.toString() is a primitive) return it    //else
   convert to string, if possible
             else error
      }
```

The conversion of [] to string yields the empty string, while the conversion of {} to string yields "[object]". This explains the first two results. For the latter, we must know that {} can also be interpreted as a code-block, and this is the case here. Hence, the Javascript interpreter sees:

```
+ []
+ {}
```

where + is interpreted as a unary operator. JavaScript has such an implementation overloading for +. Without going into more details, `+ x` behaves like a ***conversion to Number*** for x. Thus, `[]` converted to `Number` is `0`, while `{}` converted to `Number` is `NaN` (not a number).
paragraph[Morale] Morale

The Javascript treatment of + is unimportant by itself, and it has some advantages for the programmer (although our example shows otherwise). However, we can note that, by allowing any kind/type of operand for +:

- complicates the semantics of the language (the programmer needs to known about conversion to primitives and conversion to numbers)
- reduces errors (no Javascript error was signalled above), but makes programs more prone to bugs.
- makes a program more expressive (+ can be used in several ways) but sometimes difficult to use.

In programming language design, there is a fundamental tension between expressiveness and typing. Typing is a means for enforcing constraints on what is deemed as a *correct program*. It may be the case that, *conceptually correct* programs may not be accepted as correct from a typing perspective (this situation does seldom occur in Haskell).
A strongly-typed programming language, is more coercive with respect to typing constraints. In functional programming:

- Haskell
- Scala

are considered strongly-typed. In imperative / OOP programming, the languages:

- Java
- C++
- Scala

are considered strongly-typed.
A weakly-typed programming language is more relaxed w.r.t. typing constraints. For instance, in functional programming:

- Racket (Lisp)
- Clojure

are considered weakly-typed. In imperative / OOP programming, the languages:

- C
- Python
- PHP
- Javascript

(and especially the latter) are considered weakly-typed. In the latter languages, types are usually reduced to primitive constructs (programmers cannot create new types), or the type construction procedure is very simplistic. For instance, in `Racket` (formerly known as `Scheme`), which weakly-typed:

- lists can hold any values (e.g.) `'(1 #t "String")`, which means that the type for lists is primitive (not composed), and is simply `list`.
- functions can return values of different types - the type for functions is also primitive - `#procedure`.

However, type verification is not absent in weakly-typed languages, including Racket/Scheme. For instance, the call `(+ 1 '())` will produce an error since the plus operator is called on values with invalid types.

We shall discuss Scheme/Racket in more detail later. It is worth noting that, in Racket there exists extensions (Typed Racket) which allow programmers to define and compose types to some extent.

The weakly-typed vs strongly-typed classification is not rigid and is subject to debate and discussion. There is no objective ***right-answer***. For instance, here, the language `C` is viewed as weakly-typed. We illustrate a small motivating example:

```
int f (int x) {
    if (x != 0)
        return 1;
    return malloc(100);
}
```

In principle, the function `f` can return an integer, or a pointer to any object (of any type), and this is allowed by the compiler (which does issue a warning). Compared to, e.g. Java, this makes the `C` type system more relaxed.

There are valid arguments for considering `C` as strongly-typed and, as said before, there is no right answer.

subsubsection[] Compile-time vs runtime typing

This classification is done w.r.t. the moment when type inference occurs:

- during the compilation of a program

- at runtime

The former is also called static typing, while the latter - dynamic typing. In the literature, static and dynamic typing are also used with other meanings, hence, here, we prefer the terms compile-time and runtime.
The imperative/OOP languages:

- Java
- Scala
- C\C++

perform compile-time type checking, as well as the functional languages:

- Haskell
- Scala

The imperative/OOP languages:

- Python
- PHP
- Javascript

perform runtime type checking, as well as the functional languages:

- Scheme/Racket (Lisp)
- Clojure

Compile-time type checking is preferred for strongly-typed languages: the complexity of type verification is delegated to the compiler. Conversely, in weakly-typed languages, type verification is simpler, hence it can be performed by the interpreter, at runtime. Sometimes, a compiler may be absent. This is not a golden rule but merely an observation.

While runtime type checking is simpler to deploy, it has the disadvantage of not capturing typing bugs. Consider the following program in Racket:

```
(define f (lambda (x) (if x 1 (+ 1 '()))))
(f #t)
```

The function receives a value `x` which must be a boolean. In the program above there is no error, even though `(+ 1 '())` is an incorrectly-typed function call. However, in the execution of the program (i.e. `(f #t)`), the else branch of the function is not reached, hence no typing verification is performed.

Hence, runtime type checking only catches bugs on the current program execution trace.

subsection[] Typing in Haskell

subsection[] Type inference in Haskell

Haskell implements the `Hindley-Milner type inferenc` `algorithm` [22]. In what follows, we present a simplified, and more easy-to-follow, but incomplete algorithm which serves as an illustration for the main concepts underlying the original one.
subsubsection[] Intro
Consider the following expressions, and their types:

```
\x -> x + 1 :: Integer -> Integer
\x -> if x then 0 else 1 :: Bool -> Integer
zipWith (:) :: [a] -> [[a]] -> [[a]]
\f -> (f 1) + (f 2) :: (Integer -> Integer) -> Integer
```

We can see via the above example that types are constructed according to the following grammar:

```
type ::= <const_type> | <type_var> | (<type>) | type -> type | [<type>]
```

This grammar only tells half the story regarding Haskell typing, however, for the purposes of this lecture, this view suffices. According to the above grammar, types can be:

- constant types (e.g. `Integer`, `String`)
- type variables (e.g. `a` - which are usually used to designate any possible type, as in `[a]`)
- function types (e.g. `Integer ->Integer` or `(Integer ->Integer)` if this type appears in a larger type expression)

- list types (e.g. `[a]`).
- any combination of the above rules.

subsubsection[] Expression trees
We assume each Haskell expression is constructed via the following *construction rules*:

- functional application (e.g. `take 2 [1,2,3]`)
- function definition (e.g. `\x->[x+1]`)

We note that many Haskell definitions can be seen as such. For instance:

```
g f = (f 1) + 1
<code>

can be seen as:

<code haskell>
g = \f -> (f 1) + 1
```

Hence, we can take any Haskell expression, and construct a tree, in which each node represents a construction rule, and children represent sub-expressions:

- for functional applications, the children are the function name and the parameters
- for function definition, the children are the variable/variables and the function body.

For example, consider the expression tree for the function g shown previously (we use tabs to illustrate parent/child relationship):

```
\f -> (f 1) + 1
  f
  (f 1) + 1
    (+)
    (f 1)
      f
      1
    1
```

In what follows, we shall use expression trees to perform type inference.
subsubsection[] Typing rules
We introduce the following typing rules:
paragraph[Rule (TVar)] Rule (TVar)

If `v` is bound to a constant expression e of type `ct`, then `v :: ct`
paragraph[Rule (TFun)] Rule (TFun)

If `x :: t1` and `e :: t2` then `\x ->e :: t1 ->t2`
paragraph[Rule (TApp)] Rule (TApp)

If `f :: t1 ->t2` and `e :: t1` then `(f e) :: t2`
The above rule can be naturally generalised:
If `f :: t1 ->t2 ->... ->tn ->t` and `e1 :: t1`, ..., `en :: tn` then `(f e1 ... en) :: t`
In what follows, we will use these rules to make judgements on our types. These rules have a twofold usage:

- deduce the type of an expression, based on *existing knowledge*
- make hypotheses regarding the type of an expression (we shall not focus on this aspect in the presentation))

subsubsection[] Type inference stage 1: Expression tree construction
Type inference for an expression e can be seen as having two stages. In the first stage, we:

- construct the expression tree of e
- make hypotheses regarding the types of yet untyped expressions (e.g. variables).

We illustrate the first stage on the previous definition of g:

```
\f -> (f 1) + 1 :: ?
  f :: tf (here we introduce tf as the type of f. This is a type
    hypothesis)
  (f 1) + 1 :: ?
    (+) :: ?
    (f 1) :: ?
       f :: t1 -> t2 (this is another hypothesis, stemming from the
    fact that f is applied on 1)
       1 :: ?
    1 :: ?
```

subsubsection[] Type inference stage 2: Rule application
In this stage, we start from the previously-build tree, and:

- apply typing rules to deduce types for new sub-expressions

- perform type unification: This is one aspect that we shall not elaborate on, in this lecture.

This is equivalent to a bottom-up tree traversal: We start from the leaves, and progress to the root (i.e. the expression to be typed).
Without delving into details, type unification is an important ingredient, because it allows us to infer the most general type of an expression. Consider the following Haskell expression: `\f x ->(f x,f 1)`, which defines a function that takes another function `f`, a value `x` and returns a pair: the first element of the pair is the application `f x`, while the second - `f 1`:

- initially, we do not know what `f` is, hence it has the most general type (say) `tf` - it can be anything;
- judging by the application `f x`, we deduce that `f` must be a function, of type `a->b`, where `x::a`
- judging by the application `f 1`, we deduce that `a` must be an `Integer`

The unification process combines the information collected so far:

- `tf` must unify (coincide with) `a ->b`
- `a` must unify with `Integer`

The final type for the expression is: `(Integer ->b) ->Integer ->(b, b)`
We illustrate the second stage of the type inference on the same example:

```
\f -> (f 1) + 1 :: tf -> Integer
  f :: tf (via (TFun))
  (f 1) + 1 :: Integer (via (TApp))
    (+) :: Integer -> Integer -> Integer (this we know from Prelude,
    after the type synthesis of (+), also t2 must unify with Integer)
    (f 1) :: t2 (via (TApp); also, t1 must unify with Integer)
       f :: t1 -> t2
       1 :: Integer (via (TVar), from Prelude)
    1 :: Integer (via (TVar), from Prelude)
```

The (pseudo)-algorithmic procedure concludes with the following answer:

- `g :: tf ->Integer`, where
  - `tf` unifies with `t1 ->t2`
  - `t2` unifies with `Integer`
  - `t1` unifies with `Integer`

After unification, the result is shown to the programmer: `g :: (Integer ->Integer) ->Integer`
Exercises. Find the type of the following expressions, by applying the type synthesis pseudo-algorithm:

- `map (\x->[x+1])`
- `\f x->if x then f x else x`
- `g f x = x && (f x)`
- `g f = f (g f)`

section[Course: Abstract datatypes] Course: Abstract datatypes subsection[] Abstract Datatypes

subsubsection[] Intro
An Abstract Datatype relies on *functions* to describe the possible values of a type. We start with a simplistic example:

```
data Nat = Zero | Succ Nat
```

- the expression `data Nat` introduces a new type in the programming language
- after =, the base constructors of the type follow. In Haskell, all constructors must begin with a capital letter
- `Zero` is a nullary constructor.
- `Succ Nat` designates an internal constructor, which expects a natural number (of type `Nat`). A value (`Succ x`) is of type `Nat` (i.e. a natural number), and we may be tempted to see it as a function call, which *returns* the successor of `x`

`Zero` and `Succ` are called data constructors in Haskell. `Nat` is called a `type` or `type-constructor`. We shall distinguish between the two in a later lecture.

Note that the *internal* representation of an ADT, as perceived by the programmer, is *abstract*. We may see values as *calls* of special functions - data constructors. Except for their meaning (and language-level implementation), data constructors behave exactly as functions. For instance:

- `Zero :: Nat`
- `Succ :: Nat ->Nat`

We continue the example with addition:

```haskell
add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = Succ (add x y)
```

An important observation is that the pattern matching mechanism in Haskell relies on data constructors, and their applications. For instance, the following definition is a correct usage of the pattern matching mechanism:

```haskell
f (1:y:[]) = ...
```

I uses the data constructors `(:)` and `[]` for lists, as well as the data constructor `1` for integers. The pattern describes any list of integers which starts with a `1` and contains exactly two elements.
subsubsection[] Monomorphic List implementation

In what follows, we give an implementation for the type ***List of integers***. This type is called ***monomorphic***, since our list can only contain elements of a single type (integer):

```haskell
data IList = Void | Cons Integer IList

app :: IList -> IList -> IList
app Void l = l
app (Cons h t) l = Cons h (app t l)

convert :: IList -> [Integer]
convert Void = []
convert (Cons h t) = h : (convert t)

mfoldl :: (b -> Integer -> b) -> b -> IList -> b
mfoldl op acc Void = acc
mfoldl op acc (Cons h t) = mfoldl op (op acc h) t

mfoldr :: (Integer -> a -> a) -> a -> IList -> a
mfoldr op acc Void = acc
mfoldr op acc (Cons h t) = op h (mfoldr op acc t)

convert2 :: IList -> [Integer]
convert2 = mfoldr (:) []

showl :: IList -> String
showl = show . convert2
```

In the above code, we have defined some basic list operations, such as `app` (list concatenation) and `convert`, which transforms a list of type `IList` to a conventional Haskell list (of type `[Integer]`).
We have also implemented the two folding procedures for `IList`. Note the type signature of each fold. Finally, we have used folds to provide an alternative implementation for convert, as well as for converting lists to strings.

subsubsection[] Propositional Logic in Haskell Abstract Datatypes are a natural way to define more elaborate data-structures, such as propositional formulae. We give a possible definition below:

```
data Formula = Var String | And Formula Formula | Or Formula Formula |
    Not Formula
```

We observe that:

- `Var :: String ->Formula`
- `And :: Formula ->Formula ->Formula`
- `Or :: Formula ->Formula ->Formula`
- `Not :: Formula ->Formula`

A good exercise consists in the implementation of a display function for formulae:

```
fshow :: Formula -> String
fshow (Var v) = v
fshow (And f1 f2) = "("++(fshow f1)++" ^ "++(fshow f2)++")"
fshow (Or f1 f2) = "("++(fshow f1)++" V "++(fshow f2)++")"
fshow (Not f) = "~("++(fshow f)++")"
```

Next, we implement a function `push`, which pushes negation *inward*:

```
1 push :: Formula -> Formula
2 push (Var v) = (Var v)
3 push (Not (Var v)) = Not (Var v)
4 push (Not (And f1 f2)) = Or (push (Not f1)) (push (Not f2))
5 push (Not (Or f1 f2)) = And (push (Not f1)) (push (Not f2))
6 push (And f1 f2) = And (push f1) (push f2)
7 push (Or f1 f2) = Or (push f1) (push f2)
```

Notice, at lines 4,5 the implementation of de-Morgan's laws.

Finally, we implement a function which computes the truth-value of a formula, under a certain interpretation. First, we define the type `Interpretation`:

```haskell
type Interpretation = String -> Bool

i :: Interpretation
i "x" = True
i "y" = False
i "z" = True
```

In the first line, we have defined a type-alias: `Interpretation` is a function from strings to booleans. We have also implemented a three-variable interpretation, for testing purposes. Next, we implement `eval`:

```haskell
eval :: Interpretation -> Formula -> Bool
eval i (Var v) = i v
eval i (Not f) = not(eval i f)
eval i (And f1 f2) = (eval i f1) && (eval i f2)
eval i (Or f1 f2) = (eval i f1) || (eval i f2)
```

subsubsection[] Monomorphic Trees in Haskell

```haskell
data ITree = Leaf | Node ITree Integer ITree
```

Note that:

- `Leaf :: ITree`
- `Node :: ITree ->Integer ->ITree ->ITree`

Next, we implement a folding operation on Trees. The key to it is to conceptually define what folding should do on Trees. To grasp an intuition, recall that:

- `foldr (:) []` is the identity function on lists. Hence, `foldr (:) [] [1,2,3]` produces `[1,2,3]`.
- also, recall that the map operation can be defined as: `\f ->foldr ((:).f) []`

Similar to the list case, a fold on trees should:

- preserve the tree structure, given the appropriate operator (e.g. `Node`).
- hence, the call `mtfold Node Leaf` (where `Leaf` is the accumulator) should be the identity function on trees.

Let us define the identity function `tid` for trees:

```
tid Leaf = Leaf
tid (Node r k l) = Node (tid r) k (tid l)
```

As already said, `(tid t)` is equivalent to the call `(mtfold Node Leaf t)`, for any arbitrary tree `t`. To obtain the general `mtfold` implementation, we simply generalise `Node` by an arbitrary operation `op`:

- if `Node :: ITree ->Integer ->ITree ->ITree`, then
- `op :: b ->Integer ->b ->b`

We also generalise `Leaf` by an arbitrary accumulator:

- if `Leaf :: ITree` then

- `acc :: b`

The code generalisation becomes:

```
mtfold :: (b -> Integer -> b -> b) -> b -> ITree -> b
mtfold op acc =
    let f (Node left key right) = f (f left) key (f right)
        f Leaf = acc
    in f
```

Notice that `f` is precisely the generalisation of `tid` according to our observations. We can use `mtfold` to implement various tree operations such as:

```
tsum = mtfold (\k r l->k + r + l) 0
tmirror = mtfold (\k r l -> Node l k r) Leaf
tflatten = mtfold (\k r l -> r ++ [k] ++ l) []
```

subsection[Lab: Abstract datatypes] Lab: Abstract datatypes section[] Tipuri de date abstracte
Scopul laboratorului:

- Recapitularea conceptului de TDA
- Definirea de TDA-uri în Haskell
- Familiarizarea cu conceptul de TDA-uri polimorfice
- Introducerea tipului de date `Maybe`

subsection[] Tipuri de date abstracte (TDA)

Tipurile de date abstracte sunt modele pentru a defini tipuri de date în funcție de comportamentul acestora (valori posibile, axiome, operații), contrastând cu *structurile de date*, definite din punct de vedere al implementării.
Un TDA familiar este list.
Pentru a lucra cu liste într-un limbaj ca Java, am scris două *implementări* distincte, și anume `LinkedList` și `ArrayList`. Ambele implementări respectă specificațiile din primul paragraf.

Unii autori includ în descrierea TDA-urilor și complexități. Din acest punct de vedere, implementările `LinkedList` și `ArrayList` diferă.

subsubsection[] Constructori
Valorile posibile ale unui TDA sunt date de constructorii acestuia.
Să considerăm un tip simplu de date: `Bool`, care poate avea două valori: `False` și `True`.
Sintaxa Haskell pentru a defini acest tip de date este următoarea:

```
data Bool = False | True
```

Atât numele tipului de date cât și numele constructorilor trebuie scris cu literă mare.

`False` și `True` nu sunt parametrizate și pot fi privite ca valori. Se mai numesc și *constructori nulari*.
Să considerăm un alt TDA care descrie un arbore binar cu valori întregi. Arborele poate fi gol sau un nod cu doi copii.

```haskell
data Tree = Nil | Node Int Tree Tree
```

`Node` este un constructor care primește un argument de tipul `Int` și două argumente de tip `Tree`, pentru a creea o valoare de tip `Tree`. Putem să ne gândim la el ca la o funcție:

```
Prelude> :t Node
Node :: Int -> Tree -> Tree -> Tree
```

Constructorii sunt foarte importanți în *pattern matching*. Ca exemplu, vom scrie o funcție care însumează toate valorile dintr-un arbore:

```haskell
treeSum :: Tree -> Int
treeSum Nil = 0
treeSum (Node v l r) = v + (treeSum l) + (treeSum r)
```

Observăm prezența a două pattern-uri: unul pentru arborele vid, celălalt pentru un nod, într-un fel foarte asemănător cu modul în care lucram cu liste, făcând pattern matching pe `[]` și `(x:xs)`.

Observați că ghci nu știe cum să afișeze arborele nostru:

```
Prelude> Node 1 Nil Nil

<interactive>:38:1:
    No instance for (Show Tree) arising from a use of 'print'
```

```
    In a stmt of an interactive GHCi command: print it
```

Pentru a vă fi mai ușor să vă verificați, puteți obține o reprezentare a tipurilor definite de voi astfel:

```
data Tree = Nil | Node Int Tree Tree deriving (Show)
```

Pe scurt, acest lucru înrolează TDA-ul vostru în clasa `Show`, oferind o implementare implicită a funcției `show` (apelată de ghci). Mai multe detalii în laboratorul următor.

```
Prelude> Node 1 Nil Nil
Node 1 Nil Nil
```

subsubsection[] Înregistrări
Vrem să definim un TDA pentru a descrie un student, folosind următoarele câmpuri: nume, prenume, an de studiu, medie.

```
-- tipul de date si constructorul au acelasi nume, ceea ce este ok
-- pentru ca reprezinta concepte diferite.
data Student = Student String String Int Float
```

Observăm că semnficația câmpurilor nu este evidentă din definiție. Vrem să extragem valori dintr-un Student; definim funcțiile:

```
nume :: Student -> String
nume (Student n _ _ _) = n

prenume :: Student -> String
prenume (Student _ p _ _) = p

an :: Student -> Int
an (Student _ _ a _) = a

medie :: Student -> Float
medie (Student _ _ _ m) = m
```

Pentru a face descrierea tipului de date mai clară și a evita să scriem manual acele funcții, putem folosi sintaxa:

```
data Student = Student { nume :: String
                       , prenume :: String
                       , an :: Int
                       , medie :: Float
                       }
```

subsection[] TDA-uri polimorfice
Până acum, am definit TDA-uri care lucrează doar cu întregi. Dorim să scăpăm de această limitare și să definim TDA-uri care pot lucra cu orice alt tip de date. De exemplu, are sens să considerăm capul unei liste care conține elemente de tip `a`, ca fiind un element de tip `a`, indiferent de cum arată acest tip.
Într-adevăr, aceasta este definiția funcției `head` existentă în Haskell:

```
Prelude> :t head
head :: [a] -> a
```

a se numește variabilă de tip și poate să reprezinte orice tip de date (evident, cu condiția ca toate a-urile care apar să se refere la același tip).

Să ne uităm la tipul altei funcții cunoscute:

```
```

```
map :: (a -> b) -> [a] -> [b]
```

map primește ca argument o funcție - care primește un element de tip a și întoarce un element de tip b - și o listă de elemente de tip a și întoarce o listă de elemente de tip b. De precizat că a și b pot fi distincte dar nu este necesar.

Listele din Haskell sunt polimorfice. Ele pot conține date de orice tip. O instanță a listelor are însă un tip concret:

```
Prelude> :t ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]
```

subsubsection[] Constante polimorfice
Care este tipul listei vide?

```
Prelude> :t []
[] :: [a]
```

Observăm că tipul listei vide este general. Asta înseamnă că lista vidă poate fi tratată ca o listă de orice tip.

```
Prelude> [1,2,3] ++ "123" -- eroare, tipuri diferite
Prelude> [1,2,3] ++ []
[1,2,3]
Prelude> "123" ++ []
"123"
```

Spunem că `[]` este o constantă polimorfică.

O altă constantă polimorfică este, de exemplu `1`. De aceea putem scrie:

```
Prelude> 1 + 2
3
Prelude> 1 + 2.71
3.71
```

Dacă forțăm `1` să fie întreg, vom primi o eroare la a doua adunare:

```
Prelude> (1 :: Int) + 2.71

<interactive>:60:14:
    No instance for (Fractional Int) arising from the literal
    ’2.71’
    In the second argument of ’(+)’, namely ’2.71’
    In the expression: (1 :: Int) + 2.71
    In an equation for ’it’: it = (1 :: Int) + 2.71
```

`1` are însă o constrângere despre care vom discuta în laboratorul următor.

Pentru a implementa în Haskell un TDA polimorfic, folosim sintaxa:

```haskell
data List a = Empty | Cons a (List a)
```

Ce este `List`?
`List` este un constructor de tip, nu un tip de date. Constructorii de tip primesc ca parametrii *tipuri* și întorc un tip (sau un alt constructor de tip dacă nu primesc suficienți parametrii).
Asta înseamnă că, `List` nu este un *tip*, dar `List Int`, `List Char` etc. sunt tipuri.

subsubsection[] Maybe și Either
Unul dintre TDA-urile puse la dispoziție de Haskell este `Maybe`, un tip *polimorfic* care modelează prezența unei anumite valori, sau absența acesteia:

```haskell
data Maybe a = Nothing | Just a
```

Pentru a ilustra importanța acestui TDA, considerăm următoare situație: vrem să implementăm o funcție care returnează suma valoriilor celor doi copii ai unui nod dat:

```haskell
childrenSum :: Tree -> Maybe Int
childrenSum Nil = Nothing
childrenSum (Node _ Nil _) = Nothing
childrenSum (Node _ _ Nil) = Nothing
childrenSum (Node _ (Node v _ _) (Node w _ _)) = Just (v + w)
```

Această sumă există doar pentru arbore diferit de Nil, cu ambii copii diferiți de Nil.
Putem scrie o funcție care ia un `Maybe` și returnează un șir pentru printare. Combinăm apoi cele două funcții:

```
pretty :: Maybe Int -> String
pretty Nothing = "No value"
pretty (Just v) = "Sum is " ++ show v

printChildrenSum :: Tree -> String
printChildrenSum = pretty . childrenSum
```

Încărcăm modulele în ghci și voilà:

```
*Main> printChildrenSum (Node 1 (Node 1 Nil (Node 2 Nil Nil)) (Node 3
    (Node 4 (Node 5 Nil Nil) Nil) (Node 6 Nil Nil)))
"Sum is 4"
*Main> printChildrenSum Nil
"No value"
*Main> printChildrenSum (Node 1 Nil (Node 1 Nil Nil))
"No value"
*Main> printChildrenSum (Node 1 (Node 1 Nil Nil) Nil)
"No value"
```

Un TDA similar este `Either`, care modelează valori cu două tipuri posibile:

```
data Either a b = Left a | Right b
```

Prin convenție, constructorul `Left` reprezintă o eroare, iar constructorul `Right` o valoare corectă. Astfel putem extinde mecanisme de error-handling pentru a conține mai multe informații despre ce a cauzat eroarea (spre deosebire de `Maybe`, care doar indică existența unei erori):

```haskell
childrenSum :: Tree -> Either String Int
childrenSum Nil = Left "Tree is Nil"
childrenSum (Node _ Nil _) = Left "Left child is Nil"
childrenSum (Node _ _ Nil) = Left "Right child is Nil"
childrenSum (Node _ (Node v _ _) (Node w _ _)) = Right (v + w)

pretty :: Either String Int -> String
pretty (Left s) = s
pretty (Right i) = "Sum is " ++ show i

printChildrenSum :: Tree -> String
printChildrenSum = pretty . childrenSum
```

```
*Main> printChildrenSum (Node 1 (Node 1 Nil (Node 2 Nil Nil)) (Node 3
    (Node 4 (Node 5 Nil Nil) Nil) (Node 6 Nil Nil)))
"Sum is 4"
*Main> printChildrenSum Nil
"Tree is Nil"
*Main> printChildrenSum (Node 1 Nil (Node 1 Nil Nil))
"Left child is Nil"
*Main> printChildrenSum (Node 1 (Node 1 Nil Nil) Nil)
"Right child is Nil"
```

subsubsection[] Exerciții

Încercați să vă obișnuiți să scrieți explicit tipurile tuturor funcțiilor definite de voi în continuare.

subparagraph[1. List] 1. List

a) implementați TDA-ul pentru propria voastră listă care poate conține doar numere întregi
b) scrieți o funcție care convertește valori ale TDA-ului vostru în liste din Haskell

Dacă tipul vostru de listă se numește `List`, atunci tipul funcției va fi:

```haskell
convertList :: List -> [Int]
```

c) modificați tipul `List` pentru a fi polimorfic, apoi modificați și funcția `convertList`

subparagraph[2. Tree] 2. Tree
a) Implementați TDA-ul pentru arbori polimorfici.
b) Implementați `foldT`, echivalentul lui `foldr` (puteți vizualiza rezultatul ca fiind înlocuirea tuturor constructorilor nulari cu o valoare constantă și a tuturor celorlalți constructori cu funcții care iau același numă de argumente.
c) Implementați o funcție `mapT` (echivalentul lui `map`), folosindu-vă de `foldT`.
d) Implementați o funcție `zipWithT` (echivalentul lui `zipWith`).
subparagraph[3. Numere naturale extinse] 3. Numere naturale extinse
a) Implementați un TDA care să modeleze numere naturale extinse cu un punct la infinit ($\hat{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$).
b) Implementați o funcție `extSum` care să evalueze suma a două numere naturale extinse.
c) Implementați o funcție `extDiv` care să evalueze raportul a două numere naturale extinse (împărțirea la 0 și la infinit are rezultat definit).
d) Implementați o funcție `extLess` care spune dacă un număr natural extins e mai mic decât un altul.

În laboratorul următor, vom vedea cum putem implementa aceste operații într-un mod consistent cu restul limbajului, folosind operatori cunoscuți (`+`, `div`, `<`).
subparagraph[4. Înregistrări] 4. Înregistrări

a) Scrieți un TDA care codifică un tuplu (înregistrare) format din următoarele câmpuri:

- Nume (codificat ca String)
- Vârstă (codificat ca Integer)
- Prieteni (codificat ca o lista de String-uri)
- Notă PP (codificat ca Integer). Acest câmp este opțional!

b) Scrieți o funcție care primește o listă de înregistrări și le întoarce pe acelea pentru care vârsta este mai mare ca 20.
c) Scrieți o funcție care primește o listă de înregistrări și le întoarce pe acelea care conțin cel puțin un prieten cu numele "Matei".
d) Scrieți o funcție care primește o listă de înregistrări și le întoarce pe acelea care conțin câmpul "Notă PP".
e) Scrieți o funcție care primește o listă de nume, o listă de vârste, o listă ce conține liste de prieteni, și întoarce o listă de înregistrări corespunzătoare.

subsection[] Referințe

- `Abstract Data Type - Haskell wiki` [23]
- `Algebraic Data Type - Haskell wiki` [24]
- `Maybe - Haskell wiki` [25]

section[Course: Polymorphism in functional languages] Course: Polymorphism in functional languages section[] Polymorphism in functional languages
subsection[] Polymorphism in Object-Oriented languages
Translated literally, the term polymorphism means multiple shapes. Polymorphism in programming languages is a powerful modularisation tool, which allows programmers to:

- abstract from implementation details (in the case of ad-hoc polymorphism)
- define a unique implementation for a range of types (in the case of parametric polymorphism / genericity)

Polymorphism is a mechanism supported by virtually any strongly-typed programming language, including functional and object-oriented ones.
subsubsection[] Ad-hoc polymorphism

In an Object-Oriented language (say, Java), ad-hoc polymorphism allows the programmer to ***(dynamically) select the implementation of a function, based on the type(s) of the variable(s) on which the former is applied.***
Consider the following class definitions:

```
class Animal {
  public void talk (){
    System.out.println("I am an Animal");
  }
}

class Bird extends Animal {
  public void talk (){
    System.out.println("Cip-cirip");
  }
}
```

Finally, consider the following code:

```
 Animal [] v = new Animal [2];
    v[0] = new Animal();
    v[1] = new Bird();
    for (Animal a:v)
      a.talk();
```

whose output is:

```
I am Animal
Cip-cirip
```

The example illustrates ad-hoc polymorphism, or method overriding:

- the `talk` method from the class `Animal` has been overridden in the class `Bird` which extends `Animal`.

- moreover, establishing the implementation of `a.talk()` is done at runtime, based on the actual (here `Bird`) not declared (`Animal`) type of `a`.

Method overriding should not be confused with method overloading which means that the same function name can be used for different function implementations, each having a different signature. The following code, which continues the previous example, illustrates overloading:

```
public void listen_to(Animal a){
  System.out.println("An animal is listening:");
  a.talk();
}
public void listen_to(Bird b){
  System.out.println("A bird is listening:");
  b.talk();
}
```

The function `listen_to` has been overloaded. Now, consider the calls:

```
listen_to(v[0]);
listen_to(v[1]);
listen_to(new Bird());
```

The output is:

```
An animal is listening:
I am an Animal
An animal is listening:
Cip-cirip
A bird is listening:
Cip-cirip
```

The interesting call is `listen_to(v[1])`. Note that the code for `listen_to(Animal a)` has been called (which outputs `An animal is listening:`, although the actual type of `v[1]` is `Bird`, as shown by the next line of the output `Cip-cirip`. The example illustrates an important point:

- method overloading is performed at compile-time, based on the declared not the actual type of the parameters.

To illustrate this design decision, consider yet another example:

```
class Who implements Interface1, Interface2 {}

interface Interface1 {}

interface Interface2 {}

public class X {
    private static void method(Object o)     {}
    private static void method(Interface1 i) {}
    private static void method(Interface2 i) {}
}
```

Suppose for a moment that overloading relies on the actual type of an object, instead of the declared one. Now consider the following code:

```
   Object o = new Who();
   method(o);
```

Since `Who` implements both `Interface1` and `Interface2`, the compiler cannot make a decision about `o`'s actual type.

Question: What is the actual behaviour of the above code?
One final note regarding ad-hoc polymorphism is that the function signature cannot differ only in the returned type. This holds in both Java, and Cpp and Scala. To see why this is the case, consider the following definitions:

```
class Parent {}
class Child extends Parent {}
```

and the methods:

```
public Parent method() {}
public Child method() {}
```

as well as the invocation:

```
Object o = new Parent ();
o = method();
```

The compiler cannot decide which implementation should be called in this particular case.
subsubsection[] Genericity
While ad-hoc polymorphism (overriding) allows for several different implementations to be defined under the same function name, genericity in Java (and parametric polymorphism in general), allows for:

- a unique implementation to be defined over range of types

For instance, in:

```java
static <T> int count (List<T> l){
    int i = 0;
    for (T e:l)
        i++;
    return i;
}
```

the method `count` is defined w.r.t. lists containing any type of element (`T`). Technically, genericity in Java is a mechanism for ensuring *cast-control* and is not a part of Java's type system. For instance, in the compilation phase, the above code is translated to:

```java
static int count (List l){
    int i = 0;
    for (Object e:l)
        i++;
    return i;
}
```

This process is called type erasure. Consider another example, before type erasure:

```java
List<Animal> l = ...
Animal e = l.get(i)
```

and after:

```java
List l = ...
Animal e = (Animal)l.get(i)
```

Thus, type-safety is achieved via automatic casts. subsubsection[] Others

There are other types of polymorphism which may appear in the literature, e.g. subtype polymorphism which simply means that a variable `v` of type `T` is allowed to refer to an object of *any type derived from `T`*. Thus, subtype polymorphism is a basic OOP feature.
subsection[] Polymorphism in Haskell
subsubsection[] Parametric polymorphism
Parametric polymorphism is a fundamental trait of typed functional programming in general, and Haskell in particular. It manifests via the presence of type variables which stand for any type. Numerous functions defined so far are parametrically polymorphic:

- foldl
- foldr
- map
- zipWith

they define unique implementations which are independent of:

- the type of the contained elements of a list
- the function type which is applied on elements from a list
- etc.

Unlike Java, in Haskell, parametric polymorphism is an intrinsic (and key) feature of the type-system. To explore it in more depth, we start with a discussion regarding polymorphic types:
subsubsection[] Polymorphic types
We illustrate Haskell polymorphic types by constructing polymorphic lists precisely in the same way they are defined in Prelude:

```
data List a = Void | Cons a (List a)
```

compared to the monomorphic lists defined in the previous lecture, we observe:

- the newly defined type is `List a`, where `a` is a type-variable
- `Void :: (List a)` which means that `Void` is a polymorphic value (i.e. `Void` can be the empty list for list of integers or lists of strings etc.)
- `Cons :: a ->(List a) ->(List a)`, i.e. `Cons` takes a value of type `a`, a list of type `(List a)` (not `[a]`) and returns a list of type `(List a)`

We also illustrate a recursive conversion function, as an example:
listConvert :: (List a) ->[a] listConvert Void = [] listConvert (Cons h t) = h:(listConvert t)

Pairs (and tuples in general) are a very useful data structure, and they can be defined as follows:

```haskell
data Pair a b = Pair a b
```

this definition requires more care in reading it:

- `data Pair a b` defines a polymorphic type, where two independent type variables occur: the type of the first element of the pair, and that of the second. These two types need not coincide;
- `Pair :: a ->b ->Pair a b` is the unique data constructor for pairs: it takes an element of type `a`, one of type `b` and produces an element of type `Pair a b`.

As before, we write an illustrative conversion function:

```haskell
pairconvert :: Pair a b -> (a,b)
pairconvert (Pair x y) = (x,y)
```

The programmer should not mistake the keyword `Pair` from the type `Pair a b`, with the data constructor `Pair :: a ->b ->Pair a b`. Similar to the language C, where two namespaces exist: one for structures and one for types (with the `typedef` instruction to create new types), here we also have two *namespaces*:

- one for types (where `Pair` has been defined via the l.h.s. of the = in the `data` definition)
- one for values (and functions) (where the data constructor `Pair` has been defined)

We also define the polymorphic tree datatype:

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

paragraph[Type constructors] Type constructors

Let us recall the syntax for types, as presented in the previous lecture. It mainly consisted of type-variables (anything), *function-types* as well as *list types*. To this list we may add any other type introduced via `data`.
However, there is a more uniform and elegant way for describing these types. This approach relies on a functional approach to type construction:

- We require special functions called type constructors, which take types as parameter and return types
- To construct new types, we apply type constructors on type expressions (e.g. monomorphic types or type variables).

paragraph[The List type constructor] The List type constructor

In our previous `data List a = ...` definition:

- `List` is a type constructor
- Since - conceptually, `List` is also a function, it must have a type. The type of a type constructor is called kind in Haskell. Thus, the kind of `List` is written as: `List :: * =>*`, which reads: ***List receives a type and returns a type***
- The polymorphic type `List a` is actually a type function application. The function is `List` and the parameter is the variable `a`.
- Similarly, the monomorphic type `List Integer` (or similarly `[Integer]`) is constructed as an application of `List` on the monomporphic type `Integer`.

Exercise: Describe the construction of the following types. For what do they stand?

- `[(List a)]`
- `(List [a])`

paragraph[The Pair type constructor] The Pair type constructor

- `Pair :: * =>* =>*` is a type constructor with kind `* =>* =>*`. It takes two types and produces a type.
- The type `Pair a Integer` is polymorphic and represents the type of any pair whose second component is an integer.

With this observation, we can improve our syntax for types, as follows:

```
<type> ::= <const_type> | <type_var> | <type_constructor_application>
<type_constructor_application> ::= <type_const_1> <type> |
    <type_const_2> <type> <type> | ...
```

where:

- `<type_const_1>` is any type constructor having kind `* =>*`
- `<type_const_2>` is any type constructor having kind `* =>* =>*`
- etc.

To conclude, we observe that the function type is also constructed via the application of the type constructor:

- `(->) :: * =>* =>*`

on specific types or type expressions.
subsubsection[] Ad-hoc polymorphism
Ad-hoc polymorphism is necessary in typed functional languages, and we illustrate it via a few examples:

- to display an object the interpreter is calling the function `show :: a ->String` which takes an arbitrary type and converts it to a String. Naturally, `show` requires type-dependent implemementations
- similarly, the `+` operation has different implementations for Integers, Floats, and may be extended for other objects as well.

paragraph[Towards type-classes] Towards type-classes

Consider the types `Nat` and `List a` defined in previous lectures. To makes objects of type `Nat` or `List a` showable, we require functions of signature `Nat ->String` and `List a ->String`, respectively. We define them below:

```
showNat :: Nat -> String
showNat =
        let c Zero = 0
            c (Suc x) = 1 + (c x)
            in show . c

showList :: (List a) -> String
showList = lfoldr (\x y -> (show x)++":"++y) "[]"
```

The implementation of `showList` relies on `lfoldr :: (a ->b ->b) ->b ->List a ->b`. Also, written in Haskell as-is, `showList` has problems regarding the call `(show x)`. Consider that `x::(List Nat)` or that `x :: Nat`. Depending on `x`'s type, we need to call different show functions. What is obvious already is that we need a single function name (e.g. show) which should have type-dependent implementations.
An attempt to solve this issue is by introducing a new type:

```
data Showable a = C1 Nat | C2 (List a)

show :: (Showable a) -> String
show (C1 x) = showNat x
show (C2 x) = showList y
```

An object of type `Showable a` indicates a value which can be displayed. For each showable type, we define separate construction rules (`C1` resp. `C2`). The above code still has a problem:

- `showList :: (List a) ->String`, however, to be able to call `(show x)`, x must be showable, hence `x::Showable a`

To solve this issue, we modify the signature of `showList`:

```
showList :: (List (Showable a)) -> String
```

as well as the definition of `Showable a`:

```
data Showable a = C1 Nat | C2 (List (Showable a))
```

Our approach to handling ad-hoc polymorphism suffers from a single drawback:

- it relies on type-packing. The programmer needs to handle both user-defined values (e.g. `Zero :: Nat`), as well as showable values (e.g. `C1 Zero :: Showable a`); we have two, type-dependent representations for the same object.

paragraph[Type-classes] Type-classes

To solve the above issue, ad-hoc polymorphism is implemented in Haskell via type-classes, which are conceptually different from classes in OOP. In short:

- a type-class describes a collection of types, which can be defined by the user
- type-classes also contain *function signatures* which are traits specific to each type in the type-class
- types can be enrolled in type-classes by the user
- relationships between type-classes (e.g. inclusion) can be defined by the user.

We illustrate all the above by introducing the definition for the type-class `Show`:

```
class Show a where
   show :: a -> String
```

In the above definition, `a` is an arbitrary type which is enrolled in class `Show`. Any such type supports the function `show`, which is defined as part of the type-class. The following code enrols our previous `Nat` type in class `Show`, hence making naturals showable:

```
instance Show Nat where
    show =
      let convert Zero = 0
          convert (Succ n) = 1 + (convert n)
      in show . convert
```

In our implementation, the type of `show . convert` is `Nat ->String`, since `convert :: Nat ->Integer`. This example also shows ad-hoc polymorphism in action. In the above expression (the functional composition), the general type `show :: (Show a) =>a ->String` of `show` becomes via unification `::Integer ->String`. Thus, the compiler knows to call the integer implementation of `show`, which is part of Prelude.
Also, note the interpretation of the type signature: `show :: (Show a) =>a ->String` which tells us that:

- `show :: a ->String` where a must be enrolled in the type-class `Show`

We continue by enrolling `List a` in class `Show`. Recall that, to be able to show lists, the elements from the list need to be showable. Hence, the enrollment is:

```
instance (Show a) => Show (List a) where
    show Void = "[]"
    show (Cons h t) = (show h)++":"++(show t)
```

Finally, we illustrate another kind of enrollment. It spawns from the observation that both lists and trees support map operations, which have very similar behaviour:

```
lmap :: (a -> b) -> (List a) -> (List b)
tmap :: (a -> b) -> (Tree a) -> (Tree b)
```

We can define the class of mappable types, which contains a `fmap` operation. In Haskell, this class is called `Functor`. A tentative definition `class Functor t where` raises the question regarding who is `t`, such that all constraints from the map signatures are preserved:

- map transforms *containers of a kind* (e.g. Lists) in *containers* of the same kind
- if the mapped transformation is `(a ->b)`, then the first container must have elements of type `a` while the second - of type `b`.

The solution is:

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

where `t` is a type-constructor with kind `t :: *` =>*. Thus, we have the following enrollments:

```
instance Functor List where
  fmap = ...

instance Functor Tree where
  fmap = ...
```

subsection[Lab: Haskell type-classes] Lab: Haskell type-classes section[] Clase de tipuri (Type-classes)

Scopul laboratorului:

- Familiarizarea studenților cu tipuri de clase
- Familiarizarea studenților cu constrângeri de tip

subsection[] Typeclasses
O clasă de tipuri (typeclass) este un fel de interfață care definește un comportament. Dacă un tip de date face parte dintr-o anume clasă de tipuri, atunci putem lucra pe tipul respectiv cu operațiile definite în clasă.

Conceptul de clasă de tipuri este diferit de conceptul de clasă din programarea orientată pe obiecte. O comparație mai pertinentă este între clasele de tip din Haskell și interfețele din Java.

O clasă des întâlnită este clasa `Eq`. Orice tip care este o instanță a acestei clase poate fi comparat, utilizând funcțiile == și /=. Clasa `Eq` este definită mai jos. Observați sintaxa Haskell:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Observăm că definițiile pentru == și /= depind una de cealaltă. Acest lucru ne ușurează munca atunci când vrem să înrolăm un tip acestei clase, pentru că trebuie să redefinim doar unul dintre cei doi operatori. Pentru a vedea cum lucrăm cu clase, vom defini un tip de date simplu și îl vom înrola în `Eq`:

```haskell
data Color = Red | Green | Blue

instance Eq Color where
    Red == Red = True
    Green == Green = True
    Blue == Blue = True
    _ == _ = False
```

Am redefinit ==, iar definiția lui /= rămâne neschimbată (`x /= y = not (x == y)`), ceea ce e suficient ca să folosim ambii operatori.

```
*Main> Red == Red
True
*Main> Red /= Red
False
```

În acest caz, puteam obține același comportament folosind implementarea default oferită de cuvântul cheie `deriving`, i.e. `data Color = Red |Green |Blue deriving (Eq)`.

subsection[] Constrângeri de clasă
În laboratoarele trecute, analizând tipurile unor expresii, ați întâlnit notații asemănătoare:

```
Prelude> :t elem
elem :: (Eq a) => a -> [a] -> Bool
```

Partea de după => ne spune că funcția `elem` primește un element de un tip oarecare, `a` și o listă cu elemente de același tip și întoarce o valoare booleană.
`(Eq a)` precizează că tipul de date `a` trebuie să fie o instanță a clasei `Eq` și nu orice tip. De aceea se numește o constrângere de clasă (class constraint).

Toate constrângerile de clasă sunt trecute într-un tuplu, înaintea definiției funcției, separate de =>.

```
*Main> :t (\x -> x == 0)
(\x -> x == 0) :: (Eq a, Num a) => a -> Bool
```

O definiție posibilă a funcției `elem` este:

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ [] = False
elem e (x:xs) = e == x || elem e xs
```

Ceea ce înseamnă că, odată înrolat tipul nostru clasei `Eq`, putem folosi, printre altele, și functia `elem`:

```
Prelude> elem Red [Blue, Green, Green, Red, Blue]
True
```

subsection[] Clase de tipuri și tipuri de date polimorfice
Să considerăm tipul de date polimorfic `Either`:

```
data Either a b = Left a | Right b
```

Țineți minte că `Either` nu este un tip, ci un ***constructor de tip***. `Either Int String`, `Either Char Bool` etc. sunt tipuri propriu-zise.
O clasă utilă de tipuri este clasa `Show`, care oferă funcția `show` care primește un parametru și întoarce reprezentarea acestuia sub formă de șir de caractere.

```
show :: (Show a) => a -> String
```

Pentru a înrola tipul `Either` în clasa `Show`, vom folosi sintaxa:

```
instance (Show a, Show b) => Show (Either a b) where
    show (Left x) = "Left " ++ show x
    show (Right y) = "Right " ++ show y
```

Observați constrângerile de clasă (`Show a, Show b`)! În implementarea funcției `show` pentru tipul `Either a b`, folosim aplicațiile `show x` și `show y`, deci aceste elemente trebuie să aibă, la rândul lor, un tip înrolat în clasa Show.

subsection[] Informații despre clase în ghci
Din cadrul ghci, puteți obține informații despre o clasă anume folosind comanda `:info <typeclass>` (`:i <typeclass>`):

```
Prelude> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
        -- Defined in 'GHC.Classes'
instance Ord a => Ord [a] -- Defined in 'GHC.Classes'
instance Ord Word -- Defined in 'GHC.Classes'
instance Ord Ordering -- Defined in 'GHC.Classes'
instance Ord Int -- Defined in 'GHC.Classes'
...
```

Putem observa mai multe informații utile:

- constrângerea de clasă `Eq a =>` arată că un tip de date trebuie să fie membru al clasei `Eq` pentru a putea fi membru al clasei `Ord`.
- putem observa toate funcțiile și operatorii oferiți de clasa `Ord`: `compare`, <, <= etc.

- linia `{-# MINIMAL compare |(<= ) #-}` ne informează că e suficient să implementăm fie funcția `compare`, fie operatorul `<=`, pentru a putea utiliza toate funcțiile puse la dispoziție de clasa `Ord` (amintiți-vă de clasa `Eq` și cum `/=` rămânea definit în funcție de `==`).

- linia `- Defined in ‘GHC.Classes’` indică locul în care această clasă e definită. O căutare a numelui ne duce `aici` [26], unde putem observa implementarea clasei, exact așa cum este folosită în ghc.

- următoarele linii reprezintă o înșirare a tuturor tipurilor de date despre care ghci știe că sunt înrolate în clasa `Ord`, precum și unde se găsește această înrolare. E.g. prima linie arată că listele ce conțin elemente ordonabile sunt și ele ordonabile, comportament definit în `GHC.Classes`: `https://githubcom`/ghc/ghc/blob/m
  prim/GHC/Classes.hs#L330

subsection[] Exerciții

1. În `laboratorul anterior` [27], ați definit un tip pentru a modela numerele naturale extinse cu un punct la infinit, precum și niște operații pe acestea. Dorim să facem implementarea elegantă, pentru a putea folosi operatori deja existenți (e.g. == pentru comparare) și pentru a putea folosi alte funcții existente care impun constrângeri de tip (e.g. `Data.List.sort :: Ord a =>[a] ->[a]`). Astfel, ne dorim să înrolăm tipul de date în următoarele clase:

- Show (astfel încât să afișăm numerele fără a fi precedate de vreun nume de constructor, iar infinitul ca "Inf")
- Eq
- Ord
- Num

2. Definiți un tip de date asociat următoarei gramatici:

```
<expr> ::= <value> | <variable> | <expr> + <expr> | <expr> * <expr>
```

unde o valoare poate avea orice tip.
3. Considerăm următorul constructor de tip:

```
type Dictionary a = [(String, a)]
```

care modeleaza *dicționare* - mapări de tip "nume-variabilă"-"valoare polimorfica"
Definiți funcția:

```haskell
valueof :: Dictionary a -> String -> Maybe a
```

care intoarce valoarea asociata unui nume-variabilă, dintr-un dicționar
4. Definiți următoarea clasă:

```haskell
class Eval t a where
    eval :: Dictionary a -> t a -> Maybe a
```

Spre deosebire de clasele prezentate în exemplele anterioare, care desemnează o *proprietate* a unui tip sau constructor de tip, `Eval` stabilește o relație între un constructor de tip `t` și un tip `a`. Relația spune că orice container de tip `t a` poate fi evaluat in prezența unui dicționar cu valori de tip `a`, la o valoare de tip `Maybe a`.

Acest tip de clasă reprezintă o extensie a limbajului, Multi-parameter type-class. Cel mai probabil este nevoie de următoarele directive pentru a o defini și pentru a înrola tipuri la ea:

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

class Eval t a where
    eval :: Dictionary a -> t a -> Result a
```

Mai multe despre acestea, precum și despre Functional Dependencies, un alt feature care este strâns legat de clasele de tipuri cu mai mulți parametrii puteți găsi `aici` [28].

5. Înrolați `Expr` și `Integer` în clasa `Eval`. Care este semnificația evaluării?
6. Înrolați `Expr` și `FIFO a` în clasa `Eval`. Semnificația înmulțirii este *concatenarea* a două FIFO.
subsection[] Alte exerciții
1. Ați definit, în laboratorul anterior, tipurile polimorfice `List a` și `Tree a`. Pentru le putea reprezenta, ați folosit implementarea implicită a funcției `show`, oferită de `deriving (Show)`. Aceasta nu era însă o reprezentare citibilă.
Ne dorim să reprezentăm tipul listă, la fel ca cel existent în Haskell:

```
Prelude> show (Cons 1 (Cons 2 (Cons 3 Nil)))
[1,2,3]
```

(Pentru arbori există multe reprezentări posibile, puteți alege orice reprezentare preferați).
2. Înrolați aceleași tipuri în clasa `Eq`.
3. Implementați sortarea pentru `List a`, unde `a` e un tip oarecare înrolat în clasa `Ord`.
4. Implementați căutarea binară pentru `Tree a`, unde `a` e un tip oarecare înrolat în clasa `Ord`.
5. Înrolați tipurile de date `List` și `Tree` în clasa `Functor`.
subsection[] Resurse

- `Type Classes and Overloading - Haskell wiki`[29]
- `Typeclassopedia - Haskell wiki`[30]

- `Typeclasses 101 - Learnyouahaskell` [31]
- `Typeclasses 102 - Learnyouahaskell` [32]

section[Course: Parameter passing in Programming Languages] Course: Parameter passing in Programming Languages section[] Parameter passing in different programming languages
Different programming languages adopt different strategies for passing (and evaluating) parameters during function calls. Such strategies can be split into two categories:

- *applicative*
- *normal*

but different blends are possible, depending on how values are stored and passed on. We briefly review some of these strategies:
subsubsection[] Call-by-value
In call-by-value, parameters are passed (stored on the call stack) as values. This is the case in C, as well as Java for primitive values.

```
void swap (int x, int y){
        int t = y;
        y = x;
        x = t;
}
```

We illustrate call-by-value using the `swap` function. In the code below:

```
int x = 1, y = 2;
swap(x,y);
```

the values of x and y do not change after the call of swap, since the values have been copied on the call stack. Call-by-value is an applicative evaluation strategy.
subsubsection[] Call-by-reference
Call-by-reference is implemented in C explicitly via pointers, and is the default parameter-passing strategy in Java, for non-primitives (objects). We illustrate call-by-reference in C:

```
void swap (int* x, int* y){
       int t = *x;
       *y = *x;
       *x = t;
  }
```

In the code example below:

```
int x = 1, y = 2;
swap(&x,&y);
```

the values of x and y have been changed, since their addresses (instead of their values) have been passed on the call stack. Call-by-reference is an applicative evaluation strategy.
subsubsection[] Call-by-macro expansion
Consider the following macro-definition in C:

```
#define TWICE(X,Y) {Y = X + X;}
```

In the code example:

```
int y;
TWICE(1+1,y);
```

the macro is textually-expanded without parameter evaluation, yielding `y = 1 + 1 + 1 + 1` (instead of `y = 2 + 2`). Call-by-macro is a normal evaluation strategy, and behaves exactly like normal evaluation from the lambda calculus. Note however that macros in C are more limited than functions and do not rely on a call-stack.
subsubsection[] Call-by-name
Call-by-name is a normal evaluation strategy, which, similar to call-by-macro expansion, reproduces lambda calculus's normal evaluation. It is not implemented per-se in programming languages, because it is inefficient. We illustrate this, by the following example:

```
int g(by-name int x, by-name int y){
    return x + y;
}
int f(by-name x){
    if (x == 0)
        return 0;
    return f(g(x,x)-4);
}

main{
    f(3);
}
```

where we have introduced a *fictitious by-name* directive, which forces the parameter at hand to be evaluated using the normal strategy.

The call `f(3)` will have the following behaviour:

- since `3 == 0` is false, the following call is made:
- `f(g(3,3)-4)`
  - the condition `g(3,3)-4 == 0` triggers a call to `g`. The condition is false. Thus, the following call is made:
  - `f(g(g(3,3)-4,g(3,3)-4)-4)`. Note that, even though the parameter of `f` was evaluated during the condition check (to `2`), *call-by-name* requires that it is evaluated again:
    - the condition `g(g(3,3)-4,g(3,3)-4)-4 == 0` triggers three calls of `g`, and is true. Hence the program returns `0`.

Note that, during the call of `f(3)`, we had a total number of 4 function calls of `g`, when actually 2 would have been sufficient.
subsubsection[] Call-by-need (lazy)
Call-by-need is a normal evaluation strategy which improves call-by-name by storing a result once it is computed.
Returning to our previous example, let us replace the *by-name* directive with *by-need*. Then, the call `f(3)` will have the following behaviour:

- since `3 == 1` is false, we have the following call:

- `f(g(3,3)-4)`. The expression `g(3,3)-4` is evaluated to `2` during the comparison (which fails), and the following call is triggered:
- `f(g(g(3,3)-4,g(3,3)-4)-4)`. During this call, to evaluate the comparison with 0, we need to evaluate `g(g(3,3)-4,g(3,3)-4)-4`. However, technically, this expression is viewed by the runtime as: `g(thunk,thunk)-4`, where `thunk` is a pointer to the expression `g(3,3)-4`. This expression has already been evaluated, hence we have the call `g(2,2)-4`, and the comparison succeeds.

Note that, during *call-by-need*, we only have two function calls of `g` (instead of 4).
subsection[] Lazy evaluation in Haskell
The default evaluation strategy in Haskell is lazy or call-by-need. Each expression in Haskell can be viewed as a *thunk*: a pointer which holds the expression itself, as well as its value, once it is evaluated.
We illustrate lazy evaluation by looking at the following calls:

```
foldr (&&) True [True,False,True,True]
```

Let us consider that the implementation of `(&&)` is as follows:

```
True && True = True
_ && _ = False
```

This expression will produce the following sequence of calls:

- `True && (foldr (&&) True [False,True,True]`
- `True && (False && (foldr (&&) True [True,True]))`
- `True && False`
- `False`

In the second pattern of `&&`, the function returns `False` without irrespective of the parameters. Hence, the call `(False && (foldr (&&) True [True,True]))` returns `False` without evaluating `(foldr (&&) True [True,True])`.
As it turns out, `foldr` may be efficient even if it is not tail-recursive, in situations where reducing a list does not require exploring all its elements.
Let us also the evaluation of:

```
foldl (&&) True [True,False,True,True]
```

which triggers:

- `foldl (&&) (True && True) [False,True,True]`
- `foldl (&&) ((True && True) && False) [True,True]`
- `foldl (&&) (((True && True) && False) && True) [True]`

- `foldl (&&) ((((True && True) && False) && True) && True) []`
- `((((True && True) && False) && True) && True)`
- `(((True && False) && True) && True)`
- `((False && True) && True)`
- `(False && True)`
- `False`

Since the evaluation is lazy, the accumulator is only evaluated when needed, that is, when `foldl` returns. The result shows that `foldl` may be less eficient than `foldr` in Haskell, even if the former is tail-recursive.
section[Course: Lazy evaluation in Haskell] Course: Lazy evaluation in Haskell section[] Lazy Evaluation in Haskell
subsubsection[] Introduction Lazy evaluation means that:

1. an expression (function application) will be evaluated only when it is needed (precisely as in the Lambda Calculus's normal evaluation)
2. an expression is evaluated only once

We illustrate point 1. via the following example:

```
nats = 0:(map (+1) nats)
test = foldr (\x y-> if x > 2 then 0 else x+y) 10 nats
```

- First, note that `nats` is a recursive non-terminating expression, which will produce the list of natural numbers, until memory is depleted. To examine this, it is sufficient to call `nats` in the interpreter
- Second, `test` is an expression which evaluates to `3`, although it relies on `nats` for the computation:
  - let `op = \x y->if x >2 then 0 else x+y`. Then, in effect, `test` attempts to compute the expression:

```
0 op (1 op (2 op (3 op (4 op ....
```

- however, in the expression `(3 ‘op‘ (4 ‘op‘ ....` the value of the second operand is not used (since `x>3`), hence `(4 ‘op‘ ....` is not evaluated. The result is 0. Thus, `test` actually computes

```
0 op (1 op (2 op 0))
```

- note that the value of the accumulator (`10`) is not actually used.

To illustrate point 2. consider:

```
evens = zipWith (+) nats nats
some = take 2 evens
```

We also recall that:

```
take 0 _ = []
take n (h:t) = h:(take (n-1) t)

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

No expression is evaluated until we call `some`, in the interpreter. Thus, we start with the following un-evaluated expressions:
^Variable ^Expression ^Value ^|evens |? |unevaluated ||some |? |unevaluated ||nats |? |unevaluated |
Upon calling `some` we obtain the following result which requires us to evaluate `evens`. This happens due to the pattern-matching definition of `take`, which requires a value of the form `(x:xs)`.
|evens |? |unevaluated ||some |`take 2 evens` |unevaluated ||nats |? |unevaluated |
To evaluate `evens`, as before, the pattern-matching definition of `zipWith` requires the first element of `nats`:
|evens |`zipWith (+) nats nats` |unevaluated ||some |`take 2 evens` |unevaluated ||nats |? |unevaluated |

Note that we only require x and y to evaluate zipWith in one step, hence (map (+1) nats) is not (yet) evaluated. |evens |zipWith (+) nats nats |unevaluated ||some |take 2 evens |unevaluated ||nats |0:(map (+1) nats) |unevaluated |
Thus, the evaluation of zipWith yields: |evens |(0+0):zipWith (+) xs ys |unevaluated ||some |take 2 evens |unevaluated ||nats |0:(map (+1) nats) |unevaluated |
Although we have created additional column in the table, we stress that the expressions (map (+1) nats) which appear in the body of nats, xs and ys are actually the same, and not different identical expressions. The first-step evaluation of take 2 evens is now complete:
|evens |(0+0):zipWith (+) xs ys |unevaluated ||some |0:(take 1 t) |unevaluated ||nats |0:(map (+1) nats) |unevaluated ||xs,ys |(map (+1) nats) |unevaluated ||t |zipWith (+) xs ys |unevaluated |
As before, note that both occurrences of zipWith (+) xs ys are actually the same expression. We continue with another step in the evaluation of take, which leads to evaluating zipWith (+) xs ys, and subsequently, xs and ys:
|evens |(0+0):zipWith (+) xs ys |unevaluated ||some |0:(take 1 t) |unevaluated ||nats |0:1:(map (+1) nats) |unevaluated ||xs,ys |1:(map (+1) nats) |unevaluated ||t |zipWith (+) xs ys |unevaluated |

Note that after evaluating the expression (map (+1) nats) in one step, the expression is not re-evaluated, as shown in the table above.
|evens |(0+0):zipWith (+) xs ys |unevaluated ||some |0:(take 1 t) |unevaluated ||nats |0:1:(map (+1) nats) |unevaluated ||xs,ys |1:(map (+1) nats) |unevaluated ||t |(1+1):zipWith (+) xs' ys' |unevaluated |
We have now finished the second step in the evaluation of take. We omit adding variables xs', ys' and t'. |evens |(0+0):zipWith (+) xs ys |unevaluated ||some |0:2:(take 0 t') |unevaluated ||nats |0:1:(map (+1) nats) |unevaluated ||xs,ys |1:(map (+1) nats) |unevaluated ||t |(1+1):zipWi
(+) xs' ys' |unevaluated |
Now, take 0 t' evaluates to [], hence we finally get: |evens |(0+0):zipWith (+) xs ys |unevaluated ||some |0:2:[] |evaluated ||nats |0:1:(map (+1) nats) |unevaluated ||xs,ys |1:(map (+1) nats) |unevaluated ||t |(1+1):zipWith (+) xs' ys' |unevaluated |
and the evaluation stops.
subsubsection[] Applications of normal evaluation
subsubsection[] Dynamic programming - edit distance

Consider two strings `s1` and `s2`. We define the ***edit distance*** between `s1` and `s2` as the minimal number of edit operations which make the strings identical. The allowed ***edit operations*** are:

- character insertion (e.g. `text` and `ext` have edit distance 1)
- character deletion (e.g. `tet` and `text` have edit distance 1)
- character modification (e.g. `text` and `tent` have edit distance 1)

As an example, the strings `maple` and `apple` have edit distance 2:

- we delete the first symbol from `maple` and obtain `aple`
- we insert the symbol `p` at the second position in `aple` and obtain `apple`

Dynamic programming computes the edit distance between two strings by building a matrix `d` having `size(s1)+1` lines and `size(s2)+1` columns, where `d[i][j]` represents the edit distance between the substrings `s1[0:i]` and `s2[0:j]`:

- `d[i][0] = i` for all lines (the distance between the empty string and the (sub)-string `s1[0:i]` is `i`)

- `d[0][i] = i` for all columns (the distance between the empty string and the (sub)-string `s2[0:i]` is `i`)
- if `s1[i]` and `s2[i]` coincide, then `d[i][j] = d[i-1][j-1]`
- otherwise, `d[i][j]` is computed by applying the edit operation which minimises distance. Concretely, `d[i][j]` is the minimal of `d[i-1][j] + 1` (delete character `i` from `s1`), `d[i-1][j-1] + 1` (modify character `i` from `s1`), `d[i][j-1] + 1` (insert character `i` in `s1`)

subsubsection[] Functional implementation Dynamic programming (for edit distance) can be efficiently implemented in Haskell, by exploiting lazyness in order to compute only those necessary distances, and only once. We first start with an example of an implementation of the infinite list of Fibonacci numbers:

```
fibo = 0:1:(zipWith (+) fibo (tail fibo))
```

section[] References

- `Lazy dynamic programming` [33]

subsection[Lab: Haskell and Scheme evaluation] Lab: Haskell and Scheme evaluation subsection[] Întârzierea evaluării

- `The PP team` [34]

- `C01: Introduction` [35]
- `C02: Side-effects and recursive functions` [36]
- `L01: Introduction to Haskell` [37]
- `C03: Higher-order functions` [38]
- `L02: Higher-order functions` [39]
- `C04: Types in functional programming` [40]
- `C05: Abstract datatypes` [41]
- `L04: Abstract datatypes` [42]
- `C06: Polymorphism in functional languages` [43]
- `L05: Haskell type-classes` [44]
- `C06: Parameter passing in Programming Language` [45]
- `C07: Lazy evaluation in Haskell` [46]
- `L06: Haskell and Scheme evaluation` [47]
- `C08: The Lambda Calculus` [48]
- `H01: Haskell Lazy Dynamic Programming` [49]

subsection[]Moduri de evaluare a unei expresii:

- **evaluare aplicativă - argumentele funcțiilor sunt evaluate înaintea aplicării funcției asupra lor.**

- evaluare lenesa - întârzie evaluarea parametrilor până în momentul când aceasta este folosită efectiv.

subsubsection[]Evaluare aplicativă vs. evaluare normală
Fie urmatoarea expresie, scrisă într-o variantă relaxată a Calculului Lambda (în care valori numerice și operații pe acestea sunt permise, iar funcțiile sunt în formă uncurried):

```
(λx.λy.(x + y) 1 2)
```

Evident, în urma aplicării funcției de mai sus, expresia se va evalua la 3. Să observăm, însă, cazul în care parametrii funcției reprezintă aplicații de funcții:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
```

Desigur, evaluarea expresiei `(λz.(z + 2) 3)` va genera valoarea `5`, de unde deducem că rezultatul final al expresiei va fi `6` (adunarea lui 1 cu rezultatul anterior). În cadrul acestui raționament, am presupus că parametrii sunt evaluați înaintea aplicării funcției asupra acestora. Vom vedea, în cele ce urmează, că evaluarea se poate realiza și in alt mod.
subsubsection[]Evaluare aplicativă

Evaluarea aplicativă (eager evaluation) este cea în care fiecare expresie este evaluată imediat. În exemplul de mai sus, evaluarea aplicativă va decurge astfel:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
(λx.λy.(x + y) 1 5)
6
```

subsubsection[]Evaluare normală
Spre deosebire de evaluarea aplicativă, evaluarea normală va întarzia evaluarea unei expresii, până când aceasta este folosită efectiv. Exemplu:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
(1 + (λz.(z + 2) 3))
(1 + 5)
6
```

subsubsection[]Exerciții:
paragraph[I. Șiruri] I. Șiruri

1. Construiți șirul numerelor naturale
   - Construiți șirul numerelor pare
   - Construiți șirul numerelor lui Fibonacci

paragraph[II. Aproximații] II. Aproximații

1. definiți o funcție `build :: (a ->a) ->a ->[a]` care primeste o funcție 'generator' (pe care o numim `g` în continuare), o valoare inițială (`a0`), și generază lista: `[a0, g a0, g (g a0), g (g (g a0)), ...`

Comportamentul funcției ar trebui să fie identic cu cel al funcției `iterate`, deja existentă în Haskell.

2. definiți o funcție `select` care primește o toleranță `e`, o listă, și întoarce valoarea a_n din listă care satisface proprietatea: $abs(a_n - a_{n+1}) < e$

```
iterate :: (a -> a) -> a -> [a]
```

3. Aproximație pentru $\sqrt{k}$:

a) Scrieți o funcție care calculează $\sqrt{k}$ cu toleranța `0.01`, exploatând faptul că șirul: $a_n = 1/2(a_{n-1} + k/a_{n-1})$ converge către $\sqrt{k}$ când `n` tinde la infinit.

4. Aproximație pentru derivata unei funcții `f` într-un punct `a`:
a) Scrieți o funcție care generează lista: `[h_0, h_0/2, h_0/4, h_0/8, ...]`
b) Scrieți o funcție care calculează lista aproximarilor lui `f'(a)`, calculate astfel: $f'(a) = \lim_{h \to 0} \frac{f(a+h) -}{h}$
c) Scrieți o funcție care calculează derivata in `a` a unei funcții `f`, cu toleranța `0.01`
5. Aproximație pentru integrala unei funcții pe intervalul `[a,b]`:
a) Scrieți o funcție care aproximează valoarea integralei unei funcții `f` între `a` și `b`, cu toleranța `0.01`. Strategia de îmbunătățire a unei aproximări constă în spargerea intervalului `[a,b]` în două sub-intervale de dimensiune egală `[a,m]` si `[m,b]`, calculul integralei pe fiecare, și adunarea rezultatului. section[Course: The Lambda Calculus] Course: The Lambda Calculus section[] The Lambda Calculus
subsection[] A brief history
The Lambda Calculus was created in 1928 by Alonzo Church. (To be expanded).
subsection[] Intuition

At first glance, the Lambda Calculus formalises the fundamental concept of ***function application*** from mathematics. Consider the function $f(x) = x + 1$. Then $f(2)$ denotes the application of the function f to argument 1. In order to compute the result, all occurrences of variable x are replaced by the parameter(here 1), in the body of the function. The result is $1 + 1$, which is subsequently computed using the laws of arithmetic.
At its core, the Lambda Calculus is not (apriori) designed to describe e.g. addition, or other mathematical operators (however, as we shall further see, it is possible to encode numbers and a subset of arithmetic in the Lambda Calculus).
subsection[] Syntax
The Lambda Calculus defines three constructs:

- variables (henceforth denoted as $x, y, \ldots$)
- functions (similar to `\x->...` from Haskell)
- function applications (similar to function calls `(f x)` from Haskell)

Formally, let V stand for a finite set whose elements are called variables. A λ-expression e is recursively defined as:
$e ::= x \in V \mid \lambda x.e \mid (e_1\ e_2)$
Examples of λ-expressions:

- x

- $\lambda x.x$
- $(\lambda x.\lambda y.x\ z)$

The set V of variables will henceforth be implicit.
Note that, in the Lambda Calculus, functions are curried (each function takes one parameter at a time). Thus, a Haskell function `\x y->body` would be written as $\lambda x\lambda y.body$.
subsubsection[] What is a lambda-expression?
Lambda-expressions are not designed to suit certain programming applications, hence there is no predefined concept of primitive value (integer or char), nor are there special functions to construct lists. Hence, at first glance, most lambda expressions do not seem to stand for something tangible (yet). However, we can recognise certain basic functions such as:

- $\lambda x.x$ - the identity function
- $\lambda x.y$ - the constant function which allways returns y
- $\lambda x.\lambda y.x$ (resp. $\lambda x.\lambda y.y$) - selector functions, which are expected to be called with two parameters (curried), and return the first (resp. second) one.

In the Lambda Calculus, the naming scheme for the variables is unimportant. For instance, $\lambda x.x$ and $\lambda y.y$ stand for the same identity function. Similarly, $\lambda x.\lambda x.x$ and $\lambda x.\lambda y.y$ also stand for the same function which, called - returns the identity function.
subsection[] The semantics of the Lambda Calculus
subsubsection[] Step 1: reduction
The algebraic function application $f(c)$ of function $f(x) = body$ on parameter par is encoded in the Lambda calculus as the lambda-expression:
$(\lambda x.body\ par)$
where $body, par$ are $\lambda - expressions$ and x is a variable.
Informally, computing the ***function call*** or reducing the expression amounts to:

- the substitution of all occurrences of variable x with parameter par in the body of the function.

We denote such a substitution by

$$body[par/x]$$

, where $body, par$ are λ-expressions and x is a variable.
We define $body[par/x]$ recursively, over all types of lambda expressions, as follows:

- a. $x[par/x] = par$

- b. $y[par/x] = y$
- c. $\{\lambda x.body\}[par/x] = \lambda x.body[par/x]$ where curled brackets denote the scoping of the substitution.
- d. $(e_1\ e_2)[par/x] = (e_1[par/x]\ e_2[par/x])$

Examples:

- $(\lambda x.(x\ y)\ y)[z/x] = (\{\lambda x.(x\ y)\}[z/x]\ y[z/x]) = = (\lambda x.(x\ y)[z/x]\ y) = (\lambda x.(x[z/x]\ y[z/x])\ y) = (\lambda x.(z\ x)\ y)$
- $\{\lambda x.\lambda x.x\}[z/x] = \lambda x.\lambda x.z$

The second example from the previous list illustrates a conceptual problem with point c. from our definition. Suppose we would like to reduce:

$$(\lambda x.\lambda x.x\ y)$$

On the left-hand side we have a function which returns the identity function, hence its call should produce $\lambda x.x$. However, if we substitute x by y in the body of the function, i.e. compute $\{\lambda x.x\}[y/x]$, by c. we get $\lambda x.y$, which is not what we would expect.
The problem here is related to scoping. In general, in a programming language, each variable is in the scope of a:

- variable definition (e.g. `int x = ....`), or

- function definition (e.g. `int f(int x) {...}`),

In the Lambda Calculus, we do not have variable definitions, hence variables can be ***free*** of a particular value. However, scoping is still required. In the previous example, the single occurrence of variable x is the parameter of the function $\lambda x.x$, not of $\lambda x.\lambda x.x$. Hence - informally, replacing it by y in $\lambda x.x$, as done above - should be ***illegal.***
paragraph[Free and bound variables] Free and bound variables

We label each occurrence of x in the following expression: $(\lambda x.\lambda x.x_1\ x_2)$
Informally, the occurrence x_1 is bound, because x_1 is the parameter of function $\lambda x.x_1$. However, x_2 is free - it does not designate the parameter of some function.
An occurrence of a variable can either be bound or free. Suppose x_i is an occurrence of variable x in the λ-expression e. Then, if:

- $e = x$, then x_i is free in e.
- $e = \lambda x.e'$ is bound in e.
- $e = (e_1\ e_2)$ is bound/free in e iff it is bound-/free in e_1 or e_2 (note that x_i can occur either in e_1 or in e_2).

A variable is bound iff all its occurrences are bound.
For example, in the expression:

- $e = (\lambda x.\lambda x.x\ x)$

we have two occurrences of variable x: the first is bound in e, however the second is free. Hence variable x is free.
The definition of a free variable makes more sense when judged in the context of a reduction. Let us replace the phrase:

- the substitution of all occurrences of variable x with parameter par in the body of the function.

with:

- the substitution of all *free* occurrences of variable x with parameter par in the body of the function.

in the informal text for function application, and consider the following application:

$$(\lambda x.(x\ \lambda x.x)\ z)$$

- the first occurrence of x is the formal parameter of function $\lambda x.(x\ \lambda x.x)$
- the second occurrence of x is the formal parameter of function $\lambda x.x$

Thus, x is free in $(x\ \lambda x.x)$, and precisely those free occurrences need to be replaced by the substitution. The expected result of the application should be: $(z\ \lambda x.x)$.
Let us fix point c. with this in mind:

- c1. $\{\lambda y.body\}[par/x] = \lambda y.body[par/x]$ if $x \neq y$
- c2. $\{\lambda x.body\}[par/x] = \lambda x.body$

However, point c1. still suffers from a problem. To see this, consider the expression:

$$\lambda y.(\lambda x \lambda y.x\ y)$$

Here, we can reduce the inner expression and by applying the c1. substitution rule we would obtain: $\lambda y.\lambda y.y$. However, this result is incorrect. By re-examining the expression, we observe that $\lambda x.\lambda y.x$ is a *selector*. Application $(\lambda x \lambda y.x\ \ y)$ should return a constant function, not the identity function.
At second glance, we can observe that, by simply applying c1, the scoping of y has changed: y was initially the formal parameter of $\lambda y.(\lambda x \lambda y.x\ y)$. After the reduction, it became the formal parameter of $\backslash y.y$, hence changing the entire meaning of the expression.
To solve this, we should:

- rename all ys in the function body $\lambda y.x$ by a new, unused variable. The result is: $\lambda y.(\lambda x \lambda z.x\ y)$
  * next, we can proceed with c1. as before.

  To capture this, we replace c1 by:

  * c1'. $\{\lambda y.body\}[par/x =$ \lambda y.body[par/x]ifma
  \neq y] and y is not free in par
- c2'. $\{\lambda y.body\}[par/x] = \{\lambda z.body[z/y]\}[par/x]$ if $x \neq y$ and y is free in par

subsubsection[] Step 2. reduction order(s)
In Step 1. we have satisfactorily defined the reduction:

$$(\lambda x.body\ par) \Rightarrow body[x/par]$$

of a function application, by a thorough definition of substitution of all free occurrences of a variable (x) in an expression (body).
In step 2. we look at more complicated expressions, in which several reductions can be performed at the same time. Let $e_1 \Rightarrow e_2 \Rightarrow \ldots \Rightarrow e_n$ be a sequence of zero or more reductions (hence n can be 1). We write $e_1 \Rightarrow^* e_n$ - hence e_1 reduces to e_n in zero or more steps.
Consider:

$$\underline{(\lambda x.z\ \underline{(\lambda x.(x\ x)\ \lambda y.y)})}$$

where we have underlined the two possible reductions. If we start with the second reduction, we obtain:

- $(\lambda x.z\ \underline{(\lambda y.y\ \lambda y.y)})$ which by the underlined reduction becomes:
- $(\lambda x.z\ \lambda y.y)$ which reduces to
- z

on the other hand, had we started with the first reduction in our example, we would obtain z in a single step. The following result illustrates that selecting which application to reduce first, will not affect the ultimate result obtained via a sequence of reductions:
paragraph[Theorem (Church Rosser)] Theorem (Church Rosser)
Consider e, e_1, e_2 be $\lambda - expressions$ such that $e \Rightarrow^* e_1$ and $e \Rightarrow^* e_2$ such that $e_1 \neq e_2$. Then there exists e' such that $e_1 \Rightarrow^* e'$ and $e_2 \Rightarrow^* e'$.
However, it is possible to choose a reduction sequence which does not terminate. Consider:

$$\underline{(\lambda x.z\ \underline{(\lambda x.(x\ x)\ \lambda y.(y\ y))})}$$

If we select the outer application first, the result is, as before z. However, if we continuously select the inner application, we obtain a non-terminating reduction sequence.

There are two strategies for application selection:

- normal strategy (informally: ***reduce the function first***)
- applicative strategy (informally: ***reduce the parameter first***)

In our first example, we have applied the aplicative strategy: we have always reduced the function parameters first. In our second example, we have applied the normal strategy: we have evaluated the function and ignored the parameter, hence avoiding non-termination.
We write $e \Rightarrow^n e'$ (resp. $e \Rightarrow^a e'$) whenever e reduces to e' in zero or more reduction steps, via the normal (resp. applicative) strategy.
We now formally define $\Rightarrow^n$ and $\Rightarrow^n$:
paragraph[Applicative evaluation] Applicative evaluation

We describe the strategies using the notation:

$$\frac{A}{B}$$

Which expresses that B is true whenever A is true.

$$(TFun)\frac{e \text{ does not contain applications}}{(\lambda x.body\ e) \Rightarrow^a body[e/x]}$$

$$(TApp1)\frac{e \Rightarrow^a e'}{(\lambda x.body\ e) \Rightarrow^a (\lambda x.body\ e')}$$

The rule $(TApp1)$ ensures that, if a parameter e from an application can be reduced to e', then it will be done so before computing the application. This can only be done (via $(TFun)$), only after parameters can no longer be reduced.

$$(TApp2)\frac{e \Rightarrow^a e'}{(e\ e'') \Rightarrow^a (e'\ e'')}$$

The rule $(TApp2)$ ensures that, whenever the right-hand side is not a function but is reducible, it will be reduced.

In the following example, we illustrate a sequence of reduction rules via the applicative strategy.

$$((\lambda x.(x\ x)\ \lambda y.y)\ (\lambda x.x\ \lambda x.x))$$

In our particular example, we cannot apply $(TApp1)$, since the right-hand side of the application is not a function. However, we can apply $(TApp2)$: since $(\lambda x.(x\ x)\ \lambda y.y) \Rightarrow^a (\lambda y.y \lambda y.y)$, the expression becomes:

$$((\lambda y.y\ \lambda y.y)\ (\lambda x.x\ \lambda x.x))$$

as before, we apply $(TApp2)$ and obtain:

$$(\lambda y.y\ (\lambda x.x\ \lambda x.x))$$

since the evaluation is applicative, we are forced to apply $(TApp1)$:

$$(\lambda y.y\ \lambda x.x)$$

finally, we obtain $\lambda x.x$.
paragraph[Normal evaluation] Normal evaluation

We only require two rules to specify normal evaluation:

$$(TFun)\frac{}{(\lambda x.body\ e) \Rightarrow^a body[e/x]}$$

$$(TApp)\frac{e \Rightarrow^a e'}{(e\ e'') \Rightarrow^a (e'\ e'')}$$

Observe that parameters are never evaluated under the normal strategy. We illustrate it on the previous example:

$$((\lambda x.(x\ x)\ \lambda y.y)\ (\lambda x.x\ \lambda x.x)) \Rightarrow^n$$

, via $(TApp)$:

$$((\lambda y.y\ \lambda y.y)\ (\lambda x.x\ \lambda x.x)) \Rightarrow^n$$

, via $(TApp)$:

$$(\lambda y.y\ (\lambda x.x\ \lambda x.x)) \Rightarrow^n$$

, via $(TFun)$:

$$(\lambda x.x\ \lambda x.x) \Rightarrow^n$$

, via $(TFun)$:

$$\lambda x.x$$

A fundamental property of normal evaluation is that, for each λ-expression e, if there exists an expression e' which is irreducible and $e \Rightarrow^* e'$ (i.e. e reduces to e' via some arbitrary sequence of reductions), then $e \Rightarrow^n e'$ (i.e. by applying the normal evaluation strategy, we will obtain e'.

paragraph[Lazy evaluation] Lazy evaluation

Consider the following example:

$$(\lambda x.(x\ x)\ (\lambda y.y \lambda y.y))$$

By applying the normal evaluation strategy, we obtain:

$$((\lambda y.y \lambda y.y)\ (\lambda y.y \lambda y.y)) \Rightarrow^n$$

$$(\lambda y.y\ (\lambda y.y \lambda y.y)) \Rightarrow^n$$

$$(\lambda y.y \lambda y.y) \Rightarrow^n$$

$$\lambda y.y$$

The careful reader will observe that $(\lambda y.y \lambda y.y)$ - i.e. the parameter of the outer function, is evaluated twice, which is really inefficient.

The lazy evaluation strategy improves on this very issue. We informally illustrate Haskell lazy evaluation on this example. Note however, that lazy evaluation is a programming language-related strategy, and cannot be properly defined in the Lambda Calculus.
We start of with:

$$(\lambda x.(x\ x)\ (\lambda y.y \lambda y.y))$$

In Haskell, the parameter $par = (\lambda y.y \lambda y.y)$ is a thunk, i.e. a memory object which contains:

- the expression itself, here $(\lambda y.y \lambda y.y)$
- a value, initially unknown $?$

In the above evaluation we apply the normal strategy and obtain:

$$(par\ par)$$

only here (in Haskell) the occurrences of par are not independent as in the Lambda Calculus - they point to the same thunk. Now, par is not a function, but it can be evaluated. The result is $\lambda y.y$. At this point, the value of the expression is stored in the thunk. Since the result is the identity function, the result is:

$$par$$

but now *par* has already been evaluated hence the final result is immediately $\lambda y.y$, with no other (re-) evaluation of the inner expression.
subsection[] The Lambda Calculus as a programming language
section[H01: Haskell Lazy Dynamic Programming] H01: Haskell Lazy Dynamic Programming To access the two links you must log in with your LDAP account.

`Homework Statement`[50]

`Homework Archive`[51]

chapterLogic programming section[Course: Introduction to Prolog] Course: Introduction to Prolog section[] Introduction to Prolog
subsection[] The basics
Consider the following code:

```
student(gigel).
student(ionel).
student(john).
student(mary).

male(gigel).
male(ionel).
male(john).
female(mary).

lecture(pp).
lecture(aa).

studies(gigel,aa).
studies(mary,pp).
studies(mary,aa).
studies(john,pp).
```

subsubsection[] Atoms `gigel`, `ionel`, `john`, `mary`, `pp`, `aa` are atoms. Atoms are basic Prolog constructs. Prolog is a weakly-typed language, and primitive types such as *String* or *Integer* are part of the language. The afore-mentioned elements have type *atom* (which is different from *String*).
Atoms are specific to Prolog, and they are *inherited* from First-Order Logic, the formalism on which Prolog is based.
subsubsection[] Relations

`studies`, `male`, `female` and `lecture` are relations.
In First-Order Logic, a relation over sets $A_1, \ldots, A_n$, having arity n, is a subset $R \subseteq A_1 \times \ldots \times A_n$.
In Prolog, we make no distinction between the types (A_i) of each element from a relation. Thus, in Prolog, a relation of arity n is a subset $R \subseteq Prim$ where $Prim$ is the set of primitive values from the language. Strings, integers and atoms are such values.
The above program is simply a definition of 4 relations. Mathematically, :$studies = \{(gigel, aa), (mary, pp$
subsubsection[] Queries
After loading the previous program, the interpreter (with prompt `?-`) expects a query from the programmer. Formallly, a query is a conjuction of literals, where each literal is of the form: $R(t_1, \ldots, t_n)$ and t_i are terms. Instead of pursuing a formal approach, we illustrate examples of queries, for our above program (in what follows, we prepend `?-` to Prolog code to remind the reader we are performing a query):

- `?- student(gigel)`
- `?- student(gigel),studies(gigel,aa)` (, should be read as logical *and*)
- `?- student(X)`:
  - `X` is a variable. All tokens starting with a capital letter are (treated as) variables.

  - Variables can be bound to any value from the program. Variables do not have a type.
  - When a variable is present in a query, Prolog will try to find all values for that variable which satisfy the query. By typing ; we can iterate over all such values.

- `?- student(X), lecture(X,aa)`

subsubsection[] Clauses
As shown so far, programs behave as ***databases*** (relations are essentially ***tables***) and via the interpreter, we can use queries to interrogate the database.
However, a Prolog program is far more than just a database. We can also define new relations based on existing ones, without exhaustively enumerating all members. For instance, let us add to our initial program, the relation ***all male students which share a lecture with a female student***:

```
hasFemaleColleague(M) :- student(M), male(M), studies(M,L),
    studies(F,L), female(F).
```

The above clause or rule can be translated to First-Order Logic (FOL) where it can be read easier:
$student(M) \wedge male(M) \wedge studies(M, L) \wedge studies(F, L) \wedge female(F) \implies hasFemaleColleague(M)$

Notice the direction of the implication, as well as the usage of `,` (logical conjunction), `:-` (implication) and `.`(the end of a clause).
subsubsection[] Re-satisfaction
Let us define a new property (1-ary relation) which describes those lectures which have at least two students. The first attempt:

```
lectureOfTwo(L) :- lecture(L), studies(X,L), studies(Y,L).
```

is incorrect, as `X` and `Y` may be bound to the same student. Another attempt is:

```
lectureOfTwo(L) :- lecture(L), studies(X,L), studies(Y,L), X \= Y.
```

however following the interrogation `-? lectureOfTwo(L`
`pp` and `aa` are prompted twice. To understand this, let us write our relation in FOL:
$lecture(L) \wedge studies(X, L) \wedge studies(Y, L) \wedge X \neq Y \implies lectureofTwo(L)$
Formally, FOL sentences require that all variables are within the scope of a quantifier. Thus, to be more exact, we should write:
$\forall L(lecture(L) \wedge \exists X \exists Y(studies(X, L) \wedge studies(Y, L) \wedge X \neq Y) \implies lectureofTwo(L))$
Notice the scope of each quantification, as a change in the parentheses may modify the meaning of the sentence.

Prolog does not interpret the sentence as in FOL. Instead, it finds all combinations of values for `L`, `X` and `Y` that satisfy the left-hand side (LHS) of the implication. In our example, these are:

```
pp, mary, john
pp, john, mary
aa, gigel, mary
aa, mary, gigel
```

and then reports the values of `L` for each such combination. For this reason, we observe that both aa and pp are reported twice.
We can verify this by querying: `-? lecture(L), studies(X,L), studies(Y,L), X \= Y.`
subsubsection[] Proof trees
We briefly discuss the concept of `proof trees` which will be approached in more detail later on, when discussing resolution. To introduce it, consider the queries:

```
-? female(X), studies(X,L).
```

and

```
-? studies(X,L), female(X).
```

Viewed as FOL sentences (with appropriate quantification), the queries are the same. However, in answering those queries, Prolog searches its knowledge-base (database) differently.
For the first query:

```
Find X which satisfies female(X):
  X = mary,
  Find L which satisfies studies(mary,L):
  L = pp,
    report X = mary, L = pp.
  L = aa
    report X = mary, L = aa.
```

And for the second:

```
Find X,L which satisfy studies(X,L):
  X = gigel, L = aa
    female(gigel) could not be established, continue search
  X = mary, L = pp
    female(mary) is true, report X = mary, L = pp.
  X = mary, L = aa
    female(mary) is true, report X = mary, L = aa.
  X = john, L = pp
    female(john) could not be established, search finishes.
```

This simplistic example shows that the complexity of a Prolog program (execution), may be affected by the order in which literals appear in a clause. We will later see that even the output may be affected by this order.
subsection[] Prolog programming
subsubsection[] Instantiating variables
The question *does Prolog permit side-effects* is difficult to answer. On one hand:

- side-effects are present: variables can be instantiated. For instance, when resolving `-? female(X), studies(X,L)`, initially, the variable `X` is unbound. However, once the part `female(X)` has been satisfied, `X` is bound to `mary`.

- however, side-effects are inherently limited: during the satisfaction of a query/clause, a bound variable cannot be unbound. In our example, `X` remains ***permanently bound to Mary***. `X` can only become free upon resatisfaction of a query/clause.

Variables can be (partially) bound several ways:

- via unification: e.g. `X = 2` is not the conventional assignment - = designates unification. We illustrate this via several examples:
  - `-? X = 2, Y = X.`
  - `-? X = Y + 2, Z = 2 + Y, X = Z.` - here note that expressions are treated as such - no computation (of 2+2) is performed.
- conventional assignment is performed using the `is` operator:
  - `-? X is 2+2.`
- comparison is performed using the operator `=:=`:
  - `-? X is 2+2, X =:= 4.`

subsubsection[] Representations - lists
paragraph[The idea] The idea

One distinguishing feature of FOL is that it allows representing objects which programmers use, such as lists, trees, graphs, matrices, etc. Prolog inherits this feature. In Prolog, we can enroll any term in a relation, including other - composite terms. At the same time, variables can be bound to terms. We illustrate this via an example:

```
-? X = f(Y,Z), Y = g(W), W = Z, Z = 2.
```

Here, `X` is bound to `f(g(2),2)`. Note that `f` is a binary relation and `g(2)` (a term) is enrolled with `2` in this relation.
We can exploit this feature to represent lists as follows:

```
-? L = cons(1, cons(2, cons(3, void))).
```

Note here that:

- `cons` is a binary relation. A list is an instance of such a relation.
- it enrols an element (the head) together with a relation instance describing the tail.

We can define helpful relations over lists:

```
head(L,H) :- L = cons(H,_).
tail(L,T) :- L = cons(_,T).

empty(L) :- L = void.
nonempty(L) :- L = cons(_,_).
```

_ is used much in the same way as in Haskell - to designate an arbitrary term whose name is unimportant for the programmer.
The basic observation here is that:

- we use queries over relations in order to perform computations on lists
- we assume variable `L` is bound and stands for the input
- we assume variables `H` and `T` are unbound, and stand for the output. They will be bound to the result, or results, if there are several.
- example: `-?  L = cons(1,cons(0,void)), head(L,H)`
- essentially, this is a programmer convention. For instance, the query: `-? head(L,1).` is valid, and will produce: `L = cons(1, _G766).` Here, `_G766` is an anonymous variable which is not bound - it can be anything. Basically, what Prolog is telling us is that `L` must be *cons of 1 into something*.

We can add new relations such as `len`:

```
len(L,R) :- empty(L), R = 0.
len(L,R) :- nonempty(L), tail(L,T), len(T,R1), R is R1 + 1.
```

Note that, we have defined two clauses which treat terms of different types for `L`. We can rewrite `len` in several ways, which illustrate Prolog's flexibility:

```
len(void,R) :- R = 0.
len(cons(_,T),R) :- len(T,R1), R is R1 + 1.
```

We can perform unification *implicitly* in the left-hand side of the implication. We can also do it for `R`, which results in:

```
len(void,0).
len(cons(_,T),R) :- len(T,R1), R is R1 + 1.
```

paragraph[Actual Prolog lists] Actual Prolog lists

In Prolog (as in Haskell), lists are an essential programming construct. In the previous section, we have illustrated the principle governing list representations (as relations). Lists are already implemented in Prolog and helpful relations (or predicates) are already defined for them.
Real Prolog lists are defined as follows:

```
[] % the empty list
[H|T] % 'cons' of H into T
[1,2,3] % a list with integers 1,2 and 3
```

Thus, in order to work on Prolog Lists, we could rewrite `len` as follows:

```
len2([],0).
len2([_|T],R) :- len2(T,R1), R is R1 + 1.
```

subsubsection[] List membership

We start with the following implementation of the predicate `contains` which verifies if an element is part of a list:

```
contains(E,[E|_]).
contains(E,[H|T]) :- E \= H, contains(E,T).
```

Testing our predicate yields:

```
?- contains(1,[1,2,3]).
true
false.
```

The second `false` seems puzzling and inconvenient. We can trace it down if we build the ***proof tree*** for `contains(1,[1,2,3])`:

```
contains(1,[1|_]) is satisfied. true is reported. Prolog continues
    re-satisfaction:
contains(1,[1|[2,3]]) may be satisfied:
  1 \= 1 is not satisfied, hence Prolog reports false. Re-satisfaction
    ends.
```

We shall examine a more efficient way of defining `contains` in a future lecture.
paragraph[List reversing] List reversing

We start with the following - accumulator-based implementation:

```
rev([],R,R).
rev([H|T],Acc,R) :- rev(T,[H|Acc],R).
reverse(L,R) :- rev(L,[],R).
```

While `-? reverse([1,2,3],L).` works just as expected, `reverse(L,[1,2,3]).` reports the correct result and then loops. First of all, we note that this query violates our assumption the the *first* list is the *input*, while the second - the *output*. Again, to understand the behaviour, we build the proof tree:

```
to satisfy reverse(L,[1,2,3]), we must satisfy rev(L,[],[1,2,3]).
the first clause of rev cannot be satisfied since [] and [1,2,3] do not unify.
during the satisfaction of the second clause L = [H1|T1] (which is possible since L is unbound).
  We attempt to satisfy rev(T1,[H1|[]],[1,2,3]):
    the first clause of rev cannot be satisfied ([H1] and [1,2,3] do not unify)
    while satisfying the second clause T1 = [H2|T2].
      We attempt to satisfy rev(T2,[H2|[H1|[]]],[1,2,3]).
         the first clause of rev cannot be satisfied ([H2,H1] and [1,2,3] do not unify)
         while satisfying the second clause T2 = [H3|T3].
            We attempt to satisfy rev(T3,[H3|[H2|[H1|[]]]],[1,2,3])
              the first clause of rev CAN be satisfied ([H3,H2,H1] unify with [1,2,3]). The result is reported
              we attempt re-satisfaction via the second clause: T3 = [H4|T4]
                the first clause of rev cannot be satisfied ([H4,H3,H2,H1] and [1,2,3]) do not unify).
                ....
                the process repeats until the stack becomes full.
```

subsubsection[] Take-away

- Building and understanding *proof trees* is an essential tool for learning to program safely in Prolog.

- It is important to keep a convention regarding *input* and *output* variables when writing a Prolog program. Sometimes, such a convention can be more flexible (`contains` will be an example), however, the programmer must make sure this is indeed possible.

subsection[Lab: Prolog 101] Lab: Prolog 101 subsection[] Laborator 7 - Prolog: Introducere subsubsection[] Despre Prolog
Prolog este un limbaj de programare centrat pe un set mic de mecanisme de baza(e.g. pattern-matching, backtracking). Desi setul de mecanisme de baza este limitat, Prolog este un limbaj foarte puternic si flexibil.
subsubsection[] Structura unui program Prolog
Un program Prolog contine (doar) trei constructii:

- fapte
- reguli
- interogari

subsubsection[] Fapte

Folosim faptele pentru a exprima un adevar. De exemplu vrem sa "definim" ca Gigel este barbat. Pentru a defini o regula este suficient sa ii definim numele(de preferat sugestiv) si sa ne hotaram care este "actorul". In cazul de fata Gigel este actorul iar numele regulii este barbat =>barbat(gigel).
Observati numele regulii - barbat - numele actorului - gigel(cu litera mica!) - si punctul de la sfarsitul regulii(echivalent ";")
subsubsection[] Interogari

Interogarile sunt metodele prin care noi putem sa ii cerem ceva programului. Pentru a "intreba" ceva este suficient sa scriem in interpretor ce vrem sa aflam.
De exemplu vrem sa aflam daca Gigel este barbat, pur si simplu "intrebam" in interpretor barbat(gigel). (observati numele regulii, actorul si punctul de la final!).
Raspunsul dat de programul nostru este true. Raspunsul este afirmativ deoarece acest fapt exista in cadrul programului nostru in setul de fapte si reguli (denumit in continuare baza de cunostinte).

Ce va raspunde programul la urmatoarea interogare: copil(gigel).*?
Raspunsul este false (nu avem nicio informatie in baza de cunostinte legata de varsta lui Gigel).

Dar la intrebarea femeie(andreea)? De ce?
subsubsection[] Reguli

Regulile ne permit sa exprimam o legatura intre anumite fapte si reguli. De exemplu, pentru ca Gigel sa fie sot el trebuie sa fie insurat si sa fie barbat.
barbat(gigel).
insurat(gigel).
sot(gigel):- barbat(gigel),insurat(gigel). Ce spune aceasta regula? Ce inseamna ","?
Tocmai am definit o regula care spune ca Gigel este sot daca Gigel este barbat si daca Gigel este insurat. Virgula este modul in care se exprima conjunctia logica in Prolog
Care este raspunsul la urmatoarea interogare: sot(gigel). ?
subsubsection[] Exemplu Fie urmatoarea baza de cunostinte:

```
femeie(ioana).
barbat(mihai).
barbat(andrei).
casatorit(mihai,ioana).
parinte(ioana,andrei).
parinte(mihai,andrei).
```

Observam ca am definit fapte cu aritate doi(nu suntem limitati la fapte/reguli cu aritate 1!). Care este raspunsul la urmatoarea interogare: casatorit(ioana,mihai).?
Vrem sa definim o regula care sa ne spuna daca Mihai este tata: tata(mihai):- barbat(mihai), parinte(mihai,andrei).(Mihai este barbat si este tatal lui Andrei).
Problema cu aceasta abordare este ca Mihai poate sa fie parintele mai multor copii, nu doar al lui Andrei, ceea ce nu il invalideaza ca si tata. Cum rezolvam aceasta problema? Ne trebuie o "generalizare" a lui Andrei. Aceasta generalizare o vom "implementa" folosind variabile. subsubsection[] Variabile Variabilele in Prolog sunt stringuri care incep cu litera mare.
Noua regula devine: tata(mihai):- barbat(mihai), parinte(mihai,Copil).
Prolog va incerca(prin backtracking) sa lege variabile Copil la orice actor care respecta faptul parinte(mihai,Copil) (in cazul nostru doar andrei).
Ce se intampla daca baza noastra de cunostinte contine (mult) mai multe fapte si reguli si vrem sa aflam care sunt toate mamele.?
mama(Mama):- femeie(Mama), parinte(Mama,Copil).
Interogand mama(Mama). variabila Mama va lua, pe rand, totii actorii care satisfac cele doua reguli.(Interogarea intoarce cate o valoare pe rand. Pentru e trece la urmatoarea valoare folositi ;)

subsubsection[] Liste Un concept extrem de util in Prolog este cel de lista. In Prolog, o lista se declara in felul urmator:

```
[1, 2, 3].
[1 | [2 | [3 | []]]]. ([1,2,3])
[mihai, ioana, andrei].
[mihai, parinte(ioana), X].
[1, [2, copil(andrei)], [], [X, Y, Z]].
```

Din penultimul exemplu se poate vedea ca o lista poate contine atat variabile, cat si constante sau itemi complecsi. Din ultimul exemplu se poate vedea ca o lista poate contine ca element o alta lista.
Ne propunem in continuare sa obtinem primul element dintr-o lista, precum si restul listei(head si tail din Haskell). Pentru aceasta, se va folosi unificarea din Prolog.

```
?- [H|L] = [1, 2, 3].
H = 1,
L = [2,3].
```

In urma comenzii, capul listei va fi legat la variabila X, iar restul listei va fi legata la variabila Y.

```
?- [H1,H2|_] = [1, 2, 3].
H1 = 1,
H2 = 2.
```

In urma comenzii, primele doua elemente ale liste vor fi legate la variabilele H1 si H2. Simbolul '_' (underscore) are un comportament similar cu cel din Haskell si anume cel de variabila anonima. Folosim acest simbol cand nu dorim sa legam ceva la o variabila.
Incercam in continuare sa calculam lungimea unei liste.
Incepem prin a defini o regula prin care determinam daca ceva reprezinta o lista.

```
empty(L) :- L = [].
isList(L) :- empty(L).
isList([H|T]) :- isList(T).
```

Ne dam seama ca putem simplifica codul.

```
isList([]).
isList([_|T]) :- isList(T).
```

Calculam acum lungimea unei liste.

```
len(L, R) :- empty(L), R = 0.
len([H|T], R) :- isList(L), len(T, R1), R = R1 + 1.
```

Observati ca secventa nu functioneaza cum ar trebui pentru ca unificarea previne evaluarea lui R. Varianta finala a codului este:

```
len([], 0). (unificare implicita)
len([_|T], R) :- len(T, R1), R is R1+1.
```

Cum determinam daca un element apartine unei liste?

```
contains(E, [E|_]).
contains(E, [X |T]) :- E \= X, contains(E, T).

?- contains(1, [1,2,3]).
true;
false.
```

De ce prima data obtinem true, iar dupa aceea false? subsubsection[] Exercitii liste & diverse

1. Definiti predicatul `firstTwo(X,Y,L)` care leaga variabilele `X`, `Y` la primele doua elemente din lista `L`, daca acestea exista.
2. Definiti predicatul `notContains(E,L)` care verifica daca elementul la care este legat `E` nu exista in lista `L`.
   - Poate acest predicat sa fie folosit pentru a genera toate elementele care nu sunt in `L`? Justificati raspunsul.
3. Definiti predicatul `unique(L1,L2)`. `L2` este lista `L1` fara elemente duplicate.
4. Definiti predicatul `listOnly(L1,L2)`. `L2` este lista `L1` care contine doar elementele de tip lista. Exemplu: `listOnly([1,[2,3],4,[5],6], [ [2,3],[5]])`.
5. Implementati `insertionSort` in Prolog.
   - Idenficati predicatele necesare, aritatea fiecaruia, variabilele care reprezinta input-ul, respectiv output-ul.

subsubsection[] Arbori

1. Stabiliti o conventie de reprezentare pentru arbori. Ilustrati conventia in Prolog. Exemplu: `T=....`
2. Implementati predicatele `size(T,S)` si `height(T,H)`.
3. Definiti predicatul `flatten(T,L)` unde `T` este un arbore.

subsubsection[] Reprezentarea datelor in Prolog

1. Stabiliti o conventie de reprezentare pentru expresii generate de gramatica de mai jos, urmarind acceeasi abordare ca la arbori:

```
<expr> ::= <valoare> | <variabila> | <expr> + <expr> | <expr> *
 <expr>
<variabila> ::= string
<valoare> ::= orice
```

1. Scrieti un predicat `eval(Expr,R)` unde `R` este rezultatul evaluarii lui `Expr`. `Expr` nu contine variabile.
2. Scrieti un predicat care permite evaluarea unei expresii ce contine variabile.

subsubsection[] Resurse (functii matematice)

- `Functii utile in Prolog` [1]
- `Tutorial Prolog` [2]

section[Course: Clauses, unification, cut, negation] Course: Clauses, unification, cut, negation
section[] Formal syntax, Unification, Backtracking, Cut, Negation
subsection[] Prolog syntax
The basic programming building block in Prolog is the clause. To define it formally, we first introduce:

```
<var> ::= alphanumeric sequence starting with capital
<term> ::= <number> | <atom> | <var> | <predicate>(<term1> ...,
    <term_n>)
```

The definition of terms is slightly restrictive. For instance, terms are also arithmetic expressions (e.g. `1 + X`), list *patterns* (e.g. [1|[]]), etc. But each such term can be expressed as a predicate, as already illustrated for lists, in the previous lecture.
A clause is defined via the following grammar:

```
<clause> ::= <res> :- <goal_1>, ..., <goal_m>.
<res> ::= <predicate>(<var_1>, ... <var_k>)
<goal_i> ::= <predicate>(<var_1>, ... <var_k>) |
             <term> = <term> |                    // unification
             <var> is <term> |                    // arithmetic
    assignment
             <term> =:= <term> |                  // arithmetic
    comparison
             true |                               // default
    satisfying goal
             fail |                               // default-failing
    goal
             ! |                                  // cut (discussed
    below)
```

Let:

```
q(X,Y) :- r(X), p(Y).
```

serve as an example. Then `q(X,Y)` is called resolvent, or goal. Also, in, any query e.g. `-? q(X,Y).`, `q(X,Y)` is termed goal (which Prolog tries to *prove*). `r(X)` and `p(Y)` are called subgoals (which Prolog needs to prove, in order to prove `q(X,Y)`.
There exist special coals, such as = (unification), `is` (assignment) or `=:=` (comparison):

- the goal `<term1>= <term2>` will be satisfied iff the two terms unify. Under unification variables become bound by a substitution.
- the goal `<var>is <term>` will be satisfied iff there is no type error, and will result in binding the variable to the evaluation of the term.
- the goal `<term1>=:= <term2>` will be satisfied iff the evaluation of each term yields the same value.

We also note that the above syntax is not complete, but covers most of the Prolog programming aspects of this lecture.
subsubsection[] Shorthands
Clauses such as:

```
P(X,Y) :- true.
```

can also be written as:

```
P(X,Y).
```

Also, a clause which involves unification such as:

```
P(X) :- X = f(a).
```

can also be written as:

```
P(f(a),Y).
```

Shorthanded unification is often used for lists, and looks similar to Haskell pattern matching, e.g.

```
size([],0).
size([_|T],R) :- size(T,R1), R is R1 + 1.
```

subsubsection[] Common pitfalls
The following two clauses:

```
... :- ..., X=p(A,B) , ...
... :- ..., p(A,B) , ...
```

may look similar, but their subgoals are actually very different:

- the first is satisfied if the unification of `X` with `p(A,B)` succeeds (hence `p(A,B)` is merely a term here).
- the second is satisfied if `p(A,B)` is satisfied.

subsection[] Unification
We write $t =_S t'$ to express that ***term** t **unifies with** t' **under substitution** S*.

A substitution is a set of pairs (X, t), where X is a variable and t is a term. As a shorthand, we write t/X instead of (X, t).
A substitution S is consistent iff for all pairs t_1/X and t_2/X, $t_1 =_S t_2$ (the variable X cannot be bound to different terms which do not unify).
subsubsection[] Unification rules
In what follows, we give some basic unification rules which serve the purpose of illustration. These rules are not implemented per.se. in the Prolog unification process.

$$\frac{}{atom =_S atom}$$

$$\frac{S \cup \{t/X\} \text{ is consistent}}{X =_S t}$$

For instance, $X =_{\{cons(H,T)/X\}} cons(1, void)$ since $\{cons(H, T)/X, cons(1, void)/X\}$ is consistent. However $X =_{\{void/X\}} cons(1, void)$ is false (X cannot at the same time be the empty list, and a list with one element).

$$\frac{p_1 = p_2, n = m, t_1 =_{S_1} t'_1, \ldots, t_n =_{S_1} t'_n, S = S_1 \cup \ldots \cup S_n \text{ is c}}{p_1(t_1, ..., t_n) =_S p_2(t'_1, \ldots, t'_m)}$$

For instance:
$p(a, q(X, Y), Z) =_S p(X, q(Y, Y), q(X))$, where S is $\{a/X, a/Y, q(a)/Z\}$.

The unification algorithm from Prolog will compute the most general substitution or the most general unifier (MGU). In this lecture, we will not provide a formal definition for the MGU. However, we illustrate the concept via an example.
$p(X, q(X, Z)) =_S p(Y, q(Y, Z)$ for $S = \{Y/X, X/Z\}$ however S is not an MGU. The constraint that Y unifies with Z is superfluous. Here, an MGU is $S = \{Y/X\}$.
subsubsection[] Recursive substitutions
Does `X=f(X)` produce a valid substitution? Intuitively, such a substitution would contain $f(f(...))/X$. Prolog's unification algorithm is able to detect such recursive substitutions and will not loop. Depending on the Prolog implementation (version) at hand, the unification may fail or succeed.
subsubsection[] Goal satisfaction (Backtracking)

During the satisfaction of a goal, Prolog keeps a current substitution which it iteratively updates. When a sub-goal is satisfied, a new "current substitution" is created.
Let us look in more depth at this.

```
contains(E,[E|_]).
contains(E,[H|T]) :- E \= H, contains(E,T).
```

The execution tree for the goal `-? contains(2,[1,2,3])` is given below:

```
contains(2,[1|[2,3]])
    2 \= 1
    contains(2,[2|_]) -> true (continue goal satisfaction)
    contains(2,[2|[3]])
       2 \= 2 -> false. (re-satisfaction not possible).
```

Alternatively, let us look at how `member` is implemented:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

The execution tree for `member(2,[1,2,3])` is shown below:

```
member(2,[_|[2,3]])
   member(2,[2|_]) -> true (continue goal satisfaction).
   member(2,[_|[3])
      member(2,[_|[]])
         member(2,[]) -> false (re-satisfaction not possible).
```

The difference between `contains` and `member` is that, while the former stops once an element in the list has been found, the second continues search. Is there a good motivation for the `member` implementation?

- build a goal tree for `member(X,[1,2,3])`
- build a goal tree for `contains(X,[1,2,3])`.

paragraph[Backtracking] Backtracking

Note that goal satisfaction in Prolog is similar to *pruned backtracking*.

subsubsection[] Cut (!) - pruning the goal tree Consider the following program:

```
f(a).
f(b).
g(a).
g(b).

q(X) :- f(X), g(X).
```

The proof tree for `-? q(X)` is:

```
f(X)
  X = a (subgoal satisfied). Current substitution is {a/X}
    g(X)
      g(a) true. The goal q(X) is satisfied under {a/X}. Attempt
    re-satisfaction ...
      ... resatisfaction not possible for g(X) under {a/X}.
  X = b. Current substitution is {b/X}
     g(X)
       g(b) true. Attempt re-satisfaction....
       ... not possible
  not possible to re-satisfy f(X).
not possible to re-satisfy q(X). stop.
```

In Prolog, the special goal *cut* written `!`is satisfied according to the following rules:

1. when it is first received, it is implicitly satisfied.
2. when there is an attempt to re-satisfy it, it fails.

Let us modify the previous clause to:

```
q(X) :- f(X), !, g(X).
```

The corresponding proof tree is:

```
f(X)
  X = a (subgoal satisfied). Current substitution is {a/X}
    ! (implicitly satisfied)
    g(X)
      g(a) true. The goal q(X) is satisfied under {a/X}. Attempt
     re-satisfaction ...
      ... resatisfaction not possible for g(X) under {a/X}.
  X = b. Current substitution is {b/X}
    ! (implicitly fails).
q(X) cannot be re-satisfied. stop
```

We turn to a more elaborate example:

```
f(a).
f(b).
g(a,c).
g(a,d).
g(b,c).
g(b,d).

q(X,Y) :- f(X), !, g(X,Y).
q(test,test).
```

The example illustrates a side-effect of the cut semantics: `q(X,Y)` will not be satisfied for $\{test/X, test/Y$
This shows that the effect of cut also extends to the current goal, not only the current clause of the current goal.
Similarly, we have:

```
q(test,test) :- !.
q(X,Y) :- f(X), !, g(X,Y).
```

where cut prevents the re-satisfaction of `q` via the second clause (the second cut will not even be reached).
Also, consider the following code which extends the previous example:

```
r(X,Y) :- q(X,Y).
r(1,2).
```

The goal `-?  r(X,Y)` will be satisfied for $\{1/X, 2/Y\}$, which shows that cut does not have any effect on `r` - thus, cut should not be interpreted as a *global proof tree pruner*.

subsubsection[] Negation

Prolog implements negation as negation-as failure. This is also called the closed-world assumption. In short, the negation `not(G)` should be interpreted as *G cannot be proved in the current program.*

Note that, in First-Order Logic, the negation of a sentence p is interpreted more generally:

- it has its own *proof tree* which is independent of that for p
- it is possible to have *theories* (programs) where both p and p are true. Such theories are called inconsistent. It is also possible that p is provable while p is not. Such a theory is called incomplete.

The Prolog implementation of not is given below:

```
not(G) :- G,!,fail.
not(_).
```

It is also a good illustration of a design pattern for logical programming, which is often used in Prolog:

- The key property of `not` is that it satisfies without modifying the current substitution, even if G may bound variables:
  - In the first clause, if G is satisfied, cut is reached for the first time (satisfies). Subsequently failure occurs and re-satisfaction is prevented. The second clause is never reached.
  - However, if G is not satisfied, the cut is never reached, and `not` succeeds trivially, without binding any variable from G.

The reason while `X \= H` fails in our `contains` example is that it is actually a syntactic sugar for `not(X = H)`.
More precisely, note that, in: `-? H = 1, not(X = H).`

- `X = H` succeeds hence `not(X = H)`, fails.
- re-satisfaction of `not(X = H)` is not possible, hence any supergoal having `not(X = H)` on a branch will fail also.

subsection[Lab: Prolog 102] Lab: Prolog 102 subsection[] Laborator 08 - Programare in Prolog subsubsection[] Multimi

1. Definiti predicatul `cartesian(L1,L2,R)` care construieste produsul cartezian al `L1` cu `L2`
2. Definiti predicatul `union(L1,L2,R)` care construieste reuniunea a doua multimi codificate ca liste.
3. Definiti predicatul `intersection(L1,L2,R)`
4. Definiti predicatul `diff(L1,L2,R)` care construieste diferenta pe multimi intre `L1` si `L2`

subsubsection[] Permutari, Aranjamente, Combinari

1. Definiti predicatul `pow(S,R)` care construieste `power-set`-ul multimii `S`.
2. Definiti predicatul `perm(S,R)` care genereaza toate permutarile lui `S`.
3. Definiti predicatul `ar(K,S,R)` care genereaza toate aranjamentele de dimensiune `K` cu elemente luate din `S`
4. Definiti predicatul `comb(K,S,R)` care genereaza toate combinarile de dimensiune `K` cu elemente luate din `S`

subsection[Lab: Prolog graphs] Lab: Prolog graphs Fie un graf orientat în care fiecare nod are o culoare. Graful este reprezentat în Prolog printr-o listă ce conține doua elemente [C, E]:

- C: o listă de liste [n, c] cu semnificația că nodul n are culoarea c
- E: o listă de liste [n1, n2] cu semnificația că există o muchie de la nodul n1 la nodul n2

Exemplu: G = [[[1,galben], [2,rosu], [3,albastru], [4,verde], [5,rosu], [6,albastru], [7,rosu], [8,galben], [9,albastru], [10,mov]], [[1,2],[1,3],[2,6],[3,2],[
[6,4],[6,5],[6,7],[7,8],[8,5],[8,9],[8,10],[9,10]]].
subsubsection[] Exerciții

1. Scrieți un predicat getColors care construiește o listă cu toate culorile nodurilor
2. Scrieți un predicat getInEdges care construiește o listă cu toate elementele e din E de tipul (_, X) pentru un X anume
3. Scrieți un predicat getOutEdges care construiește o listă cu toate elementele e din E de tipul (X, _) pentru un X anume
4. Scrieți un predicat getUniqueColors (același lucru ca getColors dar fără duplicate)
5. Scrieți un predicat getNeighbors ce se folosește de predicatele getInEdges și getOutEdges pentru a construi o listă cu nodurile legate de un nod X

6. **Scrieți un predicat getPathsOfLength3 care construiește toate drumurile de lungime 3 ce pleacă dintr-un nod X. Valoarea cu care va unifica rezultatul va fi o listă [X, _, _]**

chapterNotes