**Tipi:** polimorfičentip: type 'a x=(int\*'a) list;; PoljeTip type ks = {mutable x:int};; -> let a={x=1};; a.x<-2;;

type barva = RGB of int\*int\*int| CB of int;; -> let barva1 = RGB (100, 100, 100)

**ArrayType matching:** let funkcija tip = match tip with| {stevec=x;imenovalec=y} -> x/y ;;

**ParametriziraniTipi:** type 'a b\_d = List of 'a | Tree of 'a b\_d \* 'a b\_d;;

**BinarnoDrevo** (rekurzivniTipi): type 'a grm = Noben | Eden of 'a \* 'a grm| Dva of 'a \* 'a grm \* 'a grm;;

**let** **rec** izpis d i = match d,i with

| Noben,\_ -> () | Eden (x,y),0->print\_float x | Eden (x,y),\_ -> izpis y (i-1) | Dva (x,y,z),0 -> print\_float x |

Dva (x,y,z),\_ -> izpis y (i-1);izpis z (i-1);;

let drevo\_0 = Dva (0.0,Eden (1.0,Eden (3.0,Noben)),Eden (2.0,Noben));;

izpis drevo\_0 2;;

**Float:** Obstaja 6.2e-2 potenca\*\* - dela samo na float 2. \*\* 2.;; 2.0 +. 6.9

**Char:** int\_of\_char c;; char\_of\_int 98;;

**String:** spremeni char: let str=ref (Bytes.of\_string "s");; Bytes.set !str 0 'z';; char v string: Char.escaped 'a';;

**Case:** match x with | xc when xc=0 ->();;

**If** a then x else y;;

**N-terice**: let tuple= (4, 5);; fst(x) - prvi;; snd(x) - drugi;;

**List:** fun: List.hd;; List.tl;; (indexSeznama) List.nth my\_list 0;; List.mem 1 [1;2;3] -> true;; List.rev [1;2;3];; je [3;2;1]

**Fun višjega reda:** List.map (fun x->x\*x) [4;5;1;2;7;3] -> [16;25;1;4;49;9];; List.filter (fun x->x<=1) [1;2;3;-1;-2] -> [-1; -2];; List.for\_all (fun x-> x>1) [4;2;3;4] – Če vsi izpolnjujejo pogoj vrne true;;

**Polje (Array):** Array.make 5 1;; Array.to\_list array;; Array.of\_list list;; Array.copy array;; (zamenjava) polje.(2) <- 20;;

**Matrika**: let m = Array.make\_matrix 3 2 0.0;;

**Array fun visjega reda**: Array.map (fun x->x\*x) [|4;5;1;2;7;3|];;

**Zanke:** For i=0 to 10 do print\_int i; print\_string "\n"; done;; let i= ref 0 inwhile (!i<10) i := (!i+1)done;;

**Kazalci(Pointer):** let n=ref 12;; !n -> 12;; za spremenit n := 13;; !n -> 13;;

**OOP:** class oop (x:int) = object (self) val mutable x = x method get\_x = x method set\_x y = x <- y method get\_x\_x=self#get\_x end;; **END:** let a = new oop 2;; a#get\_x\_x;; **Parametriziran razred:**

class ['a] matrika i j (a:'a) =

object

val mutable m= Array.me\_mx i a method get x y = m.(x).(y)

method set x y n = m.(x).(y) <- n

end;;(za inheritat: inherit tip m... ix iy)

**Agregacija (composition):** novClass z val mutable oopji = ([] : oop list) ;;

**Dedovanje(na osnovi razreda), Inheritance (specialization) :** inherit oop x as super;;

**Initializer:** initializer print\_string "start" ;;

**Privatna metoda:** method private get\_a = a ;; - lahko jo kličemo samo znotraj razreda

**Abstraktni (virtulaen) razred:** class virtual lik stranice =object val st\_stranic= stranice method virtual get\_plos:int end;;

**Moduli:** kot datoteka: ocamlc -c p.ml isto za d.ml;; ocamlc -o nM p.cmo d.cmo;; prviJe let x = (print\_string "jao");;drugiJe P.x;;

**Implementacija v sami datoteki**: module Ocena = (ime z veliko)struct type y={a:int} let tip y={a=y} let x=10 end;;

**Vmesnik**:module type STUDENT = sig type y val x: int end ;; module Student = (Ocena:STUDENT with type y=Ocena.y);; Student.x;; je 10;; let ocena\_0 = Ocena.tip 2;;ocena\_0.a;;

**Exception Omara**;;Raise Omara;;Exception Omara int\*char;; Raise (Omara (12,'c'));;try raise (Boo 12) when| Boo 12 -> "n"| Boo i when i<0 -> "n1";;

let rec implode l = match l with

| [] -> ""

| g::r -> String.make 1 g ^ implode r;;

let get\_height tree =

let maxh = ref 0 in

let rec height tree h = match tree with

| List k -> if h > !maxh then maxh := h

| Tree (a, sub1, sub2) -> height sub1 (h+1); height sub2 (h+1) in

height tree 1;

!maxh;;

let explode str =

let out = ref [] in

for i=0 to (String.length str-1) do out:=!out@[(str.[i])]

done;

!out ;;

**Izvor funkcijskih jezikov:** Turingovstorj in lambda racun (Turing, Church, Kleen) ;;

let sort a b = if a < b then -1 else 1

let dodaj2 seq num = List.sort sort (num::seq)

Preprosta vrednost je **ATOMIČNI TIP**(int, string, real, boolean) ;; Seznam (**LIST**) -> **glava** in **rep,** lahkoprazen ali pa ne ;;

**Struktura podatkov** v FJ = kartezijski product tipov, ki sestavljajo produkt (pari, tuples) ;; **λ-abstrakcija:** za vezanje imena in vrednosti - let x = N in M => (**λx.M)N**

**Funkcija** v OCamlu je **λ-abstrakcija** ->x je **parameter** M je **telo -** fun x -> M => **λx.M ;** fun x = x\*x => **λx.x\*x**

**Curry** oblika: funkcija ima **en parameter** prednost je v preglednosti

**let** izjava veže vrednost z imenom ; globalna definicija spremenljivke je dostopna povsod ; lokalna definicije je dostopna znotraj bloka

**Obseg spremenljivke** v blocku in podblokih ;; **Življenska doba spremenljivke** je veljavnost spremenljivke

**Linearna rekurzija** je rekurzija v eno smer ;; **Repna rekurzija** jo lahko pretvorimo v iteracijo

**Parametrični polimorfizem** ->parametri so spremenljivke (generičnost)

**Višje funkcije** -> so funkcije ki sprejemajo funkcije kot **parameter** ali pa **vračajo funkcijo** – let compose f g x = f ( g x )

**Prevajalnik** -> Front-end (analiza semantike, analiza sintakse, predobdelava), Back-end (optimizacije, analiza poteka), Cilj (prevajanaje v strojno kodo, bytecode, …)

**Rekonstrukcija vrstic** -> 1. Unicode zapis v “\” zapis 2. TAB zamenjani z presledki 3. zamenjava znakov za konec vrstice 4. odstranitev praznih vrstic

**Leksikalna Analiza** -> koda razdeljena na **SEZNAM ŽETONOV** ;; **PARSER** -> prepozna pravila v **seznamu žetonov**

**Semantična Analiza** -> preverjanje tipov in smiselnosti kode

**Vemsna koda** -> BYTECODE za uporabo na različnih OS, slabost v tem da se prevaja ob zagonu (počasnejši zagon)

**SET** ali **podatkovna množica** je množica z 0 ali več elementi določenega tipa

**Razlike med FUNKCIJSKIMI jeziki**: Program je funkcija, ni važno kako nastane rezultat, λ-račun, rekurzivno naštete funkcije

**In IMPERATIVNIMI jeziki**: program je zaporedje ukazov, ukazi spreminjajo glavni pomnilnik

Če **predrutina** vrnevrednost je **FUNKCIJA** drugače je **POSTOPEK** ;; **SLOVAR** -> KLJUČ in VREDNOST, povezava dveh elem.

**Objektno usmirjen model** -> abstrakcija podatkov, podatki so skriti, videne so metode in lastnosti

**Razred** ->definicija objekta ;; **Agencija v OOP** -> ena izmed dveh abstrakcij => **Agencija** = ‘has-a’ (ima lastnost)

Primer: (UCENEC razred IMA NASLOV razred) **Specifikacija** = ‘is-a’ (je nekaj)

**Specifikacija** -> pod razred; podeduje vse lastnosti super razreda

**Abstraktni tip** -> vpelje se zaradi verjetnosti napak programerja, defin. mnozico moznih vrendosti in operacije na elm. tega tipa

**Subtyping** = relacija na tipih (vrednost enega tip ana mestu drugega ) ;; **Objekt** -> ima variable part (spremenljivke) in fixed part (tabela metod) ;; **Generičnost** = način programiranja, postopki so v smislu tipov

**Parametrizirani razredi** omogočajo uporabo **parametričnega polimorfizma**

**Modul** = omogoča dekompozicijo v več programskih enot, programer lahko dela z prevedeno kodo modula (ni potrebna izvorrna koda), programer mora poznati vmesnik do modulov (vrednosti funkcije, tipe in pod module)

**Vmesnik** – se uporablja za omejitev dostopa do modula, skrije detajle, tipi so pbstraktni

**Vmesnik je podpis, Implementacija je struktura**

**Abstraktni podatkovni tip** = označuje množico abstraktnih struktur, zelo prilagodljiv, ime modula z **veliko,** podpis in struktura ne potrebujeta imenovanja

**Parametriziran modul** = generični modul, temelji na osnovi modulov, ki so podani kot argumenti

**Tipni sistem** = **1. Curry Haskel 2. Alonzo Church**

* implicitne vrste - eksplicitne vrste
* funkcionalni jeziki - stroge anotacije
* opcijske anotacije - preverjanje tipov spremenljivk, izrazov, ...

**Kompatibilnost tipov** = pravila, ki določajo, če lahko uporabimo vrednost z nekim tipom v določenem kontekstu

**Type-checking** = postopek, ki preverja če program spoštuje pravila jezika

**Strongly-typed** = ko jezik ne dovoli aplikaciji, da uporabi neko operacijo na objekt, ki ni namenjen za uporabo te operacije

**Strictly-typed** = ko je jezik strongly-typed in je type-checking med prevajanem

**Type-checking rule based** = 1. Ekvivalenčna pravila 2. Kompatibilna pravila 3. Interferenčna pravila

**Type-interface** = se uporablja za type-checking, je proces določanja tipa izrazov na podlagi znanih tipov, poznamo 2 pristopa