OCaml cheatsheet

Na koncu vsakega ukaza moramo napisati `;;`.

Zmnožek tipov

Urejeni pari:

```
# (1 + 2, "banana") ;;
- : int * string = (3, "banana")
```

Urejene *n*-terice:

```
# (1, "banana", false, 2) ;;
- : int * string * bool * int = (1, "banana", false, 2)
```

Projekciji *first* in *second*:

```
# fst (1, "banana") ;;
- : int = 1
# snd (1, "banana") ;;
- : string = "banana"
```

Definicije vrednosti

Definicija vrednosti z `let`:

```
# let i = 10 + 3 ;;
val i : int = 13

# let j = 100 + i * i ;;
val j : int = 269
```

Definirali smo **nespremenljivo vrednost**, ne spremenljivke! `i` in `j` ne moremo spreminjati. V spodnjem primeru nov `x` prekrije staro definicijo za `x`:

```
# let x = 2 ;;
val x : int = 2
# let x = 3 ;;
val x : int = 3
```

Enotski tip

```
# () ;;
- : unit = ()
```

Zapisi

Da si ne rabimo zapomniti vrstnega reda podatkov v n-tericah, **komponente poimenujemo** in dobimo zapise:

```
# type oseba = { ime : string; priimek : string; } ;;
type oseba = { ime : string; priimek : string; }
# { ime = "Mojca"; priimek = "Pokraculja" } ;;
- : oseba = {ime = "Mojca"; priimek = "Pokraculja"}
```

Zapišemo lahko tudi urejeno *enerico*:

```
# type zajec = { masa : int } ;;
type zajec = { masa : int; }
# { masa = 42 } ;;
- : zajec = {masa = 42}
```

Dostopanje do polj zapisov

Do polja z imenom `foo` v zapisu `s` dostopamo s `s.foo`:

```
# let mati = { ime = "Neza"; priimek = "Cankar" } ;;
val mati : oseba = {ime = "Neza"; priimek = "Cankar"}
# mati.ime ;;
- : string = "Neza"
# mati.priimek ;;
- : string = "Cankar"
```

Do polj lahko dostopamo tudi z vzorci:

```
# let {ime = i; priimek = p} = mati ;;
val i : string = "Neza"
val p : string = "Cankar"

# let {ime = i; priimek = _} = mati ;;
val i : string = "Neza"
```

Polj, ki nas ne zanimajo, v vzorcu ni treba omenjati:

```
# let {ime = i} = mati ;;
val i : string = "Neza"
```

Definicije tipov

S `type a = ...` lahko definiramo zapise:

```
type complex = { re : float; im : float }
type color = { red : float; green : float; blue : float }
```

Definiramo lahko tudi okrajšave za tipe:

```
type krneki = int * bool * string
```

Namesto `int * bool * string` lahko pišemo `krneki`.

Vsota tipov

`izdelek` je vsota treh tipov: prvi tip je zmnožek tipov `barva` in `int`, drugi in tretji tip sta oba `int`. Za oznake smo izbrali `Cevelj`, `Palica` in `Posoda`. Tem oznakam pravimo **konstruktorji**.

```
type barva = { blue : float; green : float;  red : float }

type izdelek =
  | Cevelj of barva * int
  | Palica of int
  | Posoda of int
```

Črn čevelj velikosti 42:

```
Cevelj ({blue=0.0; green=0.0; red=0.0}, 42)
```

Palica velikosti 7:

```
Palica 7
```

Posoda prostornine 7:

```
Posoda 7
```

Včasih želimo uvesti **tip iz končnega števila konstant**:

```
type t = Foo | Bar | Baz | Qux
```

Razločevanje primerov (`match`)

Denimo, da je cena izdelka `z` določena takole:

- čevelj stane 15 evrov, če je številka manjša od 25, sicer stane 20 evrov,
- palica dolžine `x` stane `1 + 2 * x` evrov,
- posoda stane 7 evrov ne glede na prostornino.

V OCaml to zapišemo z `match`:

```
match z with
  | Cevelj (b, v) -> if v < 25 then 15 else 25
  | Palica x -> 1 + 2 * x
  | Posoda y -> 7
```

Splošna oblika `match` stavka:

```
match e with
  | p₁ -> e₁
  | p₂ -> e₂
  | p₃ -> e₃
    ⋮
  | pᵢ -> eᵢ
```

p_1, ..., p_i so **vzorci**. Vrednost izraza `match` je prvi e_j, za katerega `e` zadošča vzorcu p_j.

Vzorci v stavku match so lahko poljubno gnezdeni. Denimo, da bi želeli ceno izračunati takole:

- čevelj stane 15 evrov, če je številka manjša od 25, sicer stane 20 evrov,
- palica dolžine 42 stane 1000 evrov,
- palica dolžine `x ≠ 42` stane `1 + 2 * x` evrov,
- posoda stane 7 evrov ne glede na prostornino.

`match` stavek potem izgleda tako:

```
match z with
  | Cevelj (b, v) -> if v < 35 then 15 else 25
  | Palica 42 -> 1000
  | Palica x -> 1 + 2 * x
  | Posoda y -> 7
```

Tip funkcij

Tip funkcij `a -> b` zajema funkcije, ki sprejmejo argument tipa `a` in vrnejo rezultat tipa `b`. V OCamlu lambda-abstrakcijo `\x.e` zapišemo kot `fun x -> e`:

```
# fun x -> 2 * (x + 3) + 3 ;;
- : int -> int = <fun>
```

Veljajo podobna pravila kot v lambda-računu. Funkcije lahko gnezdimo:

```
# fun x -> (fun y -> 2 * x - y + 3) ;;
- : int -> int -> int = <fun>
```

OCaml izračuna tip funkcije `int -> int -> int`. Operator `->` je **desno asociativen**, `a -> b -> c` je enako `a -> (b -> c)`.

Funkcije lahko tudi uporabljamo:

```
# (fun x -> (fun y -> 2 * x - y + 3)) 10 ;;
- : int -> int = <fun>
# (fun x -> (fun y -> 2 * x - y + 3)) 10 3 ;;
- : int = 20
```

Funkcije lahko poimenujemo:

```
# let f = fun x -> x * x + 1 ;;
val f : int -> int = <fun>
# f 10 ;;
- : int = 101
```

Namesto `let f = fun x -> ...` lahko pišemo `let f x = ...`:

```
# let g x = x * x + 1 ;;
val g : int -> int = <fun>
```

```
# g 10 ;;
- : int = 101
```

Rekurzivne funkcije

Rekurzivno definicijo funkcije **moramo naznaniti z** `let rec`:

```
# let rec fact n = (if n = 0 then 1 else n * fact (n - 1)) ;;
val fact : int -> int = <fun>
# fact 10 ;;
- : int = 3628800
```

Računanje tipov funkcije

OCaml sam izračuna tip funkcije. Včasih ostane kak tip nedoločen:

```
# fun (x, y) -> (y, x) ;;
- : 'a * 'b -> 'b * 'a = <fun>
```

Tip `x` je poljuben, prav tako tip `y`. OCaml ju zapiše z `'a` in `'b`. Apostrof označuje dejstvo, da sta to **poljubna** tipa ali parametra.

Še en primer:

```
# fun (x, y, z) -> (x, y + z, x) ;;
- : 'a * int * int -> 'a * int * 'a = <fun>
```

Ko zapišemo funkcijo, lahko **podamo tip njenih argumentov**:

```
# fun (x : string) -> x ;;
- : string -> string = <fun>
```

Združevanje nizov

Nize v OCamlu združujemo z operatorjem `^`:

```
# let str1 = "hello" ;;
val str1 : string = "hello"
# let str2 = "world" ;;
val str2 : string = "world"
```

```
# str1 ^ str2 ;;
- : string = "helloworld"
```

Seznami

Seznam `intov` - tipa `int list`:

```
# let stevila = [1; 2; 3; 4] ;;
val stevila : int list = [1; 2; 3; 4]
```

Dostopamo do glave in repa seznama:

```
# List.hd stevila ;;
- : int = 1
# List.tl stevila ;;
- : int list = [2; 3; 4]
```

Na vsakem elementu pokličemo funkcijo `string_of_int`:

```
# let nizi = List.map string_of_int stevila ;;
val nizi : string list = ["1"; "2"; "3"; "4"]
```

Kvadriramo vsak element seznama s pomočjo anonimne funkcije:

```
# let nizi = List.map (fun x -> x*x) stevila ;;
val nizi : int list = [1; 4; 9; 16]
```

Reference

Ko v OCamlu definiramo vrednost `x` z `let x = e₁ in e₂` je `x` **nespremenljiva vrednost**. Če želimo spremenljivo vrednost, moramo narediti **referenco**:

- z `ref v` naredimo novo referenco z vrednostjo `v`,
- z `!r` ali `r.contents` dobimo trenutno vrednost reference `r`,
- z `r := v` nastavimo vrednost reference `r`.

```
# let r = ref 5 ;;
val r : int ref = {contents = 5}
# !r ;;
- : int = 5
```

```
# !r + 10 ;;
- : int = 15
```

Vrednost reference nastavimo z `:=`:

```
# r := 8 ;; (* <-- tu smo nastavili vrednost r na 8 *)
- : unit = ()
# !r ;;
- : int = 8
# !r + 10 ;;
- : int = 18
```

Zanke

OCaml ima tudi zanki `while` in `for`:

```
while ⟨pogoj⟩ do
  ...
done

for i = ⟨spodnja-meja⟩ to ⟨zgornja-meja⟩ do
  ...
done
```

Program, ki sešteje prvih 42 števil:

```
let vsota_lihih_42 =
  let v = ref 0 in
  let i = ref 0 in
  while !i < 42 do
    v := !v + (2 * !i + 1) ;
    i := !i + 1
  done ;
  !v
```

Vmesniki

V algebri poznamo algebraične strukture, na primer vektorske prostore, grupe, monoide, kolobarje, Boolove algebre, ... Definicija takih struktur poteka v dveh korakih:

- **signatura** pove, kakšne množice, konstante in operacije imamo,
- **aksiomi** povedo, kakšnim zakonam morajo zadoščati operacije.

V OCamlu lahko podamo poljubno **signaturo** (tipe in vrednost), ne moremo pa zapisati aksiomov, ki jim zadoščajo. Takole zapišemo signaturo za usmerjeni graf:

```
module type DIRECTED_GRAPH =
sig
  type v
  type e
  val src : e -> v
  val trg : e -> v
end
```

Implementacija

Implementacija v OCamlu se imenuje **modul** (angl. module). Modul je skupek definicij tipov in vrednosti, lahko pa vsebuje tudi še nadaljnje podmodule. Primer za usmerjeni graf:

```
module K4 : DIRECTED_GRAPH =
struct

    type v = V0 | V1 | V2 | V3
    type e = E0 | E1 | E2 | E3 | E4 | E5

    let src = function
      | E0 -> V0
      | E1 -> V1
      | E2 -> V2
      | E3 -> V3
      | E4 -> V0
      | E5 -> V1

    let trg = function
      | E0 -> V1
      | E1 -> V2
      | E2 -> V3
      | E3 -> V0
      | E4 -> V2
      | E5 -> V3
end
```

Generično programiranje

V OCamlu je generično programiranje omogočeno s **funktorji**. Funktor `F`, ki sprejme strukturo `A`, ki zadošča signaturi `S`, in vrne strukturo `B` zapišemo takole:

```
module F(A : S) =
struct
```

```
  ⟨definicija strukture B⟩
end
```

Primer preprostega funktorja `Cycle`, ki sprejme strukturo s številom `n` in vrne usmerjeni cikel na `n` vozliščih:

```
(* Takole pa naredimo modul, ki je parametriziran s
   strukturo. Kasneje bomo videli bolj uporabne primere. *)
module Cycle (S : sig val n : int end) : DIRECTED_GRAPH =
struct
  type v = int
  type e = int
  let src k = k
  let trg k = (k + 1) mod S.n
end

module C5 = Cycle(struct let n = 5 end)
module C15 = Cycle(struct let n = 15 end)
```