Programiranje I: 2. izpit

6. februar 2023

Čas reševanja je 120 minut. Veliko uspeha!

1. naloga

a) Napišite funkcijo `sum_two_smallest: int * int * int -> int`, ki sprejme trojico in vrne seštevek dveh najmanjših elementov.

b) Napišite funkcijo `flatten: 'a option option -> 'a option`.

c) Napišite funkcijo `dot_product: int list -> int list -> int`, ki izračuna skalarni produkt dveh vektorjev podanih s seznamom.

d) Napišite funkcijo `smallest_modulo: int list -> int -> int option`, ki vrne tisto vrednost v podanem seznamu, ki da najmanjši ostanek pri deljenju z drugim argumentom. Če je seznam prazen naj funkcija vrne `None`. Za vse točke naj bo funkcija repno rekurzivna.

```
# smalest_modulo [ 3; 4; 5; 6; 8; 9; 10 ] 7;;
- : int option = Some 8
```

e) Napišite funkcijo `target_product: int list -> int * int option`, ki v seznamu najde par elementov, katerih zmnožek je enak dolžini seznama. Element je lahko v paru sam s seboj. Če je rešitev več, vrnite tisto, ki je leksikografsko najmanjša, če pa rešitve ni, potem vrnite `None`.

```
# target_product [ 6; 5; 4; 3; 2; 1 ]
- : int option = Some (1, 6)
```

2. naloga

Logične formule v naslejdnji nalogi bomo predstavili s spodnjimi tipi. Formula je ali atomarna (torej spremenljivka ali konstanta), ali pa je konjunkcija/disjunkcija več logičnih formul.

```
type combinator_operator = And | Or
type atom = Variable of string | Constant of bool

type combinator = {
  operator : combinator_operator;
  children : formula list;
  negated : bool;
}

and formula =
  | Combinator of combinator
  | Leaf of { value : atom; negated : bool }

let example =
  Combinator
    {
      operator = And;
      negated = false;
      children =
        [
          Leaf { value = Variable "x"; negated = true };
          Combinator
            {
              operator = Or;
              negated = true;
              children = [ Leaf { value = Constant true; negated = false } ];
            };
        ];
    }
```

a) Napišite funkcijo `map_shallow: (formula -> formula) -> combinator -> combinator`, ki funkcijo uporabi na vseh direktnih otrocih podanega kombinatorja. Funkcija `map_shallow` ne gre v globino, ampak spremeni samo prvi nivo.

b) Napišite funkcijo `collect_variables: formula -> string list`, ki vrne seznam vseh spremenljivk nastopajočih v formuli. Če se katera spremenljivka pojavi večkrat je vseeno, ali jo vrne enkrat ali večkrat. Za vse točke naj bo funkcija repno rekurzivna.

```
# collect_variables example;;
- : string list = ["x"]
```

c) Napišite funkcijo `update: string -> formula -> formula -> formula`, ki sprejme ime spremnljivke, novo formulo in staro formulo. Funkcija naj vse pojavitve spremenljivke v stari formuli zamenja z novo formulo, kot to naredimo pri matematičnih izrazih. Posebno bodite pozorni, kaj se zgodi pri negacijah. Ker podajanje dvojne negacije v podanem tipu ni najlepše, v primeru dvojne negacije to kar takoj odpravite.

```
# update "x" (Leaf { value = Constant true; negated = true }) example
- : formula =
Combinator
 {operator = And;
  children =
   [Leaf {value = Constant true; negated = false};
    Combinator
     {operator = Or;
      children = [Leaf {value = Constant true; negated = false}];
      negated = true}];
  negated = false}
```

d) Napišite funkcijo `push_negation: formula -> formula`, ki negacije potisne do atomov (in jih v primeru dvojnih negacij odstrani). Po uporabi mora biti nova formla logično ekvivalentna, in za vse podformule oblike `Combinator` mora veljati, da je negated nastavljen na `false`

```
# push_negation example;;
- : formula =
Combinator
  {operator = Or;
  children =
    [Leaf {value = Variable "x"; negated = true};
    Combinator
      {operator = And;
      children = [Leaf {value = Constant true; negated = true}];
      negated = false}];
  negated = false}
```

e) Napišite funkcijo `eval: (string -> bool) -> formula -> bool`, ki sprejme formulo in funkcijo, ki spremenljivkam priredi booleovo vrednost in izračuna končno vrednost formule. Za vse točke naj bo funkcija repno rekurzivna. Funkciji `List.for_all` in `List.exists` sta repno rekurzivni.

```
# eval (fun _ -> true) example;;
- : bool = false
```

3. naloga

Nalogo lahko rešujete v Pythonu ali OCamlu.

Miha med večerjo hodi po dvorani in zbira investitorje za svoj novi projekt. Investitorji stojijo v ravni vrsti drug za drugim. Miha se lahko hrkati pogovarja z natanko enim investitorjem in ima omejeno časa za prepričevanje, zato bi rad ugotovil kako naj čim bolje razporedi svoj čas, da bo pridobil kar največ srestev. Za vsakega investitorja je podan seznam parov pozitivnih celih števil, ki pove, koliko dodatnega denarja bo dobil od investitorja, če ga prepričuje vsaj toliko časa. Če je torej za investitorja podan seznam `[(1, 10), (4, 4)]`, pomeni, da bo za porabljeno eno, dve ali tri enote časa dobil 10 enot denarja, če pa bo porabil 4 ali več enot časa pa dobil 14 enot denarja (najprej 10, potem pa še 4). Vsakič ko konča pogovor z nekim investitorjem porabi še 3 enote časa, da se od njega poslovi (če je med poslavljanjem večerje konec ni s tem nič narobe, saj se lahko poslovi zunaj) Za premik od enega do drugega investitorja porabi toliko časa, kolikor mest narazen stojita. Napišite funkcijo `vecerja(investitorji, cas)`, ki vrne maksimalen znesek, ki ga lahko iztrži. V OCamlu pa ima funkcija signaturo `vecerja -> (int * int) list list -> int -> int`.

```
investitorji = [
  [(1, 10), (3, 15), (4, 2), (8, 20)],
  [(1, 15), (3, 5), (4, 20), (7, 20)],
  [(3, 10), (6, 15), (9, 2), (10, 20)],
]

print(vecerja(investitorji, 15), "$")
```

Izpiše `87 $`.