Programiranje II

Funkcijski jeziki

1. **Kakšen je zgodovinski izvor funkcionalnih jezikov?**
   Temelja sta turingov stroj in lambda račun.
   Matematiki: Turing, Church, Kleene, Post...

---

2. **Razloži koncept vrednosti v funkcijski jezikih. Predstavi možne tipe vrednosti.**
   Atomični tipi: integer, real, boolean, string
   Preprosta vrednost je atomičnega tipa.
   Funkcijski jeziki uporabljajo *vrednosti*
   Imperativni jeziki pa *spremenljivke*

---

3. **Predstavi strukturo seznama v funkcijskih jezikih (list)**
   Seznam je lahko prazen ali pa vsebuje elemente določenega tipa.
   List ima glavo in rep.

---

4. **Predstavi strukturo produktov v funkcijskem jeziku**
   Interpretacija produktnega tipa je kartezijski produkt tipov, ki sestavljajo produkt.
   Primeri produktov so: pari, triples, n-tuples.

---

5. **Razloži, kako se uporablja λ abstrakcija v funkcijskih jezikih.**
   Primer: let izjava: let x = N in M.
   → (λx.M) N
   Uporabljena za vezanje imena in vrednosti.

V Ocamlu: globalna in lokalna deklaracija.

---

6. **Predstavi koncept funkcije v OCaml-u.**
   Funkcija je lambda abstrakcija λx.M.
   x je parameter in M je telo funkcije.

   Funkcija je sestavljena iz parametra x in telesa M.
   `function x -> M`
   `function x -> x*x;;` → λx.x*x

   Telo funkcije je lahko druga funckija.
   `function x -> (function y -> 3*x +y);;`

   Vrednost funkcije je lahko druga funkcija.
   `(function x -> function y -> 3*x + y) 5`

7. **Opiši Curry obliko funkcije. Kakšne so prednosti uporabe te oblike?**
   Arity funkcije je število parametrov.
   Curry oblika: **Funkcije imajo en parameter**

   Ta parameter je lahko tudi tuple ali produkt...
   Prednost uporabe *Curryeve* oblike je ta, da je koda bolj pregledna in jedrnata.

   ```
   let square = function x -> x*x;;
   ```

---

8. **Opiši načina za vezavo med imeno in vrednostjo v bloku OCaml.**
   `let` izjava veže vrednost z imenom. `let x = M in N` = (λ.N) M.

   Vrednost je lahko definirana globalno ali lokalno.
   - globalna definicija: dostopna povsod.
   - lokalna: dostopna samo v lokalnem kontekstu.

---

9. **Predstavi koncept obsega in življenske dobe spremenljivke.**
   Obseg je programsko območje, kjer je opredeljena. Vrednost definirana v določenem blocku, je definirana v vseh pod-blockih...

   Življenska doba vrednosti je veljavnost definicije.

---

10. **Predstavi koncept rekurzije v funkcijskih jezikih.**
    Definicija rekurzije je referenca samo nase.

    Prvič predstavljena v Lisp.

    Linearna rekurzija → rekurzija, ki poteka v eni smeri.
    Primer: `let rec factorial n -> if n=1 then 1 else n * factorial(n-1);;`

    Repna rekurzija → je lahko pretvorjena v iteracijo.

---

11. **Predstavi parametrični polimorfizem.**
    Funkcija ima lahko *več različnih oblik*.
    Tipi parametrov so spremenljivke.
    - 'a, 'b, 'c.

    Generičnost...

---

12. **Višje funkcije. Kako jih uporabljamo in predstavi...**
    Višja funkcija vzame funkcijo kot parameter ali pa vrne funkcijo.
    Primer: Kompozitum: `let compose f g x = f (g x);;`
    Uporaba: map, filter, reduce...

---

13. **Predstavi in razloži ujemanje vzorcev.**
    To je kompleksen case statement.
    Omogoča preprost dostop do komponent kompleksnih podatkovnih struktur.

```
_: wildcard
(*
    Vzorci v match morjo bit istega tipa.
    Vzorci morajo biti linearni. Ena spremenljivka le enkrat.
    Vzorci grejo po vrsti. Od zgoraj navzdol.
    Prvo ujemanje se izrazi.
    Seznam vzorcev mora biti izčrpen.
*)
let isApple x = match x with
    | "Apple" -> true   |
    | ----------------- |
    | "Orange" -> false |
    | _ -> false;;      |
```

---

14. **Kako definiramo polimorfičen tip v OCamlu?**

```
type name = typedef;;
type ('a, 'b) pair = 'a * 'b;;
```

---

15. **Opiši tip unije v OCaml-u.**
    `type figure = Knight | Pawn | ... ;;`

Prevajalniki

1. **Predstavi strukturo prevajalnika.**
   - Front-end:
     - rekostrukcija vrstic
     - leksikalna analiza
     - pred-obdelava
     - analiza sintakse
     - analiza semantike
   - Back-end:
     - analiza poteka
     - vmesna optimizacija
     - generacija kode in optimizacija odvisna od cilja.
   - Cilj:
     - Prevajanje v: strojno kodo, bytecode, runnable file, drugi jeziki...

---

2. **Opiši proces rekonstrukcije vrstic.**
   1. Znaki Unicode v zapis z backslashom \x10D.
   2. Tabulatorji zamenjani s presledki.
   3. Znaki za konec vrstice so zamenjani z `; ...

4. Prazne vrstice so odstranjene.

---

3. **Kaj je leksikalna analiza?**
   Napisana koda je razdeljena v seznam *tokenov* (žetonov):
   - literals: 12345, 0x0, "oi", 'a', true.
   - ključne besede: if, else, for, function, void, return.
   - identifikatorji: varX, $, MyClass.
   - tipi: int, char, 'a, void.
   - operatorji: +, -, *, /.
   - oklepaji...

   Ustvarjeno je zaporedje žetonov, in ločila so odstranjena.
   Regularni izrazi se uporabljajo.
   DKA (deterministični končni automati = Regularni izraz).

---

4. **Kaj je slovnica programskega jezika?**
   Programski jeziki imajo preprosto sintakso, katero lahko ponazorimo z CFG oz. KNS.
   KNS (kontekstno neodvisna slovnica) ima strukturo:

```
G = (V, T, R, S)
V = spremenljivke
T = žetoni (simboli)
R = produkcije
S = začetna spremenljivka
```

- Slovnica nam pove, kateri nizi spadajo v določen jezik.

---

5. **Opiši parser.**
   Parser je algoritem, ki sprejme seznam tokenov in prepozna, katera pravila sestavljajo ta seznam.
   Ustvari **leksično drevo**.
   → root (zgoraj) = začetna točka.
   → vozlišča = pravila.
   → listi = žetoni.

Parse tree (Sintaktično drevo):

---

6. **Opiši naloge semantične analize.**

- preverjanje tipov.
- preverjanje, če koda ima smisel.
- primer: null + null ali int + int.
- semantičnost je 100% postavljena s strani pisatelja jezika.
  Uporablja sintaktično drevo za vhodni podatek.
  Pretrovi **sintaktično drevo** → **abstraktno sintaktično drevo**.
  Obstaja tudi: **Tabela simbolov**.
  Poveže (zmappa) identifikator z njegovim: **tipom**, **strukturo** in **območje**.

---

7. **Opiši vlogo vmesne kode.**
   Pojavi se pri jezikih (ponavadi višje-stopenjskih), ki jo želimo uporabljati na več različnih *operacijskih sistemih*.
   `(bytecode)`
   Prednosti:
   - lahko poganjaš na različnih OS.
   - kodo se lahko pretvori nazaj v izvirno kodo.
   - Lahko nosi dodatne informacije.
   - Lahko se izvede optimizacija, specifična za tisti računalnik.
   Slabosti:
   - Prevajanje ob zagonu → počasnejši zagon.

---

8. **Predstavi načine optimizacije vmesne kode.**

- Eliminacija mrtve / neuporabne kode.
- Alokacija spomina za konstante.
- Skok do skoka zamenjan z enim direktnim skokom.
- inline expansion: inline funkcije so "cenejše" kot normalne.
  JMP → JMP → X &arr; JMP → X
- Enkratni check za indexe pri arrayih.
- Pogosto uporabljene spremenljivke so shranjene v registru.

---

9. **Opiši funkcije back-end-a.**
   Optimalno vmesno kodo prevede v strojno kodo (objektna koda):

- Glede na cilje, lahko prevede v `bytecode` za navidezno napravo.
- Lahko se prevede v drug jezik.
- Lahko se prevede v *assembly code*.

> ⓘ **Imperativni jeziki**

1. **Predstavi razlike med funkcijskim in imperativnim modelom računanja.**
   Razlika v programiranju:

Funkcijski	**Imperativni**
Program je funkcija	Program je zaporedje ukazov
Rezultat se opravi z oceno funkcije.	Rezultat temelji na izvajanju ukazov
Ni važno, kako nastane rezultat.	Ukazi spremenijo vsebino glavnega pomnilnika

Razlika v modelih:

Funkcijski	**Imperativni**

Funkcijski	Imperativni
λ-račun, rekurzivno naštete funkcije	
Abstraktni stroj je	Abstraktni stroj je **Turingov stroj**
	Rezultat dobimo, ko je doseženo končno stanje.

---

2. **Kaj je bila ideja strukturiranega programiranja, razvitega leta 1970?**

- Top-down dizajn. (Od uzgari dol.) Main prva.
- Modularizacija kode.
- Strukturirani tipi (multi-dimenzionalna polja).
- Opisna imena spremenljivk in konstant.
- Komentarji.

---

3. **Opiši koncept spremenljivke v imperativnih jezikih.**
   Spremenljivka je simbolično ime za del spomina, katero lahko uporabljamo za dostop do shranjene vrednosti v tej celici (spominska celica).

   Vsak program ima *tabelo simbolov*, ki ima shranjene vse informacije glede spremenljivk.
   [ime, naslov, velikost]

   Program mora dodeliti spominski prostor preden se uporabi spremenljivka.

---

4. **Opiši koncepte imperativnih jezikov uporabljenih za kontrolo zaporedja.**
   - **Zaporedja:**
     Telemljna abstrakcija za opisovanje algoritmov.
     Zaporedje ukazov spremeni stanje spremenljivk.
   - **Blok:**
     Blok je predstavljen z aktivacijskim zapisom.
     Imperativni jeziki so ponavadi strukturirani z bloki.
     `begin-end | loop | telo funkcije`
     Vsak blok ima:
     - lokalne spremenljivke
     - lokacijo spomina za vrednost, ki jo vrne
   - **Pogoj:**
     Strojna koda uporablja ukaze za pogojne skoke
     [JE, JNE, JA, ...]
   - **Zanka:**
     - `while (x > 0):` Ponavljaj ukaze med / dokler določen pogoj ni sprejet.
     - `for (i in 0..10):`
     - `do-while:` Pogoj je na koncu (vsaj ekrat se izvede).

       Kontrola zanke:
       - skok iz loopa: `break;`

    - skok do pogoja: `continue;`

---

5. **Predstavi koncept procedure in funkcije v imperativnih jezikih.**
   Podrutine so temeljni mehanizem za kontrolo.

- Del programa, ki ima definiran vhod in izhod, je podrutina, postopek ali funkcija.
- Podrutina izvede operacijo na željo klicatelja.
- S pomočjo parametrov se podrutini prenesejo argumenti.
- Če podrutina vrne vrednost je **funkcija** drugače pa je **postopek**.

---

6. **Opiši najbolj pomembne načine prenašanja parametrov.**
   - Podajanje vrednosti
     Parameter je lokalna spremenljivka postopka.
   - Podajanje referenc
     Koda postopka spreminja podano spremenljivko.
     Vse spremembe spremenljivke ostanejo tudi po koncu postopka.

---

7. **Predstavi strukturo podatka (record) in njeno uporabo v IPL.**
   To je produkt imenovanih komponent.

```
    type name = { name1 : t1; ... ; nameN : tN };;
```

```
struct complex {
    double rp;
    double ip;
}
```

---

8. **Predstavi koncept pointerja.**
   Pointer je referenca na objekt v spominu.
   Operacije z pointerji:
   - alokacija in delokacija objektov na kopici (heap).
   - dodelitev enega kazalca drugemo.
     Rekurziven tip v OCamlu:

```
    type ctree = Empty | Node of char * ctree;;
```

---

9. **Predstavi strukturo polja in njenega namena v IPL.**
   Polja so podatkovne strukture, ki vsebujejo določeno število podatkov, ki so določenega tipa.
   Polje je po definiciji spremenljivo, vendar je velikost fiksna.
   Dostop do elementov je ponavadi preko indexov.

---

10. **Opiši strukturo podatkovne množice.**
    Set (podatkovna množica) ima 0 ali več elementov določenega tipa.

---

11. **Opiši strukturo slovarja.**
    Slovar ima **KLJUČ** in **VREDNOST**, katera dva elementa sta povezana.
    Ta strukturja je zelo popularna.

Upravljanje spomina

1. **Opiši koncept *časa vezave*.**
   Vezava je povezava dveh stvari: ime in stvar, ki jo imenuje.

   Čas vezave je čas, ki preteče, preden se ustvari vezava.
   Čas:
   - krajši = hitrejši
   - daljši = fleksibilen

Statičen = pred zagonom.
Dinamičen = med tekom.

---

2. **Opiši življensko dobo objektov in povezav.**
   To je čas, ki preteče med nastankom in uničenjem te povezave.

   Čas, ki preteče med nastankom in uničenjem objekta, je **življenska doba objekta**.

---

3. **Predstavi pristope dodelitve pomnilnika v programskih jezikih.**
   - Statični objekti (static object) → dobijo naslov, ki ga obdržijo tekom izvajanja.
   - Skladovni objekti (stack object) → LIFO (last in first out)
   - Heap objects → so lahko dodeljeni kadarkoli, vendar je za to potreben bolšje upravljanje s spominom.

---

4. **Predstavi statično dodelitev spomina. Kaj je statičen aktivacijski zapis?**
   Globalne spremenljivke so statični objekti.
   - Enostavno in hitro.
   - Alociraj en aktivacijski zapis za eno funkcijo.
   - Statična alokacija.
   - Starejši dialekti Fortrana uporabljajo ta sistem.

     Slabe lastnosti:
   - Nemogoče je uporabljati rekurzijo.
   - Multi-nitenje ne more biti implementirano.

   Statičen aktivacijski zapis vsebuje:
   - lokalne spremenljivke + začasne vrednosti
   - argumente podrutin.
   - naslov *?* vrnitve *?*
   - vrednost, ki jo vrne.
   - referenco na aktivacijski zapis klicatelja.

---

5. **Opiši skladovno alokacijo prostora za podrutine**
   Aktivacijski zapis je alociran ko je blok ali podrutina aktivirana.
   Upravljanje s skladom je pod okriljem subrutine.
   TODO

---

6. **Predstavi alokacijo na kopico. Kateri problemi obstajajo v povezavi s tako alokacijo?**
   Pri kopici so lahko pod-bloki alocirani in delocirani kadarkoli.
   Kopica je potrebna za dinamično alokacijo povezanih podatkovnih struktur kot so nizi, seznami, seti. Oz. tisto, čemur se velikost lahko spremeni.
   Problemi:
   - počasnejši dostop
   - ni garantirano, da bo prostor porabljen z najboljšo efektivnostjo.
     TODO

---

7. **Katere strategije uporabljamo, pri manipuliranju s spominom v kopici?**
   Kopica je `linked list` .
   Obstaja čiščenje spomina in re-alokacija spomina.
   - First fit:
     Free list je skeniran, dokler se ne najde dovolj velik blok.
     Če je blok prevelik, je splittan in se uporabi samo potrebna velikost.
   - Best fit:
     Poišče najbolj ustrezen blok.

---

8. **Opiši ekplicitno upravljanje pomnilnika.**
   v C in C++ imamo `malloc` in `free` , s katero lahko razpolagamo in dinamično upravljamo s pomnilnikom.
   Program alocira bloke spomina in ima popolno kontrolo nad njimi.

   Problemi:
   - če se pointer "izgubi", imamo `memory leak` .
   - če je objekt pomotoma prevzet, imamo `viseči pointer` .

     Prevzemi so ponavadi avtomatični. (lokalne spremenljivke funkcije)

---

9. **Opiši avtomatično upravljanje s spominom. Zakaj garbage collection?**
   Funkcijski jeziki imajo skoraj vedno GC.
   Alokacija objektov je vedno sprožena ob določeni operaciji: **[nov objekt, dodajanje seznamu, ...]**

Delokacija je lahko:

Eksplicitna	**Implicitna**
C, C++, Pascal	Jezik mora priskrbeti mehanizem, za identifikacijo nedosegljivih objektov in jih odstraniti.

Zakaj **JA** GC?

- Napake pri delokaciji spomina so ene izmed najbolj pogoste in najdražje napake, pri programih.

Zakaj **NE** GC?

- Počasnejše.
- Ni lahka implementacija.

---

10. **Primerjaj avtomatično in eksplicitno čiščenje spomina.**
    Avtomatično čiščenje je zelo pomembno.
    V modernih sistemih, je "cena" avtomatičnega čiščenja kompenzirana z močnejšo opremo.

---

11. **Opiši GC z štetjem referenc.**
    Objekt ni več uporaben, ko ni več pointerjev, ki kažejo na ta objekt.

    Števnik pointerjev je v objektu, in je nastavljen na 1.
    Ko je število 0, lahko objekt prevzamemo.
    PROBLEM:
    Objekt je lahko neuporaben, čeprav obstajajo reference, ki kažejo nanj.
    (cirkularna struktura je problem).

---

12. **Opiši `mark-and-sweep` mehanizem za GC. Predstavi mogoče izboljšave.**
    1. Collector gre čez vsak blok v kopici, in ga označi kot useless.
    2. Začetno z pointerji zunaj kopice, gre rekurzivno čez vse povezane podatkovne strukture, in vsak na novo "najden" blok označi kot "uporaben".
    3. Vse, ki so "useless", počisti.
       TODO

Objektno-usmerjeni jeziki

1. **Opiši objektno-usmerjeni model.**
   Namen objektno-usemerjenega modela je abstrakcija podatkov.

Podatki implementacije so skriti in vse interakcije z objektom so preko njegovih funkcij.
Objekt ima:

- metode
- lastnosti

---

2. **Opiši definicijo razreda in nastanka primerka (instance).**
   Razred je definicija objekta, ki definira lastnosti in obnašanje. Obnašanje razreda je implementirana z metodami. Nove instance so lahko narejene v že obstoječih razredih.
   - možnost dedovanja

   Večina jezikov ima operator `new()` .
   - Ocaml nima konstruktorjev
   - parametri so dosegljivi kot spremenljivke
   - Razred se obnaša kot generator → funkcija, ki generira objekte.

   Privatne metode in spremenljivke so dosegljive samo v tem razredu.

---

3. **Opiši agregacijo in njeno implementacijo v OOP.**
   Agregacija je ena izmed dveh abstrakcij, uporabljenih v OO.
   - **Agregacija** = 'has-a' (ima lastnost).
   - **Specializacija** = 'is-a' (je nekaj).

   Objekt je sestavljen iz drugih objektov.
   Primer: (Učenec $_{\text{razred}}$ ima naslov $_{\text{razred}}$).

---

4. **Opiši specializacijo in njeno implementacijo v OOP.**
   Pod-razred podeduje vse lastnosti super-razreda.
   Pod-razred je specializacija super-razreda.

---

5. **Opiši koncept method-overriding-a.**
   Deklaracija metode, ki že obstaja v starševskem razredu, velja za nadomestitev že obstoječe metode.
   Tako lahko pod-razred implementira funkcijo na način, ki njemu ustreza.

---

6. **Opiši koncept dedovanja večih stvari hkrati, in problemov, ki so navzoči.**
   En objekt deduje več stvari.
   Problem:
   - **Diamantni problem** → problemi pri poimenovanju, in več kopij določenega starša.

---

7. **Kaj je abstraktni tip? Podaj primer.**
   To je tip, ki se vpelje zaradi velike nevarnosti napak s strani programerja.
   Je model, ki poleg množice možnih vrednosti definira tudi vse operacije, ki so dovoljene na elementih tega tipa.

---

8. **Predstavi koncept subtyping-a in subsitivity-a**
   Sub-typing je relacija na tipih, ki omogoča da je vrednost enega tipa uporabljena na mestu drugega.

TODO

---

9. **Predstavi dinamično vezavo in subsumption...**

TODO

---

10. **Opiši implementacijo razredov in objektov.**
    Vsak objekt je sestavljen iz dveh delov:
    - Variable part: vsebuje spremenljivke.
    - Fixiran del: tabela metod in je enak za vse instance.

---

11. **Opiši koncept abstraktnega razreda.**
    Abstraktni razred vsebuje virtualne metode. Njim se definira samo podpis.
    Razred je abstrakten, če vsebuje eno virtualno metodo.
    Ko pod-razred implementira virtualno metodo, postane metoda "realna", ali pa je tudi pod-razred abstrakten.

---

12. **Kaj je generičnost?**
    To je način programiranja, pri katerem so postopki napisani v smislu tipov, ki bodo natančno določeni pozneje, in so instancirani šele, ko jih potrebujemo za konkretne tipe.

    Parametrizirani razredi omogocajo uporabo ˇ parametricnega polimorfizma ˇ . Kot pri deklaracijah tipov so lahko tudi razredi parametrizirani s spremenljivami tipov.

Moduli

1. **Opiši koncept modula.**
   Modularna zasnova programov omogoča dekompozicijo v več programskih enot, ki jih lahko razvijamo samostojno od preostalega dela sistema.

   Programer lahko dela z prevedo kodo modula. Ni potrebna izvirna koda.

   Programer mora poznati vmesnik do modulov, ki vsebuje vrednosti, funkcije tipe in pod-module, ki jih modul nudi uporabnikom.

---

2. **Predstavi module kot enote prevajanja.**
   Lahko so predstavljeni kot ena ali več datotek.
   Naprimer v ocamlu imamo modul in vmesnik, lahko pa uporabljamo modul tudi brez vmesnika. Tako je tudi v C.

   Vmesnik se uporablja za to, da se omeji dostop do modula.
   Vmesnik = podpis
   Implementacija modula = struktura.

---

3. **Opiši koncept abstraktnega podatkovnega tipa in njegovih relacij z moduli.**
   Ideja abstraktnega podatkovnega tipa je ta, da označuje določeno množico abstraktnih struktur.
   Reče se mu abstrakten, ker ponuja svojo implementacijo, in je zato prilagodljiv.

---

4. **Opiši koncept vmesnika in implentacije modulov v OCaml-u.**
   Vmesnik omejuje dostop do modula.

   Vmesnik skrije implementacijske detajle
   Vse kar programer rabi vedeti, je definirano v vmesniku.
   V vmesniku so tipi abstraktni.

---

5. **Predstavi zgradbo modularnega jezika v Ocaml-u.**
   Vmesnik = podpis.
   Implementacija = struktura.
   - ime modula z VELIKO.
   - podpis in struktura ne potrebujeta imenovanja.

---

6. **Kako doseči skrivanje informacij / omejevanje dostopa do modulov? **
   To dosežemo z vmesnikom, kjer lahko skrijem kodo implementacije in omejimo druge informacije.

---

7. **Opiši parametirizirane module (funktorje). Dej primer.**
   To so generični moduli, ki temeljijo na osnovi modulov, ki so podani kot argumenti.
   TODO

Tipi

1. **Predstavite tipni sistem programskega jezika**
   - Curry Haskell:
     - implicitne vrste (ocaml, haskell, ml)
     - funkcionalni jeziki

    - opcijske anotacije
    - anotacije so podane, kjer so potrebne
    - tipi izhajajo iz izrazov
- Alonzo Church:
    - eksplicitne vrste
    - stroge anotacije
    - implementacije jezikov vključujejo preverjanje tipov spremenljivk, izrazov itd...

---

2. **Razloži pravila ekvivalence tipov.**
   Ta pravila določijo, če in kdaj se dva tipa različnih vrednosti ujemata.

---

3. **Razloži pravila kompatibilnosti tipov.**
   Ta pravila določijo, če je lahko vrednost s tem tipom uporabljena v določenem kontekstu.

---

4. **Kaj je type-checking?**
   Preverjanje tipa je postopek, ki zagotavlja, da program spoštuje pravila jezika.

   Jezik je **strongly typed**, ko aplikaciji ne dovoli kakršnih koli operacij na objekt, ki ni namenjen za uporabo te operacije.

   Jezik je **statically typed**, ko je **strongly typed** in je type checking lahko med prevajanjem.

---

5. **Opiši type checking rule based.**
    - ekvivalenčna pravila.
    - kompatibilnostna pravila.
    - interferenčna pravila.

---

6. **Opiši type inference.**
   Uporablja se za type-checking.
   To je proces določanja tipa izrazov na podlagi znanih tipov.

   Poznamo 2 pristopa:
    - temelji na typing-rules (pišemo tip)
    - prebere tip (`var x = 40`)