Miksi tyypitystä?

- Keino saada joukko-opista konsistentti.
- Kertoo tietokoneelle tuleeko dataa käsitellä esim. kokonaislukuna, liukulukuna, merkkijonona, tms.
- Helpottaa ohjelmointia monin eri tavoin.

Peruskäsitteitä

- Vahva tyypitys: Järjestelmä havaitsee jos dataa yritetään käyttää tavalla joka ei ole sen tyypin mukaista.
  Esim. yritetään vähentää toisistaan kahta merkkijonoa.
- Heikko tyypitys: Järjestelmä ei havaitse ...
  Lopputulos usein määrittelemätön.

```
char *a = "foo", *b = "bar";
return *((int *) a) - *((int *) b);

union { int i; char *s; } u;
u.s = "foo";
return u.i;
```

- Staattinen tyypitys: Järjestelmä havaitsee *käännösaikana* jos dataa yritetään käyttää tavalla, joka ei ole sen tyypin mukaista.
- Dynaaminen (latentti) tyypitys: Virhe havaitaan (jos havaitaan) suoritusaikana.
  Esimerkkejä: Lispit, muutamat vanhat funktionaaliset kielet, SmallTalk, monet skripti- ja laajennuskielet.

- Yleensä staattisesti tyypitetty ohjelmointikieli on nopeampi suorittaa ja turvallisempi ohjelmoida, dynaamisesti tyypitetty yleensä joustavampi.

- Tyypintarkistus (type checking): Kääntäjän se osa/vaihe, joka tarkistaa ohjelman tyyppien korrektin käytön.
- Tyyppipäättely (type inference): Kääntäjän se osa/vaihe, joka päättelee ohjelman (kunkin lausekkeen) yleisimmän mahdollisen tyypin.

```
# fun x -> x + 1;;
- : int -> int = <fun>
# fun x -> x = 0;;
- : int -> bool = <fun>
# fun x -> x;;
- : 'a -> 'a = <fun>
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let rec length l = match l with
      [] -> 0
    | _ :: t -> 1 + length t;;
val length : 'a list -> int = <fun>
```

  Tässä muuttujat `'a`, `'b` ja `'c`, eli α, β, ja γ tarkoittavat mitä tahansa tyyppiä.

- Tyyppipäättely tuottaa usein yleisempiä tyyppejä kuin mitä ohjelmoija alunperin ajatteli, mikä usein lisää koodin uudelleenkäytettävyyttä ja evolvoitavuutta. Koodi on yleensä tiiviimpää.

  Toisaalta tyyppipäättely virheilmoituksia voi joskus olla vaikea ymmärtää ja joissakin tapauksissa päätellyt tyypit voivat olla varsin vaikeaselkoisia. Lisäksi esim. ML:ssä käytettävä Hindley-Milner –tyyppipäättely on vain puoliratkeava (semidecidable).

Tyypillisiä tyyppejä

- Perustyypit, esim. `Int`, `String`, `Real`, `Bool`, `File`, jne.
- Luetellut tyypit (enumerated types).
- Listat, esim. `[Int]`, taulukot, joukot jne.
- Monikko, l. tuppeli, l. karteesinen tulo, esim. `<Int, String>`. Monikon osien järjestyksellä on merkitystä.
- Tietue (record, struct), esim. `<id:Int, name:String>`. Kenttien järjestyksellä ei ole väliä.
- Luokat ja oliot. Niistä paljon lisää ensi kerralla.
- Variantit tyypit, esim.

```
  Person <Int, String>
| Animal <Int, Owner>
| Car <Owner>
```

  Varianttien järjestyksellä ei ole väliä.

  Pascalin tietueiden vaihtelevat osat ovat eräs esimerkki varianteista tyypeistä. Esimerkki Korpela, Larmela, Planman, "Pascal-ohjelmointikieli", s. 185:

```
TYPE kuvionlaji = (suorakulmio, ympyra, kolmio);
     kuvio = RECORD
               keskipiste : RECORD x, y : Real END;
               CASE laji : kuvionlaji OF
                 suorakulmio : (leveys, korkeus : Real);
                 ympyra      : (sade : Real);
                 kolmio      : (kanta, kork : Real);
             END
```

  C:n unionit ovat turvattomia variantteja tyyppejä.

  (Nolangissa ei tulle olemaan variantteja tyyppejä.)

- Funktiot, esim. `Int`$\rightarrow$`Int`, `Int`$\rightarrow$`Bool`, $\alpha \rightarrow \alpha$,
  $(\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \beta$.

  Funktiotyyppien esittämisessä, ja joskus funktioiden toteuttamisessa käytetään usein ns. *curryingi*ä: n-argumenttinen funktio käsitellään kuin yksiargumenttinen funktio, joka palauttaa yksiargumenttisen funktion, joka $\ldots$ n kertaa $\ldots$, joka palauttaa lopullisen paluuarvon. Eli
  $f(x : \alpha, y : \beta, z : \gamma) : \delta \ =$
  $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta \ =$
  $\alpha \rightarrow (\beta \rightarrow (\gamma \rightarrow \delta))$

  Esimerkki

  ```
  # let compose f g x = f (g x);;
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  # let id x = x;;
  val id : 'a -> 'a = <fun>
  # compose id;;
  - : ('_a -> '_b) -> '_a -> '_b = <fun>
  # compose id id;;
  - : '_a -> '_a = <fun>
  # compose id id 1;;
  - : int = 1
  ```

- Monet rakenteiset tyypit voivat olla *rekursiivisia*, esim. binääripuun tyyppi
  `BinTree = <item:Item, left: BinTree, right: BinTree>`.

Alityypitys

- Tyyppi S on tyypin T *alityyppi*, $S \leq T$, joss tyyppiä S oleva arvo voi esiintyä missä tahansa missä tyyppiä T oleva arvokin voi (subsumption).

  (Muita kirjallisuudessa käytettyjä symboleja alityyppisuhteelle ovat $<$, $\preceq$, $\rhd$, $\lhd$, $\subset$, $\subseteq$, $\sqsubset$ ja $\sqsubseteq$.)

- Alityyppisuhde määritellään induktiivisesti kahden tyypin rakenteen yli *esimerkiksi* näin:

  1. Mikä tahansa tyyppi on alityyppisuhteessa itsensä kanssa, eli $S \leq S$.
  2. Perustyypeillä saattaa olla alityyppisuhteita, esim. `String` $\leq$ `Char`.
  3. Listoille `[S]` $\leq$ `[T]` joss $S \leq T$.
  4. Monikoille $<S_1, \ldots, S_n> \leq <T_1, \ldots, T_m>$ joss $m \leq n$ ja $\forall i \leq m : S_i \leq T_i$.
  5. Tietueille $<f_1{:}S_1, \ldots, f_n{:}S_n> \leq <f_1{:}T_1, \ldots, f_m{:}T_m>$ joss $m \leq n$ ja $\forall i \leq m : S_i \leq T_i$.
  6. Varianteille tyypeille $(l_1\ S_1 \mid \ldots \mid l_n\ S_n) \leq (l_1\ T_1 \mid \ldots \mid l_m\ T_m)$ joss $m \leq n$ ja $\forall i \leq m : S_i \leq T_i$.
  7. Funktioille $S_a \to S_r \leq T_a \to T_r$ joss $S_r \leq T_r$ ja $T_a \leq S_a$ **(HUOM!)**

- Alityyppisuhde on transitiivinen.

- $S \leq T$ ja $T \leq S$ joss $S = T$.

- Jos tyypeille on annettu nimiä, niin päättyykö alityyppivertailu tyypinnimiin ja niistä tunnettuihin piirteisiin, vai jatkaako alityyppivertailu edelleen tyyppien rakenteisiin? Esim.

  ```
  struct S { int x; int y; };
  struct T { int x; int y; };
  ```

  Päteekö tässä $\mathtt{S} \leq \mathtt{T}$?

- Määritelty alityypitys (declared subtyping): $\mathtt{S} \not\leq \mathtt{T}$, koska missään ei ole toisin sanottu.
- Rakenteellinen alityypitys (structural subtyping): $\mathtt{S} \leq \mathtt{T}$, nimistä välittämättä säännön 5 mukaan.

- Vastaavasti tyyppien samuutta kysyttäessä, $\mathtt{S} = \mathtt{T}$?
- Nimiekvivalenssi (esim. Pascal): $\mathtt{S} \neq \mathtt{T}$, koska niillä on eri nimi.
- Rakenteellinen ekvivalenssi (esim. C, joskin kenttien järjestyksellä *on* C:ssä väliä): $\mathtt{S} = \mathtt{T}$.

Varianssi

- Kielen tyyppijärjestelmän kaikki sallitut tyypit voidaan ilmaista yleensä varsin yksinkertaisilla kieliopeilla, esim.

  type: basic_type | constructed_type | type_variable
  basic_type: `Int` | `String` | ...
  constructed_type: type → type | ...

- Tyyppimuuttuja τ esiintyy tyyppikaavassa $T(\tau)$ *invariantissa* positiossa joss $\forall\sigma \leq \tau \Rightarrow T(\sigma) = T(\tau)$.
- ... *kovariantissa* positiossa joss $\forall\sigma \leq \tau \Rightarrow T(\sigma) \leq T(\tau)$.
- ... *kontravariantissa* positiossa joss $\forall\sigma \leq \tau \Rightarrow T(\sigma) \geq T(\tau)$.
- ... *bivariantissa* positiossa joss $\forall\sigma \leq \tau \Rightarrow T(\sigma) \neq T(\tau)$.
- Esimerkki:

`Int`→`Int`	invariantti
`Int`→ τ	kovariantti
τ →`Int`	kontravariantti
$\tau \to \tau$	bivariantti

(Jos tyyppimuuttujan kvantifikaatio ei välttämättä ole universaali tai jos se on tyyppitarkistuksen tai tyyppipäättelyn sisäinen muuttuja, se on tapana valita aakkosiston loppupäästä, esim. τ ja σ.)

Polymorfia

- Ylikuormitus (overloading): Funktion (tai operaattorin) nimi toteuttaa itseasiassa monta eri koodin sisältävää aliohjelmaa, joista valitaan se, joka sopii kutsun argumenttien tyyppeihin.

  Esim. C++

  ```
  void print(int i) { printf("%d", i); }
  void print(char *s) { printf("%s", s); }
  ```

- Koersio (coersion): Argumentin tyyppi muutetaan funktion (tai operaattorin) argumenttilistaa vastaavaksi.
- Ylikuormitus ja koersio ovat kaksi *ad-hoc*-polymorfian lajia. Molemmissa tuotettu ohjelmakoodi riippuu funktion tai sen parametrien tyypeistä.

- Parametrisessa polymorfiassa funktio toimii täsmälleen samoin argumentin tyypistä riippumatta ja se käännetään vain kerran.
- Myös tyyppi voi olla polymorfinen, esim. `[α]` on lista, jossa on α-tyyppisiä alkioita, ja
  ```
  BinTree(α) = <item:α, left:BinTree(α), right:BinTree(α)>
  ```
  on vastaava binääripuu.
- Jos kielessä on alityypitys, niin parametrinen polymorfia (voi?) tarkoittaa sisällyttävää polymorfiaa (inclusion polymorphism), jossa esim. lista `[α]` sisältää joko tyypin α tai sen alityypin alkioita samanaikaisesti.
- Rajoitettu polymorfia (bounded polymorphism) tarkoittaa, että tyypin määrittelyssä sanotaan, että tyyppimuuttujan pitää olla korkeintaan jotain tyyppiä, esim.
  ```
  BinTree(α ≤Ordered) =
    <item:α, left:BinTree(α), right:BinTree(α)>
  ```

Pohdittavaa

- Listojen loppua ilmaisee yleensä jokin erityinen alkio, esim. `[]`. Mikä on sen tyyppi?
- Osaatko löytää funktion, jonka tyyppi on $\alpha \to \beta$?
- Mikä on τ:n position varianssi tyyppikaavassa ($\tau \to$`Int`) $\to$`Int` ?
- Kehitä esitettyjä alityyppisääntöjä vastaava tyyppikielioppi, jossa tyyppimuuttuja voi esiintyä vain kovarianteissa positioissa.
- Olkoon tyyppikaavojen T(τ) ja S(τ) varianssit tunnetut τ:n suhteen. Mitkä ovat tyyppikaavojen <T(τ),S(τ)>, T(τ) $\to$ S(τ), ja T(S(τ)) varianssit τ:n suhteen?
- Onko esimerkiksi C:n yhteenlasku `1 + 2.0` esimerkki koersiosta vai ylikuormituksesta?
- Ovatko C++:n templatet parametrista vai ad-hoc –polymorfiaa?