Milleks tüübid?

- Mida teeb järgmine programmijupp?

$$
\begin{aligned}
x_1 \quad &:= \quad \texttt{"}\textit{Pii siinus on}: \ \texttt{"}; \\
x_2 \quad &:= \quad 3.1415926; \\
&\ldots \\
&\texttt{print}\, x_2; \\
&\texttt{print}\,(\texttt{sin}(x_1));
\end{aligned}
$$

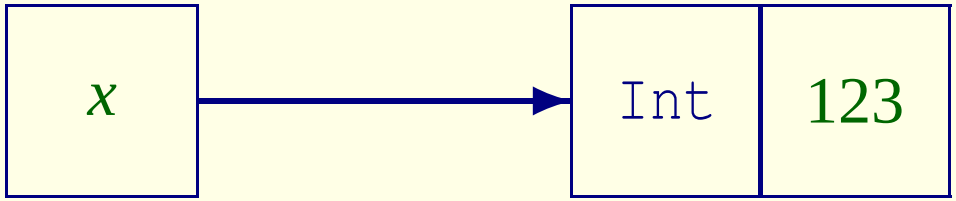
- Ei tea (loodetavasti siiski mitte midagi väga hullu :-)

Milleks tüübid?

- Peaeesmärk on vältida selliste vigade tekkimist
- Idee:
  - Seome väärtustega/muutujatega/… tüübid
  - Kontrollime, kas tüübid klapivad
- Sõltuvalt tüübikontrolli toimumise ajast jaotatakse keeled:
  - dünaamiliselt tüübitud keeled
  - staatiliselt tüübitud keeled
- Tüüpimata = täpselt üks tüüp!

Dünaamiline tüüpimine

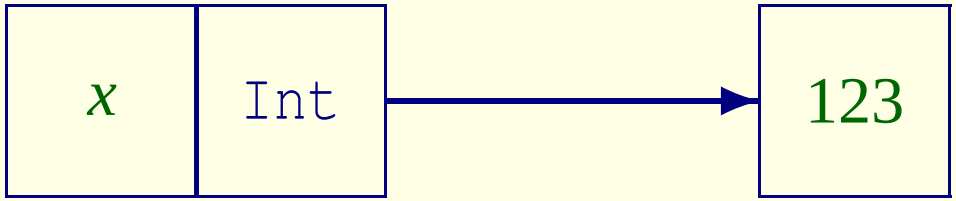
- Tüübikontroll toimub programmi täitmise ajal



- Scheme, Lisp, APL, AWK, Tcl/Tk, Perl, . . .
- Väga paindlik (++)
- Ebaefektiivne (−)
- Tüübivigade leidmine raske (—)

Staatiline tüüpimine

- Tüübikontroll toimub programmi transleerimise ajal



- Fortran, Cobol, Algol, Pascal, C, Java, . . .
- Tüübivigade leidmine lihtne (++)
- Efektiivne (+)
- Tugevalt (?!) kitsendav (—)

Staatiline tüüpimine

- Tüübikontroll on range, kui korrektselt tüübitud programm ei saa anda tüübivigu
  - Pascal vs. C
- Paljud (enamus ?!) staatiliselt tüübitud keeled nõuavad muutujate/protseduuride/. . . tüüpide deklareerimist
- Mõned keeled (ML, Haskell, . . .) kasutavad tüüpide tuletamist (ingl. *type inference*)

Lihtne tüübitud keel

- Süntaks:

⟨*exp*⟩	::=	⟨*integer-literal*⟩ | *true* | *false*	`lit (datum)`
	|	⟨*varref*⟩	`varref (var)`
	|	⟨*operator*⟩ ⟨*operands*⟩	`app (rator rands)`
	|	`if` ⟨*exp*⟩ `then` ⟨*exp*⟩ `else` ⟨*exp*⟩	`if (test-exp then-exp else-exp)`
	|	`proc` ⟨*varlist*⟩ ⟨*exp*⟩	`proc (formals body)`
	|	`let` ⟨*decls*⟩ `in` ⟨*exp*⟩	`let (decls body)`
	|	`letrec` ⟨*decls*⟩ `in` ⟨*exp*⟩	`letrec (decls body)`
	|	`assert` ⟨*type*⟩ `:` ⟨*exp*⟩	`assert (type exp)`

Lihtne tüübitud keel

- Süntaks (järg):

$$
\begin{array}{rcll}
\langle \textit{type-list} \rangle & ::= & \texttt{()} & \\
 & | & \texttt{(} \langle \textit{type} \rangle \; \{\texttt{,} \langle \textit{type} \rangle\}^* \texttt{)} & \\
\langle \textit{type} \rangle & ::= & \langle \textit{prim-type} \rangle & \\
 & | & \langle \textit{arrow-type} \rangle & \texttt{tcons (name types)} \\
\langle \textit{prim-type} \rangle & ::= & \textit{int} \mid \textit{bool} & \\
\langle \textit{arrow-type} \rangle & ::= & \texttt{(->} \langle \textit{type-list} \rangle \; \langle \textit{type} \rangle \texttt{)} &
\end{array}
$$

Tüübikontroll

- Näited:

```
--> "let decr = assert (-> (int) int) : proc (n) +(n,-1);
       comp = assert (-> ((-> (int) bool), (-> (int) int))
                         (-> (int) bool)) :
                  proc (f, g) assert (-> (int) bool) :
                                 proc (n) f(g(n))
    in let isone = comp(zero, decr)
       in isone(2)"
false : bool
--> "letrec fac = assert (-> (int) int) :
              proc (n) if zero(n) then 1 else *(n,fac(-(n,1)))
    in fac(5)"
120 : int
```

Tüübikontroll

Abifunktsioonid:

```
(define domains (compose cdr tcons->types))
(define range (compose car tcons->types))

(define integer-type (make-tcons 'int '()))
(define boolean-type (make-tcons 'bool '()))

(define type-of-datum
  (lambda (datum)
    (if (integer? datum) integer-type boolean-type)))
```

Tüübikontroll

Abifunktsioonid (järg):

```
(define proc-type?
  (lambda (type)
    (and (tcons? type) (eq? '-> (tcons->name type)))))

(define make-proc-type
  (lambda (domain-types range-type)
    (make-tcons '-> (cons range-type domain-types))))
```

Tüübikontroll

- Tüüpide võrdlemine:

```
(define match-types
  (letrec
    ((same-type?
       (lambda (type1 type2)
         (let ((types1 (tcons->types type1))
               (types2 (tcons->types type2)))
           (and (eq? (tcons->name type1) (tcons->name type2))
             (= (length types1) (length types2))
             (andmap2 same-type? types1 types2))))))
    (lambda (type1 type2)
      (if (not (same-type? type1 type2))
        (type-error "Incompatible types:" type1 type2)))))
```

Tüübikontroll

- Tüüpide võrdlemine (järg):

```
(define andmap2
  (lambda (predicate list1 list2)
    (or (null? list1)
      (and (predicate (car list1) (car list2))
        (andmap2 predicate (cdr list1) (cdr list2))))))

(define match-types-pairwise
  (lambda (types1 types2)
    (for-each match-types types1 types2)))
```

Tüübikontroll

- Avaldise tüübikontroll:

```
(define type-of-exp (lambda (exp tenv)
  (let ((answer (variant-case exp
                  (lit (datum) (type-of-datum datum))
                  (varref (var) (apply-tenv tenv var))
                  (if (test-exp then-exp else-exp) ...)
                  (app (rator rands) ...)
                  (let (decls body) ...)
                  (letrec (decls body) ...)
                  (assert (type exp) ...)
                  (proc (formals body) ...)
                  (else (error "Invalid syntax:" exp)))))
    answer)))
```

Tüübikontroll

- Tingimusavaldise tüübikontroll:

```
(if (test-exp then-exp else-exp)
  (let ((test-exp-type (type-of-exp test-exp tenv))
        (then-exp-type (type-of-exp then-exp tenv))
        (else-exp-type (type-of-exp else-exp tenv)))
    (match-types test-exp-type boolean-type)
    (match-types then-exp-type else-exp-type)
    then-exp-type))
```

Tüübikontroll

- Aplikatsiooni tüübikontroll:

```
(app (rator rands)
  (type-of-app (type-of-exp rator tenv)
               (types-of-rands rands tenv)))
```

```
(define types-of-rands (lambda (rands tenv)
  (map (lambda (rand) (type-of-exp rand tenv)) rands)))
```

```
(define type-of-app (lambda (rator-type rand-types)
  (if (proc-type? rator-type)
      (match-types-pairwise (domains rator-type) rand-types)
      (type-error "Not a procedure type:" rator-type))
  (range rator-type)))
```

Tüübikontroll

- Let-avaldiste tüübikontroll:

```
(let (decls body)
  (type-of-exp body
    (extend-tenv
      (map decl->var decls)
      (types-of-let-rands decls tenv)
      tenv)))

(define types-of-let-rands
  (lambda (decls tenv)
    (types-of-rands (map decl->exp decls) tenv)))
```

Tüübikontroll

- Letrec-avaldiste tüübikontroll:

```
(letrec (decls body)
  (for-each (compose check-letrec-decl-exp decl->exp)
    decls)
  (type-of-exp body (letrec-tenv decls tenv)))
```

```
(define check-letrec-decl-exp
  (lambda (exp)
    (if (not (and (assert? exp) (proc? (assert->exp exp))))
      (error "Invalid declaration expression:" exp))))
```

Tüübikontroll

- Letrec-avaldiste tüübikontroll (järg):

```
(define letrec-tenv
  (lambda (decls tenv)
    (let ((vars (map decl->var decls))
          (assert-exps (map decl->exp decls)))
      (let ((var-types (map assert->type assert-exps)))
        (let ((new-tenv (extend-tenv vars var-types tenv)))
          (for-each
            (lambda (assert-exp)
               (type-of-exp assert-exp new-tenv))
            assert-exps)
          new-tenv)))))
```

Tüübikontroll

- Assert-avaldiste ja protseduuride tüübikontroll:

```
(assert (type exp) (type-of-assert type exp tenv))
(proc (formals body) (type-of-proc formals body tenv))
```

```
(define type-of-proc
  (lambda (formals body tenv)
    (error "Procedure not inside an assert:" formals body)))
```

Tüübikontroll

```
(define type-of-assert
 (lambda (type exp tenv)
  (variant-case exp
    (proc (formals body)
      (if (proc-type? type)
        (if (= (length formals) (length (domains type)))
          (match-types (range type)
            (type-of-exp body
              (extend-tenv formals (domains type) tenv)))
          (type-error "Wrong number of formals" formals type))
        (type-error "Assert type is not a procedure:" type)))
    (else (match-types (type-of-exp exp tenv) type)))
   type))
```

Järgmiseks korraks

- Lugeda läbi EOPL ptk. 13.1, 13.2
- (EOPL ptk. 13 veebis saadaval!!)
- “Mängida” interpretaatoriga `loeng23.ss`