HW 7

For HW 7, your task is to finish a type checker that ensures only same-shaped trees are combined.

- +(1,2) is ok
- +(1,**cons**(1,2)) must be rejected

Is the resulting type system realistic (i.e., useful in practice)?

No.

To see why, look at the **heavier**-**tree** function of exercise 5.3, since tree-processing functions are highly practical...

Program from Exercise 5.3

```
proc(nt)
 let ht = proc(ntf)
               let nt = car(ntf)
                     f = cdr(ntf)
               in
                 if iscons(nt)
                   then cons((f cons(car(nt), f)),
                                     (f cons(cdr(nt), f)))
                   else +(nt,1)
 in (ht cons(nt, ht))
```

Program from Exercise 5.3

```
proc(nt)
 let ht = proc(ntf)
              let nt = car(ntf)
                    f = cdr(ntf)
              in
                if iscons(nt)
                  then cons((f cons(car(nt), f)),
                                   (f cons(cdr(nt), f)))
                  else +(nt,1)
 in (ht cons(nt, ht))
```

Assume **f**'s type is $(\mathbf{T}_1 \rightarrow \mathbf{T}_2)$.

Can't solve $\mathbf{T}_1 = [\mathbf{T}_3 : (\mathbf{T}_1 \rightarrow \mathbf{T}_2)]$.

Exercise 5.3 with letrec

Does **letrec** help?

```
proc(nt)
 letrec ht = proc(nt)
               if iscons(nt)
                 then cons((ht car(nt)),
                           (ht cdr(nt)))
                 else +(nt,1)
 in (ht nt)
```

Exercise 5.3 with letrec

Does **letrec** help?

```
proc(nt)
 letrec ht = proc(nt)
               if iscons(nt)
                 then cons((ht car(nt)),
                           (ht cdr(nt)))
                 else +(nt,1)
 in (ht nt)
```

Assume **ht**'s type is $([T_1 : T_2] \rightarrow T_3)$.

Can't solve $[T_1 : T_2] = T_1$.

Exercise 5.3 with letrec

Does **letrec** help?

```
proc(nt)
 letrec ht = proc(nt)
               if iscons(nt)
                 then cons((ht car(nt)),
                           (ht cdr(nt)))
                 else +(nt,1)
 in (ht nt)
```

Assume **ht**'s type is $([T_1 : T_2] \rightarrow T_3)$.

Can't solve $[T_1 : T_2] = T_1$.

The problem is that **ht** must work on many shapes of trees.

Exercise 5.3 with letrec and Polymorphism

Does polymorphism help?

```
proc(nt)
 letrec ht = proc(nt)
              if iscons(nt)
                then cons((ht car(nt)),
                          (ht cdr(nt)))
                else +(nt,1)
 in (ht nt)
```

Assume **ht**'s type is $([T_1 : T_2] \rightarrow T_3)$.

Need to solve $[T_{11} : T_{12}] = T_1$ *could be ok!*

Exercise 5.3 with letrec and Polymorphism

Does polymorphism help?

```
proc(nt)
 letrec ht = proc(nt)
                if iscons(nt)
                  then cons((ht car(nt)),
                            (ht cdr(nt)))
                  else +(nt,1)
 in (ht nt)
```

Assume **ht**'s type is $([T_1 : T_2] \rightarrow T_3)$.

Need to solve $[T_{11} : T_{12}] = T_1$ *could be ok!*

But can't solve $[T_1 : T_2] =$ `int`.

Why Exercise 5.3 Works

How can both we take the **car** of **nt** and add 1 to **nt**?

```
proc(nt)
 letrec ht = proc(nt)
               if iscons(nt)
                then cons((ht car(nt)),
                          (ht cdr(nt)))
                else +(nt,1)
 in (ht nt)
```

Why Exercise 5.3 Works

How can both we take the **car** of **nt** and add 1 to **nt**?

```
proc(nt)
 letrec ht = proc(nt)
               if iscons(nt)
                 then cons((ht car(nt)),
                           (ht cdr(nt)))
                 else +(nt,1)
 in (ht nt)
```

Because we test, first.

We need to tell the type system about this test.

Telling a Type System About Datatypes

Forms like **define-datatype** and **cases** are exactly what we need to tell the type system about variants and tests.

```
tree = [ tree : tree ]  or  int

proc(nt)
 letrec ht = proc(nt)
              cases nt of
                [tree : tree]: cons((ht car(nt)),
                                    (ht cdr(nt)))
                int: +(nt,1)
 in (ht nt)
```

Telling a Type System About Datatypes

Forms like **define-datatype** and **cases** are exactly what we need to tell the type system about variants and tests.

`tree` = [`tree : tree`] **or** `int`

Yet another technical problem:

- Is the type of 5 `int` or `tree`?
- Is the type of **cons**(5,5) [`int : int`] or [`tree : int`] or [`tree : tree`] or `tree`?

Solution: use explicit constructors.

Datatype Constructors

```
tree = (Branch [ tree : tree ])  or  (Leaf int)

let grow =
  proc(nt)
   letrec ht = proc(nt)
                    cases tree nt of
                      [(Branch p) (Branch cons((ht car(p)),
                                                    (ht cdr(p)))]
                      [(Leaf n) (Leaf +(n,1))]
   in (ht nt)
in (grow (Branch cons((Leaf 1), (Leaf 2))))
```

With a more flexible datatype type declaration, we can simplify this and get rid of the **cons** primitive completely.

Datatype Constructors without cons

```
tree = (Branch tree tree)  or  (Leaf int)

let grow =
  proc(nt)
   letrec ht = proc(nt)
                   cases tree nt of
                     [(Branch l r) (Branch (ht l) (ht r))]
                     [(Leaf n) (Leaf +(n,1))]
   in (ht nt)
in (grow (Branch (Leaf 1) (Leaf 2)))
```

Using Datatypes Types

- More examples in DrScheme
- HW 8: implement an interpreter using this book language
  - A type-checker and interpreter for this book language is provided
  - You write *another* interpreter layer on top of the book language interpreter

HW 8 interpreter

↓

latest book-language interpreter

↓

DrScheme interpreter

HW 8 Setup

```
exp = (var num) or (lit num) or (lam num exp)
   or (app exp exp) or (plus exp exp)
   or (minus exp exp) or (ifz exp exp exp)
in let sum10 =
 (app (lam 19 (app (app (var 19) (var 19))
                   (lit 10)))
      (lam 18
       (lam 17
         (ifz (var 17)
              (lit 0)
              (plus (var 17)
                    (app (app (var 18) (var 18))
                         (minus (var 17) (lit 1))))))))
in letrec ... eval ... = ...
in (eval sum10 ...)
```

Elements of a Realistic Type System

- A few primitive types
- Datatype declarations and dispatching

- Inference
- Polymorphism
- Datatype Polymorphism

The last three are not necessary for HW 8.

Toward Datatype Polymorphism

```
numlst = (empty) or (cons num numlst)
in
letrec numlst map((num -> num) f, numlst l)
           = cases numlst l of
              [(empty) (empty)]
              [(cons i l) (cons (f i) (map f l))]
in let l = (cons 1 (cons 20 (cons 14 (empty))))
in (map proc(num x)+(x,1) l)
```

Toward Datatype Polymorphism

```
numlstlst = (lempty) or (lcons numlst numlstlst)
numlst = (empty) or (cons num numlst)
in
letrec numlstlst map((num -> numlst) f, numlst l)
           = cases numlst l of
              [(empty) (lempty)]
              [(cons i l) (lcons (f i) (map f l))]
in let l = (cons 1 (cons 20 (cons 14 (empty))))
in (map proc(num x)(cons x (empty)) l)
```

Can we avoid multiple lst datatypes and multiple **map**s?

Datatype Polymorphism

```
<'a>lst = (empty) or (cons 'a <'a>lst)
twolsts = (together <num>lst <<num>lst>lst)
in
letrec <'b>lst map(('a -> 'b) f, <'a>lst l) =
              cases <'a>lst l of
               [(empty) (empty)]
               [(cons i l) (cons (f i) (map f l))]
in let l = (cons 1 (cons 20 (cons 14 (empty))))
   in (together (map proc(num x)+(x,1) l)
                (map proc(num x)(cons x (empty)) l))
```

- Use `'`<id> as a type variable
- Multiple uses of `'`<id> in one type definition or function signature correspond to the same type variable

Languages with Datatype Polymorphism

- The checker and interpreter in **`lecture16.scm`** implements the core of ***ML***

`http://caml.inria.fr/`

- Java will eventually get a similar kind of data polymorphism

Alternate HW 8

If you prefer, you can implement HW 8 using OCaml.

$$\begin{array}{c}\boxed{\text{HW 8 interpreter}}\\ \downarrow \\ \boxed{\text{OCaml interpreter}}\end{array}$$