# Outline of Lecture 12

- ADT example: search trees
- Lazy evaluation in Haskell
- List comprehension revisited
- Infinite lists

# ADT example: search trees

- A binary search tree is an object of type `Tree a`

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

  whose elements are **ordered**

- The tree `(Node val t1 t2)` is ordered if
  - all values in `t1` are smaller than `val`,
  - all values in `t2` are larger than `val`, and
  - the trees `t1` and `t2` are themselves ordered.

- The operations for building and manipulations such trees must preserve the order. We can ensure that only such approved operations are used by making the type into an abstract data type

# ADT example: search trees (cont.)

The signature of the abstract data type for such trees

```
module STree
  (Tree,
   nil,          -- Tree a
   isNil,        -- Tree a -> Bool
   isNode,        -- Tree a -> Bool
   leftSub,      -- Tree a -> Maybe (Tree a)
   rightSub,     -- Tree a -> Maybe (Tree a)
   treeVal,      -- Tree a -> Maybe a
   insertVal,    -- Ord a => a -> Tree a -> Tree a
   deleteVal,    -- Ord a => a -> Tree a -> Tree a
   minTree       -- Ord a => Tree a -> Maybe a
  )
where
```

The constructors for `Tree` datatype are hidden!

# ADT example: search trees (cont.)

```
data Tree a = Nil | Node a (Tree a) (Tree a)

nil ::  Tree a
nil = Nil

isNil ::  Tree a -> Bool
isNil Nil = True
isNil _ = False

isNode ::  Tree a -> Bool
isNode Nil = False
isNode _ = True

leftSub ::  Tree a -> Maybe (Tree a)
leftSub (Node _ t1 _) = Just t1
leftSub _ = Nothing
```

# ADT example: search trees (cont.)

```
rightSub ::  Tree a -> Maybe (Tree a)
rightSub (Node _ _ t2) = Just t2
rightSub _ = Nothing

treeVal ::  Tree a -> Maybe a
treeVal (Node v _ _) = Just v
treeVal _ = Nothing

minTree ::  Ord a => Tree a -> Maybe a
minTree t
  | isNil t = Nothing
  | isNil t1 = v
  | otherwise = minTree t1
   where
    (Just t1) = leftSub t
    v = treeVal t
```

# ADT example: search trees (cont.)

```
insertVal ::  Ord a => a -> Tree a -> Tree a
insertVal val Nil = (Node val Nil Nil)
insertVal val (Node v t1 t2)
  | v==val = Node v t1 t2
  | val < v = Node v (insertVal val t1) t2
  | val > v = Node v t1 (insertVal val t2)

deleteVal ::  Ord a => a -> Tree a -> Tree a
deleteVal val Nil = Nil
deleteVal val (Node v t1 t2)
  | val < v = Node v (deleteVal val t1) t2
  | val > v = Node v t1 (deleteVal val t2)
  | isNil t2 = t1
  | isNil t1 = t2
  | otherwise = joinTrees t1 t2
```

# ADT example: search trees (cont.)

```
-- auxiliary function

joinTrees ::  Ord a => Tree a -> Tree a -> Tree a
joinTrees t1 t2 = Node mini t1 newt2
  where
   (Just mini) = minTree t2
   newt2 = deleteVal mini t2
```

# Lazy evaluation in Haskell

- The underlying (lazy) evaluation strategy: Haskell will only evaluate an argument to a function if that argument's value is needed to compute the overall result
- If an argument is structured (e.g., a list or a tuple), only those parts of the argument that are needed for computation will be evaluated
- Since an intermediate result (e.g., list) will be only generated on demand, using such a list will not necessarily will be expensive computationally
- One of the consequences: a possibility to describe infinite data structures. Under lazy evaluation, often only parts of such a data structure need to be examined

# Lazy evaluation in Haskell (cont.)

- When the Haskell evaluation process starts, a *thunk* is created for each expression
- A *thunk* – a placeholder in the underlying graph of the program. It will be evaluated (reduced), if necessary. Otherwise, the garbage collector will eventually sweep it away
- If it is evaluated, because it's in the graph, it can be shared between expressions without re-calculation
- Lazy evaluation is often compared to non-strictness

# Strict vs non-strict languages

- Strict languages evaluate *inside out*; nonstrict languages like Haskell evaluate *outside in*
- *Outside in* means that evaluation proceeds from the outermost parts of expressions and works inward based on what values are needed. Thus, the order of evaluation and what gets evaluated can vary depending on inputs
- While in strict languages, evaluation starts with subexpressions. When all of them are evaluated, their enclosing expressions are calculated, etc. Thus, it goes *inside out*
- The following would work only in a nonstrict language:

```
Prelude> fst (1,undefined)
1
Prelude> tail [undefined,2,3]
[2,3]
```

# Lazy evaluation and function application

- Now, let's consider different evaluation scenarios in Haskell
- The argument which is not needed for producing the overall result will not be evaluated, e.g.

```
switch ::  Integer -> a -> a -> a
switch n x y
  | n>0 = x
  | otherwise = y
```

If the integer `n` is positive, only `x` is evaluated while the value `y` is "ignored". And vice versa in the `otherwise` case

# Lazy evaluation and function application

- The duplicated argument is never evaluated more than once, e.g.

```
hh ::  Integer -> Integer -> Integer
hh x y
  | x>0 = x+x
  | otherwise = ...
```

If the first guard succeeds, the value of `x` is evaluated only once (and stored in the internal Haskell data graph). For instance, in the function application

```
hh 12 (34^4 - hh 99 5)
```

the second argument expression is never evaluated

# Lazy evaluation and function application

- An argument is not necessarily evaluated fully. Only the parts that are needed are examined, e.g.

```
pm ::  (Integer,Integer) -> Integer
pm (x,y) = x+1
```

If we apply this function to the pair `(3+2,4-17)`, only the first part of the pair will be fully evaluated

# Evaluation order for a function application

A reminder: general form of a function declaration:

```
f p1 p2 ... pk
  | g1        = e1
  | g2        = e2
  ...
  | otherwise = er
   where
    l1 a1,1 ... = r1
    l2 a2,1 ... = r2
    ...

f q1 q2 ... qk
  = ...
```

where $p_i, q_i, a_{i,j}$ are argument patterns, $g_i$ are boolean expressions, and $l_i$ are local identifiers.

# Evaluation order for a function application (cont.)

- A function declaration may contain a number of equations (with pattern matching), then a number of guarded declarations with each equation, as well as several local definitions for each equation
- While applying a function, pattern matching expressions in function equations are evaluated in the order they come (until the first success)
- Moreover, for each applied pattern, only the necessary parts of argument expressions are evaluated
- Similarly, the guards are evaluated in the defined order (until the first success)
- Only those local definitions that are needed (either in guards or result expressions) are evaluated

# General evaluation order in an Haskell expression

- Evaluation is **from outside in**. In situations like

$$\underline{\mathtt{f}_1\ \mathtt{e}_1\ \underline{(\mathtt{f}_2\ \mathtt{e}_2\ 17)}}$$

  where one application encloses another, the outer one is evaluated first

- Otherwise, evaluation is **from left to right**. In the expression like

$$\underline{\mathtt{f}_1\ \mathtt{e}_1} + \underline{(\mathtt{f}_2\ \mathtt{e}_2)}$$

  the underlined expressions are both to be evaluated, however, the left one will be evaluated first. In some cases like `False && p`, the evaluation of the left expression is sufficient for the overall result

# List comprehensions revisited

- From the evaluation order standpoint ...
- A reminder: a list comprehension is an expression of the form

$$[\mathrm{e} \mid \mathtt{q}_1, \mathtt{q}_2, \ldots, \mathtt{q}_\mathtt{k}]$$

  where each $\mathtt{q_i}$ is either
  - a **generator** of the form `p <- lExp`, where `p` is a pattern and `lExp` is an expression of the list type
  - a **test**, `bExp`, which is a boolean expression
- Multiple generators allow to combine elements from two or more lists. What is the evaluation order?

# List comprehensions revisited (cont.)

- Example:

```
pairs ::  [a] -> [b] -> [(a,b)]
pairs xs ys = [(x,y) | x <-xs, y<-ys]
```

  Then calling `pairs [1,2,3] [4,5]` gives us

  `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

- First, the first value from `xs`, 1, is fixed and all possible values from `ys` are chosen. Then, the process is repeated for the remaining values from `xs` (2 and 3)

# List comprehensions revisited (cont.)

- This order is not accidental, since we can have the second generator to depend on the value given by the first generator, e.g:

```
triangle ::  Int -> [(Int,Int)]
triangle n = [(x,y) | x <-[1..n], y<-[1..x]]
```

Then calling `triangle 3` gives us

`[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]`

Thus, the value of `x` restricts how many values are considered for `y`

# List comprehensions revisited (cont.)

- Example: Pythagorean triples (where the sum of squares of the first two numbers is equal to square of the third one):

```
pyTriples ::  Integer -> [(Integer,Integer,Integer)]
pyTriples n = [(x,y,z) | x <-[2..n], y<-[x+1..n],
  z <- [y+1..n], x*x + y*y == z*z]
```

Here the test combines the values from the three generators

# List comprehensions revisited (cont.)

- Generators may rely on (recursive) function calls. Example of calculating permutations:

```
perms ::  Eq a => [a] -> [[a]]
perms [] = [[]]
perms xs = [x:ps | x<-xs, ps <- perms (xs\\[x])]
```

where \\ is the list subtraction (difference) operator from Data.List

# List comprehensions revisited (cont.)

- If some generator patterns are **refutable**, i.e., may sometimes fail, the corresponding elements are filtered out from (not counted in) the result. For instance,

```
heads ::  [[a]] -> [a]
heads zs = [x | (x:_) <- zs]
```

If we apply

```
> heads [[],[2],[4,5],[]]
```

the result is simply `[2,4]`

# Infinite lists

- One important consequence of lazy evaluation is a possibility for the language to describe **infinite** structures, where only the necessary finite portion will be actually evaluated
- Any recursive type will contain infinite objects. We will concentrate on infinite lists here
- A simple example:

```
ones ::  [Integer]
ones = 1 : ones
```

- Evaluation of `ones` in Haskell produces a list of ones, indefinitely:

```
[1,1,1,1,1,1,1,^C,1,1,1,Interrupted
```

# Infinite lists (cont.)

- We can sensibly evaluate functions applied to `ones`, e.g.,

```
> take 20 ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

- Built in the system are the lists of the form `[n ..]` and `[n,m]` so that

```
[3 ..]   == [3,4,5,6, ...
[3,5, ..]  == [3,5,7,9, ...
```

- We can define these functions ourselves, e.g.,

```
from ::  Integer -> [Integer]
from n = n : from (n+1)
```

# Infinite lists (cont.)

- List comprehensions can also define infinite lists. Example (all Pythagorean triples):

```
pyTriples = [(x,y,z) | z <- [2..], y <- [2..z-1],
  x <- [2 ..  y-1], x*x + y*y == z*z]
```

- Another example: generating prime numbers (*Sieve of Eratosthenes*):

```
primes ::  [Integer]
primes = sieve [2 ..]

sieve (x:xs) =
  x :  sieve [y | y <- xs, y `mod` x > 0]
```

  - Sieve the infinite list and then add the first "survived" element to the prime list
  - Then use this last found prime as the number to sieve on
  - Repeat indefinitely

# Infinite lists (cont.)

- Example: generating pseudo-random numbers:

```
nextRand ::  Integer -> Integer
nextRand n = (multiplier*n + increment) `mod` modulus

randomSequence ::  Integer -> [Integer]
randomSequence = iterate nextRand

seed = 17489
multiplier = 25173
increment = 13849
modulus = 65536
```

```
> randomSequence seed
[17489,59134,9327,52468,43805,8378,18395, ...
```

# Why infinite lists?

- **Data-directed computing** (a sequence of data generating processes and generic data transformations)
- Constructing and manipulating potentially **infinite/unlimited resources**. We don't know how much of the resource will be needed while constructing the program
- More abstract and simpler to write