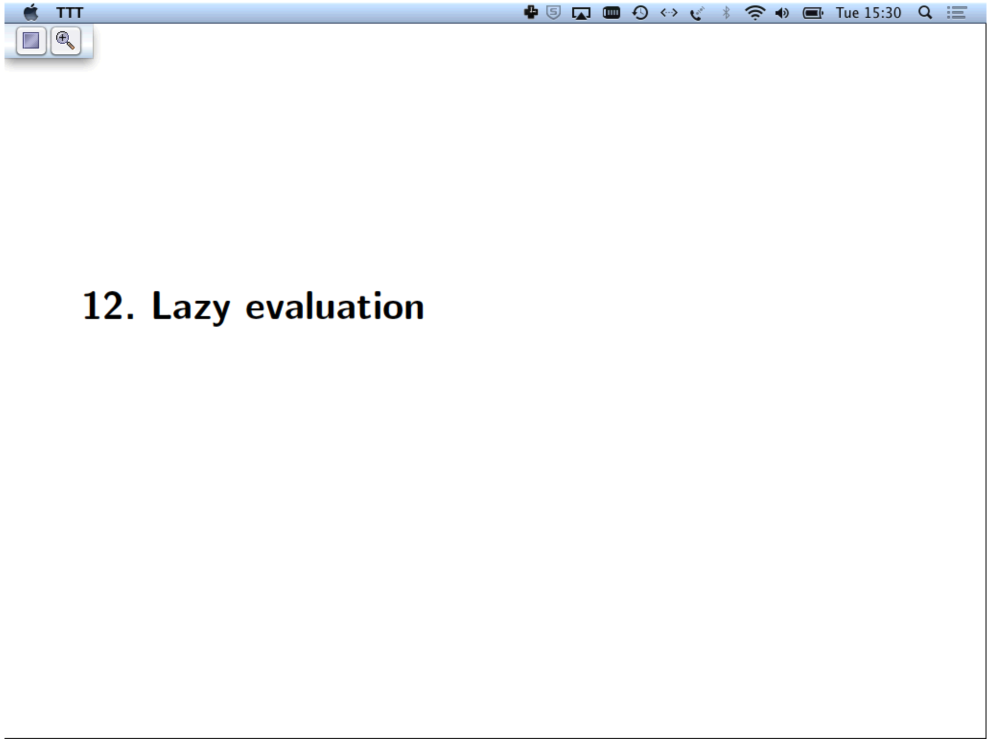
Script generated by TTT

Title: Nipkow: Info2 (21.01.2014)

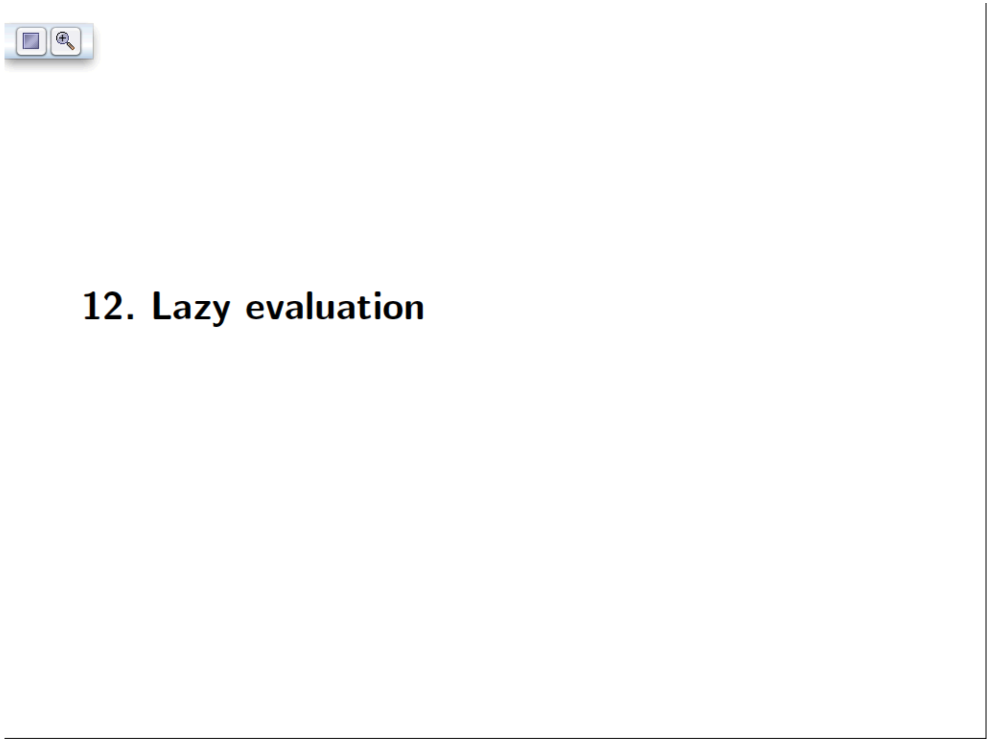
Date: Tue Jan 21 15:30:38 CET 2014

Duration: 89:01 min

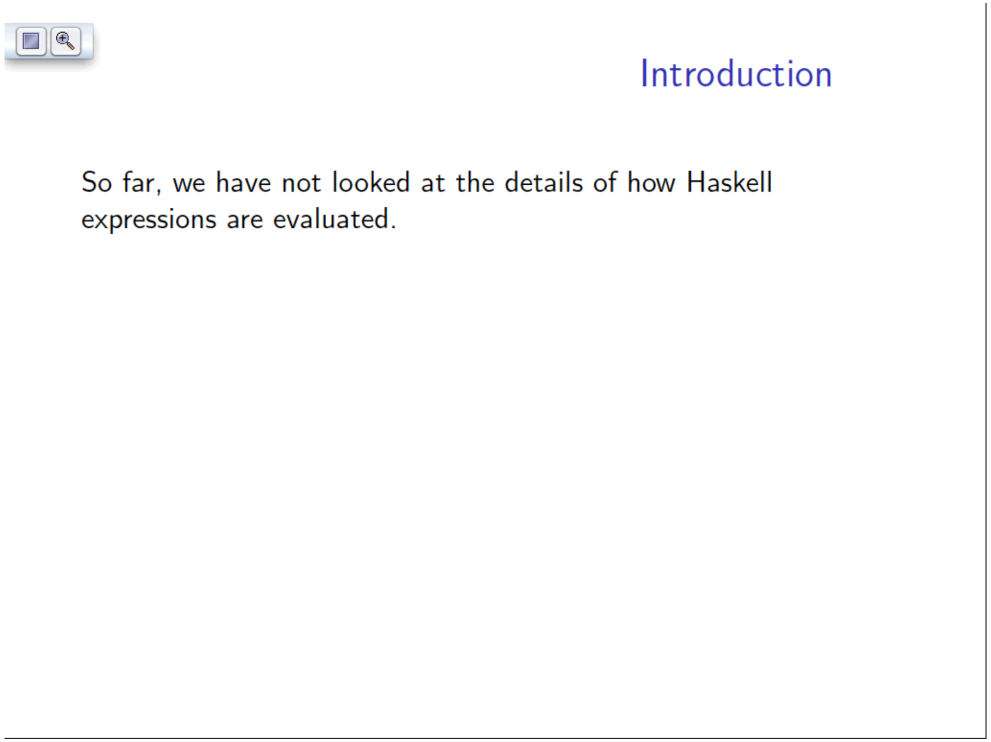
Pages: 151



12. Lazy evaluation



12. Lazy evaluation



Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated.

Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called

lazy evaluation („verzögerte Auswertung")

Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called

lazy evaluation („verzögerte Auswertung")

Advantages:

- Avoids unnecessary evaluations

Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called

lazy evaluation („verzögerte Auswertung")

Advantages:

- Avoids unnecessary evaluations
- Terminates as often as possible
- Supports infinite lists
- Increases modularity

Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called

lazy evaluation („verzögerte Auswertung")

Advantages:

- Avoids unnecessary evaluations
- Terminates as often as possible
- Supports infinite lists
- Increases modularity

Therefore Haskell is called a *lazy functional langauge*.

Evaluating expressions

Expressions are evaluated (*reduced*) by successively applying definitions until no further reduction is possible.

Evaluating expressions

Expressions are evaluated (*reduced*) by successively applying definitions until no further reduction is possible.

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

One evaluation:

```
sq(3+4)
```

Evaluating expressions

Expressions are evaluated (*reduced*) by successively applying definitions until no further reduction is possible.

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

One evaluation:

```
sq(3+4)  =  sq 7
```

Evaluating expressions

Expressions are evaluated (*reduced*) by successively applying definitions until no further reduction is possible.

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

One evaluation:

```
sq(3+4)  =  sq 7  =  7 * 7
```

Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example
Let `n` have value `0` initially.

Two evaluations:

`n + (n := 1)`

Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example
Let `n` have value `0` initially.

Two evaluations:

$\underline{\texttt{n}}\ \texttt{+ (n := 1)}$

Theorem

Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example

Let `n` have value 0 initially.

Two evaluations:

$\underline{\mathtt{n}} \texttt{ + (n := 1)} \quad = \quad \texttt{0 + (}\underline{\texttt{n := 1}}\texttt{)} \quad = \quad \texttt{0 + 1}$

Theorem

Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example

Let `n` have value 0 initially.

Two evaluations:

$\underline{\mathtt{n}} \texttt{ + (n := 1)} \quad = \quad \texttt{0 + (}\underline{\texttt{n := 1}}\texttt{)} \quad = \quad \underline{\texttt{0 + 1}}$

Reduction strategies

An expression may have many reducible subexpressions:

$$\underline{\texttt{sq }(\underline{3+4})}$$

Reduction strategies

An expression may have many reducible subexpressions:

$$\underline{\texttt{sq }(\underline{3+4})}$$

Terminology: *redex* = reducible expression

Reduction strategies

An expression may have many reducible subexpressions:

$$\underline{\texttt{sq (}\underline{\texttt{3+4}}\texttt{)}}$$

Terminology: *redex* = reducible expression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:

Reduction strategies

An expression may have many reducible subexpressions:

$$\underline{\texttt{sq (}\underline{\texttt{3+4}}\texttt{)}}$$

Terminology: *redex* = reducible expression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:
Arguments are evaluated
before they are substituted into the function body

Reduction strategies

An expression may have many reducible subexpressions:

$$\underline{\texttt{sq (}\underline{\texttt{3+4}}\texttt{)}}$$

Terminology: *redex* = reducible expression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:
Arguments are evaluated
before they are substituted into the function body
`sq (3+4)  =  sq 7  =  7 * 7`

Outermost reduction Always reduce an outermost redex.

Reduction strategies

An expression may have many reducible subexpressions:

$$\underline{\texttt{sq (}\underline{\texttt{3+4}}\texttt{)}}$$

Terminology: *redex* = reducible expression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:
Arguments are evaluated
before they are substituted into the function body
`sq (3+4)  =  sq 7  =  7 * 7`

Outermost reduction Always reduce an outermost redex.
Corresponds to *call by name:*
The unevaluated arguments
are substituted into the the function body

Reduction strategies

An expression may have many reducible subexpressions:

```
sq (3+4)
```

Terminology: *redex* = reducible expression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:
Arguments are evaluated
before they are substituted into the function body

```
sq (3+4)  =  sq 7  =  7 * 7
```

Outermost reduction Always reduce an outermost redex.
Corresponds to *call by name:*
The unevaluated arguments
are substituted into the the function body

```
sq (3+4)  =  (3+4) * (3+4)
```

Comparison: termination

Definition:

```
loop = tail loop
```

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)
```

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)  =  fst(1,tail loop)
              =  fst(1,tail(tail loop))
              =  ...
```

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)  =  fst(1,tail loop)
              =  fst(1,tail(tail loop))
              =  ...
```

Outermost reduction:

```
fst (1,loop)  =  1
```

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)  =  fst(1,tail loop)
              =  fst(1,tail(tail loop))
              =  ...
```

Outermost reduction:

```
fst (1,loop)  =  1
```

Theorem If expression e has a terminating reduction sequence, then outermost reduction of e also terminates.

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)  =  fst(1,tail loop)
              =  fst(1,tail(tail loop))
              =  ...
```

Outermost reduction:

```
fst (1,loop)  =  1
```

Theorem If expression e has a terminating reduction sequence, then outermost reduction of e also terminates.

Outermost reduction terminates as often as possible

Why is this useful?

Example

Can build your own control constructs:

```
switch :: Int -> a -> a -> a
```

Why is this useful?

Example

Can build your own control constructs:

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

Why is this useful?

Example

Can build your own control constructs:

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

fac :: Int -> Int
fac n = switch n (n * fac(n-1)) 1
```

Comparison: Number of steps

Innermost reduction:

```
sq (3+4) =  sq 7  =  7 * 7  =  49
```

Comparison: Number of steps

Innermost reduction:

```
sq (3+4) =  sq 7  =  7 * 7  =  49
```

Outermost reduction:

```
sq(3+4)  =  (3+4)*(3+4)  =  7*(3+4)  =  7*7  =  49
```

Comparison: Number of steps

Innermost reduction:

`sq (3+4) = sq 7 = 7 * 7 = 49`

Outermost reduction:

`sq(3+4) = (3+4)*(3+4) = 7*(3+4) = 7*7 = 49`

More outermost than innermost steps!

Comparison: Number of steps

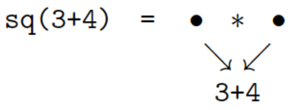
Innermost reduction:

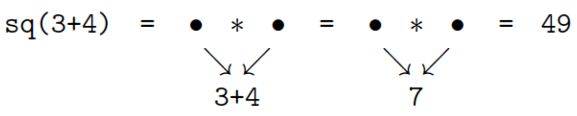
`sq (3+4) = sq 7 = 7 * 7 = 49`

Outermost reduction:

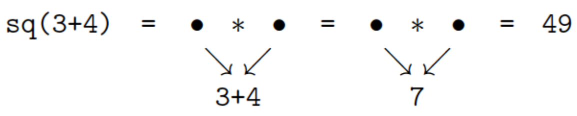
`sq(3+4) = (3+4)*(3+4) = 7*(3+4) = 7*7 = 49`

More outermost than innermost steps!
How can outermost reduction be improved?
Sharing!



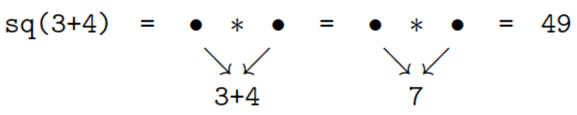


The expression 3+4 is only evaluated *once!*



The expression 3+4 is only evaluated *once!*

Lazy evaluation := outermost reduction + sharing



The expression 3+4 is only evaluated *once!*

Lazy evaluation := outermost reduction + sharing

Theorem
Lazy evaluation never needs more steps than innermost reduction.

The principles of lazy evaluation:

- Arguments of functions are evaluated only
  if needed to continue the evaluation of the function.

The principles of lazy evaluation:

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function. (Remember `fst (1,loop)`)

The principles of lazy evaluation:

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function. (Remember `fst (1,loop)`)
- Each argument is evaluated at most once (sharing!)

The principles of lazy evaluation:

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function. (Remember `fst (1,loop)`)
- Each argument is evaluated at most once (sharing!)

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]
```

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]                      -- does f.1 match?
```

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]                      -- does f.1 match?
=  f (1 : [2..3]) [7..9]
```

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]                      -- does f.1 match?
=  f (1 : [2..3]) [7..9]             -- does f.2 match?
```

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]                  -- does f.1 match?
=  f (1 : [2..3]) [7..9]         -- does f.2 match?
=  f (1 : [2..3]) (7 : [8..9])   -- does f.3 match?
```

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]                  -- does f.1 match?
=  f (1 : [2..3]) [7..9]         -- does f.2 match?
=  f (1 : [2..3]) (7 : [8..9])   -- does f.3 match?
=  1+7
=  8
```

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

`f (2+3) (4-1) (3+9)`

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

```
f (2+3) (4-1) (3+9)
  ?  2+3 >= 4-1 && 2+3 >= 3+9
```

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

```
f (2+3) (4-1) (3+9)
  ?  2+3 >= 4-1 && 2+3 >= 3+9
  ?  =  5 >= 3 && 5 >= 3+9
```

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

```
f (2+3) (4-1) (3+9)
  ?  2+3 >= 4-1 && 2+3 >= 3+9
  ?  =  5 >= 3 && 5 >= 3+9
  ?  =  True    && 5 >= 3+9
  ?  =  5 >= 3+9
  ?  =  5 >= 12
  ?  =  False
  ?  3 >= 5 && 3 >= 12
```

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

```
f (2+3) (4-1) (3+9)
  ?  2+3 >= 4-1 && 2+3 >= 3+9
  ?  =  5 >= 3 && 5 >= 3+9
  ?  =  True    && 5 >= 3+9
  ?  =  5 >= 3+9
  ?  =  5 >= 12
  ?  =  False
  ?  3 >= 5 && 3 >= 12
  ?  =  False && 3 >= 12
  ?  =  False
  ?  otherwise = True
```

Guards

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

```
f (2+3) (4-1) (3+9)
  ? 2+3 >= 4-1 && 2+3 >= 3+9
  ?  =  5 >= 3 && 5 >= 3+9
  ?  =  True   && 5 >= 3+9
  ?  =  5 >= 3+9
  ?  =  5 >= 12
  ?  =  False
  ? 3 >= 5 && 3 >= 12
  ?  =  False && 3 >= 12
  ?  =  False
  ? otherwise = True
=  12
```

`where`

Same principle: definitions in `where` clauses are only evaluated when needed and only as much as needed.

`where`

Same principle: definitions in `where` clauses are only evaluated when needed and only as much as needed.

Lambda

Haskell never reduces inside a lambda

Lambda

Haskell never reduces inside a lambda

Example: `\x -> False && x` cannot be reduced

Lambda

Haskell never reduces inside a lambda

Example: `\x -> False && x` cannot be reduced
Reasons:

- Functions are black boxes

Lambda

Haskell never reduces inside a lambda

Example: `\x -> False && x` cannot be reduced
Reasons:

- Functions are black boxes
- All you can do with a function is apply it

Lambda

Haskell never reduces inside a lambda

Example: `\x -> False && x` cannot be reduced
Reasons:

- Functions are black boxes
- All you can do with a function is apply it

Example:

```
(\x -> False && x) True  =  False && True  =  False
```

Built-in functions

Arithmetic operators and other built-in functions
evaluate their arguments first

Built-in functions

Arithmetic operators and other built-in functions
evaluate their arguments first

Example

`3 * 5` is a redex

Built-in functions

Arithmetic operators and other built-in functions
evaluate their arguments first

Example

`3 * 5` is a redex

`0 * head (...)` is not a redex

Predefined functions from Prelude

They behave like their Haskell definition:

```
(&&) :: Bool -> Bool -> Bool
True  && y  =  y
False && y  =  False
```

Slogan

Lazy evaluation evaluates an expression only when needed and only as much as needed.

Slogan

Lazy evaluation evaluates an expression only when needed and only as much as needed.

(*“Call by need”*)

12.1 Applications of lazy evaluation

Minimum of a list

```
min  =  head . inSort
```

Minimum of a list

```
min  =  head . inSort

inSort :: Ord a => [a] -> [a]
inSort []       =  []
inSort (x:xs)   =  ins x (inSort xs)
```

Minimum of a list

```
min  =  head . inSort

inSort :: Ord a => [a] -> [a]
inSort []       =  []
inSort (x:xs)   =  ins x (inSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []   =  [x]
ins x (y:ys) | x <= y      =  x : y : ys
             | otherwise   =  y : ins x ys
```

Minimum of a list

```
min  =  head . inSort

inSort :: Ord a => [a] -> [a]
inSort []       =  []
inSort (x:xs)   =  ins x (inSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []   =  [x]
ins x (y:ys) | x <= y      =  x : y : ys
             | otherwise   =  y : ins x ys

⟹  inSort [6,1,7,5]
    =  ins 6 (ins 1 (ins 7 (ins 5 [])))
```

```
min [6,1,7,5]   =  head(inSort [6,1,7,5])
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
```

Minimum of a list

```
min  =  head . inSort

inSort :: Ord a => [a] -> [a]
inSort []      =  []
inSort (x:xs)  =  ins x (inSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []  =  [x]
ins x (y:ys) | x <= y     =  x : y : ys
             | otherwise  =  y : ins x ys

⟹  inSort [6,1,7,5]
    =  ins 6 (ins 1 (ins 7 (ins 5 [])))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
=  1
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
=  1
```

Lazy evaluation needs only linear time

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
=  1
```

Lazy evaluation needs only linear time
although `inSort` is quadratic

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
=  1
```

Lazy evaluation needs only linear time
although `inSort` is quadratic
because the sorted list is never constructed completely

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
=  1
```

Lazy evaluation needs only linear time
although `inSort` is quadratic
because the sorted list is never constructed completely

Warning: this depends on the exact algorithm and does not work so nicely with all sorting functions!

Maximum of a list

```
max  =  last . inSort
```

Complexity?

12.2 Infinite lists

Example

A recursive definition

```
ones :: [Int]
ones  =  1 : ones
```

Example

A recursive definition

```
ones :: [Int]
ones  =  1 : ones
```

that defines an infinite list of 1s:

```
ones
```

Example

A recursive definition

```
ones :: [Int]
ones  =  1 : ones
```

that defines an infinite list of 1s:

```
ones  =  1 : ones
```

Example

A recursive definition

```
ones :: [Int]
ones  =  1 : ones
```

that defines an infinite list of 1s:

```
ones  =  1 : ones  =  1 : 1 : ones  =  ...
```

What GHCi has to say about it:

```
> ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Haskell lists can be finite or infinite

But Haskell can compute with infinite lists, thanks to lazy evaluation:

But Haskell can compute with infinite lists, thanks to lazy evaluation:

```
> head ones
1
```

Remember:

> Lazy evaluation evaluates an expression only as much as needed

---

But Haskell can compute with infinite lists, thanks to lazy evaluation:

```
> head ones
1
```

Remember:

> Lazy evaluation evaluates an expression only as much as needed

Outermost reduction: `head ones   =   head (1 : ones)   =   1`

But Haskell can compute with infinite lists, thanks to lazy evaluation:

```
> head ones
1
```

Remember:

> Lazy evaluation evaluates an expression only as much as needed

Outermost reduction: `head ones   =   head (1 : ones)   =   1`

Innermost reduction:

```
head ones
=  head (1 : ones)
=  head (1 : 1 : ones)
=  ...
```

Haskell lists are never actually infinite but only potentially infinite

Haskell lists are never actually infinite but only potentially infinite

Lazy evaluation computes as much of the infinite list as needed

This is how partially evaluated lists are represented internally:

1 : 2 : 3 : $\boxed{\text{code pointer to compute rest}}$

Haskell lists are never actually infinite but only potentially infinite

Lazy evaluation computes as much of the infinite list as needed

This is how partially evaluated lists are represented internally:

1 : 2 : 3 : $\boxed{\text{code pointer to compute rest}}$

In general: finite prefix followed by code pointer

Why (potentially) infinite lists?

Why (potentially) infinite lists?

- They come for free with lazy evaluation
- They increase modularity:
  list producer does not need to know
  how much of the list the consumer wants

Example: The sieve of Eratosthenes

1. Create the list 2, 3, 4, . . .

Example: The sieve of Eratosthenes

1. Create the list 2, 3, 4, . . .
2. Output the first value p in the list as a prime.
3. Delete all multiples of p from the list
4. Goto step 2

Example: The sieve of Eratosthenes

1. Create the list 2, 3, 4, . . .
2. Output the first value p in the list as a prime.
3. Delete all multiples of p from the list
4. Goto step 2

2 3 4 5 6 7 8 9 10 11 12 . . .

In Haskell:

```haskell
primes :: [Int]
primes  =  sieve [2..]

sieve :: [Int] -> [Int]
sieve (p:xs)   =  p : sieve [x | x <- xs,
```

In Haskell:

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

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

In Haskell:

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

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

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
```

In Haskell:

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

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

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x `mod` 2 /= 0]
```

In Haskell:

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

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

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x `mod` 2 /= 0]
=  2 : sieve [x | x <- 3:[4..],  x `mod` 2 /= 0]
```

In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs)  =  p : sieve [x | x <- xs, x ‘mod‘ p /= 0]
```

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x ‘mod‘ 2 /= 0]
=  2 : sieve [x | x <- 3:[4..],  x ‘mod‘ 2 /= 0]
=  2 : sieve (3 : [x | x <- [4..],  x ‘mod‘ 2 /= 0])
```

In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs)  =  p : sieve [x | x <- xs, x ‘mod‘ p /= 0]
```

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x ‘mod‘ 2 /= 0]
=  2 : sieve [x | x <- 3:[4..],  x ‘mod‘ 2 /= 0]
=  2 : sieve (3 : [x | x <- [4..],  x ‘mod‘ 2 /= 0])
=  2 : 3 : sieve [x | x <- [x|x <- [4..],  x ‘mod‘ 2 /= 0]
                      x ‘mod‘ 3 /= 0]
```

In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs)  =  p : sieve [x | x <- xs, x ‘mod‘ p /= 0]
```

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x ‘mod‘ 2 /= 0]
=  2 : sieve [x | x <- 3:[4..],  x ‘mod‘ 2 /= 0]
=  2 : sieve (3 : [x | x <- [4..],  x ‘mod‘ 2 /= 0])
=  2 : 3 : sieve [x | x <- [x|x <- [4..],  x ‘mod‘ 2 /= 0]
                      x ‘mod‘ 3 /= 0]
= ...
```

Modularity!

Modularity!

The first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The primes between 100 and 150:

```
> takeWhile (<150) (dropWhile (<100) primes)
```

Modularity!

The first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The primes between 100 and 150:

```
> takeWhile (<150) (dropWhile (<100) primes)
[101,103,107,109,113,127,131,137,139,149]
```

Modularity!

The first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The primes between 100 and 150:

```
> takeWhile (<150) (dropWhile (<100) primes)
[101,103,107,109,113,127,131,137,139,149]
```

All twin primes:

```
> [(p,q) | (p,q) <-                         , p+2==q]
```

Modularity!

The first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The primes between 100 and 150:

```
> takeWhile (<150) (dropWhile (<100) primes)
[101,103,107,109,113,127,131,137,139,149]
```

All twin primes:

```
> [(p,q) | (p,q) <- zip primes (tail primes), p+2==q]
```

Primality test?

```
> 101 `elem` primes
True

> 102 `elem` primes
```

Primality test?

```
> 101 `elem` primes
True

> 102 `elem` primes
```

nontermination

Primality test?

```
> 101 `elem` primes
True

> 102 `elem` primes
```

nontermination

```
prime n  =  n ==
```

Primality test?

```
> 101 `elem` primes
True

> 102 `elem` primes
```

nontermination

```
prime n  =  n == head (dropWhile (<n) primes)
```

Sharing!

There is only one copy of `primes`

Primality test?

```
> 101 `elem` primes
True
```

```
> 102 `elem` primes
```

nontermination

```
prime n  =  n == head
```

Sharing!

There is only one copy of `primes`

Every time part of `primes` needs to be evaluated

Example: when computing `take 5 primes`

`primes` is (invisibly!) updated to remember the evaluated part

Example: `primes = 2 : 3 : 5 : 7 : 11 : sieve ...`

Sharing!

There is only one copy of `primes`

Every time part of `primes` needs to be evaluated

Example: when computing `take 5 primes`

`primes` is (invisibly!) updated to remember the evaluated part

Example: `primes = 2 : 3 : 5 : 7 : 11 : sieve ...`

The next uses of `primes` are faster:

Example: now `primes !! 2` needs only 3 steps

Sharing!

There is only one copy of `primes`

Every time part of `primes` needs to be evaluated

Example: when computing `take 5 primes`

primes is (invisibly!) updated to remember the evaluated part

Example: `primes = 2 : 3 : 5 : 7 : 11 : sieve ...`

The next uses of `primes` are faster:

Example: now `primes !! 2` needs only 3 steps

Nothing special, just the automatic result of sharing

The list of Fibonacci numbers

Idea: 0 1 1 2 ...

The list of Fibonacci numbers

$$\begin{array}{lll} \text{Idea:} & & 0\ 1\ 1\ 2\ \ldots \\ & + & \quad 0\ 1\ 1\ \ldots \end{array}$$

The list of Fibonacci numbers

$$\begin{array}{lll} \text{Idea:} & & 0\ 1\ 1\ 2\ \ldots \\ & + & \quad 0\ 1\ 1\ \ldots \\ & = & 0\ 1\ 2\ 3\ \ldots \end{array}$$

The list of Fibonacci numbers

Idea: 0 1 1 2 ...
$+$ 0 1 1 ...
$=$ 0 1 2 3 ...

From Prelude: `zipWith`

The list of Fibonacci numbers

Idea: 0 1 1 2 ...
$+$ 0 1 1 ...
$=$ 0 1 2 3 ...

From Prelude: `zipWith`
Example: `zipWith f [a1, a2, ...] [b1, b2, ...]`

The list of Fibonacci numbers

Idea: 0 1 1 2 ...
$+$ 0 1 1 ...
$=$ 0 1 2 3 ...

From Prelude: `zipWith`
Example: `zipWith f [a1, a2, ...] [b1, b2, ...]`
`= [f a1 a2, f a2 b2, ...]`

The list of Fibonacci numbers

Idea: 0 1 1 2 ...
$+$ 0 1 1 ...
$=$ 0 1 2 3 ...

From Prelude: `zipWith`
Example: `zipWith f [a1, a2, ...] [b1, b2, ...]`
`= [f a1 a2, f a2 b2, ...]`

```
fibs :: [Integer]
fibs = 0 : 1 :
```

The list of Fibonacci numbers

Idea: 0 1 1 2 ...
$+$ 0 1 1 ...
$=$ 0 1 2 3 ...

From Prelude: `zipWith`
Example: `zipWith f [a1, a2, ...] [b1, b2, ...]`
`= [f a1 a2, f a2 b2, ...]`

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

The list of Fibonacci numbers

Idea: 0 1 1 2 ...
$+$ 0 1 1 ...
$=$ 0 1 2 3 ...

From Prelude: `zipWith`
Example: `zipWith f [a1, a2, ...] [b1, b2, ...]`
`= [f a1 a2, f a2 b2, ...]`

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

How about

```
fibs = 0 : 1 : [x+y | x <- fibs, y <- tail fibs]
```

Hamming numbers

Game tree

```
data Tree p v = Tree p v [Tree p v]
```

Separates move computation and valuation from move selection

Game tree

```
data Tree p v = Tree p v [Tree p v]
```

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally, as much as is needed

---

Game tree

```
data Tree p v = Tree p v [Tree p v]
```

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice
- Supports infinitely broad and deep trees (useful??)

Game tree

```
data Tree p v = Tree p v [Tree p v]
```

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice
- Supports infinitely broad and deep trees (useful??)

```
gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
```

Game tree

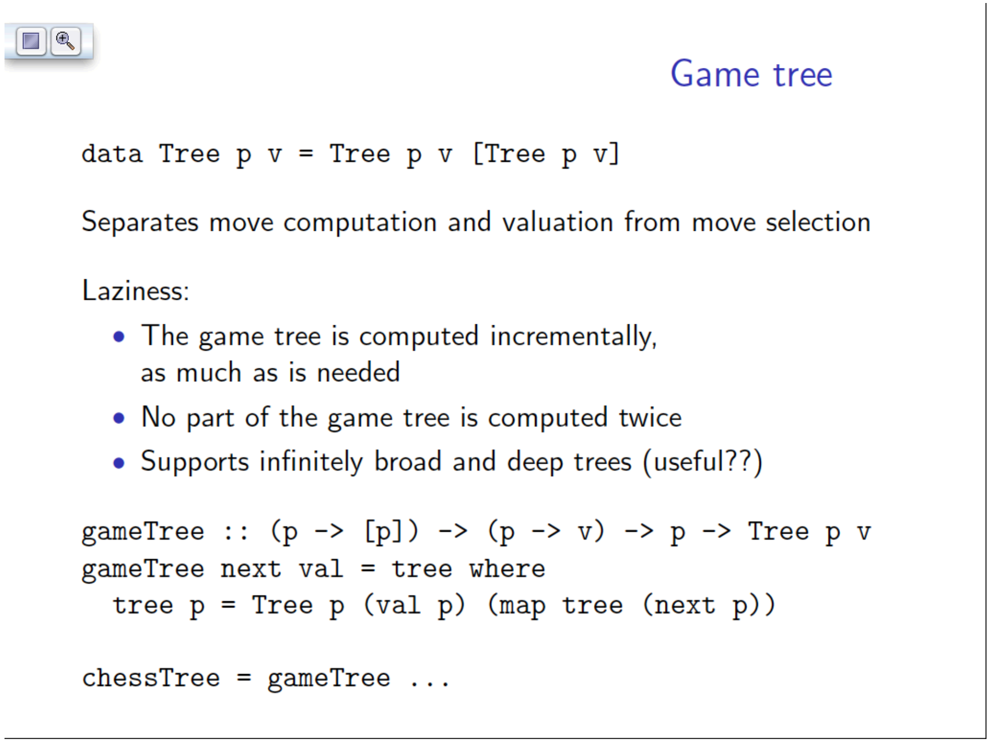
```
data Tree p v = Tree p v [Tree p v]
```

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice
- Supports infinitely broad and deep trees (useful??)

```
gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
gameTree next val = tree where
  tree p = Tree p (val p) (map tree (next p))
```



Game tree

```
data Tree p v = Tree p v [Tree p v]
```

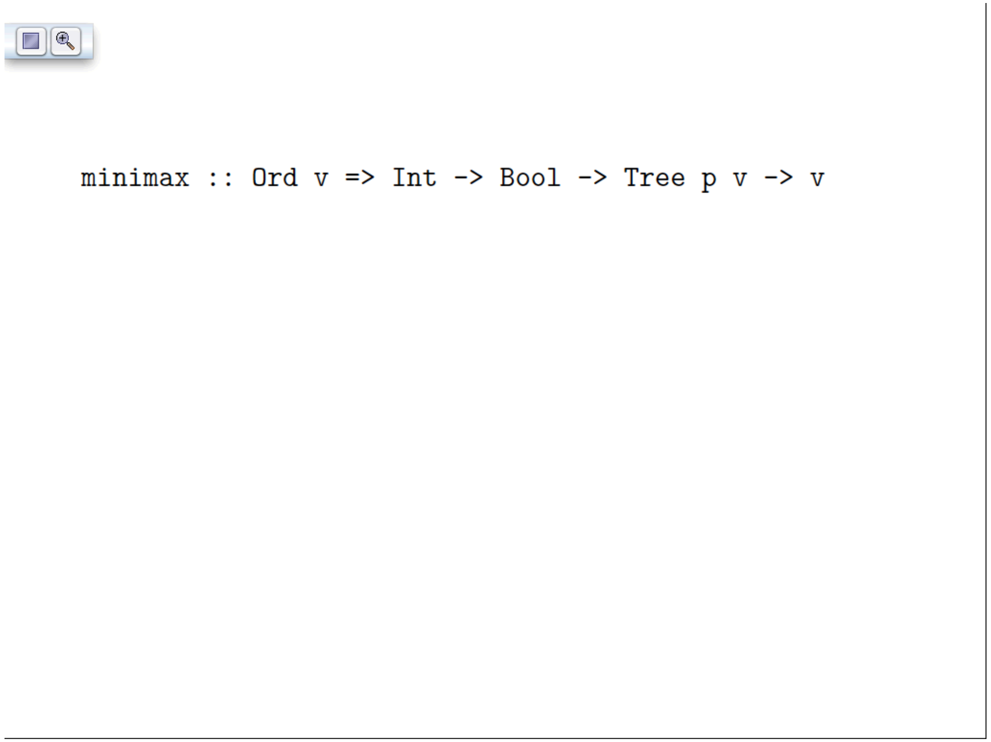
Separates move computation and valuation from move selection

Laziness:

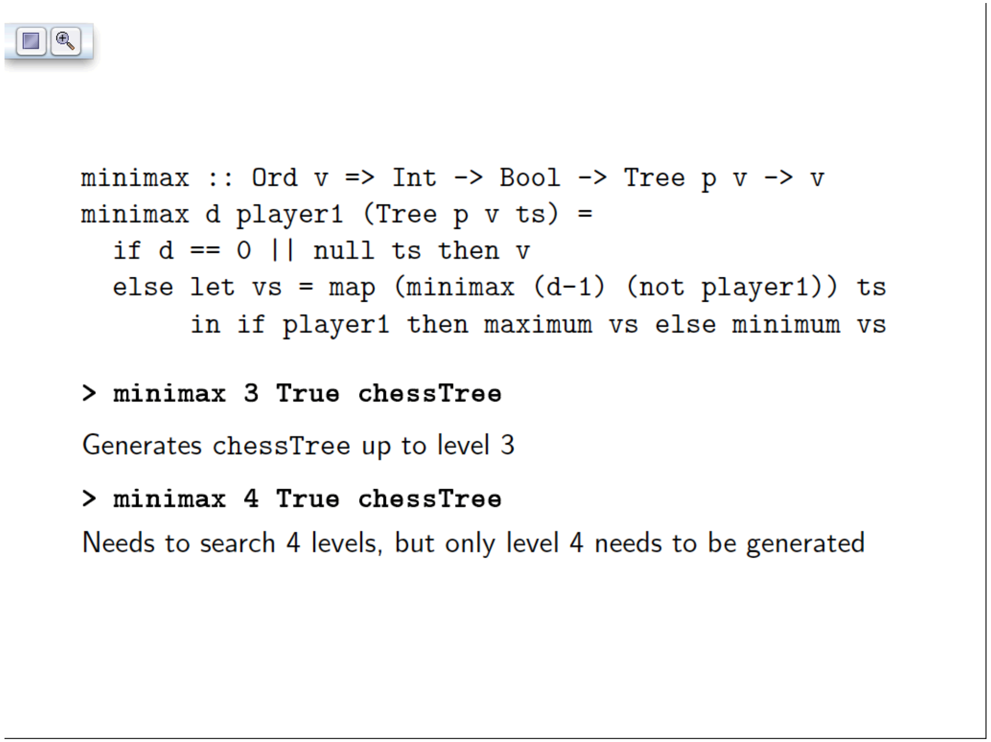
- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice
- Supports infinitely broad and deep trees (useful??)

```
gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
gameTree next val = tree where
  tree p = Tree p (val p) (map tree (next p))

chessTree = gameTree ...
```



```
minimax :: Ord v => Int -> Bool -> Tree p v -> v
```



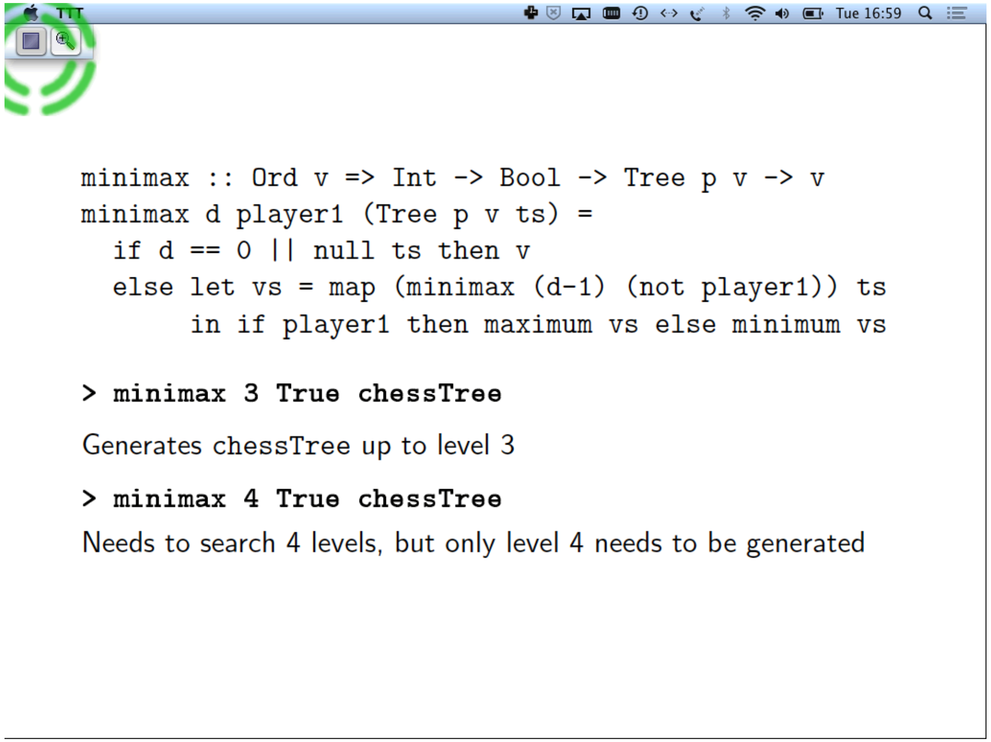
```
minimax :: Ord v => Int -> Bool -> Tree p v -> v
minimax d player1 (Tree p v ts) =
  if d == 0 || null ts then v
  else let vs = map (minimax (d-1) (not player1)) ts
       in if player1 then maximum vs else minimum vs
```

> minimax 3 True chessTree

Generates chessTree up to level 3

> minimax 4 True chessTree

Needs to search 4 levels, but only level 4 needs to be generated



```
minimax :: Ord v => Int -> Bool -> Tree p v -> v
minimax d player1 (Tree p v ts) =
  if d == 0 || null ts then v
  else let vs = map (minimax (d-1) (not player1)) ts
       in if player1 then maximum vs else minimum vs
```

> minimax 3 True chessTree

Generates chessTree up to level 3

> minimax 4 True chessTree

Needs to search 4 levels, but only level 4 needs to be generated