12. Lazy evaluation

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
- Terminates as often as possible

Therefore Haskell is called a _lazy functional language_.

Evaluating expressions

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

Example:

\[
\text{sq :: Integer -> Integer} \\
\text{sq \, n \, = \, n \, \ast \, n}
\]

One evaluation:

\[
\text{sq(3+4) = sq \, 7 \, = \, 7 \, \ast \, 7 \, = \, 49}
\]
Evaluating expressions

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

Example:

\[
\text{sq :: Integer } \rightarrow \text{ Integer} \\
\text{sq \ n \ = \ n * n}
\]

One evaluation:

\[
\text{sq(3+4) = sq 7 = 7 * 7 = 49}
\]

Another evaluation:

\[
\text{sq(3+4) = (3+4) * (3+4) = 7 * (3+4) = 7 * 7 = 49}
\]

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:
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:
\[
\text{\( n + (n := 1) \)}
\]

Evaluation strategies
An expression may have many reducible subexpressions:

\[
\text{\( \text{sq (3+4)} \)}
\]

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

Example:
\[
\text{\( \text{sq :: Integer -> Integer} \)}
\]
\[
\text{\( \text{sq \ n = n \times n} \)}
\]

One evaluation:
\[
\text{\( \text{sq(3+4) = sq 7 = 7 \times 7 = 49} \)}
\]

Another evaluation:
\[
\text{\( \text{sq(3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49} \)}
\]
Reduction strategies

An expression may have many reducible subexpressions:

\[ \text{sq} (3+4) \]

Terminology: \textit{redex} = reducible expression

Two common reduction strategies:

\textbf{Innermost reduction} Always reduce an innermost redex.
- Corresponds to \textit{call by value}:
  - Arguments are evaluated before they are substituted into the function body
  - \[ \text{sq} (3+4) = \text{sq} 7 = 7 \times 7 \]

\textbf{Outermost reduction} Always reduce an outermost redex.
- Corresponds to \textit{call by name}:
Reduction strategies

An expression may have many reducible subexpressions:

\[ \text{sq} \ (3+4) \]

Terminology: \textit{redex} = \textit{reducible expression}

Two common reduction strategies:

\textbf{Innermost reduction} Always reduce an innermost redex.
- Corresponds to \textit{call by value}:
- Arguments are evaluated before they are substituted into the function body
  \[ \text{sq} \ (3+4) = \text{sq} \ 7 = 7 \times 7 \]

\textbf{Outermost reduction} Always reduce an outermost redex.
- Corresponds to \textit{call by name}:
- The unevaluated arguments are substituted into the function body
  \[ \text{sq} \ (3+4) = (3+4) \times (3+4) \]

Comparison: termination

Definition:
\[ \text{loop} = \text{tail loop} \]

Innermost reduction:
\begin{align*}
\text{fst} (1, \text{loop}) &= \text{fst}(1, \text{tail loop}) \\
&= \text{fst}(1, \text{tail}(\text{tail loop})) \\
&= \ldots
\end{align*}

Outermost reduction:
Comparison: termination

Definition:
loop = tail loop

Innermost reduction:
\[ \text{fst } (1, \text{loop}) = \text{fst}(1, \text{tail loop}) \]
\[ = \text{fst}(1, \text{tail(tail loop)}) \]
\[ = \ldots \]

Outermost reduction:
\[ \text{fst } (1, \text{loop}) = 1 \]

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

\textbf{Outermost reduction terminates as often as possible}

Comparison: termination

Definition:
loop = tail loop

Innermost reduction:
\[ \text{fst } (1, \text{loop}) = \text{fst}(1, \text{tail loop}) \]
\[ = \text{fst}(1, \text{tail(tail loop)}) \]
\[ = \ldots \]

Outermost reduction:
\[ \text{fst } (1, \text{loop}) = 1 \]

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

Why is this useful?
Why is this useful?

Example
Can build your own control constructs:

```haskell
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:
\[ \text{sq}(3+4) = \text{sq} \ 7 = 7 \times 7 = 49 \]

Outermost reduction:
\[ \text{sq}(3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49 \]
sq(3+4) = \bullet \ast \bullet = \ast \ast = 49

The expression 3+4 is only evaluated once!

sq(3+4) = \bullet \ast \bullet = \ast \ast = 49

The expression 3+4 is only evaluated once!

Lazy evaluation := outermost reduction + sharing

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 \texttt{fst (1,loop)})

- 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 \texttt{fst (1,loop)})
- Each argument is evaluated at most once (sharing!)

Pattern matching

Example

\texttt{f :: [Int] \rightarrow [Int] \rightarrow Int}
\begin{align*}
  f [] & \quad ys = 0 \\
  f (x:xs) [] & \quad = 0 \\
  f (x:xs) (y:ys) & \quad = x+y \\
\end{align*}

Example

\texttt{f :: [Int] \rightarrow [Int] \rightarrow Int}
\begin{align*}
  f [] & \quad ys = 0 \\
  f (x:xs) [] & \quad = 0 \\
  f (x:xs) (y:ys) & \quad = x+y \\
\end{align*}

Lazy evaluation:
\texttt{f [1..3] [7..9]}
Pattern matching

Example

\[ f :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Int} \]
\[ f \text{ [] } ys = 0 \]
\[ f \text{ (x:xs) [] } = 0 \]
\[ f \text{ (x:xs) (y:ys) } = x+y \]

Lazy evaluation:
\[ f \text{ [1..3] [7..9] } \quad \text{-- does } f\text{.1 match?} \]
\[ = f \text{ (1 : [2..3]) [7..9] } \quad \text{-- does } f\text{.2 match?} \]
Pattern matching

Example

\[ f :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Int} \]
\[ f \ [] \quad y s \quad = \quad 0 \]
\[ f \ (x:xs) \ [] \quad = \quad 0 \]
\[ f \ (x:xs) \ (y:ys) \quad = \quad x + y \]

Lazy evaluation:
\[ f \ [1..3] \ [7..9] \quad -- \text{does } f.1 \text{ match?} \]
\[ = \ f \ (1 : \ [2..3]) \ [7..9] \quad -- \text{does } f.2 \text{ match?} \]
\[ = \ f \ (1 : \ [2..3]) \ (7 : \ [8..9]) \quad -- \text{does } f.3 \text{ match?} \]
\[ = \ 1 + 7 \]
\[ = \ 8 \]

Guards

Example

\[ f \ m \ n \ p \ | \ m >= n \&\& m >= p \Rightarrow p = m \]
\[ | \ n >= m \&\& n >= p \Rightarrow p = n \]
\[ | \ \text{otherwise} \Rightarrow p = p \]

Lazy evaluation:
\[ f \ (2+3) \ (4-1) \ (3+9) \]
Example

\[
\begin{align*}
  f & \quad m & \quad n & \quad p \\
  | & \quad m \geq n & \& & m \geq p = m \\
  | & \quad n \geq m & \& & n \geq p = n \\
  | & \quad \text{otherwise} & = & p
\end{align*}
\]

Lazy evaluation:
\[
\begin{align*}
  f & \quad (2+3) & \quad (4-1) & \quad (3+9) \\
  ? & \quad 2+3 \geq 4-1 & \& & 2+3 \geq 3+9
\end{align*}
\]

Example

\[
\begin{align*}
  f & \quad m & \quad n & \quad p \\
  | & \quad m \geq n & \& & m \geq p = m \\
  | & \quad n \geq m & \& & n \geq p = n \\
  | & \quad \text{otherwise} & = & p
\end{align*}
\]

Lazy evaluation:
\[
\begin{align*}
  f & \quad (2+3) & \quad (4-1) & \quad (3+9) \\
  ? & \quad 2+3 \geq 4-1 & \& & 2+3 \geq 3+9 \\
  ? & \quad 5 \geq 3 & \& & 5 \geq 3+9 \\
  ? & \quad \text{True} & \& & 5 \geq 3+9 \\
  ? & \quad 5 \geq 3+9 \\
  ? & \quad 5 \geq 12 \\
  ? & \quad \text{False}
\end{align*}
\]
Example

\[
\begin{align*}
\text{f m n p} & | m > n \land m \geq p = m \\
& | n > m \land n \geq p = n \\
& | \text{otherwise} = p
\end{align*}
\]

Lazy evaluation:

\[
\begin{align*}
\text{f (2+3) (4-1) (3+9)} \\
\ ? \ 2+3 >= 4-1 \land 2+3 >= 3+9 \\
\ ? \ = 5 >= 3 \land 5 >= 3+9 \\
\ ? \ = \text{True} \land 5 >= 3+9 \\
\ ? \ = 5 >= 3+9 \\
\ ? \ = 5 >= 12 \\
\ ? \ = \text{False} \\
\ ? \ 3 >= 5 \land 3 >= 12 \\
\ ? \ = \text{False} \land 3 >= 12 \\
\ ? \ = \text{False} \\
\ ? \ \text{otherwise} = \text{True}
\end{align*}
\]

Example

\[
\begin{align*}
\text{f m n p} & | m > n \land m \geq p = m \\
& | n > m \land n \geq p = n \\
& | \text{otherwise} = p
\end{align*}
\]

Lazy evaluation:

\[
\begin{align*}
\text{f (2+3) (4-1) (3+9)} \\
\ ? \ 2+3 >= 4-1 \land 2+3 >= 3+9 \\
\ ? \ = 5 >= 3 \land 5 >= 3+9 \\
\ ? \ = \text{True} \land 5 >= 3+9 \\
\ ? \ = 5 >= 3+9 \\
\ ? \ = 5 >= 12 \\
\ ? \ = \text{False} \\
\ ? \ 3 >= 5 \land 3 >= 12 \\
\ ? \ = \text{False} \land 3 >= 12 \\
\ ? \ = \text{False} \\
\ ? \ \text{otherwise} = \text{True}
\end{align*}
\]

Guard

where

Same principle: definitions in where clauses are only evaluated when needed and only as much as needed.
Haskell never reduces inside a lambda

Example: \( x \rightarrow \text{False} \land x \) cannot be reduced

Reasons:
- Functions are black boxes
- All you can do with a function is apply it
Haskell never reduces inside a lambda

Example: \( x \rightarrow \text{False} \land x \) cannot be reduced
Reasons:
- Functions are black boxes
- All you can do with a function is apply it

Example:
\[(\lambda x \rightarrow \text{False} \land x) \text{True} = \text{False} \land \text{True} = \text{False}\]

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

Example
\[3 \ast 5 \text{ is a redex}\]

Predefined functions from Prelude

They behave like their Haskell definition:

\[\&\& : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}\]
\[\text{True} \&\& y = y\]
\[\text{False} \&\& y = \text{False}\]
Lazy evaluation evaluates an expression only when needed and only as much as needed.

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

(“Call by need”)
Minimum of a list

min = head . insSort

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

min = head . insSort

insSort [] = []
insSort (x:xs) = ins x (insSort xs)

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

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

⇒ insSort [6,1,7,5]
   = ins 6 (ins 1 (ins 7 (ins 5 [])))

⇒ insSort [6,1,7,5]
   = ins 6 (ins 1 (ins 7 (ins 5 [])))
\[ \text{min } [6,1,7,5] = \text{head}(\text{inSort } [6,1,7,5]) \\
= \text{head}(\text{ins } 6 (\text{ins } 1 (\text{ins } 7 (\text{ins } 5 [])))) \\
= \text{head}(\text{ins } 6 (\text{ins } 1 (\text{ins } 7 (5 : [])))) \]
Minimum of a list

\[
\text{min} \ = \ \text{head} \ . \ \text{insSort}
\]
\[
\text{insSort} :: \text{Ord} \ a \Rightarrow [a] \rightarrow [a]
\]
\[
\text{insSort} \ [[] \ = \ []
\]
\[
\text{insSort} \ (x:x:xs) \ = \ \text{ins} \ x \ (\text{insSort} \ xs)
\]
\[
\text{ins} :: \text{Ord} \ a \Rightarrow a \rightarrow [a] \rightarrow [a]
\]
\[
\text{ins} \ x \ [[] \ = \ [x]
\]
\[
\text{ins} \ x \ (y:ys) \ | \ x <= y \ = \ x : y : ys
\]
\[
| \ \text{otherwise} \ = \ y : \text{ins} \ x \ ys
\]

\[
\text{min} \ [6,1,7,5] \ = \ \text{head} \ (\text{insSort} \ [6,1,7,5])
\]
\[
= \ \text{head} \ (\text{ins} \ 6 \ (\text{ins} \ 1 \ (\text{ins} \ 7 \ ([] )))))
\]
\[
= \ \text{head} \ (\text{ins} \ 6 \ (\text{ins} \ 1 \ (\text{ins} \ 7 \ ([] )))))
\]
\[
= \ \text{head} \ (\text{ins} \ 6 \ (1 : 5 : \text{ins} \ 7 \ []))
\]

\[
\text{min} \ [6,1,7,5] \ = \ \text{head} \ (\text{insSort} \ [6,1,7,5])
\]
\[
= \ \text{head} \ (\text{ins} \ 6 \ (\text{ins} \ 1 \ (\text{ins} \ 7 \ ([] )))))
\]
\[
= \ \text{head} \ (\text{ins} \ 6 \ (\text{ins} \ 1 \ (\text{ins} \ 7 \ ([] )))))
\]
\[
= \ \text{head} \ (\text{ins} \ 6 \ (1 : 5 : \text{ins} \ 7 \ []))
\]
\[
= \ \text{head} \ (1 : \text{ins} \ 6 \ (5 : \text{ins} \ 7 \ []))
\]
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

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

\[ \text{max} = \text{last \ inSort} \]

Takeuchi Function

\[
\begin{align*}
t &: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
t \ x \ y \ z &| \ x \leq y \ = \ y \\
&| \ \text{otherwise} \ = \ t \ (t \ (x-1) \ y \ z) \\
&| \ \quad \ (t \ (y-1) \ z \ x) \\
&| \ \quad \ (t \ (z-1) \ x \ y)
\end{align*}
\]

In C:

```c
int t(int x, int y, int z) {
    if (x <= y)
        return y;
    else
        return t(t(x-1, y, z), t(y-1, z, x), t(z-1, x, y));
}
```
Takeuchi Function

t :: Int -> Int -> Int

\[ t(x, y, z) = \begin{cases} x & \text{if } x \leq y \\ t(x-1, y, z) & \text{otherwise} \end{cases} \]

In C:

```c
int t(int x, int y, int z) {
    if (x <= y)
        return y;
    else
        return t(x-1, y, z); // Note: The third argument is z, not y-1 as shown.
}
```
Takeuchi Function

\[ t : \text{Int} \to \text{Int} \to \text{Int} \to \text{Int} \]
\[ t \; x \; y \; z \mid x \leq y = y \]
\[ \text{otherwise} = t \; (t \; (x-1) \; y \; z) \]
\[ (t \; (y-1) \; z \; x) \]
\[ (t \; (z-1) \; x \; y) \]

In C:

```c
int t(int x, int y, int z) {
    if (x <= y)
        return y;
    else
        return t(t(x-1, y, z), t(y-1, z, x), t(z-1, x, y));
}
```

Try \texttt{t 15 10 0} — Haskell beats C!
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]

Haskell lists can be finite or infinite

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

Haskell lists can be finite or infinite
Printing an infinite list does not terminate
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

Lazy evaluation computes as much of the infinite list as needed

This is how partially evaluated lists are represented internally:
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 : code pointer to compute rest
```

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
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

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

In Haskell:

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

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

3 5 7 9 11 2
In Haskell:

```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:

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

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

Lazy evaluation:

```haskell
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:

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

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

Lazy evaluation:

```haskell
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]
= ...
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]`
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

Nothing special, just the automatic result of sharing

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 b1, f a2 b2, ...]

fibs :: [Integer]
fibs = 0 :

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 b1, f a2 b2, ...]

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

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