Haskell

- **functional** — programs are function definitions; functions are “first-class citizens”
- **pure** (referentially transparent) — “no side-effects”
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading

- Standardised language version: **Haskell 98**
- Several compilers and interpreters available
- Comprehensive web site: [http://haskell.org/](http://haskell.org/)

Simple Expression Evaluation

The Haskell interpreters hugs, ghci, and hi accept any expression at their prompt and print (after the first ENTER) the value resulting from evaluation of that expression.

Prelude> 4*(5+6)-2
42

Expression evaluation proceeds by applying rules to subexpressions:

\[
4 \times (5+6) - 2 \\
= 4 \times 11 - 2 \\
= 44 - 2 \\
= 42
\]

Unfolding Definitions

Assume the following definitions to be in scope:

\[
\text{answer} = 42 \\
\text{magic} = 7
\]

Expression evaluation will **unfold** (or **expand**) definitions:

Prelude> (answer - 1) * (magic * answer - 23)
11111

\[
(\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23) \\
= (42 - 1) \times (7 \times 42 - 23) \\
= 41 \times 291 \\
= 11111
\]

How did I find those numbers?

Easy!

Prelude> \[n \mid n <- [1..400], 11111 \mod n == 0 \] \[1,41,271\]

This is a list comprehension:

- return all \( n \)
- where \( n \) is taken from then list \([1..400]\)
- and a result is returned only if \( n \) divides 11111.
Expanding Function Definitions

perimeter :: Double -> Double
perimeter \( r \) = 2 * \( r \) * \( \pi \)

square :: Integer -> Integer
square \( x \) = \( x \) * \( x \)

\[
\begin{align*}
\text{perimeter } (1 + 2) &= 2 * (1 + 2) * \pi \\
&= 2 * 3 * \pi \\
&= 6 * \pi \\
&= 18.84955592153876
\end{align*}
\]

\[
\begin{align*}
\text{square } (1 + 2) &= (1 + 2) * (1 + 2) \\
&= 3 * 3 \\
&= 9
\end{align*}
\]

Matching Function Definitions

fact :: Integer -> Integer
fact 0 = 1
fact \( n \) = \( n \) * \( \text{fact} \ (n-1) \)

\[
\begin{align*}
\text{fact } 3 &= 3 * \text{fact } (3-1) \\
&= 3 * \text{fact } 2 \\
&= 3 * (2 * \text{fact } (2-1)) \\
&= 3 * (2 * (2 * \text{fact } (1-1))) \\
&= 3 * (2 * (1 * \text{fact } (1-0))) \\
&= 3 * (2 * (1 * \text{fact } 0)) \\
&= 3 * (2 * (1 * 1)) \\
&= 3 * (2 * 1) \\
&= 3 * 2 \\
&= 6
\end{align*}
\]

Simple Expression Evaluation — Explanation

- Arguments to a function or operation are evaluated only when needed.
- If for obtaining a result from an application of a function \( f \) to a number of arguments, the value of the argument at position \( i \) is always needed, then \( f \) is called strict in its \( i \)-th argument.
- Therefore: If \( f \) is strict in its \( i \)-th argument, then the \( i \)-th argument has to be evaluated whenever a result is needed from \( f \).
- Simpler: A one-argument function \( f \) is strict iff \( f \ \text{undefined} = \text{undefined} \).
  - Constant functions are non-strict: \( (\text{const } 5) \ \text{undefined} = 5 \)
  - Checking a list for emptiness is strict: \( \text{null} \ \text{undefined} = \text{undefined} \)
  - List construction is non-strict: \( \text{null} (\text{undefined} : \text{undefined}) = \text{False} \)
  - Standard arithmetic operators are strict in both arguments: \( 0 * \text{undefined} = \text{undefined} \)

Conditional Expressions

\[
\text{Prelude}\geq \text{if } 11111 \mod 41 = 0 \text{ then } 11111 \div 41 \text{ else } 5 \\
= 271
\]

The pattern is:

\[
\text{if } \text{condition} \ \text{then} \ \text{expression1} \ \text{else} \ \text{expression2}
\]

- If the condition evaluates to True, the conditional expression evaluates to the value of \( \text{expression1} \).
- If the condition evaluates to False, the conditional expression evaluates to the value of \( \text{expression2} \).
- If the condition does not evaluate to anything, the conditional expression also does not evaluate to anything.

Therefore: “if _ then _ else _” is strict in the condition.

In C: \( (\text{condition} ? \text{expression1} : \text{expression2}) \)
Expanding Function Definitions

```haskell
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```haskell
fact 3
= if 3 == 0 then 1 else 3 * fact (3-1)
= if False then 1 else 3 * fact 2
= 3 * fact (3-1)
= 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
= 3 * if 2 == 0 then 1 else 2 * fact (2-1)
= 3 * if False then 1 else 2 * fact (2-1)
= 3 * 2 * fact (2-1)
= 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
= 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
= 3 * 2 * if False then 1 else 1 * fact (1-1)
= 3 * 2 * 1 * fact (1-1)
= 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
= 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
= 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
= 3 * 2 * 1 * 1
= 3 * 2 * 1
= 3 * 2
= 6
```

List Construction

Display and enumeration lists are *syntactic sugar*: A list is
- either the **empty list**: `[]`,
- or **non-empty**, and constructed from a **head** `x` and a **tail** `xs` (read: “xes”) `x : xs` — read: “`x cons xs`”.

“:” is used as **infix list constructor**:

\[
\begin{align*}
1 &: [2, 3] &= [1, 2, 3] \\
\end{align*}
\]

As an infix operator, “:” **associates to the right**:

```haskell
x : y : ys = x : (y : ys)
```

Example:

\[
1 : 2 : [3,4] = 1 : (2 : [3,4]) = 1 : [2,3,4] = [1,2,3,4]
\]

Cons is Not Associative

The convention that “:” **associates to the right** allows to save parentheses in certain circumstances.

However, “:” is **not** associative:

- A list of integers:
  \[
  1 : (2 : [3,4]) = 1 : 2 : [3,4] = [1, 2, 3, 4]
  \]
  \[
  (1 : 2) : [3,4] \text{ is nonsense, since 2 is not a list!}
  \]
- A list of lists of integers:
  \[
  [2] : [[3,4,5],[6,7]] = [[2],[3,4,5],[6,7]]
  \]
  \[
  \text{Another list of lists of integers:}
  (1 : [2]) : [[[3,4,5],[6,7]]] = [[[1,2],[3,4,5],[6,7]]]
  \]
  \[
  1 : ([2] : [[3,4,5],[6,7]]) \text{ is nonsense again!}
  \]
  \[
  \text{Reason: 1 and [2] cannot be members of the same list (type error).}
  \]
List Comprehensions

General shape:

\[
\left\{ \text{term} \mid \text{generator} \mid \{, \text{generator_or_constraint} \}^* \right\}
\]

Examples:

\[
\left\{ n \times n \mid n \leftarrow [1 \ldots 5] \right\} = [1, 4, 9, 16, 25]
\]
\[
\left\{ n \times n \mid n \leftarrow [1 \ldots 10], \text{even } n \right\} = [4, 16, 36, 64, 100]
\]
\[
\left\{ m \times n \mid m \leftarrow [1.3.5], n \leftarrow [2.4.6] \right\} = [2.4.6.12.18.10.20.30]
\]

Note:

- The left generator “generates slower”.

- Haskell code fragments will frequently be presented like above in a form that is more readable than plain typewriter text — in that case, the “comes from” arrow “<-” in generators turns into “←”

The Type Language

Haskell has a full-fledged type language, with

- Simple predefined datatypes: Bool, Char, Integer, ...
- Predefined type constructors: lists, tuples, functions, ...
- Type synonyms
- User-defined datatypes and type constructors
- Type variables — to express parametric polymorphism
- ...

Simple Predefined Datatypes

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>truth values</td>
<td>False, True</td>
</tr>
<tr>
<td>Char</td>
<td>“Unicode” characters</td>
<td>(in GHC: ISO-10646)</td>
</tr>
<tr>
<td>Integer</td>
<td>integers</td>
<td>arbitrary precision</td>
</tr>
<tr>
<td>Int</td>
<td>“machine integers”</td>
<td>≥ 32 bits</td>
</tr>
<tr>
<td>Float</td>
<td>real floating point</td>
<td>single precision</td>
</tr>
<tr>
<td>Double</td>
<td>real floating point</td>
<td>double precision</td>
</tr>
<tr>
<td>Complex Float</td>
<td>complex floating point</td>
<td>single precision</td>
</tr>
<tr>
<td>Complex Double</td>
<td>complex floating point</td>
<td>double precision</td>
</tr>
</tbody>
</table>

Important Points

- Execution of Haskell programs is expression evaluation
- Defining functions in Haskell is more like defining functions in mathematics than like defining procedures in C or classes and methods in Java
- One Haskell function may be defined by several “equations” — the first that matches is used.
- Lists are an easy-to-use datastructure with lots of language and library support.
  For this reason, lists are heavily used especially in beginners’ material.
  In many cases, advanced Haskell programmers will use other datastructures, for example FiniteMaps instead of association lists.
**List Types**

If \( t \) is a type, then the **list type** \([t]\) is the type of **lists** with elements of type \( t \).

```haskell
answer :: Integer
answer = 42
limit :: Int
limit = 100
```

Then:

- \([ 1, 2, 3, answer]\) :: \([\text{Integer}]\)
- \([ 1 .. \text{limit}\] \) :: \([\text{Int}]\)
- \([ [ 1 .. \text{limit}\] , [ 2 .. \text{limit}\] ] :: [[\text{Int}]]
- \([ 'h', 'e', 'l', 'l', 'o' ]\) :: \([\text{Char}]\)
- \("hello" \) :: \([\text{Char}]\)
- \([ "hello", "world" ]\) :: [[\text{Char}]]
- \(["first", "line"], ["second", "line"]\] :: [[[\text{Char}]]]

**Product Types (Pairs)**

If \( t \) and \( u \) are types, then the **product type** \((t, u)\) is the type of **pairs** with first component of type \( t \) and second component of type \( u \) (mathematically: \( t \times u \)).

**Examples:**

- (answer, limit) :: (Integer, Int)
- (limit, answer) :: (Int, Integer)
- ("??", answer) :: ([Char], Integer)
- ("??", (limit, answer)) :: ([Char], (Int, Integer))
- ("??", 'X') :: ([Char], Char)
- (limit, ("??", 'X')) :: (Int, ([Char], Char))
- (True, (["X", limit], ("Y", 5)]) :: (Bool, ([[Char], Int]])

**Tuple Types**

If \( n \neq 1 \) is a natural number and \( t_1, \ldots, t_n \) are types, then the **tuple type** \((t_1, \ldots, t_n)\) is the type of \( n \)-tuples with the \( i \)th component of type \( t_i \).

**Examples:**

- (answer, 'c', limit) :: (Integer, Char, Int)
- (answer, 'c', limit, "all") :: (Integer, Char, Int, [Char])
- () :: ()

---

**Simple Type Synonyms**

If \( t \) is a type not containing any type variables, and \( Name \) is an identifier with a capital first letter, then

```haskell
type Name = t
```

defines \( Name \) as a **type synonym** for \( t \), i.e., \( Name \) can now be used interchangeably with \( t \).

**Examples:**

- type String = [Char] -- **predefined**
- type Point = (Double, Double) -- (1.5, 2.7)
- type Triangle = (Point, Point, Point)
- type CharEntity = (Char, String) -- ('Ā', "āuml;")
- type Dictionary = [(String, String)] -- [("day", "jour")]}
Type Variables and Polymorphic Types

• Identifiers with lower-case first letter can be used as type variables.

• Type variables can be used like other types in the construction of types, e.g.:
  
  \[(a,b)\]
  (Bool, (a, Int))
  [(String, [(key, val)])]

• A type containing at least one type variable is called **polymorphic**

• Polymorphic types can be instantiated by instantiating type variables with types, e.g.:
  
  \[(a,b)\] ⇒ [(Char, b)]
  \[(a,b)\] ⇒ [(Char, Int)]
  \[(a,b)\] ⇒ [(a, [(String, Int)])]
  \[(a,b)\] ⇒ [(a, [(String, c)])]

Typing of List Construction

• The empty list can be used at any list type: [] :: [a]

• If an element x :: a and a list xs :: [a] are given, then
  
  (x : xs) :: [a]

Examples:

2 :: Int
[] :: [Int]
[2] = 2 :: []
[[3,4,5],[6,7]] :: [[[Int]]]
[2] :: [[3,4,5],[6,7]] :: [[[Int]]]
1 : ([2] : [[3,4,5],[6,7]]) -- cannot be typed!

Function Types and Function Application

If t and u are types, then the **function type** t->u is the type of all functions accepting arguments of type t and producing results of type u (mathematically: \( t \to u \)).

Then:

• If a function \( f :: a \to b \) and an argument \( x :: a \) are given, then we have \( (f x) :: b \).

• If a function \( f :: a \to b \) is given and we know that \( (f x) :: b \), then the argument x is used at type a.

• If an argument \( x :: a \) is given and we know that \( (f x) :: b \), then the function \( f \) is used at type \( a \to b \).

Type Inference Examples

\(\text{fst} :: (a,b) \to a\)
\(\text{fst} (x,y) = x\)

\(\text{fst} ('c', \text{False}) :: \text{Char}\)

\(["hello", \text{fst} (x, 17)] \Rightarrow x :: \text{String}\)

\(\text{f p = limit + fst p} \Rightarrow p :: \text{(Int,a)}\)

\(\text{f :: (Int,a) \to Int}\)

\(\text{g h = fst (h "") :: [limit]} \Rightarrow h :: \text{String \to (Int,a)}\)
Let’s Play the Evaluation Game Again — 1

```haskell```

```haskell
h1 :: String -> (Int, String)
    h1 str = (length str, ' ' : str)

    g h1 = fst (h1 "") : [limit]

    Then:
    g h1
    = fst (h1 "") : [limit]
    = fst (length "", ' ' : "") : [limit]
    = length "" : [limit]
    = 0 : [limit]
    = [0, 100]
```

Let’s Play the Evaluation Game Again — 2

```haskell```

```haskell
h2 :: String -> (Int, Char)
    h2 str = (sum (map ord (notOccCaps str)), head str)

    notOccCaps :: String -> String
    notOccCaps str = filter ('notElem' str) ['A' .. 'Z']

    g h2 = fst (h2 "") : [limit]

    Then:
    g h2
    = fst (h2 "") : [limit]
    = fst (sum (map ord (notOccCaps "")), head "") : [limit]
    = sum (map ord (notOccCaps "")) : [limit]
    = ... = 2015 : [limit]
    = [2015, 100]
```

Higher-Order Functions

```haskell```

```haskell
g h = fst (h "") : [limit]

Functional Programming: Functions are first-class citizens

- Functions can be arguments of other functions: g h2
- Functions can be components of data structures: (7, h1), [h1, h2]
- Functions can be results of function application: succ . succ

A first-order function accepts only non-functional values as arguments.

A higher-order function expects functions as arguments.

```haskell```

```haskell
g is a second-order function: it expects first-order functions like h1, h2 as arguments.
```

map and filter

```haskell```

```haskell
map :: (a -> b) -> ([a] -> [b])
    map f [] = []
    map f (x:xs) = f x : map f xs

    filter :: (a -> Bool) -> ([a] -> [a])
    filter p [] = []
    filter p (x : xs) = if p x then x : rest else rest
        where rest = filter p xs

    These functions could also be defined via list comprehension:
    map f xs = [ f x | x <- xs ]
    filter p xs = [ x | x <- xs, p x ]

    Examples:
    map (7 *) [1 .. 6] = [7, 14, 21, 28, 35, 42]
    filter even [1 .. 6] = [2, 4, 6]```
### Operator Sections

- Infix operators are turned into functions by surrounding them with parentheses:
  
  \[
  (+) \ 2 \ 3 \ = \ 2 + 3
  \]

- This is necessary in type declarations:

  \[
  (+) \ :: \ Int -> Int -> Int \quad -- \ not \ the \ "natural" \ type \ of \ (+)
  \]

  \[
  (:) \ :: \ a \ -> \ [a] \ -> \ [a]
  \]

- It is also possible to supply only one argument (which has to be an atomic expression):

  \[
  (2 \ +) \ 3 \ = \ 2 + 3 \ = \ (+ 3) \ 2
  \]

  \[
  (8,3 \ /) \ 2.5 \ = \ 8.3 / 2.5 \ = \ (/ 2.5) \ 8.3
  \]

  \[
  (7 \ :) \ [] \ = \ 7 \ : \ [] \ = \ (: [] \ 7
  \]

  \[
  ((2^17) :) \ (16:[])) = (2^17) : 16 : [] = (: (16:[])) \ (2^17)
  \]

### Type Inference Examples

```haskell
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False) :: Char

("hello", fst (x, 17)) \Rightarrow x :: String

f p = limit + fst p \Rightarrow p :: (Int,a)
   f :: (Int,a) -> Int
```

### Curried Functions

- **Function application associates to the left**, i.e.,

  \[
  f \times y = (f \times) y
  \]

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., “they accept their arguments one at a time”:

  \[
  cylVol \ r \ h = (\pi \ :: \ Double) \ * \ r \ * \ r \ * \ h
  \]

  Since the right-hand side, \( r \) and \( h \) obviously all have type \( \text{Double} \), we have;

  \[
  (cylVol \ r) :: \text{Double} \rightarrow \text{Double}
  \]

  \[
  cylVol :: \text{Double} \rightarrow (\text{Double} \rightarrow \text{Double})
  \]

- **Function type construction associates to the right**, i.e.,

  \[
  a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)
  \]

### “Partial Application”

Let values with the following types be given:

```haskell
f :: a \rightarrow b \rightarrow c
x :: a
y :: b
```

The type of \( f \) is the function type \( a \rightarrow (b \rightarrow c) \), with

- argument type \( a \),
- result type \( b \rightarrow c \).

Therefore, we can apply \( f \) to \( x \) and obtain:

\[
(f \ x) :: b \rightarrow c
\]

The application of a “two-argument function” to a single argument is a “one-argument function”, which can then be applied to a second argument:

\[
(f \ x) \ y :: c = f \ x \ y
\]
Partial Application — Example

\[ g :: (\text{String} \to (\text{Int}, \text{a})) \to [\text{Int}] \]
\[ g \ h = \text{fst} (h \ "") : [\text{limit}] \]

\[ k :: \text{Int} \to \text{String} \to (\text{Int}, \text{String}) \]
\[ k \ n \ str = (n * (\text{length} \ str + 1), \text{unwords} (\text{replicate} \ n \ str)) \]

\[ g \ (k \ 3) = \text{fst} (k \ 3 \ "") : [\text{limit}] \]
\[ = \text{fst} (3 * (\text{length} \ "" + 1), \text{unwords} (\text{replicate} \ 3 \ "")) : [\text{limit}] \]
\[ = (3 * (\text{length} \ "" + 1)) : [\text{limit}] \]
\[ = (3 * 1) : [\text{limit}] \]
\[ = 3 : [\text{limit}] \]
\[ = [3, 100] \]

Operations on Functions

\[ \text{id} :: \text{a} \to \text{a} \quad \text{-- identity function} \]
\[ \text{id} \ x = x \]

\[ (\ . \) :: (\text{b} \to \text{c}) \to (\text{a} \to \text{b}) \to (\text{a} \to \text{c}) \quad \text{-- function composition} \]
\[ (f \ . \ g) \ x = f \ (g \ x) \]

\[ \text{flip} :: (\text{a} \to \text{b} \to \text{c}) \to (\text{b} \to \text{a} \to \text{c}) \quad \text{-- argument swapping} \]
\[ \text{flip} \ f \ x \ y = f \ y \ x \]

\[ \text{curry} :: ((\text{a}, \text{b}) \to \text{c}) \to (\text{a} \to \text{b} \to \text{c}) \quad \text{-- currying} \]
\[ \text{curry} \ g \ x \ y = g \ (x,y) \]

\[ \text{uncurry} :: (\text{a} \to \text{b} \to \text{c}) \to ((\text{a}, \text{b}) \to \text{c}) \]
\[ \text{uncurry} \ f \ (x,y) = f \ x \ y \]

Exercise (necessary!): Copy only the definitions to a sheet of paper, and then infer the types yourself!

Turning Functions into Infix Operators

Surrounding a function name by backquotes turns it into an infix operator.

Frequently used examples (not the “natural” types throughout):

\[ \text{div}, \text{mod}, \text{max}, \text{min} :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ \text{elem} :: \text{Int} \to [\text{Int}] \to \text{Bool} \]

\[ 12 \ '\text{div}' \ 7 = 1 \]
\[ 12 \ '\text{mod}' \ 7 = 5 \]
\[ 12 \ '\text{max}' \ 7 = 12 \]
\[ 12 \ '\text{min}' \ 7 = 7 \]
\[ 12 \ '\text{elem}' \ [1 .. 10] = \text{False} \]

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via pattern matching:

\[ \text{null} :: [\text{a}] \to \text{Bool} \]
\[ \text{null} \ [\text{\_}] = \text{True} \]
\[ \text{null} \ (\text{x} : \text{xs}) = \text{False} \]

\[ \text{head} :: [\text{a}] \to \text{a} \]
\[ \text{head} \ (\text{x} : \text{xs}) = \text{x} \]

\[ \text{tail} :: [\text{a}] \to [\text{a}] \]
\[ \text{tail} \ (\text{x} : \text{xs}) = \text{xs} \]

(head and tail are partial functions — both are undefined on the empty list.)
Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via structural induction:

\[
\begin{align*}
\text{length} & : \{a\} \rightarrow \text{Int} \\
\text{length} \ [\] & = 0 \\
\text{length} \ (x : xs) & = 1 + \text{length} \ xs \\
\text{++} & : \{a\} \rightarrow \{a\} \rightarrow \{a\} \\
\text{++} \ [\] & = [\] \\
\text{++} \ (x : xs) & = x : (\text{++} \ xs) \\
\text{sum} & : \{a\} \rightarrow \text{Int} \\
\text{sum} \ [\] & = 0 \\
\text{sum} \ (x : xs) & = x + \text{sum} \ xs \\
\text{product} & : \{a\} \rightarrow \text{Int} \\
\text{product} \ [\] & = 1 \\
\text{product} \ (x : xs) & = x \ast \text{product} \ xs
\end{align*}
\]

\(x \ '\text{elem}' \ [\] = \text{False}\)
\(x \ '\text{elem}' \ (y : ys) = x \equiv y \ || \ x \ '\text{elem}' \ ys\)

(All these functions are in the standard prelude.)

Exercise: Positional List Splitting

- \(\text{take} : \text{Int} \rightarrow \{a\} \rightarrow \{a\}\)

  \(\text{take}\), applied to a \(k : \text{Int}\) and a list \(xs\), returns the longest prefix of \(xs\) of elements that has no more than \(k\) elements.

- \(\text{drop} : \text{Int} \rightarrow \{a\} \rightarrow \{a\}\)

  \(\text{drop} \ k \ xs\) returns the suffix remaining after \(\text{take} \ k \ xs\).

Laws:

- \(\text{take} \ k \ xs + \ \text{drop} \ k \ xs = xs\)

- \(\text{length} \ (\text{take} \ k \ xs) \leq k\)

Note: \(\text{splitAt} \ k \ xs = (\text{take} \ k \ xs, \ \text{drop} \ k \ xs)\)

Unfolding Definitions

A simple definition:

\[
\text{limit} = 10 \wedge 2
\]

Expanding this definition:

\[
4 \ast (\text{limit} + 1) = 4 \ast (10 \wedge 2 + 1) = \ldots
\]

Another definition:

\[
\text{concat} = \text{foldr} \ (\text{++}) \ [\]
\]

Expanding this definition:

\[
\text{concat} \ [[1,2,3], [4,5]] = (\text{foldr} \ (\text{++}) \ [\]) \ [[1,2,3], [4,5]] = \ldots
\]

Guarded Definitions

\[
\text{sign} \ x | \ x > 0 = 1 \\
| \ x == 0 = 0 \\
| x < 0 = -1
\]

\[
\text{choose} : \text{Ord} \ a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b
\]

\[
\text{choose} \ (x, v) \ (y, w) | x > y = v \\
| x < y = w \\
| \text{otherwise} = \text{error} \ "I cannot decide!"
\]

If no guard succeeds, the next pattern is tried:

\[
\text{take} \ 0 \ = \ [\] \\
\text{take} \ k \ | \ k < 0 = \text{error} \ "\text{take}: negative argument" \\
\text{take} \ k \ [\] = [\] \\
\text{take} \ k \ (x : xs) = x : \text{take} \ (k - 1) \ xs
\]

\[
\begin{align*}
\text{take} \ 2 \ [5, 6, 7] & = \text{take} \ 2 \ (5 : 6 : 7 : [\]) \\
& = 5 : \text{take} \ (2 - 1) \ (6 : 7 : [\]) \\
& = 5 : \text{take} \ 1 \ (6 : 7 : [\]) \\
& = 5 : 6 : \text{take} \ (1 - 1) \ (7 : [\]) \\
& = 5 : 6 : \text{take} \ 0 \ (7 : [\]) \\
& = 5 : 6 : [\] = [5, 6]
\end{align*}
\]