Programmation fonctionnelle

Mini-haddock

Table des matières

Certaines explications ont été simplifiées, notamment en spécialisant au type des listes certaines signatures utilisant les classes `Foldable` ou `Traversable` (comparez le type de `foldr` indiqué dans ce document avec celui indiqué par `ghci`).

1 `Control.Applicative`

This module describes a structure intermediate between a functor and a monad.

`class Functor f => Applicative f where`

A functor with application, providing operations to

— embed pure expressions (`pure`), and
— sequence computations and combine their results (`<*>`).

Methods

`pure :: a -> f a`

Lift a value.

`(<*>) :: f (a -> b) -> f a -> f b` (`infixl 4`)

Sequential application.

`class Applicative f => Alternative f where`

Methods

`empty :: f a`
The identity of `<|>`

`(<|>) :: f a -> f a -> f a`   (**infixl** 3)
An associative binary operation

`some :: f a -> f [a]`
One or more.

`many :: f a -> f [a]`
Zero or more.

`optional :: Alternative f => f a -> f (Maybe a)`
One or none.

2 Control.Monad

`class Applicative m => Monad m where`

From the perspective of a Haskell programmer, however, it is best to think of a monad as an *abstract datatype* of actions. Haskell's **do** expressions provide a convenient syntax for writing monadic expressions.

Instances of `Monad` should satisfy the following laws:

- `return a >>= k  =  k a`
- `m >>= return  =  m`
- `m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h`

Furthermore, the `Monad` and `Applicative` operations should relate as follows:

- `pure = return`
- `(<*>) = ap`

The above laws imply:

- `fmap f xs  =  xs >>= return . f`

Methods

`(>>=) :: m a -> (a -> m b) -> m b`   (**infixl** 1)
Sequentially compose two actions, passing any value produced by the first as an argument to the second.

`(>>) :: m a -> m b -> m b`   (**infixl** 1)
Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

Default implementation:

```
m >> k = m >>= \_ -> k
```

`return :: a -> m a`
Inject a value into the monadic type.

Default implementation:

```
return = pure
```

Instances

- `Monad []`
- `Monad Maybe`
- `Monad IO`

Basic `Monad` functions

`mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
: Map each element of a structure to a monadic action, evaluate these actions from left to right, and collect the results.

`mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`
: Map each element of a structure to a monadic action, evaluate these actions from left to right, and ignore the results.

`forM :: Monad m => [a] -> (a -> m b) -> m [b]`
: `forM` is `mapM` with its arguments flipped.

`forM_ :: Monad m => [a] -> (a -> m b) -> m ()`
: `forM_` is `mapM_` with its arguments flipped.

`sequence :: Monad m => [m a] -> m [a]`
: Evaluate each monadic action in the structure from left to right, and collect the results.

`sequence_ :: Monad m => [m a] -> m ()`
: Evaluate each monadic action in the structure from left to right, and ignore the results.

`(=<<) :: Monad m => (a -> m b) -> m a -> m b` **`(infixr 1)`**
: Same as `>>=`, but with the arguments interchanged.

`forever :: Applicative f => f a -> f b`
: `forever act` repeats the action infinitely.

Generalisations of list functions

`join :: Monad m => m (m a) -> m a`
: The join function is the conventional monad join operator. It is used to remove one level of monadic structure, projecting its bound argument into the outer level.

`replicateM :: Applicative m => Int -> m a -> m [a]`
: `replicateM n act` performs the action `n` times, gathering the results.

`replicateM_ :: Applicative m => Int -> m a -> m ()`
: Like `replicateM`, but discards the result.

Conditional execution of monadic expressions

`guard :: Alternative f => Bool -> f ()`
: `guard b` is `pure ()` if `b` is `True`, and `empty` if `b` is `False`.

`when :: Applicative f => Bool -> f () -> f ()`
: Conditional execution of Applicative expressions. For example,

```
when debug (putStrLn "Debugging")
```

will output the string `Debugging` if debug is `True`, and otherwise do nothing.

`unless :: Applicative f => Bool -> f () -> f ()`
: The reverse of when.

3 `Control.Parallel`

Parallel Constructs

`par :: a -> b -> b`
: Indicates that it may be beneficial to evaluate the first argument in parallel with the second. Returns the value of the second argument.

`a `par` b` is exactly equivalent semantically to `b`.

`par` is generally used when the value of a is likely to be required later, but not immediately. Also it is a good idea to ensure that a is not a trivial computation, otherwise the cost of spawning it in parallel overshadows the benefits obtained by running it in parallel.

`pseq :: a -> b -> b`

Semantically identical to `seq`, but with a subtle operational difference: `seq` is strict in both its arguments, so the compiler may, for example, rearrange ``a `seq` b`` into ``b `seq` a `seq` b``. This is normally no problem when using `seq` to express strictness, but it can be a problem when annotating code for parallelism, because we need more control over the order of evaluation; we may want to evaluate `a` before `b`, because we know that `b` has already been sparked in parallel with `par`.

This is why we have `pseq`. In contrast to `seq`, `pseq` is only strict in its first argument (as far as the compiler is concerned), which restricts the transformations that the compiler can do, and ensures that the user can retain control of the evaluation order.

4 Data.Bool

`data Bool :: *`

Constructors

- `False`
- `True`

`(&&) :: Bool -> Bool -> Bool` (infixr 3)

Boolean “and”

`(||) :: Bool -> Bool -> Bool` (infixr 2)

Boolean “or”

`not :: Bool -> Bool`

Boolean “not”

`otherwise :: Bool`

`otherwise` is defined as the value `True`. It helps to make guards more readable. eg.

```
f x | x < 0     = ...
    | otherwise = ...
```

5 Data.Char

`data Char :: *`

The character type `Char` is an enumeration whose values represent Unicode characters.

To convert a `Char` to or from the corresponding `Int` value defined by Unicode, use `toEnum` and `fromEnum` from the `Enum` class respectively.

`isSpace :: Char -> Bool`

Returns `True` for any Unicode space character, and the control characters `\t`, `\n`, `\r`, `\f`, `\v`.

`isLower :: Char -> Bool`

Selects lower-case alphabetic Unicode characters (letters).

`isUpper :: Char -> Bool`

Selects upper-case or title-case alphabetic Unicode characters (letters).

`isAlpha :: Char -> Bool`

Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters).

`isAlphaNum :: Char -> Bool`

Selects alphabetic or numeric digit Unicode characters.

`isDigit :: Char -> Bool`

Selects ASCII digits, i.e. ‘0’..’9’.

isHexDigit :: Char -> Bool

Selects ASCII hexadecimal digits, i.e. ‘0’..’9’, ‘a’..’f’, ‘A’..’F’.

Case conversion

toUpper :: Char -> Char

Convert a letter to the corresponding upper-case letter, if any. Any other character is returned unchanged.

toLower :: Char -> Char

Convert a letter to the corresponding lower-case letter, if any. Any other character is returned unchanged.

6 Data.Functor

Functors: uniform action over a parameterized type, generalizing the map function on lists.

class Functor f where

The `Functor` class is used for types that can be mapped over. Instances of Functor should satisfy the following laws:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

The instances of `Functor` for lists, `Maybe` and `IO` satisfy these laws.

Methods

fmap :: (a -> b) -> f a -> f b

Instances

- `Functor []`
- `Functor Maybe`
- `Functor IO`

7 Data.List

Basic functions

(++) :: [a] -> [a] -> [a] (**infixr** 5)

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
```

If the first list is not finite, the result is the first list.

head :: [a] -> a

Extract the first element of a list, which must be non-empty.

last :: [a] -> a

Extract the last element of a list, which must be finite and non-empty.

tail :: [a] -> [a]

Extract the elements after the head of a list, which must be non-empty.

init :: [a] -> [a]

Return all the elements of a list except the last one. The list must be non-empty.

null :: [a] -> Bool

Test whether the list is empty.

length :: [a] -> Int
Returns the size/length of a finite list as an `Int`.

List transformations

map :: (a -> b) -> [a] -> [b]
map f xs is the list obtained by applying f to each element of xs, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

reverse :: [a] -> [a]
`reverse xs` returns the elements of `xs` in reverse order. `xs` must be finite.

intersperse :: a -> [a] -> [a]
The `intersperse` function takes an element and a list and "intersperses" that element between the elements of the list. For example,

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

intercalate :: [a] -> [[a]] -> [a]
`intercalate xs xss` is equivalent to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

subsequences :: [a] -> [[a]]
The subsequences function returns the list of all subsequences of the argument.

```
subsequences "abc" == ["","a","b","ab","c","ac","bc","abc"]
```

permutations :: [a] -> [[a]]
The `permutations` function returns the list of all permutations of the argument.

```
permutations "abc" == ["abc","bac","cba","bca","cab","acb"]
```

Reducing lists (folds)

foldl :: (b -> a -> b) -> b -> [a] -> b
Left-associative fold of a list.

`foldl`, when applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

Note that to produce the outermost application of the operator the entire input list must be traversed. This means that `foldl'` will diverge if given an infinite list.

Also note that if you want an efficient left-fold, you probably want to use `foldl'` instead of `foldl`. The reason for this is that latter does not force the "inner" results (e.g. ``z `f` x1`` in the above example) before applying them to the operator (e.g. to ``(`f` x2)``). This results in a thunk chain $O(n)$ elements long, which then must be evaluated from the outside-in.

foldl' :: (b -> a -> b) -> b -> [a] -> b
Left-associative fold of a structure but with strict application of the operator.

This ensures that each step of the fold is forced to weak head normal form before being applied, avoiding the collection of thunks that would otherwise occur. This is often what you want to strictly reduce a finite list to a single, monolithic result (e.g. length).

foldl1 :: (a -> a -> a) -> [a] -> a
A variant of `foldl` that has no base case, and thus may only be applied to non-empty structures.

foldl1' :: (a -> a -> a) -> [a] -> a
A strict version of `foldl1`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

Right-associative fold of a structure.

`foldr`, when applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

Note that, since the head of the resulting expression is produced by an application of the operator to the first element of the list, `foldr` can produce a terminating expression from an infinite list.

`foldr1 :: (a -> a -> a) -> [a] -> a`

A variant of `foldr` that has no base case, and thus may only be applied to non-empty lists.

Special folds

`concat :: [[a]] -> [a]`

The concatenation of all the elements of a list of lists.

`concatMap :: (a -> [b]) -> [a] -> [b]`

Map a function over all the elements of a list and concatenate the resulting lists.

`and :: [Bool] -> Bool`

`and` returns the conjunction of a list of `Bool`s. For the result to be `True`, the list must be finite; `False`, however, results from a `False` value finitely far from the left end.

`or :: [Bool] -> Bool`

`or` returns the disjunction of a list of `Bool`s. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value finitely far from the left end.

`any :: (a -> Bool) -> [a] -> Bool`

Determines whether any element of the list satisfies the predicate.

`all :: (a -> Bool) -> [a] -> Bool`

Determines whether all elements of the list satisfy the predicate.

`sum :: Num a => [a] -> a`

The sum function computes the sum of the numbers of a list.

`product :: Num a => [a] -> a`

The product function computes the product of the numbers of a list.

`maximum :: Ord a => [a] -> a`

The largest element of a non-empty list.

`minimum :: Ord a => [a] -> a`

The least element of a non-empty list.

Building lists

`scanl :: (b -> a -> b) -> b -> [a] -> [b]`

`scanl` is similar to `foldl`, but returns a list of successive reduced values from the left:

```
scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
```

Note that

```
last (scanl f z xs) == foldl f z xs
```

Infinite lists

`iterate :: (a -> a) -> a -> [a]`

`iterate f x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

`repeat :: a -> [a]`

`repeat x` is an infinite list, with `x` the value of every element.

`replicate :: Int -> a -> [a]`

`replicate n x` is a list of length `n` with `x` the value of every element.

`cycle :: [a] -> [a]`

`cycle` ties a finite list into a circular one, or equivalently, the infinite repetition of the original list. It is the identity on infinite lists.

Unfolding

`unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`

The `unfoldr` function is a "dual" to `foldr`: while `foldr` reduces a list to a summary value, `unfoldr` builds a list from a seed value. The function takes the element and returns `Nothing` if it is done producing the list or returns `Just (a,b)`, in which case, `a` is a prepended to the list and `b` is used as the next element in a recursive call. For example,

```
iterate f == unfoldr (\x -> Just (x, f x))
```

A simple use of `unfoldr`:

```
>>> unfoldr (\b -> if b == 0 then Nothing else Just (b, b-1)) 10
[10,9,8,7,6,5,4,3,2,1]
```

Sublists

Extracting sublists

`take :: Int -> [a] -> [a]`

`take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`:

```
take 5 "Hello World!" == "Hello"
take 3 [1,2,3,4,5] == [1,2,3]
take 3 [1,2] == [1,2]
take 3 [] == []
take (-1) [1,2] == []
take 0 [1,2] == []
```

`drop :: Int -> [a] -> [a]`

`drop n xs` returns the suffix of `xs` after the first `n` elements, or `[]` if `n > length xs`:

```
drop 6 "Hello World!" == "World!"
drop 3 [1,2,3,4,5] == [4,5]
drop 3 [1,2] == []
drop 3 [] == []
drop (-1) [1,2] == [1,2]
drop 0 [1,2] == [1,2]
```

`splitAt :: Int -> [a] -> ([a], [a])`

`splitAt n xs` returns a tuple where first element is `xs` prefix of length `n` and second element is the remainder of the list:

```
splitAt 6 "Hello World!" == ("Hello ","World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])
splitAt 1 [1,2,3] == ([1],[2,3])
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])
```

`takeWhile :: (a -> Bool) -> [a] -> [a]`

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`:

```
takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
takeWhile (< 9) [1,2,3] == [1,2,3]
takeWhile (< 0) [1,2,3] == []
```

`dropWhile :: (a -> Bool) -> [a] -> [a]`

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`:

```
dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]
dropWhile (< 9) [1,2,3] == []
dropWhile (< 0) [1,2,3] == [1,2,3]
```

`span :: (a -> Bool) -> [a] -> ([a], [a])`

`span`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that satisfy `p` and second element is the remainder of the list:

```
span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
span (< 9) [1,2,3] == ([1,2,3],[])
span (< 0) [1,2,3] == ([],[1,2,3])
```

`span p xs` is equivalent to `(takeWhile p xs, dropWhile p xs)`

`group :: Eq a => [a] -> [[a]]`

The `group` function takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements. For example,

```
group ”Mississippi” = [”M”,”i”,”ss”,”i”,”ss”,”i”,”pp”,”i”]
```

Predicates

`isPrefixOf :: Eq a => [a] -> [a] -> Bool`

The `isPrefixOf` function takes two lists and returns `True` iff the first list is a prefix of the second.

`isSuffixOf :: Eq a => [a] -> [a] -> Bool`

The `isSuffixOf` function takes two lists and returns `True` iff the first list is a suffix of the second. The second list must be finite.

`isInfixOf :: Eq a => [a] -> [a] -> Bool`

The `isInfixOf` function takes two lists and returns `True` iff the first list is contained, wholly and intact, anywhere within the second.

```
isInfixOf ”Haskell” ”I really like Haskell.” == True
isInfixOf ”Ial” ”I really like Haskell.” == False
```

Searching lists

Searching by equality

`elem :: Eq a => a -> [a] -> Bool`

Does the element occur in the list?

`notElem :: Eq a => a -> [a] -> Bool`

`notElem` is the negation of `elem`.

`lookup :: Eq a => a -> [(a, b)] -> Maybe b`

`lookup key assocs` looks up a key in an association list.

Searching with a predicate

`find :: (a -> Bool) -> [a] -> Maybe a`

The `find` function takes a predicate and a list and returns the leftmost element of the list matching the predicate, or `Nothing` if there is no such element.

`filter :: (a -> Bool) -> [a] -> [a]`

`filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

```
filter p xs = [ x | x <- xs, p x]
```

partition :: (a -> Bool) -> [a] -> ([a], [a])
The `partition` function takes a predicate a list and returns the pair of lists of elements which do and do not satisfy the predicate, respectively; i.e.,

```
partition p xs == (filter p xs, filter (not . p) xs)
```

Indexing lists

These functions treat a list `xs` as a indexed collection, with indices ranging from `0` to `length xs - 1`.

(!!) :: [a] -> Int -> a (infixl 9)
List index (subscript) operator, starting from 0.

elemIndex :: Eq a => a -> [a] -> Maybe Int
The `elemIndex` function returns the index of the first element in the given list which is equal (by `==`) to the query element, or `Nothing` if there is no such element.

findIndex :: (a -> Bool) -> [a] -> Maybe Int
The `findIndex` function takes a predicate and a list and returns the index of the first element in the list satisfying the predicate, or `Nothing` if there is no such element.

Zipping and unzipping lists

zip :: [a] -> [b] -> [(a, b)]
`zip` takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded.

zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
`zip3` takes three lists and returns a list of triples, analogous to `zip`.

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
`zipWith` generalises `zip` by zipping with the function given as the first argument, instead of a tupling function. For example, `zipWith (+)` is applied to two lists to produce the list of corresponding sums.

zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
The `zipWith3` function takes a function which combines three elements, as well as three lists and returns a list of their point-wise combination, analogous to `zipWith`.

unzip :: [(a, b)] -> ([a], [b])
`unzip` transforms a list of pairs into a list of first components and a list of second components.

“Set” operations

nub :: Eq a => [a] -> [a]
$O(n^2)$. The `nub` function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. (The name nub means “essence”.)

delete :: Eq a => a -> [a] -> [a]
`delete x` removes the first occurrence of `x` from its list argument. For example,

```
delete 'a' ”banana” == ”bnana”
```

Ordered lists

sort :: Ord a => [a] -> [a]
The `sort` function implements a stable sorting algorithm. It is a special case of `sortBy`, which allows the programmer to supply their own comparison function.

sortOn :: Ord b => (a -> b) -> [a] -> [a]
Sort a list by comparing the results of a key function applied to each element. `sortOn f` is equivalent to `sortBy (comparing f)`, but has the performance advantage of only evaluating `f` once for each element in the input list. This is called the decorate-sort-undecorate paradigm, or Schwartzian transform.

`insert :: Ord a => a -> [a] -> [a]`
The `insert` function takes an element and a list and inserts the element into the list at the first position where it is less than or equal to the next element. In particular, if the list is sorted before the call, the result will also be sorted.

Generalized functions

By convention, overloaded functions have a non-overloaded counterpart whose name is suffixed with “By”.

User-supplied equality (replacing an Eq context)

`nubBy :: (a -> a -> Bool) -> [a] -> [a]`
The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded `==` function.

`deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]`
The `deleteBy` function behaves like `delete`, but takes a user-supplied equality predicate.

User-supplied comparison (replacing an Ord context)

The function is assumed to define a total ordering.

`sortBy :: (a -> a -> Ordering) -> [a] -> [a]`
The `sortBy` function is the non-overloaded version of `sort`.

`maximumBy :: (a -> a -> Ordering) -> [a] -> a`
The largest element of a non-empty list with respect to the given comparison function.

`minimumBy :: (a -> a -> Ordering) -> [a] -> a`
The least element of a non-empty list with respect to the given comparison function.

8 Data.Maybe

`data Maybe a`
The `Maybe` type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented as `Just a`), or it is empty (represented as `Nothing`). Using `Maybe` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.

The `Maybe` type is also a monad. It is a simple kind of error monad, where all errors are represented by `Nothing`. A richer error monad can be built using the `Either` type.

Constructors

— `Nothing`
— `Just a`

`maybe :: b -> (a -> b) -> Maybe a -> b`
The maybe function takes a default value, a function, and a `Maybe` value. If the `Maybe` value is `Nothing`, the function returns the default value. Otherwise, it applies the function to the value inside the Just and returns the result.

Basic usage:

```
>>> maybe False odd (Just 3)
True
>>> maybe False odd Nothing
False
```

`isJust :: Maybe a -> Bool`
The `isJust` function returns `True` iff its argument is of the form `Just _`.

`isNothing :: Maybe a -> Bool`
The `isNothing` function returns `True` iff its argument is `Nothing`.

fromJust :: Maybe a -> a

The `fromJust` function extracts the element out of a `Just` and throws an error if its argument is `Nothing`.

```
>>> fromJust (Just 1)
1
>>> 2 * (fromJust (Just 10))
20
>>> 2 * (fromJust Nothing)
*** Exception: Maybe.fromJust: Nothing
```

fromMaybe :: a -> Maybe a -> a

The `fromMaybe` function takes a default value and and Maybe value. If the `Maybe` is `Nothing`, it returns the default values; otherwise, it returns the value contained in the `Maybe`.

```
>>> fromMaybe "" (Just "Hello, World!")
"Hello, World!"
>>> fromMaybe "" Nothing
""
```

listToMaybe :: [a] -> Maybe a

The `listToMaybe` function returns `Nothing` on an empty list or `Just a` where a is the first element of the list.

maybeToList :: Maybe a -> [a]

The `maybeToList` function returns an empty list when given `Nothing` or a singleton list when not given `Nothing`.

catMaybes :: [Maybe a] -> [a]

The `catMaybes` function takes a list of `Maybes` and returns a list of all the `Just` values.

```
>>> catMaybes [Just 1, Nothing, Just 3]
[1,3]
```

mapMaybe :: (a -> Maybe b) -> [a] -> [b]

The `mapMaybe` function is a version of map which can throw out elements. In particular, the functional argument returns something of type `Maybe b`. If this is `Nothing`, no element is added on to the result list. If it is `Just b`, then `b` is included in the result list.

9 Data.String

The String type and associated operations.

type String = [Char]

A String is a list of characters. String constants in Haskell are values of type String.

lines :: String -> [String]

`lines` breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

Note that after splitting the string at newline characters, the last part of the string is considered a line even if it doesn't end with a newline. For example,

```
lines "" == []
lines "\n" == [""]
lines "one" == ["one"]
lines "one\n" == ["one"]
lines "one\n\n" == ["one",""]
lines "one\ntwo" == ["one","two"]
lines "one\ntwo\n" == ["one","two"]
```

Thus `lines s` contains at least as many elements as newlines in `s`.

`words :: String -> [String]`
: `words` breaks a string up into a list of words, which were delimited by white space.

`unlines :: [String] -> String`
: `unlines` is an inverse operation to `lines`. It joins lines, after appending a terminating newline to each.

`unwords :: [String] -> String`
: `unwords` is an inverse operation to words. It joins words with separating spaces.

10 `Data.Tuple`

The tuple data types, and associated functions.

`fst :: (a, b) -> a`
: Extract the first component of a pair.

`snd :: (a, b) -> b`
: Extract the second component of a pair.

`curry :: ((a, b) -> c) -> a -> b -> c`
: `curry` converts an uncurried function to a curried function.

`uncurry :: (a -> b -> c) -> (a, b) -> c`
: `uncurry` converts a curried function to a function on pairs.

`swap :: (a, b) -> (b, a)`
: Swap the components of a pair.

11 `Prelude`

`data Ordering :: *`

Constructors

- `LT`
- `EQ`
- `GT`

Basic type classes

`class Eq a where`

The Eq class defines equality (`==`) and inequality (`/=`). All the basic datatypes exported by the `Prelude` are instances of `Eq`, and `Eq` may be derived for any datatype whose constituents are also instances of `Eq`.

Methods

- `(==) :: a -> a -> Bool`
- `(/=) :: a -> a -> Bool`

`class Eq a => Ord a where`

The `Ord` class is used for totally ordered datatypes.

Instances of `Ord` can be derived for any user-defined datatype whose constituent types are in `Ord`. The declared order of the constructors in the data declaration determines the ordering in derived `Ord` instances. The `Ordering` datatype allows a single comparison to determine the precise ordering of two objects.

Methods

- `compare :: a -> a -> Ordering`
- `(<) :: a -> a -> Bool`
- `(<=) :: a -> a -> Bool`
- `(>) :: a -> a -> Bool`
- `(>=) :: a -> a -> Bool`
- `max :: a -> a -> a`

— `min :: a -> a -> a`

`class Enum a where`

Class `Enum` defines operations on sequentially ordered types.

The `enumFrom`... methods are used in Haskell's translation of arithmetic sequences.

Instances of `Enum` may be derived for any enumeration type (types whose constructors have no fields). The nullary constructors are assumed to be numbered left-to-right by `fromEnum` from 0 through n-1.

Methods

`succ :: a -> a`
the successor of a value. For numeric types, `succ` adds 1.

`pred :: a -> a`
the predecessor of a value. For numeric types, `pred` subtracts 1.

`toEnum :: Int -> a`
Convert from an `Int`.

`fromEnum :: a -> Int`
Convert to an `Int`. It is implementation-dependent what `fromEnum` returns when applied to a value that is too large to fit in an `Int`.

`enumFrom :: a -> [a]`
Used in Haskell's translation of `[n..]`.

`enumFromThen :: a -> a -> [a]`
Used in Haskell's translation of `[n,n'..]`.

`enumFromTo :: a -> a -> [a]`
Used in Haskell's translation of `[n..m]`.

`enumFromThenTo :: a -> a -> a -> [a]`
Used in Haskell's translation of `[n,n'..m]`.

`class Bounded a where`

The `Bounded` class is used to name the upper and lower limits of a type.

Methods

— `minBound :: a`
— `maxBound :: a`

Numbers

Numeric types

`data Int :: *`
A fixed-precision integer type with at least the range $[-2^{29}..2^{29}-1]$. The exact range for a given implementation can be determined by using `minBound` and `maxBound` from the `Bounded` class.

`data Integer :: *`
Type for arbitrary-precision integers.

`data Float :: *`
Single-precision floating point numbers. It is desirable that this type be at least equal in range and precision to the IEEE single-precision type.

`data Double :: *`
Double-precision floating point numbers. It is desirable that this type be at least equal in range and precision to the IEEE double-precision type.

Numeric type classes

`class Num a where`

Basic numeric class.

Methods

`(+), (-), (*) :: a -> a -> a`

`negate :: a -> a`
: Unary negation.

`abs :: a -> a`
: Absolute value.

`signum :: a -> a`
: Sign of a number.

`fromInteger :: Integer -> a`
: Conversion from an `Integer`. An integer literal represents the application of the function `fromInteger` to the appropriate value of type `Integer`, so such literals have type `(Num a) => a`.

Instances

- `Num Int`
- `Num Integer`
- `Num Float`
- `Num Double`

`class (Real a, Enum a) => Integral a where`

Integral numbers, supporting integer division.

Methods

`quot :: a -> a -> a infixl 7`
: integer division truncated toward zero

`rem :: a -> a -> a infixl 7`
: integer remainder, satisfying

```
(x `quot` y)*y + (x `rem` y) == x
```

`div :: a -> a -> a infixl 7`
: integer division truncated toward negative infinity

`mod :: a -> a -> a infixl 7`
: integer modulus, satisfying

```
(x `div` y)*y + (x `mod` y) == x
```

`quotRem :: a -> a -> (a, a)`
: simultaneous quot and rem

`divMod :: a -> a -> (a, a)`
: simultaneous div and mod

`toInteger :: a -> Integer`
: conversion to `Integer`

Instances

- `Integral Int`
- `Integral Integer`

`class Num a => Fractional a where`

Fractional numbers, supporting real division.

Methods

`(/) :: a -> a -> a`
: fractional division

`recip :: a -> a`
reciprocal fraction

`fromRational :: Rational -> a`
Conversion from a `Rational`. A floating literal stands for an application of `fromRational` to a value of type `Rational`, so such literals have type `(Fractional a) => a`.

Numeric functions

`even :: Integral a => a -> Bool`

`odd :: Integral a => a -> Bool`

`(^) :: (Num a, Integral b) => a -> b -> a`
raise a number to a non-negative integral power

`(^^) :: (Fractional a, Integral b) => a -> b -> a`
raise a number to an integral power

`fromIntegral :: (Integral a, Num b) => a -> b`
general coercion from integral types

Miscellaneous functions

`id :: a -> a`
Identity function.

`const :: a -> b -> a`
`const x` is a unary function which evaluates to `x` for all inputs.

For instance,

```
>>> map (const 42) [0..3]
[42,42,42,42]
```

`(.) :: (b -> c) -> (a -> b) -> a -> c`
Function composition.

`flip :: (a -> b -> c) -> b -> a -> c`
`flip f` takes its (first) two arguments in the reverse order of `f`.

`($) :: (a -> b) -> a -> b      (infixr 0)`
Application operator. This operator is redundant, since ordinary application `(f x)` means the same as `(f $ x)`. However, `$` has low, right-associative binding precedence, so it sometimes allows parentheses to be omitted; for example:

```
f $ g $ h x  =  f (g (h x))
```

It is also useful in higher-order situations, such as `map ($ 0) xs`, or `zipWith ($) fs xs`.

`until :: (a -> Bool) -> (a -> a) -> a -> a`
`until p f` yields the result of applying `f` until `p` holds.

`error :: String -> a`
`error` stops execution and displays an error message.

`undefined :: a`
A special case of `error`. It is expected that compilers will recognize this and insert error messages which are more appropriate to the context in which `undefined` appears.

`seq :: a -> b -> b`
The value of `seq a b` is bottom if `a` is bottom, and otherwise equal to `b`. `seq` is usually introduced to improve performance by avoiding unneeded laziness.

A note on evaluation order: the expression `seq a b` does not guarantee that `a` will be evaluated before `b`. The only guarantee given by `seq` is that the both `a` and `b` will be evaluated before `seq` returns a value. In particular, this means that `b` may be evaluated before `a`. If you need to guarantee a specific order of evaluation, you must use the function `pseq` from `Control.Parallel`.

Converting to and from String

`class Show a where`

Conversion of values to readable `String`s.

Methods

`show :: a -> String`

Convert a value to a readable `String`.

`class Read a where`

Parsing of `String`s, producing values.

`read :: Read a => String -> a`

The `read` function reads input from a string, which must be completely consumed by the input process.

12 System.IO

The standard I/O library.

The IO monad

`data IO a :: * -> *`

A value of type `IO a` is a computation which, when performed, does some I/O before returning a value of type `a`.

There is really only one way to "perform" an I/O action: bind it to `Main.main` in your program. When your program is run, the I/O will be performed. It isn't possible to perform I/O from an arbitrary function, unless that function is itself in the `IO` monad and called at some point, directly or indirectly, from `Main.main`.

`IO` is a monad, so `IO` actions can be combined using either the **do**-notation or the >> and >>= operations from the `Monad` class.

Instances

- `Monad IO`
- `Functor IO`
- `Applicative IO`
- `Alternative IO`

Files and handles

`type FilePath = String`

File and directory names are values of type `String`, whose precise meaning is operating system dependent. Files can be opened, yielding a handle which can then be used to operate on the contents of that file.

`readFile :: FilePath -> IO String`

The `readFile` function reads a file and returns the contents of the file as a string. The file is read lazily, on demand, as with `getContents`.

`writeFile :: FilePath -> String -> IO ()`

The computation `writeFile file str` function writes the string `str`, to the file `file`.

appendFile :: FilePath -> String -> IO ()
The computation `appendFile file str` function appends the string `str`, to the file `file`.

Note that `writeFile` and `appendFile` write a literal string to a file. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first.

```
main = appendFile "squares" (show [(x,x*x) | x <- [0,0.1..2]])
```

Special cases for standard input and output

interact :: (String -> String) -> IO ()
The `interact` function takes a function of type `String -> String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.

putChar :: Char -> IO ()
Write a character to the standard output device.

putStr :: String -> IO ()
Write a string to the standard output device.

putStrLn :: String -> IO ()
The same as `putStr`, but adds a newline character.

print :: Show a => a -> IO ()
The `print` function outputs a value of any printable type to the standard output device. Printable types are those that are instances of class `Show`; `print` converts values to strings for output using the `show` operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2 could be written as:

```
main = print ([(n, 2^n) | n <- [0..19]])
```

getChar :: IO Char
Read a character from the standard input device.

getLine :: IO String
Read a line from the standard input device.

getContents :: IO String
The `getContents` operation returns all user input as a single string, which is read lazily as it is needed.

13 Parser

Bibliothèque simple d'analyseurs syntaxiques.

Types

type Resultat a = Maybe (a, String)
Type des résultats de l'exécution d'un `Parser a`.

data Parser a = MkParser (String -> Resultat a)
Type d'un analyseur syntaxiques qui retourne une valeur de type a s'il réussit.

Instances

- `Monad Parser`
- `Functor Parser`
- `Applicative Parser`
- `Alternative Parser`

runParser :: Parser a -> String -> Resultat a
Exécute un analyseur syntaxique sur une entrée donnée comme une chaîne.

`resultat :: Resultat a -> a`
: Extrait le résultat, en cas de réussite du parseur ; déclenche une exception sinon.

Briques de base et combinateurs

`unCaractereQuelconque :: Parser Char`
: Analyseur qui réussit si l'entrée n'est pas vide. Consomme et renvoie le premier caractère (quelconque) de l'entrée.

`pure  :: a -> Parser a` et son synonyme `return`
: Analyseur qui réussit toujours. Ne consomme rien et renvoie le résultat passé en argument (voir `Control.Applicative`).

`empty :: Parser a`
: Analyseur qui échoue toujours (voir `Control.Applicative`).

`(<|>) :: Parser a -> Parser a -> Parser a`
: Alternative entre deux analyseurs (voir `Control.Applicative`).

  Combinateur qui retourne le résultat du premier analyseur s'il réussit, sinon celui du second.

`(>>=) :: Parser a -> (a -> Parser b) -> Parser b`
: Séquence de deux analyseurs (voir `Control.Monad`).

  `p >>= fp` retourne le résultat du parseur `fp v` où `v` est la valeur parsée par `p` s'il a réussi, et échoue si `p` a échoué.

`(>>) :: Parser a -> Parser b -> Parser b`
: Séquence de deux analyseurs quand le résultat du premier analyseur n'est pas utile (voir `Control.Monad`). Ce combinateur est particulièrement naturel quand le premier parseur d'une séquence est de type `Parser ()`.

  `p1 >> p2` retourne le résultat du parseur `p2` si `p1` a réussi, et échoue si `p1` a échoué.

`many :: Parser a -> Parser [a]`
: Répétition zéro ou plusieurs fois d'un analyseur (voir `Control.Applicative`).

  Étant donné un analyseur `p`, retourne l'analyseur qui itère `p` tant qu'il réussit et retourne la liste des résultats.

  Correspond à l'étoile des expressions régulières.

`some :: Parser a -> Parser [a]`
: Répétition une ou plusieurs fois d'un analyseur (voir `Control.Applicative`).

  Étant donné un analyseur `p`, retourne l'analyseur qui itère `p` tant qu'il réussit, échoue si `p` échoue initialement, sinon retourne la liste des résultats s'il a réussi au moins une fois.

  Correspond au « $+$ » des expressions régulières.

Analyseurs construits avec les briques de base

`carQuand :: (Char -> Bool) -> Parser Char`
: Analyseur qui réussit avec le premier caractère s'il vérifie la condition donnée, puis le consomme et le renvoie.

`car :: Char -> Parser Char`
: Analyseur qui réussit si l'entrée commence par le caractère donné, puis le consomme et le renvoie.

`chaine :: String -> Parser String`
: Analyseur qui réussit si l'entrée commence par la chaîne donnée, puis la consomme et la renvoie.