

title Cheat Sheet for JinJings MPS Stuff

author code by Jinjing Wang
text by Henry Laxen

date July 3, 2009

This file can be loaded into ghci and executed. You are encouraged to do so.

Haskell Code

```
1 import Prelude hiding ((.), (^), (>), (/), elem, foldl, foldl1)
2 import MPS
3 import Control.Category hiding ((.))
```

Haskell Code

```
4 hamlet = ["2", "b", "or", "not", "to", "be"]
```

. is used to pass an argument to the function on the right, also called **apply** and **send_to** later

$(.) :: a \rightarrow (a \rightarrow b) \rightarrow b$

Notice that . binds less tightly than function arguments, so $[1..5].drop\ 2$ is the same as $drop\ 2\ [1..5]$

Haskell Code

```
5 t1 = [1..5].head -- 1
6 t1a = apply [1..5] head -- 1
7 t1b = send_to [1..5] head -- 1
8 t2 = [1..5].drop 2 -- [3,4,5]
```

> is left to right function composition, read it as *then*

$(>) :: (\text{Category } \text{cat}) \Rightarrow \text{cat } a\ b \rightarrow \text{cat } b\ c \rightarrow \text{cat } a\ c$

Haskell Code

```
9 t3 = tail > length $ [1..5] -- tail then length, is 4
```

^ is an infix fmap

$(^) :: (\text{Functor } f) \Rightarrow f\ a \rightarrow (a \rightarrow b) \rightarrow f\ b$

Haskell Code

```
10 t4 = [1..5] ^ (1+) -- is [2,3,4,5,6]
```

`/` is a `FilePath` joiner, not division, see `System.FilePath.Posix.combine`
`(/) :: FilePath -> FilePath -> FilePath`

Haskell Code

```
11 t5 = "usr" / "local" / "bin" -- is "usr/local/bin" on Linux
```

join is `intercalate`, it inserts the first list, (usually a `String`) in between each of the second lists, (usually a list of `Strings`)

`join :: [a] -> [[a]] -> [a]`

Haskell Code

```
12 t6 = ["2 b", "to be"].join " or not " -- is "2 b or not to be"
```

first .. tenth select the corresponding element of a list

`first :: [a] -> a`

Haskell Code

```
13 t7 = [1..5].third -- is 3
```

unique is a synonym for `nub`, but requires `ord`. It returns the unique elements of list

`unique :: (Ord a) => [a] -> [a]`

Haskell Code

```
14 t8 = unique ([1..5] ++ [3..8]) -- is [1,2,3,4,5,6,7,8]
```

is_unique returns `True` if there are no duplicate elements in a list

`is_unique :: (Ord a) => [a] -> Bool`

Haskell Code

```
15 t9 = (is_unique [1,2,3,4], is_unique [1,1,2,3]) -- is (True,False)
```

same returns `True` if all the elements of a list are the same

`same :: (Ord a) => [a] -> Bool`

Haskell Code

```
16 t10 = same "aaa" -- is True
```

times is like an infix `replicate`, to be used with the new `.`

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

Haskell Code

```
17 t11 = 3.times 'a' -- is "aaa"
```

upto and **downto** are infix ranges, to be used with the new `.`

`upto :: (Enum a) => a -> a -> [a]`

Haskell Code

```
18 t12 = 1.upto 3 -- is [1,2,3]
19 t13 = 3.downto 1 -- is [3,2,1]
```

remove_at, **insert_at** and **replace_at** are all similar, and do their thing at a particular position of a given list.

```
remove_at :: Int -> [a] -> [a]
insert_at, replace_at :: Int -> a -> [a] -> [a]
```

Haskell Code

```
20 t14 = insert_at 2 'a' "blb" -- is "blab"
```

at is list indexing

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

Haskell Code

```
21 t15 = at 2 "abcd" -- is 'c'
```

slice takes a slice out of a list, if the list is $[0..n]$ then slice a b is $[x]$ such that $a \leq x < b$

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

Haskell Code

```
22 t16 = slice 2 4 [1..5] -- is [3,4] (remember 0 indexing!)
```

cherry_pick picks a set of elements out of a list based on a list of indices

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

Haskell Code

```
23 t17 = cherry_pick [5,0,1] hamlet -- ["be","2","b"]
```

reduce and **reduce'** are synonyms for `foldl1`, but all the arguments must be of the same type. Very useful for chaining functions together. The `'` version is strict

```
reduce, reduce' :: (a -> a -> a) -> [a] -> a
```

Haskell Code

```
24 t18 = reduce (>) [(2*), (1+)] 3 -- is 7 (2*3 + 1)
25 t19 = reduce (<<<) [(2*), (1+)] 3 -- is 8 (1+3) * 2
```

inject and **inject'** is `foldl` with the arguments flipped

```
inject, inject' :: (Foldable t) => a -> (a -> b -> a) -> t b -> a
```

Haskell Code

```
26 t20 = [2,3].inject 1 (-) -- is -4 (1-2-3)
```

none_of takes a predicate (function `a -> Bool`) and a list and returns `True` if none of the elements of the list satisfy the predicate.

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

Haskell Code

```
27 t21 = [1..5].none_of (==6) -- is True
```

select and **reject** filter those that match, and filter those that don't match.
select, reject :: (a -> Bool) -> [a] -> [a]

Haskell Code

```
28 t22 = [1..5].reject even -- is [1,3,5]
29 t23 = [1..5].select odd -- is [1,3,5]
```

inner_map maps a function over the each element of an inner list
inner_map :: (a -> b) -> [[a]] -> [[b]]

Haskell Code

```
30 t24 = inner_map succ hamlet -- is ["3","c","ps","opu","up","cf"]
```

inner_reduce maps reduce over an inner list
inner_reduce :: (a -> a -> a) -> [[a]] -> [a]

Haskell Code

```
31 t25 = inner_reduce (+) [[1..5].select odd, [1..5].select even] -- is [9,6]
```

inner_inject maps inject from above over an inner list
inner_inject :: (Foldable t) => a -> (a -> b -> a) -> [t b] -> [a]

Haskell Code

```
32 t26 = inner_inject 1 (-) [[1..5].select odd, [1..5].select even] -- is [-8,-5]
```

label_by takes a function and a list, and returns a list of tuples where the first element of the tuple is f applied to the element of the list, and the second element is the original element. This is an easy way of zipping f of a list with itself.

label_by :: (a -> c) -> [a] -> [(c, a)]

Haskell Code

```
33 t27 = hamlet.label_by length .take 3 -- is [(1,"2"),(1,"b"),(2,"or")]
```

labeling is label_by, with the label coming first
labeling :: (a -> c) -> [a] -> [(a, c)]

Haskell Code

```
34 t28 = hamlet.labeling length .take 3 -- is [("2",1),("b",1),("or",2)]
```

in_group_of turns a list into a list of lists where the length of each inner list is given by an Int. The length of the last element of the inner list may be less than the Int. in_group_of :: Int -> [t] -> [[t]]

Haskell Code

```
35 t29 = join " " hamlet.in_group_of 3 -- is ["2 b"," or"," no","t t","o b","e"]
```

split_to turns a list into a list of lists where the number of sublists is given by an Int. If the length of the original list is not divisible by the Int, there will be an additional element.

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

Haskell Code

```
36 t30 = join " " hamlet.split_to 3 -- is ["2 b o","r not"," to b","e"]
```

belongs_to and **has** are like `elem` with different arguments.

`belongs_to :: (Foldable t, Eq a) => t a -> a -> Bool`

`has :: (Foldable t, Eq b) => b -> t b -> Bool`

Haskell Code

```
37 t31 = "not".belongs_to hamlet -- is True
38 t32 = hamlet.has "not" -- is True
```

indexed takes a list and returns a list of tuples, the first element of the tuple being the 0 based index of the position in the list.

`indexed :: (Num t, Enum t) => [b] -> [(t, b)]`

Haskell Code

```
39 t33 = hamlet.indexed -- is [(0,"2"),(1,"b"),(2,"or"),(3,"not"),(4,"to"),(5,"be")]
```

map_with_index first indexes a list, and then applies a map to the result. This makes it easy to manipulate each element of a list based on its position.

`map_with_index :: (Num t, Enum i) => ((i, a) -> b) -> [a] -> [b]`

Haskell Code

```
40 t34 = hamlet.map_with_index (\(x,y) -> if x.even then y.upper else y)
41 -- is ["2","b","OR","not","T0","be"]
```

ljust and **rjust** left or right justify a string to a fixed length.

`ljust, rjust :: Int -> a -> [a] -> [a]`

Haskell Code

```
42 t35 = "12".rjust 5 '0' -- is "00012"
43 t36 = "12".ljust 5 '0' -- is "12000"
```

swap exchanges the two elements of a two-tuple

`swap :: (a, b) -> (b, a)`

Haskell Code

```
44 t37 = swap (2 , "be") -- is ("be",2)
```

only_fst and **only_snd** retrieves only the first or second element of a list of tuples

`only_fst :: [(a, b)] -> [a]`

`only_snd :: [(a, b)] -> [b]`

Haskell Code

```
45 t38 = only_fst (zip hamlet [1..]) -- is ["2","b","or","not","to","be"]
46 t39 = only_snd (zip hamlet [1..]) -- is [1,2,3,4,5,6]
```

map_fst and **map_snd** maps a function over only the first or second element of a list of tuples

```
map_fst :: (a -> b) -> [(a, c)] -> [(b, c)]
map_snd :: (a -> b) -> [(c, a)] -> [(c, b)]
```

Haskell Code

```
47 t40 = map_fst same (zip hamlet [1..3]) -- is [(True,1),(True,2),(False,3)]
48 t41 = map_snd (1+) (zip hamlet [1..3]) -- is [("2",2),("b",3),("or",4)]
```

filter_fst and **filter_snd** filters a list of tuples by looking only at the fst or snd element of each pair.

```
filter_fst :: (a -> Bool) -> [(a, b)] -> [(a, b)]
filter_snd :: (b -> Bool) -> [(a, b)] -> [(a, b)]
```

Haskell Code

```
49 t42 = filter_fst even (zip [1..] hamlet) -- is [(2,"b"),(4,"not"),(6,"be")]
```

lower_upper and **capitalize** convert a string to lower or upper case
lower, upper, capitalize :: String -> String

Haskell Code

```
50 t43 = map capitalize hamlet . join " " -- is "2 B Or Not To Be"
51 t44 = map capitalize > join " " $ hamlet -- is "2 B Or Not To Be"
```

The raw lhs file is available for you to play with if you are so inclined. You can also read this page in pdf format Please address any complaints, comments, or misc. personal abuse to Henry Laxen Thanks.