Reference Sheet for CO120.1 Programming I

Autumn 2016

This reference sheet does not cover many interesting types and classes, such as `Either`, `IO`, `Complex`, and `Monad`. These are not included in the 120.1 syllabus, but even a limited understanding of them may be helpful in examinations.

1 Bools

```
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not  :: Bool -> Bool
```

2 Maybes

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Given default value, a function, and `Maybe` value: Returns default value if `Maybe` value is `Nothing`; Otherwise applies function to value inside `Just`.

```
isJust    :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
```

Requires `Data.Maybe`.

```
fromJust :: Maybe a -> a
fromMaybe :: a-> Maybe a -> a
```

Requires `Data.Maybe`.

`fromJust`: Given a `Maybe` value: Returns value inside `Just` or error if `Nothing`.

`fromMaybe`: Given default value and `Maybe` value: Returns value inside `Just` or default value if `Nothing`.

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

Requires `Data.Maybe`.

`catMaybes`: Given list of `Maybe` values: Returns a list of all `Just` values.

`mapMaybe`: Given function from value to `Maybe` value and list of values: Returns a list of all `Just` values from mapping function to list of values.

3 Tuples

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

It is usually better to use pattern matching unless used with higher order functions.

```
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: a -> b -> c -> (a, b) -> c
```

`curry`: Given uncurried function $f(x, y)$: Returns curried function $f\ x\ y$.

`uncurry`: Given curried function $f\ x\ y$: Returns uncurried function $f(x, y)$.

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

Requires `Data.Tuple`.

4 Enums

```
succ :: a -> a
pred :: a -> a
```

`succ`: Given a value: Returns its successor.

`pred`: Given a value: Returns its predecessor.

```
[n..]
[n,n'..]
[n..m]
[n,n'..m]
```

Returns enum from n.
Returns enum from n then n'.
Returns enum from n to m.
Returns enum from n then n' to m.

5 Numbers

5.1 Common Operators

```
(+), (-), (*) :: Num a => a -> a -> a
(/)           :: Fractional a => a -> a -> a
(^), (^^)     :: (Fractional a, Integral b) => a -> b -> a
```

```
negate :: Num a => a -> a
recip  :: Fractional a => a -> a
```

```
abs    :: Num a => a -> a
signum :: Num a => a -> a
```

`abs`: Given a value, returns its absolute value.
`signum`: Given a value, returns its sign.

5.2 Mathematical Constants and Functions

```
pi :: Floating a => a
```

```
exp, log, sqrt :: Floating a => a -> a
(**), logBase  :: Floating a => a -> a -> a
```

`log`: Natural log.

```
sin, cos, tan, asin, acos, atan :: Floating a => a -> a
```

In Radians.

5.3 Number Theoretical Functions

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

```
div    :: Num a => a -> a -> a
mod    :: Num a => a -> a -> a
divMod :: Num a => a -> a -> (a, a)
```

`divMod`: Simultaneous div and mod.

```
gcd :: Integral a => a -> a -> a
lcm :: Integral a => a -> a -> a
```

5.4 Conversion between Numerical Types

```
round, ceiling, floor :: (Floating a, Num b) => a -> b
```

`round`: Rounds to the closest integer, or even integer if closest integers are equidistant.

```
fromIntegral :: (Integral a, Num b) a -> b
```

6 Characters

Requires `Data.Char`.

6.1 Classification

```
isSpace       :: Char -> Bool
isLower       :: Char -> Bool
isUpper       :: Char -> Bool
isAlpha       :: Char -> Bool
isAlphaNum    :: Char -> Bool
isDigit       :: Char -> Bool
isPunctuation :: Char -> Bool
isSeparator   :: Char -> Bool
```

6.2 Case Conversion

```
toUpper :: Char -> Char
toLower :: Char -> Char
```

6.3 Numeric Conversion

```
ord :: Char -> Int
chr :: Int -> Char
```

7 Lists

7.1 Working with Lists

7.1.1 List Operations and Transformations

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

(:): Cons: Adds single element to beginning of a list.
(++): Appends two lists.

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

map: Maps a function over a list. The more general function fmap (infix <$> works for all functor types).
filter: Returns a list of elements for which a predicate holds.

```
head :: [a] -> a
last :: [a] -> a
tail :: [a] -> [a]
init :: [a] -> [a]
```

head, last: Returns the first, last element respectively.
tail, init: Returns all elements but the first, last respectively.

```
null :: [a] -> Bool
```

Returns True if and only if the given list is empty.

```
reverse :: [a] -> [a]
```

```
intersperse :: a -> [a] -> [a]
intercalate :: [a] -> [[a]] -> [a]
```

Requires Data.List.
intersperse: Given an element and a list: Returns a list with element interspersed between each element of the list.
intercalate: Intersperses elements between each list in a set of lists.

```
transpose :: [[a]] -> [[a]]
```

Requires Data.List. E.g. transpose $[[1,2,3,4],[11,12],[1,2,13],[5]] = [[1,11,1,5],[2,12,2],[3,13],[4]]$

```
subsequences :: [a] -> [[a]]
permutations :: [a] -> [[a]]
```

subsequences: Returns a list of all subsequences.
permutations: Returns a list of all permutations.

7.1.2 Sublists

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

splitAt n xs: take n xs and drop n xs.

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

Don't forget to use takeWhile and dropWhile!
span: Given a predicate and a list: Returns a tuple where the first element is the longest prefix of the list that satisfy the predicate and the second element is the remainder.
break: As for span, but for the longest prefix that does not satisfy the predicate.

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

Requires Data.List. Drops the largest suffix of the list for which the predicate holds.

```
stripPrefix :: [a] -> [a] -> Maybe [a]
```

Requires Data.List. Drops the given prefix from a list, or returns Nothing if the list did not start with the given prefix.

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

Requires Data.List. Given a list; Returns a list of lists, that when concatenated reform the original list, and such that each sublist only contains equal elements.

```
inits :: [a] -> [[a]]
tails :: [a] -> [[a]]
```

Requires Data.List.
inits: Returns all initial segments, shortest first.
tails: Returns all final segments, shortest first.

7.1.3 Predicates

```
isPrefixOf      :: [a] -> [a] -> Bool
isSuffixOf      :: [a] -> [a] -> Bool
isInfixOf       :: [a] -> [a] -> Bool
isSubsequenceOf :: [a] -> [a] -> Bool
```

Requires `Data.List`.

`isPrefixOf`: Returns `True` if the first list is a prefix of the second.

`isSuffixOf`: Returns `True` if the first list is a suffix of the second.

`isInfixOf`: Returns `True` if the first list is contained, wholly and intact, in the second.

`isSubsequenceOf`: Returns `True` if the first list is contained, in order, in the second (elements not necessarily consecutive).

7.1.4 Indexing Lists

```
(!!) :: [a] -> Int -> a
```

Returns element at given index.

```
elemIndex   :: a -> [a] -> Maybe Int
elemIndices :: a -> [a] -> [Int]
findIndex   :: (a -> Bool) -> [a] -> Maybe Int
findIndices :: (a -> Bool) -> [a] -> [Int]
```

Requires `Data.List`.

`elemIndex, findIndex`: Returns the index of the first element in the given list that is equal to the given value, satisfies the given predicate respectively; Or returns `Nothing` if none exists.

`elemIndices, findIndices`,: Returns the indices of all elements in the given list that are equal to the given value, satisfy the given predicate respectively.

7.1.5 Searching Lists

```
elem    :: a -> [a] -> Bool
notElem :: a -> [a] -> Bool
```

`elem, notElem`: Returns `True` if the given element is, is not an element of the given list respectively.

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

Looks up a key in a dictionary; Returns `Nothing` if key not found.

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

Requires `Data.List`. Returns the first element that satisfies the predicate, or `Nothing` if none exists.

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

Requires `Data.List`. Returns the pair of lists of elements which do and do not satisfy the predicate respectively.

7.1.6 Zipping and Unzipping Lists

```
zip  :: [a] -> [b] -> [(a, b)]
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
```

```
zipWith  :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

```
unzip  :: [(a, b)] -> ([a], [b])
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
```

7.2 Building Lists

7.2.1 Building Lists

```
scanr           :: (a -> b -> b) -> b -> [a] -> [b]
scanl           :: (b -> a -> b) -> b -> [a] -> [b]
scanr1, scanl1 :: (a -> a -> a) -> [a] -> [a]
```

Returns a list of successive values from respective fold function.

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

Given a function that produces a `Maybe` pair and a starting value; Applies the function to the starting value; If the function returns a `Just` pair, the first element is added to the resulting list and the function is applied again to the second element; If the function returns `Nothing`, the resulting list is returned.

```
mapAccumR (or L) :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
```

Given a function to a pair of an accumulator and a result, and a starting value for the accumulator; Returns a pair of the final accumulator and the list of results.

7.2.2 Infinite Lists

```
iterate   :: (a -> a) -> a -> [a]
repeat    :: a -> [a]
replicate :: Int -> a -> [a]
cycle     :: [a] -> [a]
```

`iterate`: Returns an infinite list of applications of a function to a starting value.

`repeat`: Returns an infinite list with each element of the given value.

`replicate`: Returns a list of given length of a repeated element.

`cycle`: Returns an infinite list produced by cycling the given list.

7.3 Reducing Lists

7.3.1 Folds

```
foldr  :: (a -> b -> b) -> b -> [a] -> b
foldl  :: (b -> a -> b) -> b -> [a] -> b
foldr1 :: (a -> a -> a) -> [a] -> a
foldl1 :: (a -> a -> a) -> [a] -> a
```

Don't forget to use folds!

Given a binary operator, starting value and list: Reduces list using the binary operator in the given direction, starting from the given starting value.

e.g. `foldr` $f\ x_0\ [x_1, x_2, x_3, \ldots, x_n] = f\ x_1 (f\ x_2 (f\ x_3 (\ \ldots\ (f\ x_n\ x_0))))$,
and `foldl` $f\ x_0\ [x_1, x_2, x_3, \ldots, x_n] = (f\ (f\ (f\ (f\ x_0\ x_1)\ x_2)\ x_3) \ldots\ x_n)$.

`fold1`: Has no starting value.

7.3.2 Special Folds

```
length  :: [a] -> Int
sum     :: Num a => [a] -> a
product :: Num a => [a] -> a
```

```
maximum :: [a] -> a
minimum :: [a] -> a
```

```
or  :: [Bool] -> Bool
and :: [Bool] -> Bool
any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
```

`or, any`: Returns `True` if `True` for any element.
`and, all`: Returns `True` if `True` for all elements.

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

`concat`: Concatenates a list of lists into a list.
`concatMap`: Maps a function, then concatenates result.

7.4 Special Lists

7.4.1 Set Functions

```
nub :: [a] -> [a]
```

Requires `Data.List`. Removes duplicates.

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

Requires `Data.List`. Deletes first occurrence of given element, each element of given list.

```
union     :: [a] -> [a] -> [a]
intersect :: [a] -> [a] -> [a]
```

Requires `Data.List`. Returns union, intersection of two lists.

7.4.2 Ordered Lists

```
sort :: [a] -> [a]
```

Requires `Data.List`. Sorts list.

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

Requires `Data.List`. Sorts list by comparing values generated by given function.

```
insert :: a -> [a] -> [a]
```

Requires `Data.List`. Inserts element so that a sorted list remains sorted.

7.4.3 Functions on Strings

```
lines   :: String -> [String]
words   :: String -> [String]
unlines :: [String] -> String
unwords :: [String] -> String
```

`words`: Breaks up into a list of words, delimited by white space.
`lines`: Breaks up into a list of strings separated by newlines.

8 Functions

```
id       :: a -> a
const    :: a -> b -> a
asTypeOf :: a -> a -> a
```

`id`: Identity function.
`const`: Evaluates to constant value for all inputs.
`asTypeOf`: Forces first argument to have the same type as the second.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
($) :: (a -> b) -> a -> b
```

`(.)`: Function composition: $(f.g)\ x = f(g\ x)$. Note the requirements for the type and number of arguments. You can use this to combine two higher order functions into one.

`($)`: Function application: $f\ \$\ g\ \$\ h\ x = f(g(h\ x))$. Often used to omit parentheses. To be avoided in general!

```
flip :: (a -> b -> c) -> b -> a -> c
```

Use to invert arguments. Alternative to back-quotes or lambdas.

```
until :: (a -> Bool) -> (a -> a) -> a -> a
```

Given a predicate and function; Applies function until predicate holds.

```
error "Error␣string"
```

Stops execution and displays error message.

9 Types and Common Type Classes

Use `data` to define a new data-type (you *must* define type constructors), `type` to define a type synonym, `class` to define a new type-class, and `instance` to make a type an instance of a type-class.

Don't forget you can use `deriving` in most cases. Example:

```
data Tree a = Empty | Leaf a | Node a (Tree a) (Tree a)
            deriving (Show)
```

Remember you can use `:i` in ghci for information about types and type-classes.

10 Syntactic Features and Good Practice

Spacing Use proper spacing and alignment. Keep lines short.

Function Names Give functions meaningful names. Don't ever give two different things the same name!

Comments Use to keep code ordered and to explain complicated functions. Begin with `--`. Multli-line comments use `{-` and `-}`.

Multi-Line Strings End and start each line with \.

Where clauses Use instead of `let` statements to avoid computing something multiple times or to clean up code.

Helper Functions Use helper functions to provide complicated functionality (e.g. creating a list *and* checking for convergence as it is created). Accumulating parameters are also particularly useful:

```
function input = function' input startingAcc
  where
    function' baseCase  acc = acc
    function' otherCase acc = function nextCase updatedAcc
```

Lambdas Use as an alternative to higher-order functions for very complicated functions.

List comprehensions Often functions involving lists can be written using comprehensions, recursion, or higher-order functions. Choose carefully.

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

Pattern matching Use often. Especially helpful for breaking up lists using `:` and for working with tuples. Don't forget to use `_` and `@` where appropriate.

Guards Use instead of `if` ... `then` ... `else` statements.

Tips for Lexis Tests

- Read instructions carefully. Don't rewrite a given function!
- Be prepared to write very little for the first questions and much more (helper functions, etc.) towards the end. You should check your answers to the first parts before beginning the last part. You should spend time planning your answer (on paper) for the last part.
- Aim firstly to get a working implementation, then consider optimisations.
- Don't forget to use functions defined earlier in the program.
- Test often (using `undefined` where necessary to allow compilation) and read any error messages carefully.
- Make use of any provided test cases but do not rely upon them.
- Use an editor with syntax highlighting and `ghci` in terminal.
- Use `:set -W` to turn on warnings and `:set -w` to turn off warnings in `ghci`. Use `:browse` to see a list of all functions included in a module.