University of Canberra
Faculty of Information Sciences and Engineering

Computer Engineering 1 — 1997
Part 1 : Introductory Programming

Standard Prelude Functions

Pure Haskell is a very small language. The language is made more powerful and convenient by providing a collection of pre-defined functions. These are in a file called the *standard prelude* which is automatically loaded when Haskell is started.

This handout describes some of the most useful such functions. For actual definitions and more details, have a look at the standard prelude file `Prelude.hs`. Follow the link from the unit home page or look at the actual path given when you start up Hugs.. There are also a number of library modules that you can import into your own programs. Again, the easiest way to take a look at them is to follow the links from the unit home page.

Basic boolean functions:

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

`exp1 && exp2` gives the infix boolean conjunction of `exp1` and `exp2`.

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

`exp1 || exp2` gives the infix boolean disjunction of `exp1` and `exp2`.

```
> not  :: Bool -> Bool
```

`not exp` gives the boolean negation of `exp`. The constant boolean values are `True` and `False`.

Basic numerical functions:

```
> (+), (-), (*), (/)  :: Num a => a -> a -> a
> negate              :: Num a => a -> a
```

Standard arithmetic operators (only defined on `Int` and `Float`).

```
> div, quot, rem, mod  :: Int -> Int -> Int
```

Integer division, quotient, remainder, modulus.

```
> even, odd  :: Int -> Bool
```

Predicates to test whether integers are even and odd respectively.

```
> gcd  :: Int -> Int -> Int
```

Greatest common divisor.

```
> lcm  :: Int -> Int -> Int
```

Lowest common multiple.

```
> (^)  :: Num a => a -> Int -> a
```

Exponentiation - note that the exponent must be an `Int`.

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

Absolute value of a `Float` or `Int`.

```
> sin, asin, cos, acos, tan, atan,
> log, exp, sqrt             :: Float -> Float
> logBase                    :: Float -> Float -> Float
```

Some standard functions.

```
> truncate  :: Float -> Int
```

Truncate a `Float` to an `Int`.

```
> pi  :: Float
> pi  = 3.1415926535
```

Character functions:

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

`ord` (short for *ordinal*) converts a character to its ASCII code. `chr` is the inverse of `ord`.

```
> isAlpha, isDigit, isSpace, isUpper, isLower,
> isAscii, isAlphanum, isControl, isPrint   :: Char -> Bool
```

`isAlpha c` tests whether character `c` is alphabetic.
`isDigit c` tests whether character `c` is a digit.
`isSpace c` tests whether character `c` is 'white space'.
`isUpper c` tests whether character `c` is an upper case letter.
`isLower c` tests whether character `c` is a lower case letter.
`isAscii n` tests whether integer `n` is within the ASCII range (i.e. < 128).
`isAlphanum c` tests whether character `c` is alphabetic or a digit.
`isControl c` tests whether character `c` is a control character.
`isPrint c` tests whether character `c` is printable.

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

`toUpper` converts a lower case alphabetic character to upper case. Other characters are left unchanged.

`toLower` converts an upper case alphabetic character to lower case. Other characters are left unchanged.

Relational functions:

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

Equality and inequality predicates are defined for all standard types except functions and `IO`.

```
> (<), (<=), (>), (>=)  :: Ord a => a -> a -> Bool
```

Ordering predicates are defined on ordered types. That is, all standard types except functions and `IO`.

```
> max, min  :: Ord a => a -> a -> a
```

Maximum and minimum of two values (defined on ordered types).

Operator precedence table:

```
> infixl 9 !!
> infixr 9 .
> infixr 8 ^
> infixl 7 *
> infix  7 /, `div`, `quot`, `rem`, `mod`
> infixl 6 +, -
> infix  5 \\
> infixr 5 ++, :
> infix  4 ==, /=, <, <=, >=, >
> infix  4 `elem`, `notElem`
> infixr 3 &&
> infixr 2 ||
> infixr 1 >>, >>=
> infixr 0 $
```

Note: It can be helpful to think of function application as a left associative infix operator with a precedence of 10 (i.e. highest).

Standard tuple functions:

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

The first element of a pair.

```
> snd       :: (a,b) -> b
> snd (_,y) = y
```

The second element of a pair.

Basic output functions:

```
> show  :: Text a => a -> String
```

Convert values to a string representation (valid for all standard types).

```
> putStr  :: String -> IO
```

Resolves the printing of a string. Here are example session that should demonstrate its use:

```
? "Hello World,\nNice day!"
"Hello World,\nNice day!"
?

? putStr "Hello World,\nNice day!"
Hello World,
Nice day!
?
```

Basic list processing functions:

```
> head     :: [a] -> a
```

Return the head (first element) of a list.

```
> tail     :: [a] -> [a]
```

Return the tail of a list (i.e. all but the first element).

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

Cons a new element onto the head of a list.

```
> last     :: [a] -> a
```

Return the last element of a list.

```
> init     :: [a] -> [a]
```

Return all but the last element of a list.

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

Return the length of list (number of elements).

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

Tests whether a list is empty.

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

`xs!!n` selects the `n`th element of the list `xs` for any `n < length xs`. Note that the first element is given by `xs!!0`.

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

`elem x ys` tests whether `x` is a member of the list `ys`. `notElem x ys` tests whether `x` is not a member of the list `ys`.

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

Find the maximum (resp. minimum) element in non-empty list.

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

Append lists (e.g. `[2,5,6] ++ [1,9,8,7] = [2,5,6,1,9,8,7]`).

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

Reverse the order of the elements of a list.

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

Make a list of `n` copies of `x`.

```
> repeat    :: a -> [a]
```

Generate the infinite list `[x, x, x, x, ...`

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

Generate the infinite list `xs ++ xs ++ xs ++ ...`

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

Concatenate a list of lists.

Boolean list functions:

```
> and   :: [Bool] -> Bool
```

`and xs` gives the conjunction of all elements of a list `xs` of boolean values.

```
> or    :: [Bool] -> Bool
```

`or xs` gives the disjunction of all elements of a list `xs` of boolean values.

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

`any p xs` applies some predicate `p` to every element `x` of a list `xs` and returns `True` if `p x` is `True` of *any* element of `xs`.

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

`all p xs` applies some predicate `p` to every element `x` of a list `xs` and returns `True` if `p x` is `True` of *every* element of `xs`.

Numerical list functions:

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

Returns the sum of all elements of a list of `Int` or a list of `Float`.

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

Returns the product of all elements of a list of `Int` or a list of `Float`.

Text processing functions:

```
> lines     :: String -> [String]
```

`lines s` returns the list of lines in the string `s`. Lines delimited by newline characters `\n`.

```
> words     :: String -> [String]
```

`words s` returns the list of words in the string `s`. Words are delimited by any character satisfying `isSpace`.

```
> unlines   :: [String] -> String
```

`unlines ls` joins the list of lines `ls` into a single string with lines separated by newline characters `\n`.

```
> unwords   :: [String] -> String
```

`unwords ws` joins the list of words `ws` into a single string with words separated by spaces.

List breaking functions:

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

`take n xs` returns the first `n` elements of `xs`. `drop n xs` returns the remaining elements of `xs`.

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

`splitAt n xs = (take n xs, drop n xs)`.

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

`takeWhile p xs` returns the longest initial segment of `xs` whose elements all satisfy predicate `p`. `dropWhile p xs` returns the remaining portion of the list.

zip and zipWith functions:

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

`zip` converts two lists into a list of pairs, truncating the longer list if necessary.
e.g. `zip [a1,a2,a3] [b1,b2,b3,b4,b5,b6] = [(a1,b1),a2,b2),(a3,b3)]`.

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

Similarly converts three lists into a list of triples. There are also functions `zip4` ...`zip7` defined in the library module `List.hs`. Import this module into your script if necessary.

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

The dual of `zip`.

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

`zipWith f as bs` applies `f` to corresponding elements of `as` and `bs` to give elements of the resulting list. Often `f` will be an infix operator, e.g.
`zipWith (#) [a1,a2,a3] [b1,b2,b3,b4]  = [a1#b1, a2#b2, a3#b3]`.

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

Similarly for three lists. There are also functions `zipWith4` ...`zipWith7` defined in the library module `List.hs`. Import this module into your script if necessary.

Basic higher-order list processing functions:

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

Generate the infinite list `[x, f x, f (f x), ...`

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

`map f xs` applies the function `f` to each element of the list `xs`, returning the corresponding list of results.

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

`filter p xs` returns the sublist of `xs` containing those elements which satisfy predicate `p`.

Fold functions:

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

`foldl f a xs` 'folds' list `xs` with function `f` and initial value `a`. That is:
`foldl f a [x1, x2, ..., xn]  = f (...(f (f a x1) x2)...) xn`
Often `f` is an infix operator:
`foldl (#) a [x1, x2, ..., xn]  = (...((a # x1) # x2)...) # xn`

```
> foldl1   :: (a -> a -> a) -> [a] -> a
```

Similar to `fold` but only for non-empty lists — the initial value is taken to be the head of the list (hence the slightly more restricted type). By definition:
`foldl1 f (x:xs) = fold f x xs`
Often `f` is an infix operator:
`foldl1 (#) [x1, x2, ..., xn]  = (...((x1 # x2) # x3)...) # xn`

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

Dual to `foldl`. That is:
`foldr f a [x1, x2, ..., xn]  = f x1 (f x2 (f x3 ...(f xn a)...)`
Often `f` is an infix operator:
`foldr (#) a [x1, x2, ..., xn]  = x1 # (x2 # (x3 # ...(xn # a)...)`

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

Dual to `foldl1`. For example:
`foldr1 (#) a [x1, x2, ..., xn]  = x1 # (x2 # (x3 # ...(xn-1 # xn)...)`

Miscellaneous and standard combinators:

```
> otherwise      :: Bool
> otherwise       = True
```

This is for the 'otherwise' clause in a definition using guards.

```
> error :: String -> a
```

Reports an error and prints its argument as the message. For example:

```
safediv x y
  | y /= 0     = x/y
  | otherwise  = error "divide by 0"
```

catches division by zero errors.

```
> const          :: a -> b -> a
> const k x       = k
```

Takes two arguments and ignores the second.

```
> id             :: a -> a
> id    x         = x
```

The identity function.

```
> curry          :: ((a,b) -> c) -> a -> b -> c
> curry f a b     =  f (a,b)
```

Curry a function to take its arguments one at a time, rather than as a pair.

```
> uncurry        :: (a -> b -> c) -> (a,b) -> c
> uncurry f (a,b) = f a b
```

The complement of curry, obviously.

```
> (.)        :: (b -> c) -> (a -> b) -> (a -> c)
> (f . g) x       = f (g x)
```

Function composition operator.

```
> flip           :: (a -> b -> c) -> b -> a -> c
> flip  f x y     = f y x
```

Swap the order of the arguments to a function.