Haskell – Crib Sheet

### Arithmetic Operators

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | |  |  | |  |  | | **Prelude>** 7 + 2  59  **Prelude>** 3 \* 4  12  **Prelude>** 19 - 18  1  **Prelude>** 4 / 2  2.0  **Prelude>** ( 19 – 18 + 1 ) \* 2  4 |
| |  |  | | --- | --- | | ^ | Power | | **Prelude>** 2 ^ 3  8 |
| |  |  | | --- | --- | | `div` | Integer division | | **Prelude>** 5 `div` 2  2 |
| |  |  | | --- | --- | | `mod` | Remainder | | **Prelude>** 5 `mod` 2  1 |

### Boolean and Relational Operators

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | && | AND operation | | **Prelude>** True && False  False |
| |  |  | | --- | --- | | || | OR operation | | **Prelude>** True || False  True |
| |  |  | | --- | --- | | not | Not operation | | **Prelude>** not False  True |
| |  |  | | --- | --- | | == | Is equal to | | **Prelude>** 5 == 4  False |
| |  |  | | --- | --- | | /= | Is not equal to | | **Prelude>** 5 /= 2  True |
| >= Greater than or equal to  > Greater than    <= less than or equal to  < less than | **Prelude>** 5 >= 2  True  **Prelude>** 5 > 2  True  **Prelude>** 5 <= 2  False  **Prelude>** 5 < 2  False |

### Built-in Functions

|  |  |
| --- | --- |
| sqrt - Square Root | **Prelude>** sqrt 25  5.0 |
| even - If integer is even return True if not return False | **Prelude>** even 12  True |
| odd - If number is odd return True if not return False | **Prelude>** odd 12  False |
| round - rounds to nearest integer | **Prelude>** round 5.7  6 |
| min - returns the minimum value if a pair of numbers | **Prelude>** min 5 6 7  6 |
| max - returns the maximum value of a pair of numbers | **Prelude>** max 5 6 7  7 |
| succ - returns the (succeeding) next integer | **Prelude>** succ 12  13 |

### Create Variables

|  |  |
| --- | --- |
| Integer | **Prelude>** let a = 10  **Prelude>** a  10 |
| String | **Prelude>** let b = "Hello"  **Prelude>** b  "Hello" |

### Create Lists

|  |  |
| --- | --- |
| Create Integer list | **Prelude>** let nums = [18,17,16,13,12,19] |
| Create String list | **Prelude>** let li = ["a","b","c","d"] |
| Create Empty list | **Prelude>** let x = [] |
| As a shorthand, sequences of numbers can be given by a pair of dots (..) between the first and last element | **Prelude>** let nums = [1..10]  **Prelude>** nums  [1,2,3,4,5,6,7,8,9,10] |
| If the second element is present, this gives the interval for the number sequence | **Prelude>** let nums = [2,4..12]  **Prelude>** nums  [2,4,6,8,10,12] |

### List processing

|  |  |
| --- | --- |
| Returns the first element in a list | **Prelude>** head [3,6,9,1,2]  3  **Prelude>** head ["a","b","c","d"]  "a" |
| Returns all but the first element in a list | **Prelude>** tail [3,6,9,1,2]  [6,9,1,2] |
| returns the last element in a list | **Prelude>** last [3,6,9,1,2]  2 |
| Returns all but the last element in a list | **Prelude>** init [3,6,9,1,2]  [3,6,9,1] |
| Return the length of a list | **Prelude>** length [3,6,9,1,2]  5 |
| Check to see if a list is empty | **Prelude>** let x = []  **Prelude>** null x  True  **Prelude>** let a = [1,2,3,4]  **Prelude>** null a  False |
| Return an element from the list (starting from zero) | **Prelude>** [3,6,9,1,2] !! 1  6 |
| Select the first 2 elements from a list | **Prelude>** take 2 [3,6,9,1,2]  [3,6] |
| Remove the first 2 elements from a list | **Prelude>** drop 2 [3,6,9,1,2]  [9,1,2] |
| Append to a list | **Prelude>** [1,2,3,4] ++ [5]  [1,2,3,4,5] |
| Prepend to a list | **Prelude>** [1] ++ [2,3,4,5]  [1,2,3,4,5]  **Prelude>** 1:[2,3,4,5]  [1,2,3,4,5] |
| Append 2 lists | **Prelude>** [3,6,9] ++ [1,2]  [3,6,9,1,2] |
| Reverse a list | **Prelude>** reverse[3,6,9,1,2]  [2,1,9,6,3] |
| Calculate the product of all values in a list | **Prelude>** product [3,6,9,1,2]  324 |
| Calculate the sum of all values in a list | **Prelude>** sum [3,6,9,1,2]  21 |

### Higher order functions

Higher-order functions are function that can take a function as an argument and/or return a function as a result.

|  |  |
| --- | --- |
| **Map** applies a function to each element of a list and returns the results in a list. Eg multiply all elements by 2 | **Prelude>** map (\*2) [1,2,3,4,5]  [2,4,6,8,10] |
| **Filter** applies a function to select items from a list. Eg select all values greater than 2 | **Prelude>** filter (>2) [1,2,3,4,5]  [3,4,5] |
| **Fold** reduces a list to single value using recursion. Eg sum the numbers in a list | **Prelude>** foldl (+) 6 [1,2,3,4,5]  15 |

### Functions

|  |  |
| --- | --- |
| **Function application -** Function applied to its arguments | **Prelude>** let add x y = x + y  **Prelude>** add 3 4  7 |
| **Function Type-**Functions have argument data types and an result data types  function :: argument\_type -> result\_type | double :: Integer -> Integer  double x = x \* 2 |
| **Composition of functions -** Functions can be combined to create a new function | **Prelude>** let f x = 8 \* 5  **Prelude>** let f x = x \* 5  **Prelude>** let g x = 3 + x^2  **Prelude>** f (g 4)  95  **Prelude>** g (f 4)  403 |
| **Partial function application -** When a function is applied to fewer than the total number of parameters. | **Prelude>** let add x y = x + y  **Prelude>** let add3 = add 3  **Prelude>** add3 4  7 |

### First class object

In functional programming languages functions can be first class objects. This mean they can be:

|  |  |
| --- | --- |
| assigned to variables |  |
| Passed as arguments to other functions | **Prelude>** let double x = x \* 2  **Prelude>** map double [1,2,3]  [2,4,6] |
| returned from other functions | **Prelude>** let add x y = x + y  **Prelude>** let add3 = add 3 |
| stored in data structures | **Prelude>** let add x y = x + y  **Prelude>** let a = [add 3 4, 5, 6] |