CM500293 Week 1 Cheat Sheet: Haskell and the Lambda Calculus

Yellow = code Purple = ghci Green = template / working

Defining and calling functions

```
<function> :: <type>
<function> <parameters> = <body>
```

```
addSix :: Int -> Int
addSix x = x + 6
```

Call with spaces: `addSix 3`, `ourMax 3 8`. Negative numbers need brackets: `addSix (-4)`.

Same input, same output, always. No reassignment, no loops, no side effects.

Core types

`Bool`	True, False
`Int`	fixed size (-2^63 .. 2^63-1), fast
`Integer`	unbounded whole numbers
`Float, Double`	decimals
`Char`	`'a'` single quotes
`String`	`"hi"` is exactly `[Char]`
`[a]`, `(a,b)`	a = type variable: any type

Guards

```
grade :: Int -> Char
grade n
  | n >= 80   = 'A'
  | n >= 70   = 'B'
  | otherwise = 'F'
```

Checked top to bottom; first true wins. `otherwise` is always True. Indent the `|` lines.

Recursion

```
factorial :: Integer -> Integer
factorial x
  | x <= 1    = 1
  | otherwise = x * factorial (x-1)
```

Base case answers directly. **Progress**: every call moves closer to it.

Currying and partial application

`Int -> Int -> Int`	= `Int -> (Int -> Int)` (types group right)
`ourMax 3 8`	= `(ourMax 3) 8` (application groups left)

```
mult x y = x * y
double = mult 2      -- double :: Int -> Int
```

Uncurried: `maxUC :: (Int,Int) -> Int`, called `maxUC (3,8)`.

Operators

Infix to prefix: `(+) 5 2`. Prefix to infix: `5 `max` 3` (backticks).

Application binds tighter than any operator: `max 3 2 + 5` = `(max 3 2) + 5` = 8.

Bool: `not`, && (and), || (or). Compare: `== /= < <= > >=`. Comment: `--`

Lists

A list is `[]` (empty) or `x : xs` (head x consed onto tail xs).

```
[1,2,3] = 1 : 2 : 3 : []
```

```
head [1,2,3] = 1       tail [1,2,3] = [2,3]
init [1,2,3] = [1,2]   last [1,2,3] = 3
[1,2] ++ [3] = [1,2,3] length [1,2,3] = 3
```

`:` adds ONE element to the front. ++ joins TWO lists.

Pattern matching

```
ourLength :: [a] -> Int
ourLength []     = 0
ourLength (_:xs) = 1 + ourLength xs
```

One equation per shape, tried top to bottom. _ = wildcard. Specific patterns first.

Missing a case (e.g. `[]`) compiles, but crashes at runtime: *Non-exhaustive patterns*.

Recipe: answer for `[]`? Then, trusting the function on `xs`, use x to finish.

Pairs and tuples

```
swap :: (Int,String) -> (String,Int)
swap (n,st) = (st,n)
```

`fst (1,"a") = 1`, `snd (1,"a") = "a"`. A list of pairs is a lookup table: `((k,v):rest)`. Pair functions do not accept triples.

ghci and errors

`:l file.hs` load `:r` reload `:t e` type `:q` quit

Variable not in scope	typo, or not (re)loaded
parse error on input '|'	guards not indented
No instance for (Num (Int -> Int))	missing brackets: `addSix (-4)`
Couldn't match ... Expected / Actual	wrong argument type
Non-exhaustive patterns	runtime: missing case

Read the location (line:column), then **Expected** vs **Actual**.

λ-calculus: syntax

```
M, N ::= x  |  λx.M  |  M N
```

x	variable
λx.M	abstraction (Haskell `\x -> M`)
M N	application (M applied to N)

Bracketing conventions

Application groups left	`M N P = (M N) P`
λ body reaches right	`λx.M N = λx.(M N)`
Shared λ	`λx y.M = λx.λy.M`

Trap: `λx.x y` is `λx.(x y)`, not `(λx.x) y`.

Redexes and normal forms

Redex: any subterm `(λx.M) N`, a λ applied to something.

Normal form: no redex left; a final answer.

Free and bound variables

	FV	BV
x	{x}	∅
λx.M	FV(M) \ {x}	BV(M) ∪ {x}
M N	FV(M) ∪ FV(N)	BV(M) ∪ BV(N)

Ask about each **occurrence**: a name can be free in one place and bound in another.

α-conversion

`λx.M →α λz.M[z/x]  (z fresh: not anywhere in M)`

Rename a binder and its bound occurrences. Never rename free variables.

Capture-avoiding substitution M[N/x]

1. `x[N/x] = N`
2. `y[N/x] = y`
3. `(λx.M′)[N/x] = λx.M′` (rebound: stop)
4. `(λy.M′)[N/x] = λz.(M′[z/y][N/x])` z fresh
5. `(M₁ M₂)[N/x] = (M₁[N/x])(M₂[N/x])`

Rename in case 4 only matters when y is free in N.

β-reduction

```
(λx.M) N  →β  M[N/x]
M →β* N : zero or more steps
```

```
(λx.x x)(λy.y) →β (λy.y)(λy.y) →β λy.y
(λx.λy.x) y    →β λz.y          (rename first!)
```

```
*Main> (\x y -> x) 1 2
1        -- (λx.λy.x) 1 2 →β (λy.1) 2 →β 1
```