Haskell Cheat Sheet

Matthijs Ooms `mo@tty.nl`
Michel Rijnders `mies@tty.nl`

Joy of Coding
March 1, 2013

Data Types

Characters and Strings

- `'a'` - single character
- `"abc"` - Unicode string

Numbers

- `42` - integer
- `3.14` - floating point

Booleans

- `True`
- `False`

Lists

- `[]` - empty list
- `[1,2,3]` - list of three integers
- `1 : 2 : 3 : []` - alternate way to write a list using "cons" (`:`)
- `"abc"` - list of three characters (strings are lists)

Tuples

- `(1, "a")` - 2 element tuple of a number and a string
- `(1, 'a', 'b', 'c')` - 4 element tuple of a number and 3 characters

Operators

- `+` - addition
- `-` - subtraction
- `/` - division
- `*` - multiplication
- `==` - equals
- `<` - less than
- `<=` - less than, or equals
- `>` - greater than
- `>=` - greater than, or equals
- `++` - list concatenation

Functions

Functions are defined by declaring their name, and arguments, and a equals sign:

```
square x = x * x
```

Function names must start with a lowercase letter or an underscore.

Modules

A module is a compilation unit which exports functions. To make a Haskell file (`.hs`) a module, add a module declaration to the top:

```
module OurModule where
```

Module names must start with a uppercase letter.

Exports

If an export list is not provided, then all functions are exported. Limiting what is exported is achieved by adding a list of names before the `where` keyword:

```
module OurModule (f,g,...) where
```

Imports

To import evertyhing exported by a module just use the module name:

```
import Data.List
```

Importing selectively is achieved by giving a list of names:

```
import Data.List (intersperse)
```

GHCi

- `:cd <dir>` - change directory
- `:load <module>` - load module
- `:reload` - reload all modules
- `:help` - show help
- `:quit` - exit

Pattern Matching

Multiple "clauses" of a function can be defined by "pattern-matching" on the values of arguments.

```
agree "y" = "Great!"
agree "n" = "Too bad."
agree _ = "Huh?"
```

Lists

- `(x:xs) = [1,2,3]` binds `x` to `1`
- `(x:_) = [1,2,3]` binds `x` to `1` and `xs` to `[2,3]`
- `(_:xs) = [1,2,3]` binds `xs` to `[2,3]`
- `(x:y:z:[]) = [1,2,3]` binds `x` to `1`, `y` to 2, and `z` to 3

Note that the empty list only matches the empty list.

Currying

Functions do not have to get all their arguments at once. Consider the following function:

```
add x y = x + y
```

Using add we can now write functions that add a certain value:

```
addTwo = add 2
five = addTwo 3
```

Anonymous Functions

An anonymous function (i.e. a lambda expression or lambda for short), is a function without a name. They can be defined at any time like so:

```
addTwo = \x -> x + 2
```

Local Functions

Local functions can be defined within a function using `let`. The `let` keyword must always be followed by `in`.

```
five = let addTwo = add 2
       in addTwo 3
```

Local functions can also be defined using `where`:

```
five = addTwo 3
       where addTwo = add 2
```

List Comprehensions

A list comprehension creates a list of values based on the generators and guards given:

```
f xs = [x * x | x <- xs, mod x 2 == 0]
```

Another example:

```
up cs = [c | c <- cs, isUpper c]
```

Case

`case` is similar to a `switch` statement in C or Java, but can match a pattern.

```
agree x =
  case x of
    "y" -> "Great!"
    "n" -> "Too bad."
    _ -> "Huh?"
```

If, Then, Else

As opposed to languages like C and Java `if, then, else` is an expression, not a control statement, i.e. it always returns a value:

```
agree x =
  if x == "y"
    then "Great!"
    else "Too bad."
```