HASKELL QUICK REFERENCE

IEEE VisWeek Tutorial 2008

Lexical Syntax

Comments	
--	-- end-of-line comment
{- -}	{- multi-line comment {-with nesting-} -}
{-# #-}	{-# PRAGMA usually a helpful hint to the compiler #-}
Identifier names	
eat3Chars	functions, variables and type variables start with lowercase
Double	concrete typenames / constructors start with uppercase
a	typically, variable names in argument positions are short
foo_Bar'34baz	underscores _, primes ‘, digits, mixed case, are permitted
a ++ b	symbols are infix operator names, ++ takes two arguments
a :-: b	symbols starting with a colon : are infix constructor names
(++) a b	an infix symbol can be used prefix, by enclosing in parens
a `foo` b	a prefix name can be used infix, by enclosing in backquotes
Strings	
“hello world”	strings use double-quotes
‘c’	character constants use single quotes
Lists have two constructors, empty [], and cons (:) which joins one elem to a list	
(x : xs)	a list with x at the front, xs is the rest of the list
(x : y : z : [])	a list of three things
[x, y, z]	square brackets with commas are sugar for (x:y:z:[])
[2 .. 15]	list containing a numeric range
[2, 4 .. 16]	list containing a stepped numeric range
[40, 39 .. 0]	ranges can go down as well as up
Tuples	
(x, y)	a paired value - in round parentheses with commas
(x, y, z)	a triple of values
Numbers	
42	value of any number type: Int, Integer, Float, Double, etc
42.0	value of any fractional type: Float, Double, Rational, Complex
1.2e3	scientific notation (= 1.2 x 10^3)
Equals symbols	
=	single = is a definition of a value
==	double == is a comparison operator returning a Boolean
Lambda notation	
(\x-> foo)	backslash is a poor ASCII version of the lambda symbol
->	ASCII version of a right arrow (used in lambdas, **case** discrimination, and types of functions)
Layout	
defn **where** defn2	Indentation is used intuitively to indicate logical structuring: anything indented right to the right “belongs” in this group
{ defn; defn; }	Indentation can be overridden by using explicit braces and semicolons.

Expressions

Function application	
f x	space between function name f and argument expression x
f $ x	function f applied to expression x (but right-associative)
x ++ y	operators (symbols) are applied infix
(++) x y	an infix operator can be applied prefix by enclosing in parens
x `f` y	a prefix function can be applied infix, enclosed in backquotes
f (3+4) (not y)	round parentheses to group and nest function applications
(+1)	a function/operator can be *partially* applied to only some args
Anonymous functions	
\x **->** expr	backslash pretends to be a lambda. this anonymous function names its argument x
\ (x:xs) **->** expr	this anonymous function pattern-matches its list argument
(\x **->** x+3) 5	often need parentheses around a lambda term to apply it
Data construction	
Build (1+2) True	Values are built by applying a data constructor as a function
Local naming	
let f x = rhs **in** expr	define a function f which can only be used within the given expr
let (x:xs) = rhs **in** expr	evaluate the rhs, whose result is a list. Pattern-match the components of the list, then use the names x and xs within the expr
Conditionals	
if a **then** b **else** c	a, b, and c are any expressions of the right types
case expr **of** pat0 -> expr0 pat1 -> expr1 otherwise -> e	discriminate between alternative constructions of the value denoted by expr - alternative patterns are indented. a catch-all default case is called *otherwise*
Sequencing evaluation	

do pat <- iocomp
(x:xs) <- action
something x
return y
	evaluate the side-effecting computation *iocomp*, and pattern-match its result against *pat*, for use in later actions.
	subsequent actions are indented to match the first one.
	actions can use variables bound by patterns higher up.

Pattern-matching and binding	
f (C x 3)	functions can pattern-match their arguments. A pattern is an application of a constructor to either literal values, fresh variable names, or other patterns.
f (C (2:3:y) 3)	patterns can be nested. The value of the rest of the list after the first two elements is bound to the name y if the first two elements match the given pattern
case expr **of** pat0 -> expr0 pat1 -> expr1 otherwise -> e	when there are multiple overlapping patterns, e.g. in a case expression or in a series of equations defining a function, the patterns are matched top-to-bottom, left-to-right.

Definitions

Function definition (function names start with a lower-case letter)	
f :: t	the function named f “has type” t. Known as a *type signature*.
f arg0 arg1 = rhs	function named f has two named arguments, result is rhs
f (x:xs) = rhs	function pattern-matches on its list argument, naming its parts
f x y = rhs **where** rhs = expr	an equational definition can have local definitions contained in an indented “where” clause
f n | n <0 = rhsNeg | n >0 = rhsPos	guards on equations: tests are indented with vertical bar. there are multiple right-hand-sides, each guarded by a test
Type definition (type names and constructors start with an Upper-case letter)	
data T a = C a Int	user-defined datatype T takes a type parameter ‘a’ values of type T are constructed using C values of type T contain one value of type ‘a’ and an Int
data U = V | W | X	user-defined datatype U values of type U can be either a V construction, W, or X
type M = T Bool	M is a synonym for T Bool - the names are interchangeable
newtype N = N (T U)	N is like a synonym for (T U), except the names are *not* interchangeable
Other top-level definitions	
module M **where**	every module has a capitalised name
import Data.Word	import and use functions from another module
class C a **where** method :: type	define a predicate over types. class methods are indented, and must give a type signature
instance C Int **where** method = impl	instance of a class predicate for a specific type. the class method definition is indented - no type signature
Basic types	
Int	limited precision signed integers (e.g. 30 bits)
Integer	arbitrary precision signed integers
Rational	arbitrary precision fractional numbers
Float	floating-point limited-precision fractional numbers
Double	double-word floating-point limited-precision fractionals
Bool	Booleans (constants: True, False)
Char	single Unicode characters
String	textual sequence of characters (= [Char])
Bigger types	
(a,b)	pair of types a and b (a and b are type variables)
[a]	list with element type a (a stands for any type)
a -> b	function with argument type a, result type b
a -> b -> c	function with two arguments, of types a and b, result type c