Haskell 98 Prelude Cheat Sheet [version 0.0a Apr 2008]

Standard Haskell Types

unit type

```
data () = () deriving (Eq, Ord, Enum, Bounded,
Read, Show) -- (e) is equivalent to e
```

*unit type has nullary constructor () and bottom _|_*

abstract functions (no constructors)

```
id    :: a -> a
const :: a -> b -> a
(.)   :: (b -> c) -> (a -> b) -> a -> c
flip     :: (a -> b -> c) -> b -> a -> c
seq      :: a -> t -> t -- primitive only
infixr 0 $, $!
($), ($!) :: (a -> b) -> a -> b
asTypeOf  :: a -> a -> a -- overload selector
```

Boolean type

```
data  Bool  =  False | True deriving
   (Read, Show, Eq, Ord, Enum, Bounded)
(&&), (||) :: Bool -> Bool -> Bool
not        :: Bool -> Bool
otherwise  :: Bool -- true
until    :: (a -> Bool) -> (a -> a) -> a -> a
```

Character type

Unicode, Latin1, ASCII

```
data Char = 'a' | 'b' ...
   deriving (Eq, Ord Enum, Bounded)
```

more functions in module Char, now hidden from prelude

Generic Errors

```
error     :: [Char] -> a
undefined :: a
```

Tuples

```
data (a,b) = (a,b) deriving (Eq, Ord, Bounded)
fst     :: (a,b) -> a
snd     :: (a,b) -> b
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

(a,b,c) => fst3, snd3, more functions for big tuples

Standard List Functions

```
list type with (:) constructor
(:) :: a -> [a] -> [a]
data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
infixl 9  !!
infixr 5  ++
infix  4  `elem`, `notElem`
map       :: (a -> b) -> [a] -> [b]
(++)      :: [a] -> [a] -> [a]
filter    :: (a -> Bool) -> [a] -> [a]
concat    :: [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
head      :: [a] -> a
tail      :: [a] -> [a]
last      :: [a] -> a
init      :: [a] -> [a]
null      :: [a] -> Bool
length    :: [a] -> Int
(!!)      :: [a] -> Int -> a -- 0 origin index
foldl     :: (a -> b -> a) -> a -> [b] -> a
foldl1    :: (a -> a -> a) -> [a] -> a
scanl     :: (a -> b -> a) -> a -> [b] -> [a]
scanl1    :: (a -> a -> a) -> [a] -> [a]
foldr     :: (a -> b -> b) -> b -> [a] -> b
foldr1    :: (a -> a -> a) -> [a] -> a
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanr1    :: (a -> a -> a) -> [a] -> [a]
iterate           :: (a -> a) -> a -> [a]
repeat            :: a -> [a]
replicate         :: Int -> a -> [a]
cycle             :: [a] -> [a]
take, drop        :: Int -> [a] -> [a]
splitAt           :: Int -> [a] -> ([a],[a])
takeWhile, dropWhile:: (a -> Bool) -> [a] -> [a]
span, break   :: (a -> Bool) -> [a] -> ([a],[a])
reverse           :: [a] -> [a]
and, or           :: [Bool] -> Bool
any, all          :: (a -> Bool) -> [a] -> Bool
elem, notElem     :: (Eq a) => a -> [a] -> Bool
lookup    :: (Eq a) => a -> [(a,b)] -> Maybe b
maximum, minimum :: (Ord a) => [a] -> a
sum, product      :: (Num a) => [a] -> a
zip       :: [a] -> [b] -> [(a,b)]
zipWith   :: (a->b->c) -> [a]->[b]->[c] -- map2
unzip             :: [(a,b)] -> ([a],[b])
```

(zip3, unzip3, zipWith3, more functions in module List

String List Functions

```
type  String  =  [Char]
lines, words      :: String -> [String]
unlines, unwords :: [String] -> String
```

Monadic types

```
data  Maybe a =  Nothing | Just a  deriving
   (Eq, Ord, Read, Show)
maybe :: b -> (a -> b) -> Maybe a -> b
data  Either a b =  Left a | Right b  deriving
   (Eq, Ord, Read, Show)
either:: (a -> c) -> (b -> c) -> Either a b -> c
data  Ordering =  LT | EQ | GT deriving
   (Eq, Ord, Bounded, Enum, Read, Show)
```

more utilities in modules Monad, Maybe

Numeric types

```
data  Int  =  minBound .. -1 | 0 | 1 .. maxBound
   instances: Eq, Ord, Num, Real Integral, Enum,
Bounded
data  Integer  =  ... -1 | 0 | 1 ...
   instances: Eq, Ord, Num, Real, Integral, Enum
data  Float, data Double
   instances: Eq, Ord, Num, Real, Fractional,
   Floating, RealFrac, RealFloat, Enum
```

Standard Haskell Classes

```
class  Eq a  where
   (==), (/=)  ::  a -> a -> Bool
class  (Eq a) => Ord a  where
   compare              :: a -> a -> Ordering
   (<), (<=), (>=), (>) :: a -> a -> Bool
   max, min             :: a -> a -> a
class  Enum a  where
   succ, pred     :: a -> a
   toEnum         :: Int -> a
   fromEnum       :: a -> Int
   enumFrom       :: a -> [a]    -- [n..]
   enumFromThen   :: a -> a -> [a]  [n,n'..]
   enumFromTo     :: a -> a -> [a] -- [n..m]
   enumFromThenTo :: a -> a -> a -> [a]
       -- [n,n'..m]
class  Bounded a where minBound, maxBound :: a
```

The Read and Show Classes

```
type  ReadS a = String -> [(a,String)]
type  ShowS   = String -> String
class  Read a  where
   readsPrec :: Int -> ReadS a
   readList  :: ReadS [a]
instances: Int, Integer, Float, Double, (),
Char, (Read a) => Read [a], (Read a, Read b) =>
Read (a,b)
class  Show a  where
   showsPrec :: Int -> a -> ShowS
   show      :: a -> String
   showList  :: [a] -> ShowS
instances: Int, Integer, Float, Double, (),
Char, (Show a) => Show [a], (Show a, Show b) =>
Show (a,b)
reads   :: (Read a) => ReadS a
shows   :: (Show a) => a -> ShowS
read    :: (Read a) => String -> a
showChar          :: Char -> ShowS
showString        :: String -> ShowS
showParen         :: Bool -> ShowS -> ShowS
readParen         :: Bool -> ReadS a -> ReadS a
showLitChar       :: Char -> ShowS
readLitChar       :: ReadS Char
lexLitChar, lex   :: ReadS String
```

Monadic Classes

```
class  Functor f  where
   fmap  :: (a -> b) -> f a -> f b
class  Monad m  where
   (>>=)   :: m a -> (a -> m b) -> m b
   (>>)    :: m a -> m b -> m b
   return  :: a -> m a
   fail    :: String -> m a
sequence  :: (Monad m) => [m a] -> m [a]
sequence_ :: (Monad m) => [m a] -> m ()
mapM  :: (Monad m) => (a -> m b) -> [a] -> m [b]
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
(=<<) :: (Monad m) => (a -> m b) -> m a -> m b
```

Numeric classes

```
class  (Eq a, Show a) => Num a  where
   (+), (-), (*)    :: a -> a -> a
   negate           :: a -> a
   abs, signum      :: a -> a
   fromInteger      :: Integer -> a
class  (Num a, Ord a) => Real a  where
   toRational       ::  a -> Rational
class  (Real a, Enum a) => Integral a  where
   quot, rem, div, mod :: a -> a -> a
   quotRem, divMod  :: a -> a -> (a,a)
   toInteger        :: a -> Integer
class  (Num a) => Fractional a  where
   (/)              :: a -> a -> a
   recip            :: a -> a
   fromRational     :: Rational -> a
class  (Fractional a) => Floating a  where
   pi                  :: a
   exp, log, sqrt      :: a -> a
   (**), logBase       :: a -> a -> a
   sin, cos, tan, asin, acos, atan, sinh, cosh,
   tanh, asinh, acosh, atanh :: a -> a
class  (Real a, Fractional a) => RealFrac a
where
   properFraction  :: (Integral b) => a -> (b,a)
   truncate, round :: (Integral b) => a -> b
   ceiling, floor  :: (Integral b) => a -> b
class  (RealFrac a, Floating a) => RealFloat a
where
   floatRadix      :: a -> Integer
   floatDigits     :: a -> Int
   floatRange      :: a -> (Int,Int)
   decodeFloat     :: a -> (Integer,Int)
   encodeFloat     :: Integer -> Int -> a
   exponent        :: a -> Int
   significand     :: a -> a
   scaleFloat      :: Int -> a -> a
   isNaN, isInfinite, isDenormalized,
   isNegativeZero, isIEEE :: a -> Bool
   atan2           :: a -> a -> a
class  (RealFloat a) => Complex a
```

more functions in modules Ratio and Complex

Numeric functions

```
subtract  :: (Num a) => a -> a -> a
even, odd :: (Integral a) => a -> Bool
gcd, lcm  :: (Integral a) => a -> a -> a
(^)       :: (Num a, Integral b) => a -> b -> a
(^^)::(Fractional a, Integral b) => a -> b -> a
fromIntegral :: (Integral a, Num b) => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
```

Standard Prelude I/O Functions

```
data IO a = ... --abstract type
```

Standard Output Functions

```
putChar    :: Char -> IO ()
putStr, putStrLn :: String -> IO ()
print      :: (Show a) => a -> IO ()
```

Standard Input Functions

```
getChar      :: IO Char
getLine, getContents     :: IO String
interact     :: (String -> String) -> IO ()
readIO       :: (Read a) => String -> IO a
readLn       :: (Read a) => IO a
```

Files

```
type FilePath =  String
writeFile  :: FilePath -> String -> IO ()
appendFile  :: FilePath -> String -> IO ()
readFile    :: FilePath           -> IO String
```

IO monad

```
(>>=) :: IO a -> (a -> IO b) -> IO b -- bind
(>>)  :: IO a -> IO b        -> IO b
```

Exception Handling in I/O Monad

```
data IOError -- system dependent
userError :: String -> IOError
ioError   :: IOError -> IO a
catch     :: IO a -> (IOError -> IO a) -> IO a
```

Haskell 98 Prelude Cheat Sheet [version 0.0a Apr 2008]

Standard Library Modules

module System (98)

```
data ExitCode = ExitSuccess | ExitFailure Int
  deriving (Eq, Ord, Read, Show)
getArgs   :: IO [String]
getProgName          :: IO String
getEnv    :: String -> IO String
system    :: String -> IO ExitCode
exitWith  :: ExitCode -> IO a
exitFailure          :: IO a
```

module Ix (98)

```
class  (Ord a) => Ix a  where
  range               :: (a,a) -> [a]
  index               :: (a,a) -> a -> Int
  inRange             :: (a,a) -> a -> Bool
instances: Char, Int, Integer, Bool, Ordering,
(),  (Ix a,Ix b) => Ix (a, b)
rangeSize :: (Ix a) => (a,a) -> Int
```

module Array (98)

```
infixl 9  !, //
data (Ix i) => Array i e = MkArray (i,i) (i ->
e) deriving ()
array    :: (Ix i) => (i,i) -> [(i,e)] ->
  Array i e
listArray:: (Ix i) => (i,i) -> [e] -> Array i e
(!)      :: (Ix i) => Array i e -> i -> e
bounds   :: (Ix i) => Array i e -> (i,i)
indices  :: (Ix i) => Array i e -> [i]
elems    :: (Ix i) => Array i e -> [e]
assocs   :: (Ix i) => Array i e -> [(i,e)]
(//)     :: (Ix i) => Array i e ->
  [(i,e)] -> Array i e
accum    :: (Ix i) => (e -> a -> e) ->
  Array i e -> [(i,a)] -> Array i e
accumArray :: (Ix i) => (e -> a -> e) -> e ->
  (i,i) -> [(i,a)] -> Array i e
ixmap    :: (Ix j, Ix i) => (i,i) -> (i ->
  j) -> Array j e -> Array i e
instances: (Ix i) => Functor (Array i), (Ix i,
Eq e)  => Eq (Array i e), (Ix i, Ord e) => Ord
(Array i e), (Ix i, Show i, Show e) => Show
(Array i e), (Ix i, Read i, Read e) => Read
(Array i e)
```

Module Char (98)

```
isAscii, isLatin1, isControl, isPrint, isSpace,
isUpper, isLower, isAlpha, isDigit, isOctDigit,
isHexDigit, isAlphanum :: Char -> Bool
toUpper, toLower    :: Char -> Char
digitToInt :: Char -> Int -- all hex digits
intToDigit :: Int -> Char -- 0..15
ord        :: Char -> Int
chr        :: Int  -> Char
readLitChar :: Reads Char
showLitChar :: Char -> ShowS
```

Module IO (98)

```
data Handle = ... -- implementation-dependent
  instances: Eq, Show
data HandlePosn = ... -- impl.-dependent
  instances: Eq, Show
data IOMode =  ReadMode | WriteMode |
  AppendMode | ReadWriteMode deriving (Eq, Ord,
Ix, Bounded, Enum, Read, Show)
data BufferMode  =  NoBuffering |
LineBuffering |  BlockBuffering (Maybe Int)
deriving (Eq, Ord, Read, Show)
data SeekMode =  AbsoluteSeek | RelativeSeek |
SeekFromEnd deriving (Eq, Ord, Ix, Bounded,
Enum, Read, Show)
stdin, stdout, stderr :: Handle
openFile      :: FilePath -> IOMode -> IO Handle
hClose        :: Handle -> IO ()
hFileSize     :: Handle -> IO Integer
hIsEOF        :: Handle -> IO Bool
isEOF         :: IO Bool -- =  hIsEOF stdin
hSetBuffering :: Handle  -> BufferMode -> IO ()
hGetBuffering :: Handle  -> IO BufferMode
hFlush        :: Handle -> IO ()
hGetPosn      :: Handle -> IO HandlePosn
hSetPosn      :: HandlePosn -> IO ()
hSeek         :: Handle -> SeekMode ->
  Integer -> IO ()
hWaitForInput       :: Handle -> Int -> IO Bool
hReady              :: Handle -> IO Bool
hGetChar, hLookAhead   :: Handle -> IO Char
hGetLine, hGetContents :: Handle -> IO String
hPutChar            :: Handle -> Char -> IO ()
hPutStr, hPutStrLn :: Handle -> String -> IO ()
hPrint        :: Show a => Handle -> a -> IO ()
hIsOpen, hIsClosed, hIsReadable, hIsWritable,
hIsSeekable           :: Handle -> IO Bool
isAlreadyExistsError, isDoesNotExistError,
isAlreadyInUseError, isFullError, isEOFError,
isIllegalOperation, isPermissionError,
isUserError         :: IOError -> Bool
ioeGetErrorString  :: IOError -> String
ioeGetHandle       :: IOError -> Maybe Handle
ioeGetFileName     :: IOError -> Maybe FilePath
try       :: IO a -> IO (Either IOError a)
bracket   :: IO a -> (a -> IO b) ->
  (a -> IO c) -> IO c
bracket_  :: IO a -> (a -> IO b) -> IO c -> IO c
```

Module Directory (98)

```
data Permissions = Permissions { readable,
writable, executable, searchable :: Bool }
  instances: Eq,Ord, Read, Show
createDirectory, removeDirectory
  :: FilePath -> IO ()
removeFile   :: FilePath -> IO ()
renameDirectory :: FilePath -> FilePath -> IO ()
renameFile   :: FilePath -> FilePath -> IO ()
getDirectoryContents :: FilePath -> IO [FilePath]
getCurrentDirectory  :: IO FilePath
setCurrentDirectory  :: FilePath -> IO ()
doesDirectoryExist, doesFileExist :: FilePath ->
IO Bool
getPermissions :: FilePath -> IO Permissions
setPermissions::FilePath -> Permissions -> IO ()
getModificationTime :: FilePath -> IO ClockTime
```

module Random (98)

```
class RandomGen g where
  genRange :: g -> (Int, Int)
  next     :: g -> (Int, g)
  split    :: g -> (g, g)
data StdGen = ... -- Abstract
  instances: RandomGen, Read, Show
mkStdGen :: Int -> StdGen
class Random a where
  randomR :: (RandomGen g) =>
    (a, a) -> g -> (a, g)
  random  :: (RandomGen g) => g -> (a, g)
  randomRs :: (RandomGen g) =>
    (a, a) -> g -> [a]
  randoms  :: RandomGen g => g -> [a]
  randomRIO :: (a,a) -> IO a
  randomIO  :: IO a
instances: Int, Integer, Float, Double, Bool,
Char
newStdGen    :: IO StdGen -- global generator
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

module Monad (98)

more monadic utilities here

```
class  Monad m => MonadPlus m  where
  mzero  :: m a
  mplus  :: m a -> m a -> m a
join    :: (Monad m) => m (m a) -> m a
guard   :: (MonadPlus m) => Bool -> m ()
when    :: (Monad m) => Bool -> m () -> m ()
unless :: (Monad m) => Bool -> m () -> m ()
ap      :: (Monad m) => m (a -> b) -> m a -> m b
mapAndUnzipM :: (Monad m) =>
 (a -> m (b,c)) -> [a] -> m ([b], [c])
zipWithM     :: (Monad m) =>
 (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM_    :: (Monad m) =>
 (a -> b -> m c) -> [a] -> [b] -> m ()
foldM   :: (Monad m) =>
 (a -> b -> m a) -> a -> [b] -> m a
filterM :: (Monad m) =>
 (a -> m Bool) -> [a] -> m [a]
msum    :: (MonadPlus m) => [m a] -> m a
liftM  :: Monad m => (a -> b) -> (m a -> m b)
```

liftM2 liftM3, liftM4, liftM5 for big tuples

module Maybe

more Maybe utilities here

```
isJust, isNothing :: Maybe a -> Bool
fromJust    :: Maybe a -> a
fromMaybe   :: a -> Maybe a -> a
listToMaybe :: [a] -> Maybe a
maybeToList :: Maybe a -> [a]
catMaybes   :: [Maybe a] -> [a]
mapMaybe    :: (a -> Maybe b) -> [a] -> [b]
```

module Complex

```
infix  6  :+  -- constructor
data  (RealFloat a) => Complex a = !a :+ !a
realPart, imagPart, magnitude, phase ::
  (RealFloat a) =>  Complex a -> a
conjugate   :: (RealFloat a) =>
  Complex a -> Complex a
mkPolar     :: (RealFloat a) =>
  a -> a -> Complex a
cis         :: (RealFloat a) =>
  a -> Complex a
polar       :: (RealFloat a) =>
  Complex a -> (a,a)
instances: Eq, Read, Show, Num, Fractional,
Floating
```

(+), (-), (*), negate, abs, signum, fromInteger, (/), fromRational, pi, exp, log, sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, asinh, acosh, atanh

module Time

```
data ClockTime = ... -- Implementation-dependent
  instances: Ord, Eq
data Month =  January | ...
  deriving (Eq, Ord, Enum, Bounded, Ix, Read,
Show)
data Day   =  Sunday | ...
  deriving (Eq, Ord, Enum, Bounded, Ix, Read,
Show)
data CalendarTime = CalendarTime { ctYear::Int,
ctMonth::Month, ctDay, ctHour, ctMin, ctSec ::
Int, ctPicosec :: Integer, ctWDay :: Day,
ctYDay :: Int, ctTZName :: String, ctTZ :: Int,
ctIsDST :: Bool } deriving (Eq, Ord, Read, Show)
data TimeDiff = TimeDiff { tdYear, tdMonth,
tdDay, tdHour, tdMin, tdSec :: Int, tdPicosec ::
Integer } deriving (Eq, Ord, Read, Show)
getClockTime   :: IO ClockTime
addToClockTime :: TimeDiff -> ClockTime ->
  ClockTime
diffClockTimes :: ClockTime -> ClockTime ->
  TimeDiff
toCalendarTime :: ClockTime -> IO CalendarTime
toUTCTime      :: ClockTime -> CalendarTime
toClockTime    :: CalendarTime -> ClockTime
calendarTimeToString :: CalendarTime -> String
formatCalendarTime   :: TimeLocale -> String ->
  CalendarTime -> String
```

module CPUTime

```
getCPUTime        :: IO Integer
cpuTimePrecision  :: Integer
```

module Locale

```
data TimeLocale = TimeLocale { wDays ::
[(String, String)], week days months ::
[(String, String)], amPm :: (String, String),
dateTimeFmt, dateFmt, timeFmt, time12Fmt ::
String } deriving (Eq, Ord, Show)
defaultTimeLocale :: TimeLocale
defaultTimeLocale =  TimeLocale {
 wDays  = [("Sunday", "Sun"), ...
 months = [("January", "Jan"), ...
 amPm = ("AM", "PM"),
 dateTimeFmt = "%a %b %e %H:%M:%S %Z %Y",
 dateFmt = "%m/%d/%y",
 timeFmt = "%H:%M:%S",
 time12Fmt = "%I:%M:%S %p" }
```

Note: This is Haskell 98 Prelude only.
Find more in ***Haskell Hierarchical Libraries.***