### Defining Functions

|  |
| --- |
| double x = x quadruple x = double (double x) |

### Lists

"hello" is **syntactic sugar** for ['h','e','l','l','o'] .

A list is either:

1. Empty (nil): []
2. An element **consed** onto the front of another list: x:xs

[1,2,3] is syntactic sugar for 1:(2:(3:[])).

### Pattern Matching

|  |
| --- |
| length [] = 0 length (x:xs) = 1 + length xs  and True True = True and \_ \_ = False -- \_ will match any value.  head (x:\_) = x tail (\_:xs) = xs |

### Higher Order Functions

We can define functions that:

1. Take other **functions as arguments**.
2. **Return functions** as results.

In the following example:

* e is the value for the empty list.
* op is the function to apply to the head and the recursive result.

|  |
| --- |
| fold e op [] = e fold e op (x:xs) = op x (fold e op xs)  sum = fold 0 (+) prod = fold 1 (\*) length = fold 0 (\\_ y -> y + 1) -- Passes a lambda function. |

### Types

|  |  |
| --- | --- |
| Int | Machine-width integer |
| Integer | Big integer - expands in size |
| Char | Character |
| [Int] | List of Ints |
| [Int] -> Int | Function from Int list to Int |

### Type Polymorphism

length works for lists of arbitrary type, where a denotes a **type variable**:

|  |
| --- |
| length :: [a] -> Int |

### Laziness

Haskell is a lazy language, and only evaluates values when needed.

|  |
| --- |
| from n = n : from (n+1)  > take 10 (from 1) [1,2,3,4,5,6,7,8,9,10] |

### List Comprehensions

|  |
| --- |
| qsort [] = [] qsort (x:xs) = qsort [y | y <- xs, y < x] -- List of elems < x.  ++ [x]  ++ qsort [y | y <- xs, y >= x] -- List of elems >= x. |

### Guards

Note that there is no = after function arguments when using guards:

|  |
| --- |
| signum x | x < 0 = -1  | x == 0 = 0  | otherwise = 1 |

### Infix Functions

Functions can be used in **infix** form using backticks (these are **not** single quotes / apostrophes):

|  |
| --- |
| leapyear :: Int -> Bool leapyear y | y `mod` 400 == 0 = True  | y `mod` 100 == 0 = False  | y `mod` 4 == 0 = True  | otherwise = False |

### Kinds of Names

There are six kinds of names in Haskell:

1. **Variables**: Denoting values
2. **Data-constructors**: Denoting values
3. **Type-variables**: Denoting types
4. **Type-constructors**: Denoting “type builders”
5. **Type-classes**: Denoting groups of similar types
6. **Module-names**: Denoting program modules

**Variables** and **Type-variables** begin with **lowercase** / underscore.

Other names begin with uppercase.

### Errors

|  |
| --- |
| head [] = error "cannot get head of empty list" head (x:\_) = x |

### More Lists

The !! operator lets us select the element at the nth index of a list.

|  |
| --- |
| > [1,2,3] !! 1 2 |

++ concatenates two lists together:

|  |
| --- |
| > [1,2,3] ++ [4,5,6] [1,2,3,4,5,6] |

The time taken for concatenation is proportional the length of the **first list**.

### Function Application

Function application is **left-associative**: f x y z parses as ((f x) y) z.

### Let Expressions

A let expression has the form :

|  |
| --- |
| let { x = y + 3; z = 10; f a = a + 2\*z } in f x |

Using layout:

|  |
| --- |
| let x = y + 3  z = 10  f a = a + 2\*z in f x |

### Where Expressions

A where expression has the form :

|  |
| --- |
| solve a b c = ((droot - b) / twoa), negate ((droot + b) / twoa))  where  twoa = 2\*a  discr = b\*b - 2\*twoa\*c  droot = sqrt discr |

### Case Expressions

A case expression has the form :

|  |
| --- |
| odd x = case (x `mod` 2) of  0 -> False  1 -> True  empty x = case x of  [] -> True  \_ -> False |

### Function Composition

|  |
| --- |
| double x = x + x quadruple = double . double |

### Infix Functions

Haskell permits the definition of infix functions:

|  |
| --- |
| (f ! g) x = f (g x) |

### Lambda Application

Inline functions can be constructed using Lambdas:

|  |
| --- |
| sqr = \n -> n \* n add = \x y -> x + y |

# Defining New Types

**Type synonyms**:

|  |
| --- |
| type Name = String |

Haskell considers Name and String to be exactly the same type.

**Wrapped types**:

|  |
| --- |
| newtype Name = N String |

If s is a value of type String, then N s is a value of type Name.

Haskell considers Name and String to be different types.

**Algebraic Data Types (ADTs)**:

|  |
| --- |
| data Name = Official String String | NickName String |

If f, s and n are values of type String, then Official f s and NickName n are different values of Name.

The types defined can have type parameters:

|  |
| --- |
| type TwoList t = ([t],[t]) newtype BiList t = BiList ([t],[t]) data ListPair t = LPair [t] [t] |

Example:

|  |
| --- |
| data Day = Monday -- Monday through Sunday are Data Constructors  | Tuesday  | Wednesday  | Thursday  | Friday  | Saturday  | Sunday  weekend :: Day -> Boolean weekend Saturday = True weekend Sunday = True weekend \_ = False |

Recursive structures:

|  |
| --- |
| data List = Empty  | Node Int List  > Node 1 (Node 2 (Node 3 Empty)) |

where List is a **type** name, and Empty and Node are **data-constructor** names.

### Type Variables

|  |
| --- |
| data List t = Empty  | Node t (List t) |

### Maybe

|  |
| --- |
| data Maybe t = Nothing | Just t |

Example:

|  |
| --- |
| eval :: EDict -> Expr -> Maybe Float eval d (Mul x y) = case (eval d x, eval d y) of  (Just m, Just n) -> Just (m\*n)  \_ -> Nothing |

### Fold

|  |
| --- |
| foldr u op [] = u foldr u op (x:xs) = x `op` (fold u op xs)  sum = foldr 0 (+) length = foldr 0 (\\_ y -> y+1) |

|  |
| --- |
| foldl u op [] = u foldl u op (x:xs) = fold u op xs |

### Type Constraints

|  |
| --- |
| (==) :: Eq a => a -> a -> Bool |

a must be a member of the Eq type class.

Defining the Eq typeclass:

|  |
| --- |
| class Eq where  (==) :: a -> a -> Bool |

Defining Bool as an instance of Eq:

|  |
| --- |
| instance Eq Bool where  True == True = True  False == False = True  \_ == \_ = False |

The Num type-class:

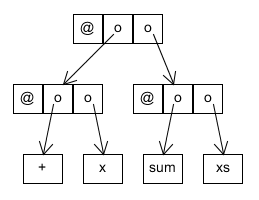
|  |
| --- |
| class Num where  (+), (-), (\*) :: a -> a -> a  negate :: a -> a  abs, signum :: a -> a  fromInteger :: Integer -> a |

Defining Expr as an instance of Num:

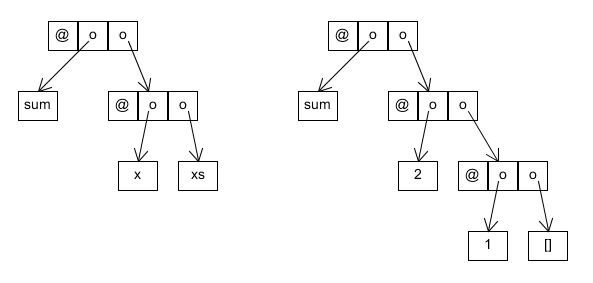
|  |
| --- |
| instance Num Expr where  e1 + e2 = addExpr e1 e2  e1 - e2 = subExpr e1 e2  e1 \* e2 = mulExpr e1 e2  negate e = negExpr e  abs e = absExpr e  signum e = signumExpr e  fromInteger i = integerToExpr i |

# Abstract Syntax Trees

|  |
| --- |
| x + sum xs |

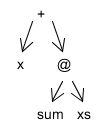


### AST Matching



### AST Shorthand

|  |
| --- |
| x + sum xs |



### Referential Transparency

1. A function whose output depends **only** on its inputs.
2. Expressions built from standard arithmetic operators.

None of the above have any side-effects.

Referential transparency does **not** mean that the language has no side-effects.

### Referentially Opaque

1. A function whose value depends on some **global variable** or state-component.
2. A procedure / function that modifies globale state.
3. The **assignment** statement.
4. A function that performs I/O.

Most of these are examples of side-effects.

### Copying

Destructive update breaks referential transparency.

We need to copy functions to prevent modifying the definition.

# Single-Threadedness

The use of a data value ds is **single-threaded** if:

* There is only ever **one** live reference to it.
* Once a function has been applied to it, the program no longer refers to ds.

Multi-threaded example:

|  |
| --- |
| f ds = (g ds, h ds) |

We have two live references to ds once f is evaluated.

Single-threaded example:

|  |
| --- |
| let ds1 = f ds  ds2 = g ds in ... |

If a function is applied to a single-threaded argument, then destructive update does not destroy referential transparency.

Copying is only necessary if arguments are multi-threaded.

# I/O

|  |
| --- |
| data IOMode = ReadMode | WriteMode | ... type FilePath = String data Handle = ... |

File handles are pointers to open files.

|  |
| --- |
| openFile :: FilePath -> IOMode -> IO Handle hClose :: Handle -> IO () |

|  |
| --- |
| hPutChar :: Handle -> Char -> IO () hGetChar :: Handle -> IO Char |

Functions that do I/O use the IO a datatype:

* Its evaluation produces an I/O side effect.
* It returns a value of type a when evaluated.

I/O actions that don’t return a value have type IO ().

I/O actions are invoked using the do notation.

### Composing I/O Actions

|  |
| --- |
| (>>) :: IO a -> IO b -> IO b -- Sequence (>>=) :: IO a -> (a -> IO b) -> IO b -- Bind return :: a -> IO a  putAB = putChar 'a' >> putChar 'b'  getPut = getChar >>= putChar |

### Do Notation

|  |
| --- |
| getChar >>= (\c -> getChar >>= (\d -> return [c, d])) |

|  |
| --- |
| do c <- getChar  d <- getChar  return [c, d] |

The final action in a do expression cannot bind its return value, since that value becomes that of the entire do expression.

Using do notation:

|  |
| --- |
| fCopyChar :: FilePath -> FilePath -> IO () fCopyChar fromF toF = do ff <- openFile fromF ReadMode  c <- hGetChar ff  hClose ff  tf <- openFile toF WriteMode  hPutChar tf c  hClose tf |

Without do notation:

|  |
| --- |
| fCopyChar fromF toF = openFile fromF ReadMode >>= \ff ->  hGetChar ff >>= \c ->  hClose ff >>  openFile toF WriteMode >>= \tf ->  hPutChar tf c >>  hClose tf |

### Classes Based on Other Classes

|  |
| --- |
| class (Eq a) => Ord a where  (<), (<=), (>=), (>) :: a -> a -> Bool  max, min :: a -> a -> a  compare :: a -> a -> Ordering  compare x y  | x == y = EQ  | x < y = LT  | otherwise = GT |

|  |
| --- |
| instance (Eq a) => Eq [a] where  [] == [] = True  (x:xs) == (y:ys) = x == y && xs == ys  \_ == \_ = False |

### Functors

|  |
| --- |
| class Functor f where  fmap :: (a -> b) -> f a -> f b |

f is a type-constructor.

|  |
| --- |
| instance Functor Maybe where  fmap f Nothing = Nothing  fmap f (Just x) = Just (f x)  instance Functor [] where  fmap = map |

# Monads

|  |
| --- |
| 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 |

### Using the Maybe Monad

|  |
| --- |
| f dict = case (lookup "foo" dict) of  Nothing -> Nothing  Just x -> case (lookup "bar" dict) of  Nothing -> Nothing  Just y -> Just (x, y) |

|  |
| --- |
| f dict = do x <- lookup "foo" dict  y <- lookup "bar" dict  return (x, y) |

|  |
| --- |
| instance Monad Maybe where  return x = Just x  Nothing >>= f = Nothing  (Just x) >>= f = f x  fail s = Nothing |

Lists are monads:

|  |
| --- |
| instance Monad [] where  return a = [a]  list >>= f = concat (map f list)  fail \_ = [] |

List comprehensions are monads:

|  |
| --- |
| myMap f xs = do { x <- xs ; return (f x) } myMap' f xs = [ f x | x <- xs ] |

|  |
| --- |
| root x = if x < 0 then fail ("negative" ++ show x)  else return (sqrt x)  sumRoots x y = do rx <- root x  ry <- root y  return (rx + ry) |

### Either as a Monad

We can use Either to show errors:

|  |
| --- |
| data Either a b = Left a | Right b  instance Monad (Either a) where  return x = Right x  (Left errStr) >>= \_ = Left errStr  (Right x) >>= f = f x |

### foldl vs. foldr

|  |
| --- |
| foldr f z [] = z foldr f z (x:xs) = f x (foldr f z xs)  foldl f z [] = z foldl f z (x:xs) = foldl f (f z x) xs |

### Lambda Abstraction

|  |
| --- |
| > (\x y -> x+y) 1 2 3 |

# Inductive Proofs

Prove:

**1) Base Case**

Prove :Expand P:

Definitions of ++ and length:

Arithmetic:

Reflexivity of =:

**2) Inductive Step**

Assume : Expand P:

Show : Expand P:

Definitions of ++ and length:

Definitions of ++ and + are associative:

Arithmetic:

By induction hypothesis: