Lecture 1 - Introduction

imperative programming: computations through updating values
functional programming: computations through applying functions

Haskell:
- -> statically typed - type check at compile time
- -> lazy - inputs evaluated when needed
- -> pure - no side effects

Agda:
- -> dependently typed - types depend on program expressions
- -> total - defined functions that terminate for all inputs
- -> proof assistant - state and prove properties of programs

Haskell Syntax

Naming rules:
- -> capitalization - constructors (True, False), concrete types (Bool, Int)
- -> small letter - functions and variables (not, length), type variables (a, t)
- -> reserved keywords (if, let, data, type, module)

Conditional expressions (if ... then ... else ...): else - required, if/then/else - nesting
Let bindings (let ... in ...)

Haskell function
- (optinal) type signature: f :: type
- definition: f x = ..., x - input
- helper functions: where ... blocks

where vs. let
- let - define variables before use, anywhere in expression
- where - define variables after use, always attached to a clause

Layout-sensitive
- Same level of identation = same block
- rule: Within a block, all definitions must start at the exact same column
- block keywords: where, let, case, do

Haskell Types

type - name for a collection of variables
Bool has two values True and False => True :: Bool
type annotations - optional, type inference of compiler

Bool: True and False
Int: 0, 42, -9000, ... (64 bit integer)
Integer: 9223372036854775805, ... (arbitrary-size integer)
Float: 1.23, Infinity, NaN, ...
Double: 1.0e40, -1.0e300, ...
Char: 'a', 'Z', '/', ...
String: "Hello, world!", "line 1\n line 2", ...

types: a -> b, applied to argument of type a to get a result of type b
side effects:
- -> modifying global variable
- -> throwing an exception
- -> reading or writing a file
- -> sending a message over the network

pure functions:
- -> no side effects
- -> return a value or loop forever

Tuple (x,y) of type (a, b) => x :: a and y :: b
curried function - function that returns another function
function arrow associates to the right: Int -> Bool -> Char = Int -> (Bool -> Char)
function application associates to the left: f x y = (f x) y

list type [a]
[Char] = String

list vs. tuple
  list - infinitely many elements all the same type
  tuple - fixed size elements, can be different types
ranges of values: from i to j = [i..j]

List Compressions

[function | generator (usually range)]
multiple generators:
  -> later generators can dependent on previous ones
  -> nested in order (first iterate through innermost one)
filtering (in generator) guard -> boolean predicate, even/odd x
no built-in controls (for, while) -> lists for iterative algorithms

Lecture 2 - Functions

Pattern Matching

not :: Bool -> Bool | not True = False | not False = True
pattern variable - matches any value that didn't match previous
wildcard (_) - pattern value for which the value doesn't matter
xor :: Bool -> Bool | xor T F = T | xor F T = T | xor _ _ = F
order of clauses -> Haskell uses first clause that matches

infix operations (+, -, ==, !!): 1 + 1 ≡ (+) 1 1
lists: [] (empty) or x:xs, [1,2,3] syntactic sugar for 1:2:3:[]
guards | b :: Bool, | otherwise is always True

Polymorphic Types

type annotations: x :: a
basic types: Bool, Int, Integer, Float, Double, Char, String, ...
list type: [a] *String = [Char]
tuple types: (a, b), (a, b, c), ..
function types: a -> b, a -> b -> c, ...

length :: [Int] -> Int } polymorphic length :: [a] -> Int
length :: [Bool] -> Int | inferred in compile time

fst :: (a, b) -> a, snd :: (a, b) -> b, swap :: (a, b) -> (b, a)

(:) :: a -> [a] -> [a] (append), head :: [a] -> a, tail :: [a] -> [a], (++) :: [a] -> [a] -> [a] (concat)

(!!) :: [a] -> Int -> a (indexing), take :: Int -> [a] -> [a], drop :: Int -> [a] -> [a]

zip :: [a] -> [b] -> [(a, b)], unzip :: [(a, b)] -> ([a], [b])

Type Classes

double x = x+x or 2*x

double :: Int -> Int (works for Float) } double :: Num a => a -> a
double :: a -> a (doesn't work for Bool) } ↳ type class constraint

type class - a collection of types that support a common interface

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  fromInteger :: Integer -> a

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

Ord is subclass of Eq.
Any instance of Ord must be instance of Eq.

class Eq a => Ord a where
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a

Lecture 3 - Data Types

Type Aliases

type alias - new name to an existing type, can't be recursive

type String = [Char] type Coordinate = (Int, Int)

Algebraic Datatypes (ADTs)

data Bool = True | False
data Ordering = LT | EQ | GT
data Shape = Circle Double | Rect Double Double
square :: Double -> Shape
square x = Rect x x

```
area :: Shape -> Double
area (Circle r) = pi * r * r
area (Rect l h) = l * h
```

constructor - function into datatype
:t Circle => Circle :: Double -> Shape
:t Rect => Rect :: Double -> Double -> Shape

newtype - data declaration with one constructor taking one argument

```
newtype EuroPrice = EuroCents Integer
newtype DollarPrice = DollarCents Integer
dollarToEuro :: DollarPrice -> EuroPrice
dollarToEuro (DollarCents x) = EuroCents (round (0.97 * fromInteger x))
```

Record Syntax

```
data Shape
  = Circle {radius :: Double}
  | Rect {width :: Double, height :: Double}
radius :: Shape -> Double, width :: Shape -> Double, height :: Shape -> Double
square :: Double -> Shape
square x = Rect {width = x}
getWidth :: Shape -> Double
getWidth (Circle {radius = r}) = 2 * r
getWidth (Rect {width = w}) = w
```

Expression Problem	Object-Oriented	Functional
new cases to type	easy	hard
new functions to type	hard	easy

language

```
Maybe a = Nothing | Just a   } optional value of type a, used in functions
                              that can fail (division by 0 -> Nothing)
data NonEmpty a = a :| [a] - list with at least one element
data Either a b = Left a  - disjoint union of a and b
                | Right b   each element is either Left x, x::a, or Right y, y::b
```

algebraic datatype - type formed from other types using sums and products
- product of a and b - tuple type (a, b)
- sum of a and b - disjoint union type Either a b

constructor - product of the types of its arguments
ADT - sum of the constructor types

Recursive Datatypes

Nat - natural numbers, Zero, Suc Zero, Suc (Suc Zero), ...
data List a = Nil | Cons a (List a)
data Tree a = Leaf a | Node (Tree a) (Tree a)

QuickCheck

prop_name inputs = conditions ==> test
conditional property - invalid test cases are discarded
chooseInt (lower, upper) - generator, generates random values of type Int

Lecture 4 - Higher-Order Functions

Higher-Order Functions

higher-order function - function that either takes a function as an argument or returns a function as a result
(curried function)

```
:t curry => curry :: ((a,b) -> c) -> (a -> b -> c)
:t uncurry => uncurry :: (a -> b -> c) -> ((a,b) -> c)
:t flip => flip :: (a -> b -> c) -> (b -> a -> c)
:t ($) => ($) :: (a -> b) -> a -> b
```

Higher-Order Functions on Lists

```
:t map => map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
:t filter => filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

lambda expression \x -> ... - single-use function
operator section - partially applied operator
(+1) = \x -> x + 1

```
:t all => all:: Foldable t => (a -> Bool) -> t a -> Bool
:t +d all => all:: (a -> Bool) -> [a] -> Bool
:t +d any => any:: (a -> Bool) -> [a] -> Bool
:t takeWhile => takeWhile:: (a -> Bool) -> [a] -> [a]
:t dropWhile => dropWhile:: (a -> Bool) -> [a] -> [a]
```

Testing Properties of Functions

higher-order property - property that takes a function as an input
Fun a b - shrinkable and printable functions

Identity & Function Composition

```
(.):: (b->c) -> (a->b) -> (a->c)
f . g = \x -> f (g x),   odd = not.even   twice f = f.f
```

(.) - compose functions without naming their arguments

```
id:: a -> a   id x = x
```

Three Laws of Function Composition:

1) id . f = f 2) f . id = f 3) f . (g . h) = (f . g) . h

Higher-Order Function foldr

foldr (#) v - replace each occurrence of (:) by (#) and the final [] by v

```
foldr :: (a->b->b) -> b -> [a] -> b
foldt :: (a->b) -> (b->b->b) -> Tree a -> b
```

foldt v f - replace each Leaf by v and each Node by f
foldl - version of foldr that associates to the left

```
foldl:: (b->a->b) -> b -> [a] -> b
```

Lecture 5 - Type Classes

Recap Type Classes

type class - family of types that share common interface
Show - show; Eq - (==), (/=); Ord - (<), (<=), ...; Num - (+), (-), (*), ...

```
class Eq a where                    -> declare type class Eq with 1 parameter a
  (==) :: a -> a -> Bool            } function declarations
  (/=) :: a -> a -> Bool            }
  x == y = not (x /= y)             } default function implementation
  x /= y = not (x == y)             }

data TrafficLight = Red | Yellow | Green
instance Eq TrafficLight where      -> declare TrafficLight instance of Eq
  Red == Red = True                 }
  Yellow == Yellow = True           } custom function implementation (for (==)),
  Green == Green = True             } other (meaning (/=)) use default implementation
  _ == _ = False                    }
```

Instance of Eq X has to define either (==) :: X -> X -> Bool or (/=) :: X -> X -> Bool.
= Minimal complete definition: (==) | (/=)

Subclasses

subclass - each instance must also be instance of the base class (inherits default implementations from base class)
Ord - subclass of Eq Ord a => ... instead of (Eq a, Ord a) => ...
(Num a, Ord a) => Real a, (Real a, Enum a) => Integral a
(Num a) => Fractional a, (Fractional a) => Floating a

Creating Classes

```
class Reversible a where
  rev :: a -> a

instance Reversible [a] where
  rev xs = reverse xs

instance Reversible (Tree a) where
  rev (Leaf x) = Leaf x
  rev (Node l r) = Node (rev r) (rev l)

instance Booly Bool where
  bool x = x

instance Booly Int where
  bool x = x /= 0

instance Booly Double where
  bool x = x /= 0.0

instance Booly (Maybe a) where
  bool Nothing = False
  bool Just(x) = True
```

```
instance Booly [a] where
  bool [] = False
  bool (_:_) = True

iffy :: Booly a => a -> b -> b -> b
iffy b x y = if bool b then x else y
```

Functors

map - apply function to every element in a list
Functor - generalized map for other data structures
class Functor f where ← container storing elements of type a
  fmap :: (a -> b) -> f a -> f b

```
instance Functor [] where
  fmap f xs = map f xs
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f Just (x) = Just (f x)
instance Functor ((->) a)
  fmap :: (b->c) -> (a->b) -> (a->c)
  fmap f g -> f . g
```

```
instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node l r) =
    Node (fmap f l) (fmap f r)
instance Functor (Either a)
  fmap :: (b->c) -> Either a b -> Either a c
  fmap f (Left x) = Left x
  fmap f (Right y) = Right (f y)
```

Applicative Functors

Applicative - subclass of Functor that adds pure and (<*>)

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

pure :: a -> f a - container with ≥1 copies of a value of type a
Maybe: pure x = Just x list: pure x = [x] Tree: pure x = Leaf x
Either a: pure x = Right x (->) a: pure x = const x
(<*>) :: f (a->b) -> f a -> f b - combines two containers by applying functions in the first to values in the second one

```
Just f <*> Just x = Just (f x)   Nothing <*> _ = _ <*> Nothing = Nothing
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
zipA :: Applicative f => f a -> f b -> f (a,b)
zipA xs ys = pure (,) <*> xs <*> ys
```

Lecture 6 - $IO & Monads

$IO Type

$IO a - type of programs that interact with the world and return of type a
$IO - no definition, built into Haskell
expression of type $IO a - action
impure/effectful programs - use $IO
putChar :: Char -> $IO (), putStr :: String -> $IO (), putStrLn :: String -> $IO ()
print :: Show a => a -> $IO (), getChar :: $IO Char, getLine :: $IO String, readLn :: Read a => $IO a
data $IOMode = ReadMode | WriteMode | ReadWriteMode | AppendMode
openFile :: FilePath -> $IOMode -> $IO Handle, hPutStrLn :: Handle -> String -> $IO ()
hGetLine :: Handle -> $IO String, hIsEOF :: Handle -> $IO Bool, hClose :: Handle -> $IO ()
do notation - write sequence of $IO actions [main = do]
vi <- ai - statement with an action ai :: $IO bi, final line must be action $IO a
return :: a -> $IO a - turns pure value into $IO action
when :: Bool -> $IO () -> $IO () - executes action when condition is true
sequence :: [$IO a] -> $IO [a] - runs list of $IO actions and collects results
$IO - functor => fmap f act - applies pure function f to the result of $IO a action
instance Applicative $IO where act, producing new $IO b action

```
pure x = return x
mf <*> mx = do f <- mf x <- mx return (f x)
```

System.$IO.unsafe, unsafePerform$IO :: $IO a -> a - $IO action inside pure function
trace :: String -> a -> a

Monad Class

(>>=) - bind, >>= :: m a -> (a -> m b) -> m b, m - monad
class Applicative m => Monad m where

```
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

monadic type m a:
-> a container data structure that holds values of type a
-> a computation that can perform side effects before returning type a

monad - type constructor that is an instance of the Monad type class
monadic type - type of form m a where m is a monad
action - expression of monadic type
monadic function - function that returns an action
(>>) - sequencing operator, executes one action after another, ignoring output of the first
when :: Applicative f => Bool -> f () -> f ()
unless :: Applicative f => Bool -> f () -> f ()
sequence :: Monad m => [m a] -> m [a]
traverse :: Applicative f => (a -> f b) -> [a] -> f [b]

Lecture 7 - More Monads

State Monad

```
newtype State s a = State (s -> (s,a))
get :: State s s, get = State (\st -> (st, st))
put :: s -> State s (), put st = State (\_ -> ((), st))
```

```
randomNumber :: StdGen -> ($int, StdGen)
randomNumber = random
```

```
randomSInt :: State StdGen $int
randomSInt = State random
```

```
instance Functor (State s) where
  fmap f (State h) =
    State (\oldSt ->
      let (x, newSt) = h oldSt
      in (f x, newSt))
```

```
instance Applicative (State s) where
  pure x = State (\st -> (x, st))
  State g <*> State h =
    State (\oldSt ->
      let (f, newSt1) = g oldSt
          (x, newSt2) = h newSt1
      in (f x, newSt2))
```

```
instance Monad (State s) where
  return x = pure x
  State h >>= f =
    State (\oldSt ->
      let (x, newSt) = h oldSt
      in runState (f x) newSt)
```

```
runState :: State s a -> s -> (a, s)
runState (State h) = h
```

reader monad - gives access to extra input of some type r
newtype Reader r a = Reader (r -> a)
writer monad - allows writing some output of type w
newtype Writer w a = Writer (w, a)

Monadic Parsing

```
type Parser a = String -> a
item :: Parser Char
item (x:[]) = x
item _ = error "Parse failed!"
```

```
type Parser a = String -> [(a, String)]
item :: Parser Char
item (x:xs) = [(x, xs)]
item [] = []
```

A parser of things is a function from strings to lists of pairs of things and strings!

```
newtype Parser a = Parser (String -> [(a, String)])
parse :: Parser a -> String -> [(a, String)]
parse (Parser f) = f
instance Functor Parser where ...
instance Applicative Parser where ...
instance Monad Parser where ...
empty :: Parser a, empty = Parser []
nat :: Parser Int
nat = do xs <- some digit
         return (read xs)
```

```
(<|>) :: Parser a -> Parser a -> Parser a
(Parser f <|> Parser g) = Parser
  (\inp -> case f inp of
     [] -> g inp
     result -> result)
some many :: Parser a -> Parser a
some x = pure (:) <*> x <*> many x
many x = some x <|> pure []
```

Monad Laws

Laws of Eq

- reflexivity: x == x = True
- symmetry: (x == y) = (y == x)
- transitivity: if (x == y && y == z) = True then x == z = True
- substitutivity: if x == y = True then f x == f y = True
  where f :: a -> b and a and b are both instances of Eq
- negation: x /= y = not (x == y)

The Functor Laws

fmap f applies f to each value stored in the container, but should leave the structure of the container unchanged.

- fmap id = id
- fmap (g . h) = fmap g . fmap h

The 4 Laws of Applicative

- pure id <*> x = x
- pure (f x) = pure f <*> pure x
- mf <*> pure y = pure (\g -> g y) <*> mf
- x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z

The Monad Laws

- Left identity: return x >>= f = f x (remove return in middle of do)
- Right identity: mx >>= (\x -> return x) = mx (remove return at end of do)
- Associativity Law: (mx >>= f) >>= g ≡ mx >>= (\x -> (f x >>= g))

Semigroup & Monoid

semigroup - set A with associative operator ◇, (x◇y)◇z = x◇(y◇z)
monoid - semigroup with a neutral element ∅, ∅◇x = x = x◇∅

```
class Semigroup a where
  (<>) :: a -> a -> a
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mappend = (<>)
  mconcat = foldr mappend mempty
```

Semigroup Law: (x <> y) <> z = x <> (y <> z)

Monoid Laws:

- mempty `mappend` x = x
- x `mappend` mempty = x
- (x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)

```
instance Semigroup [a] where (<>) = (++)
instance Monoid [a] where mempty = []
instance Semigroup Int where (<>) = (+) / (*)
```

wrapper types

```
newtype Sum a = Sum { getSum :: a }
newtype Product a = Product { getProduct :: a }
newtype Any = Any { getAny :: Bool }
newtype All = All { getAll :: Bool }
```

Lecture 8 - Laziness

Lazy Evaluation

evaluation strategy - general way to pick which subexpression to evaluate next
-> call-by-value: evaluate arguments before unfolding the definition of a function
-> call-by-name: unfold function definition without evaluating arguments
lambda expressions - black body: body never evaluated before applied
-> innermost reduction - call-by-value with evaluation under lambdas
-> outermost reduction - call-by-name with evaluation under lambdas

lazy evaluation (call-by-need) - variant of call-by-name that avoids double eval
argument -> thunk, evaluated once and stored
forcing strict evaluation - seq :: a -> b -> b
seq u v - evaluate u before returning v

strict application - ($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x - evaluate x before applying f

Infinite Data Structures

ones :: [Int], ones = 1 : ones
infinite data structures - expression that would contain an infinite number of constructors if it is fully evaluated
infinite list - stream of data that produces as much elements as required by its context
[m ...] - list of all integers starting from m

```
sieve (x:xs) =
  let xs' = [y | y <- xs, y `mod` x /= 0]
  in x : sieve xs'

primes :: [Int]
primes = sieve [2 ...]
```

```
repeat :: a -> [a]
repeat x = xs
  where xs = x : xs

cycle :: [a] -> [a]
cycle xs = xs'
  where xs' = xs ++ xs'

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Lecture 9 - Introduction to Agda

Formal verification - techniques for proving correctness of programs
Formal specification - ideas from formal logic and mathematics ensuring trustworthiness
→ model checking - explores all possible executions
→ deductive verification - uses tools to analyse and prove correctness
→ lightweight formal methods - verify properties such as type or memory safety
→ dependent types - form of deductive verification embedded in programming language

Agda

→ purely functional programming language
→ full support for dependent types
→ interactive programming with type checker

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

defines datatype Greeting with 1 constructor hello

defines function greet of type Greeting that returns hello

Syntax

Typing - single colon, `b : Bool` instead of `b :: Bool`
Naming - any symbols, capital or small letters, unicode characters
Whitespacing - `1+1` is valid function name, instead `1 + 1`
Infix operators - underscores, `_+_` instead of `(+)`

```
data Bool : Set where
  true : Bool
  false : Bool
```

```
not : Bool → Bool
not true = false
not false = true
```

```
_||_ : Bool → Bool → Bool
false || false = false
_ || _ = true
```

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
```

```
one = suc zero
two = suc one
three = suc two
```

```
_+_ : Nat → Nat → Nat
zero + y = y
(suc x) + y = suc (x + y)
```

```
isEven : Nat -> Bool
isEven zero = true
isEven suc(zero) = false
isEven suc(suc x) = isEven x
```

```
infixL 10 _+_
infixL 20 _*_
1 + 2 * 3 + 4
↳ (1 + (2 * 3)) + 4
```

```
data Int : Set where
  pos : Nat -> Int
  zero : Int
  neg : Nat -> Int
```

Types

```
myNat : Set
myNat = Nat
```

```
myFour : MyNat
myFour = suc(suc(suc(suc zero)))
```

```
id : (A : Set) -> A -> A
id A x = x
id Nat zero : Nat
id Bool true : Bool
```

```
id : {A : Set} -> A -> A
id x = x
```
hidden argument

```
id zero : Nat
id true : Bool
```

```
if_then_else : {A : Set} ->
  Bool -> A -> A -> A
if true then x else y = x
if false then x else y = y
```

polymorphic datatypes - parameter (A : Set)

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A -> List A -> List A
  infixL 5 _::_
```

```
data _×_ (A B : Set) : Set where
  _,_ : A -> B -> A × B
fst : {A B : Set} -> A × B -> A
fst (x, y) = x
snd : {A B : Set} -> A × B -> B
snd (x, y) = y
```

tuple (a, b)
product type A × B

Totality

Agda - total language
- → no runtime errors
- → no incomplete pattern matches
- → no non-terminating functions

coverage check - ensure complete pattern matching definitions

termination check - ensure termination of (structurally) recursive definitions

Lecture 10 - Dependent Types

Dependent Types

```
data Food : Set where
  pizza : Food, cake : Food, bread : Food
```

```
amountOfCheese : Food -> Nat
λamountOfCheese on cake
```

```
data Flavour : Set where
  cheesy : Flavour
  chocolatey : Flavour

data Food : Flavour → Set where
  pizza : Food cheesy
  cake : Food chocolatey
  bread : {f : Flavour} → Food f

amountOfCheese : Food cheesy → Nat
amountOfCheese pizza = 100
amountOfCheese bread = 20
```

dependent types - family of types, dependent on a base type

dependent function type - type of form $(x : A) \to B$ where the type of the output depends on the value of the input

```
data Ingredient : Flavour → Set where
  cheese, chocolate, flour, water, ... : Ingredient

ingredients : {f : Flavour} →
  Food f → List (Ingredient f)
```

Vec Type

Vec A n - type of vectors with exactly n arguments of type A

```
myVec1 : Vec Nat 4
myVec1 = 1 :: 2 :: 3 :: 4 :: []

myVec2 : Vec Nat 0
myVec2 = []

myVec3 : Vec (Bool → Bool) 2
myVec3 = not :: id :: []
```

```
zeroes : (n : Nat) → Vec Nat n
zeroes zero = []
zeroes (suc n) = 0 :: zeroes n
```

Vec A n - dependent type indexed over base type Nat

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A 0
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

argument (A : Set) - parameter, the same in type of each constructor

argument of type Nat - index, determined individually for each constructor

```
mapVec : {A B : Set} {n : Nat} → (A → B) → Vec A n → Vec B n
mapVec f [] = []
mapVec f (x :: xs) = f x :: mapVec f xs
```

```
head : {A : Set}{n : Nat} → Vec A (suc n) → A
head (x :: xs) = x
head [] - impossible

tail : {A : Set}{n : Nat} → Vec A (suc n) → Vec A n
tail (x :: xs) = xs
```

Fin Type

Fin n - natural numbers strictly smaller than n

```
data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)
```

Fin 0 - empty type

```
lookupVec : {A : Set}{n : Nat} → Vec A n → Fin n → A
lookupVec (x :: xs) zero = x
lookupVec (x :: xs) (suc i) = lookupVec xs i
```

Lecture 11 - Curry-Howard Correspondence

The Curry-Howard Correspondence

proof - sequence of statements, each direct consequence of previous ones

1) $A \to B$ 2) $A \wedge C \to B \wedge C$

3) A (from 2) 4) B (modus ponens 1, 3) 5) C (from 2) 6) $B \wedge C$ (from 4, 5)

proof tree:

annotate

$$\frac{p : A \to B^{1} \quad \frac{q : A \wedge C^{2}}{\text{fst } q : A^{3}}}{p\ (\text{fst } q) : B^{4}} \qquad \frac{q : A \wedge C^{2}}{\text{snd } q : C^{5}}}{(p\ (\text{fst } q),\ \text{snd } q) : B \wedge C^{6}}$$

proof term

Logical propositions ($A \wedge B$, $\neg A$, $A \to B$, ...) as types of all possible proofs

false proposition - empty type

conjunction $A \wedge B$ - type of pairs $A \times B$

```
fst : {A B : Set} → A × B → A    fst (x , y) = x
snd : {A B : Set} → A × B → B    snd (x , y) = y
```

$\equiv (A \wedge B) \to A$

$\equiv (A \wedge B) \to B$

implication $A \to B$ - function type $A \to B$

Modus ponens (Proof by implication): 1) P → Q, 2) P → 3) Q

```
modusPonens : {P Q : Set} → (P → Q) × P → Q
modusPonens (f, x) = f x
```

disjunction A ∨ B - sum type Either A B

Proof by cases: 1) P ∨ Q 2) P → R 3) Q → R → 4) R

```
cases : {P Q R : Set} → (P → R) × (Q → R) → R
cases (left x) (f, g) = f x
cases (right y) (f, g) = g y
```

```
data ⊤ : Set where {truth
  tt : ⊤
data ⊥ : Set where {falsity
  ⊥ - empty type
```

principle of explosion - assume false statement, prove any proposition P

```
absurd : {P : Set} → ⊥ → P,   absurd ()
```

Propositional logic		Type system
proposition	P	type
proof of a proposition	p : P	program of a type
conjunction	P × Q	pair type
disjunction	Either P Q	either type
implication	P → Q	function type
truth	⊤	unit type
falsity	⊥	empty type

negation ¬P : type P → ⊥

equivalence P ⇔ Q : type (P → Q) × (Q → P)

1. Propositions are types
2. Proofs are programs
3. Simplifying a proof is evaluating a program

```
A → A ≡ λ x → (λ y → fst y) (x, x)
      → λ x → fst (x, x)
      → λ x → x
```

$P \vee (\neg P)$ - excluded middle
$\neg\neg P \to P$ - double negation elimination
} non-constructive statements unprovable in Agda

Predicate Logic

classical logic – continuations (Lisp)
linear logic - linear types (Rust)
predicate logic - dependent types (Agda)

```
data IsEven : Nat → Set where
  e-zero : IsEven zero
  e-sucd : {n : Nat} → IsEven n → IsEven (suc (suc n))

data P : A → Set where
  c₁ : ... → P a₁
  c₂ : ... → P a₂
```

```
data IsTrue : Bool → Set where
  is-true : IsTrue true

_=Nat_ : Nat → Nat → Bool
zero =Nat zero = true
(suc x) =Nat (suc y) = x =Nat y
_ =Nat _ = false
```

Universal Quantification

$\forall (x \in A),\ P(x) \equiv (x : A) \to P\,x$

```
double : Nat → Nat
double zero = zero
double (suc m) = suc (suc (double m))
```

```
∀n - 2*n is even
double-even : (n : Nat) → isEven (double n)
double-even zero = e-zero
double-even (suc m) = e-sucd (double-even m)
```

proof by induction on natural numbers:

```
proof : (n : Nat) → P n
proof zero = ...   (base case)
proof (suc n) = ...   (inductive case)
induction hypothesis   proof n : P n
```

pattern matching -> proof by cases
recursive function -> proof by induction

```
proof : {A : Set} {xs : List A} → P xs
proof [] = ...   (base case)
proof (x :: xs) = ...   (inductive case)
proof xs = P xs   (induction hypothesis)
```

Existential Quantification

$\exists (x \in A),\ P(x) \equiv \Sigma$

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

```
fst Σ : {A : Set} {B : A → Set} → Σ A B → A
fst Σ (x , y) = x
snd Σ : {A : Set} {B : A → Set} → (z : Σ A B) → B (fst Σ z)
snd Σ (x , y) = y
```

prove: $\exists n$ s.t. $n+n=12$

```
half-a-dozen : Σ Nat (λ n → IsTrue ((n + n) == Nat 12))
half-a-dozen = 6 , is-true _ of type IsTrue ((6 + 6) == Nat 12)
```

Lecture 12 - Equational Reasoning

Identity Type

```
n-equals-n : (n : Nat) → isTrue (n == Nat n)
n-equals-n zero = is-true
n-equals-n (suc n) = n-equals-n n
```

```
data _≡_ {A : Set} → A → A → Set where
  refl : {x : A} → x ≡ x
```

```
one-plus-one : 1 + 1 ≡ 2
one-plus-one = refl
zero-not-one : 0 ≡ 1 → ⊥
zero-not-one ()
```

```
test₁ : Length (42 :: []) ≡ 1
test₁ = refl
test₂ : Length (map (1 +_) (0 :: 1 :: 2 :: [])) ≡ 3
test₂ = refl
```

```
not-not : (b : Bool) → not (not b) ≡ b
not-not true = refl
not-not false = refl
```

```
castVec : {A : Set} {m n : Nat} → m ≡ n → Vec A m → Vec A n
castVec refl xs ≡ xs
```

symmetry: $x=y \to y=x$

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

congruence, $f : A \to B$, $x=y \Rightarrow f\ x = f\ y$

```
cong : {A B : Set} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

Equational Reasoning

$(a+b)(a+b) = a(a+b) + b(a+b) = a^2 + ab + ba + b^2 = a^2 + 2ab + b^2$

head (replicate 100 "spam") = head ("spam" :: replicate 99 "spam") = "spam"

```
[] : {A : Set} → A → List A
[x] = x :: []
```

```
reverse : {A : Set} → List A → List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ [x]
```

prove reverse [x] = [x]

```
reverse-singleton : {A : Set} {x : A} → reverse [x] ≡ [x]
reverse-singleton x = refl
```

```
reverse-singleton x =
  begin
    reverse [x]
  =⟨⟩ - definition of [_]
    reverse (x :: [])
  =⟨⟩ - applying reverse (second clause)
    reverse [] ++ [x]
  =⟨⟩ - applying reverse (first clause)
    [] ++ [x]
  =⟨⟩ - applying _++_
    [x]
  end
```

Proof by Case Analysis & Induction

Case Analysis (pattern matching)

```
not-not : (b : Bool) → not (not b) ≡ b
not-not false =
  begin
    not (not false)
  =⟨⟩ - applying inner not
    not true
  =⟨⟩ - applying not
    false
  end
not-not true = ...
```

Induction (recursive function)

```
add-n-zero : (n : Nat) → n + zero ≡ n
add-n-zero zero = ...
add-n-zero (suc n) =
  begin
    (suc n) + zero
  =⟨⟩ - applying +
    suc (n + zero)
  =⟨cong suc (add-n-zero n)⟩ - IH
    suc n
  end
```

Proving Type Class Laws

Functor Laws {
- fmap id = id
- fmap (f ∘ g) = fmap f ∘ fmap g

```
map-id : {A : Set ℓ} (xs : List A)
       → map id xs ≡ xs
map-id [] =
  begin
    map id []
  =⟨⟩ - applying map
    []
  end
```

base case

```
map-id (x :: xs) =
  begin
    map id (x :: xs)
  =⟨⟩ - applying map
    id x :: map id xs
  =⟨⟩ - applying id
    x :: map id xs
  =⟨ cong (x ::_) (map-id xs) ⟩ - IH
    x :: xs
  end
```

```
_∘_ : {A B C : Set ℓ} → (B → C) → (A → B) → (A → C)
f ∘ g = λ x → f (g x)
prove map (f ∘ g) x = (map f ∘ map g) x
```

Verifying Optimizations

reverse $O(n^2)$

```
reverse : {A : Set ℓ} → List A → List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ [x]
```

reverse in $O(n)$

```
reverse-acc : {A : Set ℓ} → List A → List A → List A
reverse-acc [] ys = ys
reverse-acc (x :: xs) ys = reverse xs (x :: ys)
reverse' : {A : Set ℓ} → List A → List A
reverse' xs = reverse-acc xs []
```

```
reverse'-reverse : {A : Set ℓ} → (xs : List A) → reverse' xs ≡ reverse xs
reverse'-reverse xs =
  begin
    reverse' xs
  =⟨⟩ - def of reverse'
    reverse-acc xs []
  =⟨ reverse-acc-lemma xs [] ⟩
    reverse xs ++ []
  =⟨ append-[] (reverse xs) ⟩
    reverse xs
  end
```

```
reverse-acc-lemma : {A : Set ℓ} → (xs ys : List A) → reverse-acc xs ys ≡ reverse xs ++ ys
```

Base case

```
reverse-acc-lemma [] ys =
  begin
    reverse-acc [] ys
  =⟨⟩ - definition of reverse-acc
    ys
  =⟨⟩ - unapplying ++
    [] ++ ys
  =⟨⟩ - unapplying reverse
    reverse [] ++ ys
  end
```

Inductive case

```
reverse-acc-lemma (x :: xs) ys =
  begin
    reverse-acc (x :: xs) ys
  =⟨⟩ - def of reverse-acc
    reverse-acc xs (x :: ys)
  =⟨ reverse-acc-lemma xs (x :: ys) ⟩ - IH
    reverse xs ++ (x :: ys)
  =⟨⟩ - unapplying ++
    reverse xs ++ ([x] ++ ys)
  =⟨ sym (append-assoc (reverse xs) [x] ys) ⟩
    (reverse xs ++ [x]) ++ ys
  =⟨⟩ - unapplying reverse
    reverse (x :: xs) ++ ys
  end
```

associativity of ++

Compiler Correctness

```
data Expr : Set where
  valE : Nat → Expr
  addE : Expr → Expr → Expr

eval : Expr → Nat
eval (valE x) = x
eval (addE e₁ e₂) = eval e₁ + eval e₂

expr : Expr    expr = addE (addE (valE 2) (valE 3)) (valE 4) ≡ (2 + 3) + 4
```

```
data Op : Set where
  PUSH : Nat → Op
  ADD : Op

Stack = List Nat
Code = List Op

code : Code
code = PUSH 2 :: PUSH 3 :: ADD
     :: PUSH 4 :: ADD :: []
```

```
exec : Code → Stack → Stack
exec [] s = s
exec (PUSH x :: c) s = exec c (x :: s)
exec (ADD :: c) (m :: n :: s) = exec c (n + m :: s)
exec (ADD :: c) _ = []
```

```
comp : Expr → Code
comp (valE x) = [PUSH x]
comp (addE e₁ e₂) =
  comp e₁ ++ comp e₂ ++ [ADD]
```

$O(n^2)$ → repeated _++_

```
comp' : Expr -> Code -> Code
comp' (valE x) c = PUSH x :: c
comp' (addE e1 e2) c = comp' e1 (comp' e2 (ADD :: c))

comp : Expr -> Code
comp e = comp' e []
```

```
comp-exec-eval : (e : Expr) -> exec (comp e) [] ≡ [eval e]
comp-exec-eval e =
  begin
    exec (comp e) []
  =< comp'-exec-eval e [] [] >
    exec [] (eval e :: [])
  =<> - applying exec for []
    eval e :: []
  =<> - unapplying [_]
    [eval e]
  end
```

```
comp'-exec-eval : (e : Expr) (s : Stack) (c : Code) -> exec (comp' e c) s ≡ exec c (eval e :: s)
comp'-exec-eval (valE x) s c =
  begin
    exec (comp' (valE x) c) s
  =<> - applying comp'
    exec (PUSH x :: c) s
  =<> - applying exec for PUSH
    exec c (x :: s)
  =<> - unapplying eval for valE
    exec c (eval (valE x) :: s)
  end
```

```
comp'-exec-eval (addE e1 e2) s c =
  begin
    exec (comp' (addE e1 e2) c) s
  =<> - def of comp'
    exec (comp' e1 (comp' e2 (ADD :: c))) s
  =< comp'-exec-eval e1 s (comp' e2 (ADD :: c)) >
    exec (comp' e2 (ADD :: c)) (eval e1 :: s)
  =< comp'-exec-eval e2 (eval e1 :: s) (ADD :: c) >
    exec (ADD :: c) (eval e2 :: eval e1 :: s)
  =<> - applying exec for ADD
    exec c (eval e1 + eval e2 :: s)
  =<> - unapplying eval for addE
    exec c (eval (addE e1 e2) :: s)
  end
```