CIS 1904: Haskell

Polymorphism & Recursion Patterns

Logistics

- HW 3 due yesterday
- HW 4 will be available this tomorrow
  - Please make sure to submit just the Exercises.hs file
- HW 2 grades will be posted tomorrow
  - We will allow resubmissions of this homework
  - Resubmission details will be posted on Ed tomorrow

Polymorphism

Lists

`List Int`

- `[]`
- `Int : List Int`

`List Double`

- `[]`
- `Double : List Double`

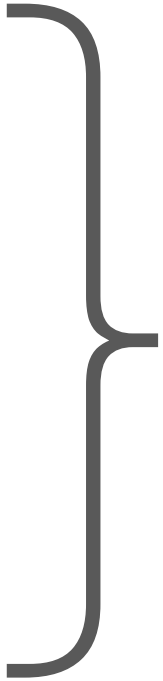
Lists

`List Int`

- `[]`
- `Int : List Int`

`List Double`

- `[]`
- `Double : List Double`



```
data List a
    = []
    | a : List a
```

Lists

```
data List a
    = []
    | a : List a
```

Lists

```
data List a
    = []
    | a : List a
```

`List` is a type *constructor*: it takes an argument and returns a type.

Lists

```
data List a
    = []
    | a : List a
```

`a` is a type *variable*: we can substitute any type in for it

Lists

```
data List a
    = []
    | a : List a
```

Note: `[a]` is just syntactic sugar for `[] a`, which is just syntactic sugar for `List a`

Parametric Polymorphism

A symbol (e.g., function name) is *polymorphic* if it can have more than one type.

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

```
foo :: [a] -> [a]
foo [] = …
foo (x : xs) = …
```

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

```
foo :: [a] -> [a]
foo [] = …
foo (x : xs) = case (typeOf x) of
    | Int -> …
    …
```

Haskell does not allow this type of casing here.

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

How many functions are there with type `a -> a`?

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

How many functions are there with type `a -> a`?
One: the identity function

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

How many terms are there with type `a`?

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

How many terms are there with type `a`?
None*

* Haskell actually uses this type for error terms like `undefined`

Maybe

```
data FailableDouble
    = Failure
    | OK Double


safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

Maybe

```
data FailableInt
    = Failure’
    | OK’ Int


safeDiv’ :: Int -> Int -> FailableInt
safeDiv’ _ 0 = Failure’
safeDiv’ x y = OK’ (x / y)
```

Maybe

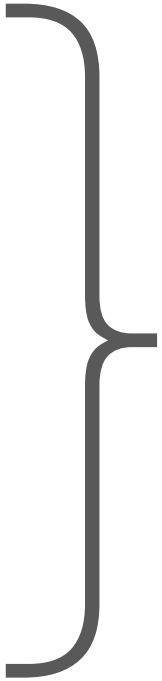
```
data FailableDouble
    = Failure
    | OK Double


data FailableInt
    = Failure’
    | OK’ Int
```

Maybe

```
data FailableDouble
    = Failure
    | OK Double


data FailableInt
    = Failure’
    | OK’ Int
```

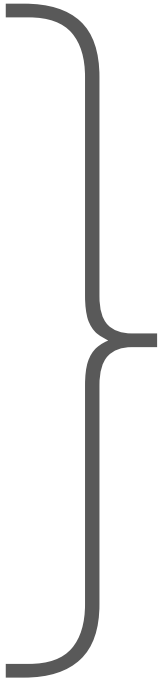


```
data Maybe a
    = Nothing
    | Just a
```

Maybe

```
data FailableDouble
    = Failure
    | OK Double


data FailableInt
    = Failure’
    | OK’ Int
```



```
data Maybe a
    = Nothing
    | Just a
```

In OCaml: `type 'a option = None | Some of 'a`

Maybe

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)

safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv m n = Just (m / n)
```

Maybe

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)

safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv m n = Just (m / n)

safeDiv' :: Int -> Int -> Maybe Int
safeDiv' _ 0 = Nothing
safeDiv' m n = Just (m / n)
```

Aside

Now that we have `Maybe`, do NOT use partial functions for the rest of this class!

- They may crash
- Without `Maybe` in the type, it is not obvious that a program using them can crash

Instead, you should:

- pattern match to avoid having to call functions like `head`
- use `Maybe`

Maybe

```
data Maybe a
    = Nothing
    | Just a
```

Exercise: Write a function `safeHead :: [a] -> Maybe a` to get the head of a list.

Maybe

```
head :: [a] -> a
head [] = errorEmptyList "head"
head (x : _) = x


safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

Parametric Polymorphism

A *parametrically polymorphic* symbol:

- Takes a type as a *parameter*
- Must be able to work with any type parameter
- Does *the same thing* for any type parameter

```
foo :: [a] -> [a]
foo [] = …
foo (x : xs) = case (typeOf x) of
    | Int -> …
    …
```

Ad Hoc Polymorphism

- Idea: Using the same “interface” (in Haskell, the same set of function names) for a related concept across a limited set of types
- Example: `+`
  - `1 + 0` `+ :: Int -> Int -> Int`
  - `2.5 + 3.8` `+ :: Double -> Double -> Double`
- In Haskell, this is done via *typeclasses*. More on this in a later lecture.

Polymorphism

A symbol is *polymorphic* if it can have more than one type

- Parametric polymorphism
- Ad-hoc polymorphism
- Others not as directly supported in Haskell
  - (Arbitrary) subtyping
    - Makes type inference very difficult
    - Makes type errors confusing
    - Multiple-inheritance issues
    - Other ways to get similar functionality

Exercise

Use pattern matching and recursion to write a function `lengths` that takes a list of lists (of any type) and returns a list of the lengths of those inner lists. You may use the `length` function from Data.List, but no others.

For example:

```
lengths [ [ ], [1,2,3], [8] ] = [0,3,1]
```

```
lengths [ [‘c’, ‘d’], [‘a’] ] = [2,1]
```

Exercise

Use pattern matching and recursion to write a function `toSingletons` that takes a list (of any type) and returns a list where each element of the input list has been replaced with a singleton list containing that element.

For example:

```
toSingletons [4,5,6] = [[4],[5],[6]]

toSingletons [‘x’] = [[‘x’]]
```

Recursion Patterns

Recursion

```
lengths :: [[a]] -> [Int]
lengths [] = []
lengths (x : xs) = length x : lengths xs


toSingletons :: [a] -> [[a]]
toSingletons [] = []
toSingletons (x : xs) = [x] : toSingletons xs
```

Recursion

```
lengths :: [[a]] -> [Int]
lengths [] = []
lengths (x : xs) = length x : lengths xs
```

```
lengths :: [[a]] -> [Int]
lengths = map length
```

```
toSingletons :: [a] -> [[a]]
toSingletons [] = []
toSingletons (x : xs) = [x] : toSingletons xs
```

```
toSingletons :: [a] -> [[a]]
toSingletons = map (: [])
```

Recursion

```
lengths :: [[a]] -> [Int]
lengths = map length
```

```
toSingletons :: [a] -> [[a]]
toSingletons = map (: [])
```

What is the type of `map`?

Recursion

```
lengths :: [[a]] -> [Int]
lengths = map length
```

```
toSingletons :: [a] -> [[a]]
toSingletons = map (: [])
```

What is the type of `map`?

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

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

- Applies a function to each element of a list
  - Remember, functions are *first class*
  - Function argument can be anonymous or named
    - e.g., `add1`
    - e.g., `(\x -> 1 + x)`
  - Functions can be operator sections `(+3)`
    - `map (+3) [0,1,2] == [0+3,1+3,2+3] == [3,4,5]`

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

Pros:

- Much more concise
- Avoids duplicated work
  - avoids potential mistakes
  - allows reuse of optimizations
- Self-documenting

All of these fit into Haskell’s overarching theme of abstracting away repeated work.

Exercise

Use `map` to write a function `squares` that squares every element of a list of integers.

Recursion

```
upperOnly :: [Char] -> [Char]
upperOnly [] = []
upperOnly (x : xs)
  | isUpper x = x : upperOnly xs
  | otherwise = upperOnly xs


posOnly :: [Int] -> [Int]
posOnly [] = []
posOnly (x : xs)
  | x > 0 = x : posOnly xs
  | otherwise = posOnly xs
```

Recursion

```
upperOnly :: [Char] -> [Char]
upperOnly [] = []
upperOnly (x : xs)
  | isUpper x = x : upperOnly xs
  | otherwise = upperOnly xs


posOnly :: [Int] -> [Int]
posOnly [] = []
posOnly (x : xs)
  | x > 0 = x : posOnly xs
  | otherwise = posOnly xs
```

Exercise: use recursion and pattern matching to write a function `foo` such that:

`foo isUpper` is the same as `upperOnly`

`foo (> 0)` is the same as `posOnly`

Filter

```
upperOnly :: [Char] -> [Char]
upperOnly [] = []
upperOnly (x : xs)
  | isUpper x = x : upperOnly xs
  | otherwise = upperOnly xs


posOnly :: [Int] -> [Int]
posOnly [] = []
posOnly (x : xs)
  | x > 0 = x : posOnly xs
  | otherwise = posOnly xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred []  = []
filter pred (x:xs)
    | pred x    = x : filter pred xs
    | otherwise = filter pred xs
```

Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

Useful for filtering elements of a list by a given predicate.

Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

Useful for filtering elements of a list by a given predicate.

Exercise: use `filter` to write a function that removes spaces from a string.

Hint: you may use `isSpace`.

Recursion

```
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs


and :: [Bool] -> Bool
and [] = True
and x : xs = x && and xs
```

Recursion

```
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

```
and :: [Bool] -> Bool
and [] = True
and x : xs = x && and xs
```

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

```
sum = foldr (+) 0
```

```
and = foldr (&&) True
```

Recursion

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```haskell
sum = foldr (+) 0
```

```haskell
and :: [Bool] -> Bool
and [] = True
and x : xs = x && and xs
```

```haskell
and = foldr (&&) True
```

Recursion

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + (sum xs)
```

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```haskell
sum = foldr (+) 0
```

```haskell
and :: [Bool] -> Bool
and [] = True
and x : xs = x && (and xs)
```

```haskell
and = foldr (&&) True
```

Fold

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

Useful for combining all the elements of a list in some way.

Fold

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

z is the “default” value

f tells us how to combine the elements of the list and the default value

Fold

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

You can think of `foldr` as replacing `:` with `f` and `[]` with `z`.

`1 : 5 : 2 : []`

`1 + 5 + 2 + 0`

Fold

```
foldr :: (b -> a -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)

foldr add 0 [1,2,3]
add 1 (foldr add 0 [2,3])
add 1 (add 2 (foldr add 0 [3]))
add 1 (add 2 (add 3 (foldr 0 [])))
add 1 (add 2 (add 3 0))
add 1 (add 2 3)
add 1 5
6
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))
```

We can see that `foldr` is right-associative.

What if we want a left-associative fold?

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

Fold

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs
```

You can think of `foldl` like an iterator in another language.

```
acc = z;
for x in xs:
    acc = f acc x
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

When are `foldr` and `foldl` equivalent (ignoring performance)?

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

`foldr` and `foldl` are equivalent if `f` is commutative and associative!

Fold

When should we use `foldr` vs. `foldl`?

Fold

When should we use `foldr` vs. `foldl`?

We almost never use `foldl`!

We do, however, use `foldl'`.

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
let x1 = 0+1 in ((x1+2)+3)
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
let x1 = 0+1 in ((x1+2)+3)
((0+1)+2)+3
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
let x1 = 0+1 in ((x1+2)+3)
((0+1)+2)+3
(1+2)+3
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
let x1 = 0+1 in ((x1+2)+3)
((0+1)+2)+3
(1+2)+3
3+3
```

Fold

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x  in foldl f z' xs

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
let x1 = 0+1 in ((x1+2)+3)
((0+1)+2)+3
(1+2)+3
3+3
6
```

Fold

`foldl'` removes some of the laziness

```
foldl' add 0 [1,2,3]
foldl' add (0 + 1) [2,3]
foldl' add 1 [2,3]
foldl' add (1 + 2) [3]
foldl' add 3 [3]
foldl' add (3 + 3) []
foldl' add 6 []
6
```

Foldl’

- Designed to be space efficient but semantically the same as `foldl`
- There are very few reasons to use `foldl`
  - `foldl’` is recommended by Haskell

Which is preferred between `foldl’` and `foldr`?

Foldl’

- Designed to be space efficient but semantically the same as `foldl`
- There are very few reasons to use `foldl`
  - `foldl’` is recommended by Haskell

Which is preferred between `foldl’` and `foldr`?

It depends.

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (+) 0 [4,5,6]

foldl' (+) 0 [4,5,6]
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (+) 0 [4,5,6] = 15

foldl' (+) 0 [4,5,6] = 15
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (-) 0 [1,1,1] ==

foldl' (-) 0 [1,1,1] ==
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (-) 0 [1,1,1] == 1

foldl' (-) 0 [1,1,1] == -3
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (\x y -> (x + y) / 2) 0 [8,4,16]

foldl' (\x y -> (x + y) / 2) 0 [8,4,16]
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (\x y -> (x + y) / 2) 0 [8,4,16] == 7

foldl' (\x y -> (x + y) / 2) 0 [8,4,16] == 10
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (++) "z" ["a","b","c"] ==

foldl' (++) "z" ["a","b","c"] ==
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl' f z [a,b,c] == f (f (f z a) b) c


foldr (++) "z" ["a","b","c"] == "abcz"

foldl' (++) "z" ["a","b","c"] == "zabc"
```

Fold

Which is preferred between `foldl'` and `foldr`?

When they are equivalent, typically `foldr`.

Foldr

- follows natural structure of the list
  - `f a (f b (f c z))`
- Allows for short-circuiting
  - `False && (b && (c && z))`
- We can sometimes use it with infinite lists!
  - We will discuss more in the laziness unit

Foldr vs. Foldl

Fold**r**:

- **Right** associative
- Goes through the list **right** to left
- **Replaces** nil and cons with z and f

Fold**l**:

- **Left** associative
- Goes through the list **left** to right
- Similar to an imperative program that **loops** through the list

Examples

```
reverse :: [a] -> [a]
reverse xs =
```

Examples

```
reverse :: [Int] -> [Int]
reverse = foldr comb []
    where
        comb x res = res ++ [x]
```

Examples

```
reverse :: [Int] -> [Int]
reverse = foldr comb []
    where
        comb x res = res ++ [x]
```

How would we write this with `foldl`?

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

Examples

```
reverse :: [Int] -> [Int]
reverse = foldl comb []
    where
        comb res x = x : res
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```