CS 457/557: Functional Languages

Lecture 4: Laws; Folds

Mark P Jones and Andrew Tolmach
Portland State University

Lawful Programming

How can we give useful information about a function without necessarily having to give all the details of its definition?

- Informal description:

  "map applies its first argument to every element in its second argument …"

- Type signature:

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

- Laws:
  - Normally in the form of equalities between expressions …

Algebra of Lists

- (++) is associative with unit []

  xs ++ (ys ++ zs) = (xs ++ ys) ++ zs

  [] ++ xs = xs = xs ++ []

- map preserves identities, distributes over composition and concatenation:

  map id = id

  map (f . g) = map f . map g

  map f (xs ++ ys) = map f xs ++ map f ys

… continued

- filter distributes over concatenation
  - filter p (xs ++ ys) = filter p xs ++ filter p ys
- Filters and maps:
  - filter p . map f = map f . filter (p . f)
- Composing filters:
  - filter p . filter q = filter r
    where r x = q x && p x

Aside: Extensionality

- Two functions are equal if they give the same results on the same arguments

$$f = g \Leftrightarrow \forall x.\ f\ x = g\ x$$

- Example: f x = 1 + 2*x and g = (1+).(2*), then:

```
g x   = ((1+) . (2*)) x
      = (1+) ((2*) x)
      = 1 + 2*x
      = f x
```

- Hence f = g

Laws Describe Interactions

- A lot of laws describe how one operator interacts with another

- Example: interactions with reverse:
  - reverse . map f = map f . reverse
  - reverse . filter p = filter p . reverse
  - map f . map g = map (f . g)
  - reverse (xs ++ ys) = reverse ys ++ reverse xs
  - reverse . reverse = reverse

- Caution: stating a law doesn't make it true! (e.g., the last two laws for reverse are not true of **all** lists…)

Uses for Laws

Laws can be used:

- To capture/document deep intuitions about program behavior
- To support reasoning about program behavior
- To optimize or transform programs (either by hand, or in a compiler)
- As properties to be tested
- As properties to be proved

concat

- concat :: [[a]] -> [a]
- concat [[1,2], [3,4,5], [6]]
  = [1,2,3,4,5,6]

- Laws:
  - filter p . concat = concat . map (filter p)
  - map f . concat = concat . map (map f)
  - concat . concat = concat . map concat

Folds:

Folds!

- A list xs can be built by applying the (:) and [] operators to a sequence of values:

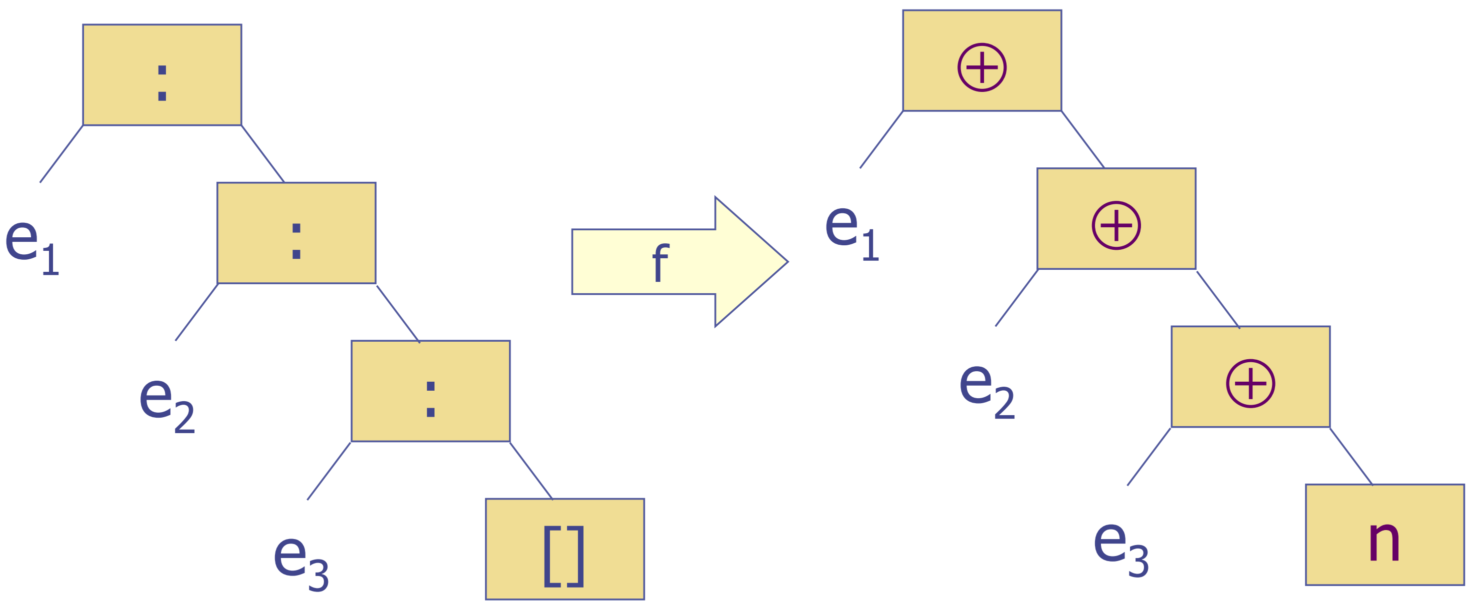
$$xs = x_1 : x_2 : x_3 : x_4 : \ldots : x_k : []$$

- Suppose that we are able to replace every use of (:) with a binary operator (⊕), and the final [] with a value n:

$$xs = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \ldots \oplus x_k \oplus n$$

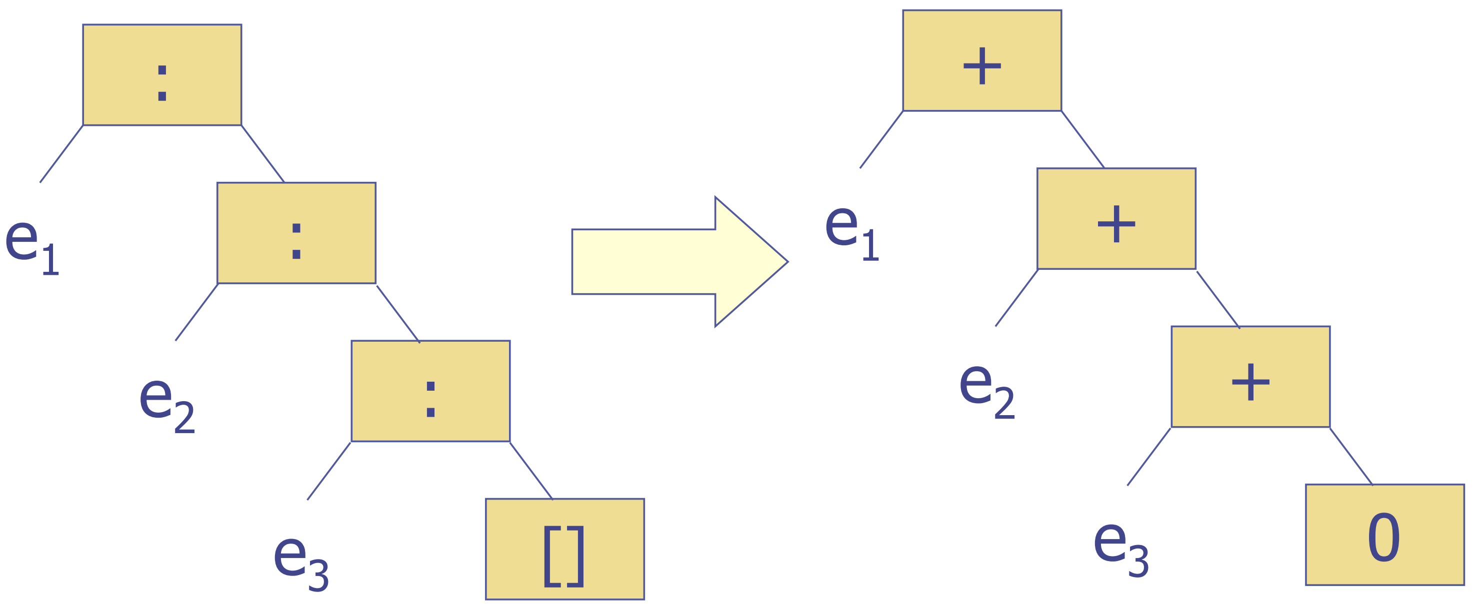
- The resulting value is called foldr (⊕) n xs
- Many useful functions on lists can be described in this way.

Graphically



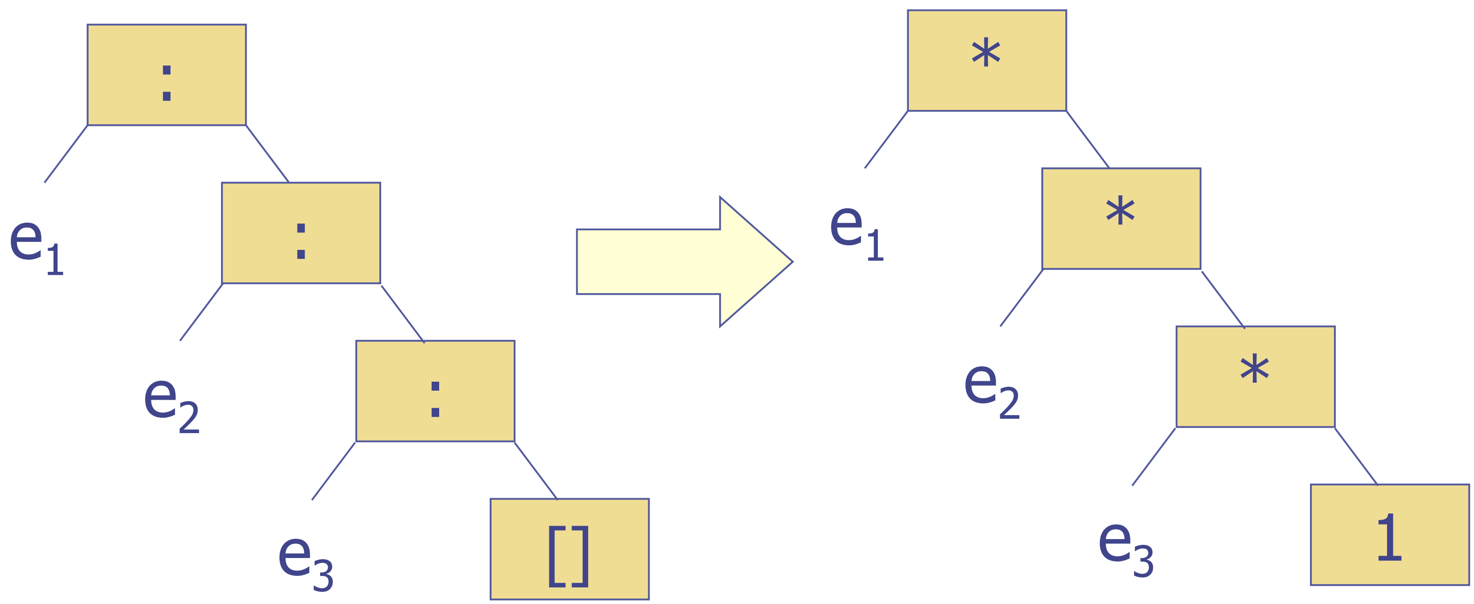
$$f = \text{foldr}\ (\oplus)\ n$$

Example: sum



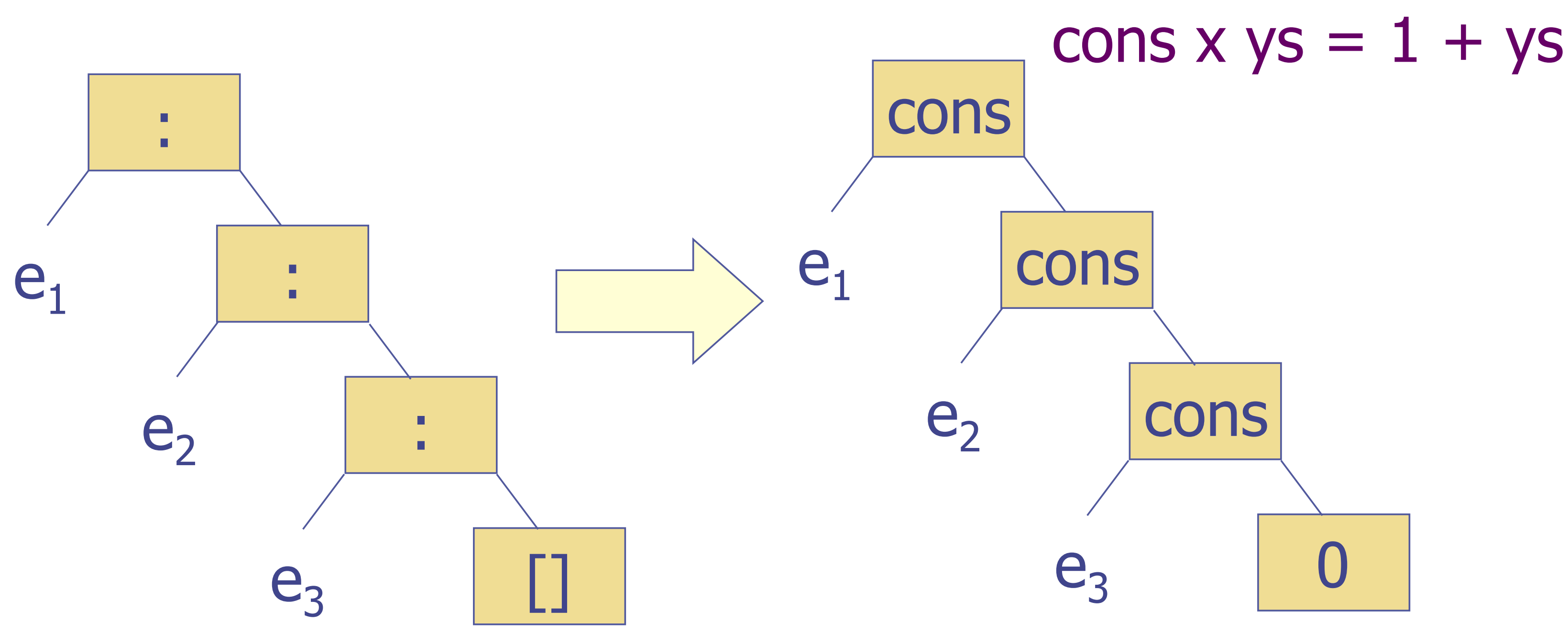
sum = foldr (+) 0

Example: product



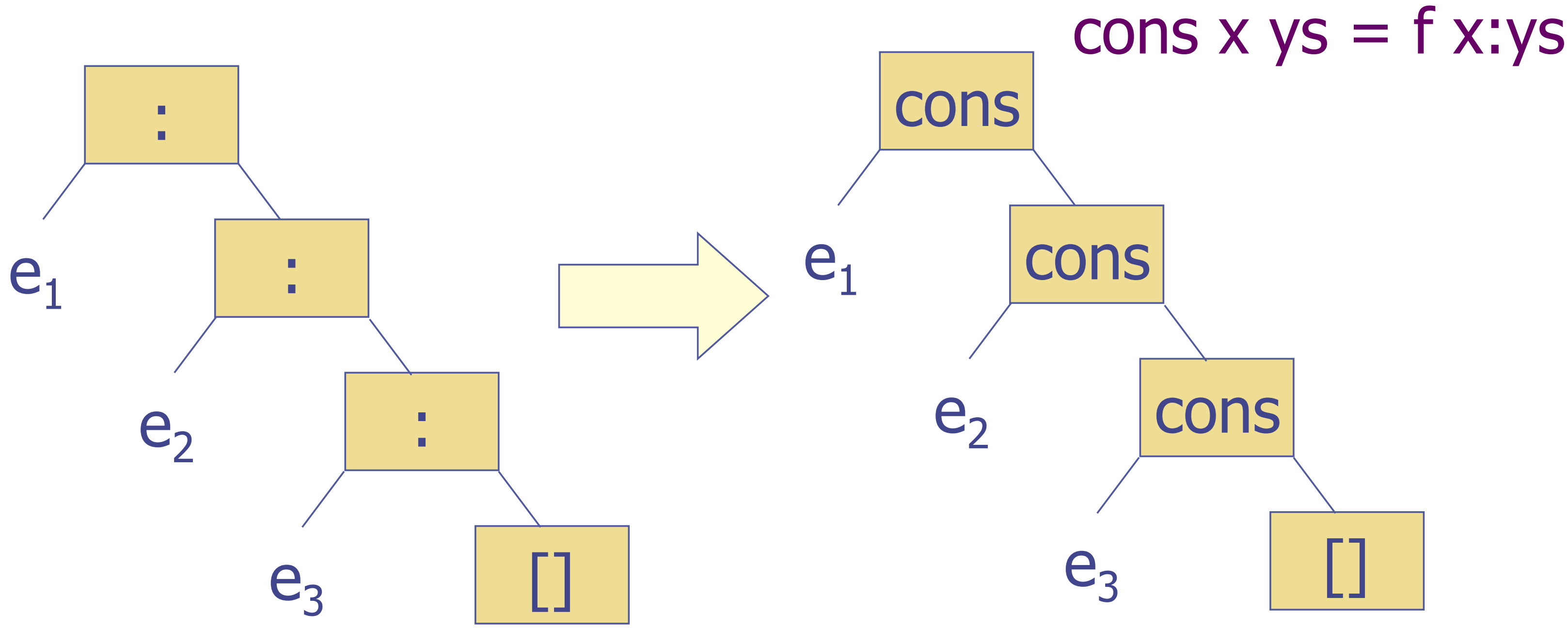
product = foldr (*) 1

Example: length



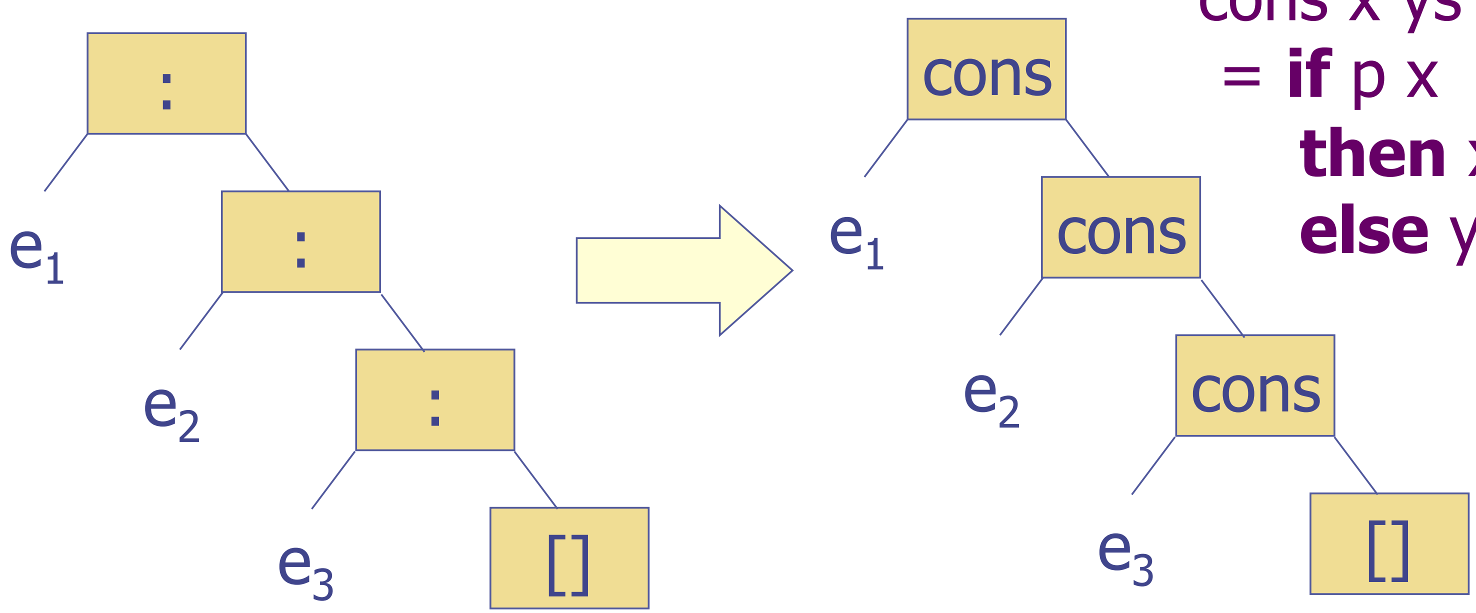
length = foldr (\x ys -> 1 + ys) 0

Example: map



map f = foldr (\x ys -> f x : ys) []

Example: filter



cons x ys
 = **if** p x
 then x:ys
 else ys

filter p = foldr (\x ys -> **if** p x **then** x:ys **else** ys) []

Formal Definition

```
foldr                   :: (a->b->b) -> b -> [a] -> b
foldr cons nil []       = nil
foldr cons nil (x:xs)   = cons x (foldr cons nil xs)
```

Applications

```
sum         = foldr (+) 0
product     = foldr (*) 1
length      = foldr (\x ys -> 1 + ys) 0
map f       = foldr (\x ys -> f x : ys) []
filter p    = foldr c []
  where c x ys = if p x then x:ys else ys
xs ++ ys    = foldr (:) ys xs
and         = foldr (&&) True
or          = foldr (||)  False
```

Patterns of Computation

- foldr captures a common pattern of computations over lists
- As such, it is a very useful function in practice to include in the Prelude
- Even from a theoretical perspective, it is very useful because it makes a deep connection between functions that might otherwise seem very different …
- From the perspective of lawful programming, one law about foldr can be used to reason about many other functions

A law about foldr

- If $(\oplus)$ is an associative operator with unit n, then

$$\text{foldr } (\oplus)\ n\ xs \quad \oplus \quad \text{foldr } (\oplus)\ n\ ys = \text{foldr } (\oplus)\ n\ (xs \mathbin{++} ys)$$

- $(x_1 \oplus \ldots \oplus x_k \oplus n) \oplus (y_1 \oplus \ldots \oplus y_j \oplus n)$
  $= (x_1 \oplus \ldots \oplus x_k \oplus y_1 \oplus \ldots \oplus y_j \oplus n)$

- All of the following laws are special cases:

```
sum xs      +   sum ys      = sum (xs ++ ys)
product xs  *   product ys  = product (xs ++ ys)
and xs      &&  and ys      = and (xs ++ ys)
or xs       ||  or  ys      = or (xs ++ ys)
```

foldl

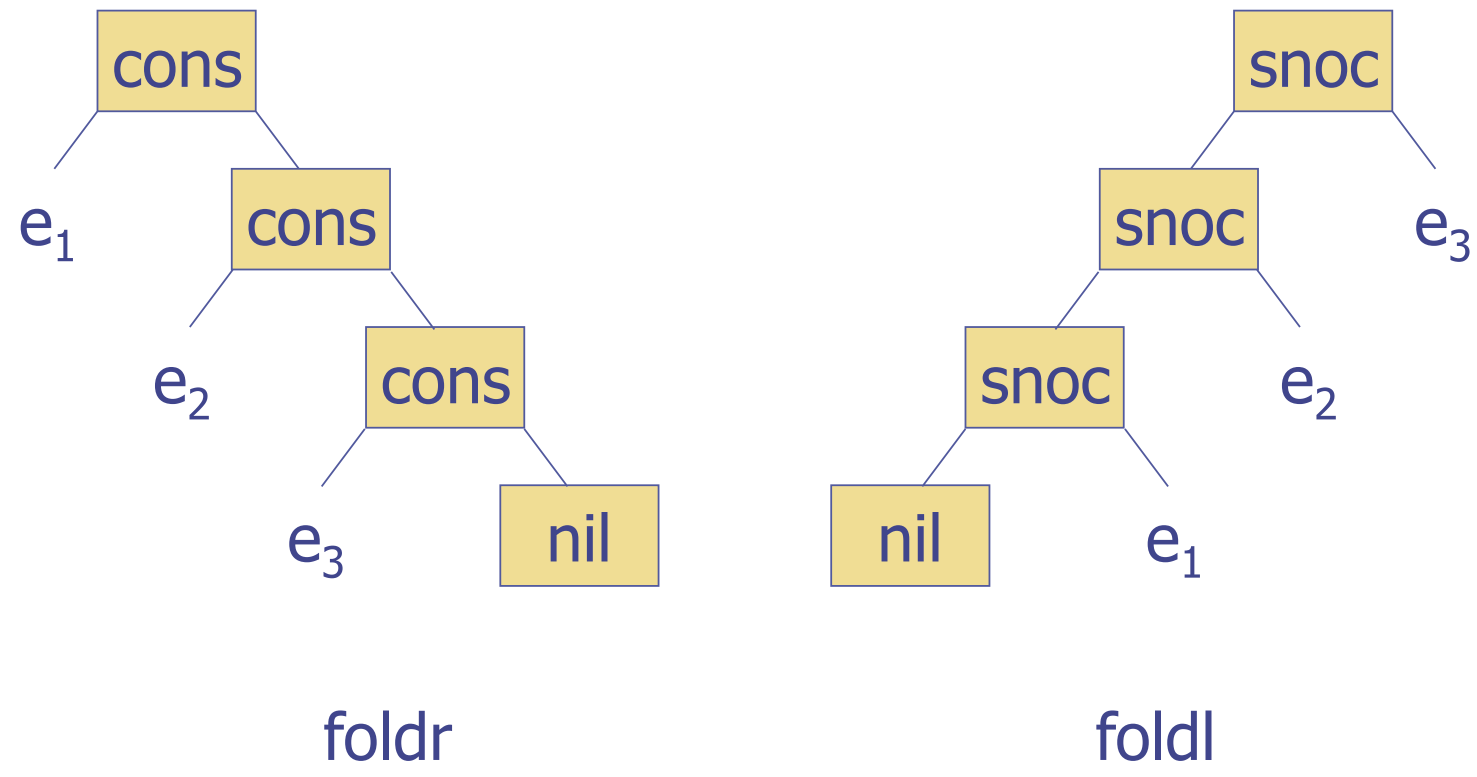
- There is a companion function to foldr called foldl:

```
foldl              :: (b -> a -> b) -> b -> [a] -> b
foldl s n []       = n
foldl s n (x:xs) = foldl s (s n x) xs
```

- For example:

$$\begin{aligned} &\text{foldl s n } [e_1, e_2, e_3] \\ &\quad = s\ (s\ (s\ n\ e_1)\ e_2)\ e_3 \\ &\quad = ((n\ \grave{}s\grave{}\ e_1)\ \grave{}s\grave{}\ e_2)\ \grave{}s\grave{}\ e_3 \end{aligned}$$

foldr vs foldl



Uses for foldl

- Many of the functions defined using foldr can be defined using foldl:

```
sum      = foldl (+) 0
product  = foldl (*) 1
```

- There are also some functions that are more easily defined using foldl:

```
reverse  = foldl (\ys x -> x:ys) []
```

- When should you use foldr and when should you use foldl? When should you use explicit recursion instead? … (to be continued)

foldr1 and foldl1

- Variants of foldr and foldl that work on non-empty lists:

```
foldr1            :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs)   = f x (foldr1 f xs)

foldl1            :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)   = foldl f x xs
```

- Notice:
  - No case for empty list
  - No argument to replace empty list
  - Less general type (only one type variable)

Uses of foldl1, foldr1

From the prelude:

```
minimum   = foldl1 min
maximum  = foldl1 max
```

Not in the prelude:

```
commaSep = foldr1 (\s t -> s ++ ", " ++ t)
```

Example: Adding Commas

To make large numbers easier to read, it is common to insert a comma after every third digit, starting from the right.

Example: 1234567 -> “1,234,567”

The show function can turn an Integer into a String, but how do we insert the commas?

Example: Adding Commas

```
commas
    =  reverse
     . foldr1 (\s t -> s++","++t)
     . group 3
     . reverse
     . show
```

1234567

"1234567"

"7654321"

["765", "432", "1"]

"765,432,1"

"1,234,567"

Summary

- Folds on lists have many uses

- Folds capture a common pattern of computation on list values

- In fact, there are similar notions of fold functions on many other algebraic datatypes …
  - (Hence the Foldable type class…)