Introduction to Programming: Lecture 6

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

27 August 2013

Folding from the Left

- Sometimes it is useful to combine the elements of a list from left to right.

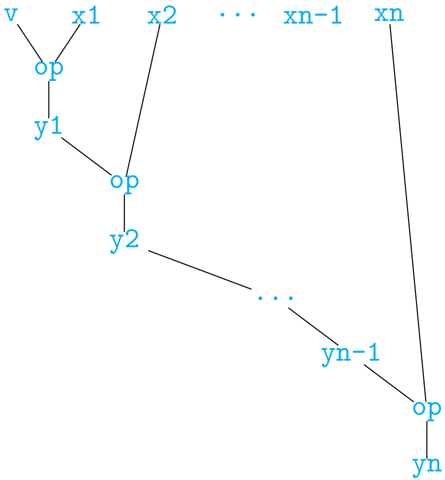
Folding from the Left

- Sometimes it is useful to combine the elements of a list from left to right.
- `foldl ::  (a -> b -> a) -> a -> [b] -> a`

  ```
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```

foldl

```
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```



Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

  `strtonum "234" = 234`

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

  `strtonum "234" = 234`

- Convert a character into the corresponding digit:

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

  ```
  strtonum "234" = 234
  ```

- Convert a character into the corresponding digit:

  ```
  chartonum :: Char -> Int
  chartonum c
    | ('0' <= c) && (c <= '9') = (ord c) - (ord '0')
  ```

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

  ```
  strtonum "234" = 234
  ```

- Convert a character into the corresponding digit:

  ```
  chartonum :: Char -> Int
  chartonum c
    | ('0' <= c) && (c <= '9') = (ord c) - (ord '0')
  ```

- Process the digits from the left and at each digit

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

  ```
  strtonum "234" = 234
  ```

- Convert a character into the corresponding digit:

  ```
  chartonum :: Char -> Int
  chartonum c
    | ('0' <= c) && (c <= '9') = (ord c) - (ord '0')
  ```

- Process the digits from the left and at each digit
  - Multiply the current sum by 10 and add the current digit to it.

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

```
strtonum "234" = 234
```

- Convert a character into the corresponding digit:

```
chartonum :: Char -> Int
chartonum c
  | ('0' <= c) && (c <= '9') = (ord c) - (ord '0')
```

- Process the digits from the left and at each digit
  - Multiply the current sum by 10 and add the current digit to it.

```
nextdigit :: Int -> Char -> Int
nextdigit i c = 10*i + (chartonum c)
```

- What next?

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

```
strtonum "234" = 234
```

- Convert a character into the corresponding digit:

```
chartonum :: Char -> Int
chartonum c
  | ('0' <= c) && (c <= '9') = (ord c) - (ord '0')
```

- Process the digits from the left and at each digit
  - Multiply the current sum by 10 and add the current digit to it.

```
nextdigit :: Int -> Char -> Int
nextdigit i c = 10*i + (chartonum c)
```

- What next? Express `strtonum` using `nextdigit` either directly via recursion or

Ascii to Integers

- Translate a given `String` of digits into the corresponding integer.

  ```
  strtonum "234" = 234
  ```

- Convert a character into the corresponding digit:

  ```
  chartonum :: Char -> Int
  chartonum c
    | ('0' <= c) && (c <= '9') = (ord c) - (ord '0')
  ```

- Process the digits from the left and at each digit
  - Multiply the current sum by 10 and add the current digit to it.

  ```
  nextdigit :: Int -> Char -> Int
  nextdigit i c = 10*i + (chartonum c)
  ```

- What next? Express `strtonum` using `nextdigit` either directly via recursion or using `foldl`

  ```
  strtonum = foldl nextdigit 0
  ```

Equality and Order

- All the basic types have equality and order defined on them

Equality and Order

- All the basic types have equality and order defined on them
- Higher order types do NOT have equality or order

  `map == map` will result in a type error. There is no equality defined on the type of `map`

Equality and Order

- All the basic types have equality and order defined on them
- Higher order types do NOT have equality or order

  `map == map` will result in a type error. There is no equality defined on the type of `map`
- Order (`<`) is also not defined on higher order types.

  In general it is not possible to check if two functions compute the same values on all inputs.

Equality over lists and tuples

- Equality is inherited from the underlying type.

Equality over lists and tuples

- Equality is inherited from the underlying type.
- Two lists are equal if and only if they have the same length and the corresponding elements are equal.

```
[1,2,3] = [1,2,3]
[1,2,3] /= [2,1,3]
```

Equality over lists and tuples

- Equality is inherited from the underlying type.
- Two lists are equal if and only if they have the same length and the corresponding elements are equal.

```
[1,2,3] = [1,2,3]
[1,2,3] /= [2,1,3]
```

- Two tuples are equal if and only if the corresponding positions have equal values.

```
(True, False) = (True,False)
(True, 17, False) = (True, 17, False)
(True, 17, False) /= (False, 17, True)
```

Order in lists and tuples

- Tuples of the same type are ordered lexicographically provided all the underlying types are ordered.

```
(False, 17, 24) < (False, 18, 1)
(False, 89) < (False, 100)
```

Order in lists and tuples

- Tuples of the same type are ordered lexicographically provided all the underlying types are ordered.

  `(False, 17, 24) < (False, 18, 1)`

  `(False, 89) < (False, 100)`

  `(89,(+)) < (90,(+))` will result in a type error.

Order in lists and tuples

- Tuples of the same type are ordered lexicographically provided all the underlying types are ordered.

  ```
  (False, 17, 24) < (False, 18, 1)
  (False, 89) < (False, 100)
  ```

  `(89,(+)) < (90,(+))` will result in a type error.

- Lists are ordered lexicographically if the underlying type is ordered.

  ```
  [1,2,3] < [1,2,3,0]
  [] < [1]
  [1,2,3] < [1,2,4]
  ```

Sorting: Insertion sorting

- Given a list of integers rearrange it in ascending order.

Sorting: Insertion sorting

- Given a list of integers rearrange it in ascending order.
- Here is one way to do this
  - remove the first element to get a shorter list
  - sort this shorter list
  - insert the first element in the correct position in the sorted list

Sorting: Insertion sorting

- Given a list of integers rearrange it in ascending order.
- Here is one way to do this
  - remove the first element to get a shorter list
  - sort this shorter list
  - insert the first element in the correct position in the sorted list

```
sort [2,1,3,2,4]
```

Sorting: Insertion sorting

- Given a list of integers rearrange it in ascending order.
- Here is one way to do this
  - remove the first element to get a shorter list
  - sort this shorter list
  - insert the first element in the correct position in the sorted list

`sort [2,1,3,2,4]`

$\leadsto$`insert 2` in `sort [1,3,2,4]`

Sorting: Insertion sorting

- Given a list of integers rearrange it in ascending order.
- Here is one way to do this
  - remove the first element to get a shorter list
  - sort this shorter list
  - insert the first element in the correct position in the sorted list

`sort [2,1,3,2,4]`

$\leadsto$ `insert 2` in `sort [1,3,2,4]`

$\leadsto$ `insert 2` in `[1,2,3,4]`

Sorting: Insertion sorting

- Given a list of integers rearrange it in ascending order.
- Here is one way to do this
  - remove the first element to get a shorter list
  - sort this shorter list
  - insert the first element in the correct position in the sorted list

`sort [2,1,3,2,4]`

$\rightsquigarrow$ `insert 2` in `sort [1,3,2,4]`

$\rightsquigarrow$ `insert 2` in `[1,2,3,4]`

$\rightsquigarrow$ `[1,2,2,3,4]`

Insertion Sorting

- How to insert a number in the correct position in a sorted list?

Insertion Sorting

- How to insert a number in the correct position in a sorted list?

```
insert 3 [1,2,2,4] = [1,2,2,3,4]
insert 3 [] = [3]
```

Insertion Sorting

- How to insert a number in the correct position in a sorted list?

  ```
  insert 3 [1,2,2,4] = [1,2,2,3,4]
  insert 3 [] = [3]
  ```

- ```
  insert Int -> [Int] -> [Int]
  insert x [] = [x]
  insert x (y:ys)
    | (x <= y) = x:y:ys
    | otherwise = y : insert x ys
  ```
  ```

Insertion Sorting

- How to insert a number in the correct position in a sorted list?

  ```
  insert 3 [1,2,2,4] = [1,2,2,3,4]
  insert 3 [] = [3]
  ```

- ```
  insert Int -> [Int] -> [Int]
  insert x [] = [x]
  insert x (y:ys)
    | (x <= y) = x:y:ys
    | otherwise = y : insert x ys
  ```

- Note that `insert` is not polymorphic, in the sense that we have seen so far, as `<=` is not necessarily defined on all types.
  ```

Insertion Sorting

- Define `isort :: [Int] -> [Int]` using `insert`

Insertion Sorting

- Define `isort :: [Int] -> [Int]` using `insert`

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Insertion Sorting

- Define `isort :: [Int] -> [Int]` using `insert`

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

or more succinctly `isort = foldr insert []`

Sorting: Merge Sorting

- Here is another method to sort a list.
  - Divide the list into two halves.
  - Sort each recursively using this algorithm.
  - Merge together the two sorted lists into a single sorted list.

Merging two sorted lists

5	16	83	99	33	55	85	93

Merging two sorted lists

5	16	83	99

33	55	85	93

Merging two sorted lists

5	16	83	99

33	55	85	93

5

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16						

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16	33

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16	33	55				

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16	33	55	83

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16	33	55	83	85

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16	33	55	83	85	93

Merging two sorted lists

5	16	83	99

33	55	85	93

5	16	33	55	83	85	93	99

Mergesort

99	5	83	16	85	33	93	55

Mergesort

99	5	83	16	85	33	93	55

99	5	83	16	85	33	93	55

Mergesort

99 5 83 16 85 33 93 55

99 5 83 16	85 33 93 55

99 5	83 16	85 33	93 55

Mergesort

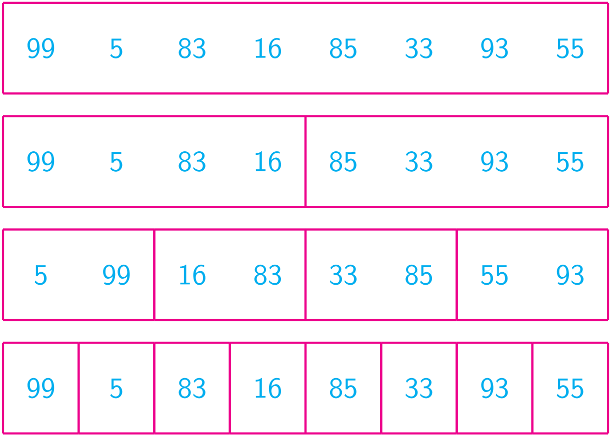
99	5	83	16	85	33	93	55

99	5	83	16	85	33	93	55

99	5	83	16	85	33	93	55

99	5	83	16	85	33	93	55

Mergesort



99 5 83 16 85 33 93 55
99 5 83 16 | 85 33 93 55
5 99 | 16 83 | 33 85 | 55 93
99 | 5 | 83 | 16 | 85 | 33 | 93 | 55

Mergesort

99	5	83	16	85	33	93	55

99	5	83	16	85	33	93	55

5	99	16	83	33	85	55	93

Mergesort

99	5	83	16	85	33	93	55

5	16	83	99	33	55	85	93

5	99	16	83	33	85	55	93

Mergesort

99	5	83	16	85	33	93	55

5	16	83	99	33	55	85	93

Mergesort

5	16	33	55	83	85	93	99

5	16	83	99	33	55	85	93

Mergesort

5	16	33	55	83	85	93	99

Merging in Haskell

- `merge` to merge two sorted lists of integers.

Merging in Haskell

- `merge` to merge two sorted lists of integers.

```
merge :: [Int] -> [Int] -> [Int]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys)
   | (x < y) = x : merge xs (y:ys)
   | otherwise = y : merge (x:xs) ys
```

Merging in Haskell

- `merge` to merge two sorted lists of integers.

```
merge :: [Int] -> [Int] -> [Int]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys)
  | (x < y) = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

- Once again, `merge` is not polymorphic, in the sense that we have seen so far, as it uses `<`.

Mergesort in Haskell

- Using `merge` to write down a `mergesort`

Mergesort in Haskell

- Using `merge` to write down a `mergesort`

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort fhalf)
                    (mergesort shalf)
              where
              fhalf = take n l
              shalf = drop n l
              n = div (length l) 2
```

Sorting: Quicksort

- Suppose we can find the median (middle element in the sorted order) in a list

Sorting: Quicksort

- Suppose we can find the median (middle element in the sorted order) in a list
- Collect all values less than median and sort

Sorting: Quicksort

- Suppose we can find the median (middle element in the sorted order) in a list
- Collect all values less than median and sort
- Collect all values more than median and sort

Sorting: Quicksort

- Suppose we can find the median (middle element in the sorted order) in a list
- Collect all values less than median and sort
- Collect all values more than median and sort
- Combine these sorted sublists using `++` (No need to `merge`)

Sorting: Quicksort

- Suppose we can find the median (middle element in the sorted order) in a list
- Collect all values less than median and sort
- Collect all values more than median and sort
- Combine these sorted sublists using `++` (No need to `merge`)

  Median is not easy to find.

Quicksort

A sorting algorithm proposed by C.A.R.Hoare. It is extensively used in practice.

Quicksort

A sorting algorithm proposed by C.A.R.Hoare. It is extensively used in practice.

- Pick a *pivot* p from the list L

Quicksort

A sorting algorithm proposed by C.A.R.Hoare. It is extensively used in practice.

- Pick a *pivot* p from the list L
- Rearrange the list as $L_1 p L_2$ where

Quicksort

A sorting algorithm proposed by C.A.R.Hoare. It is extensively used in practice.

- Pick a *pivot* p from the list L
- Rearrange the list as $L_1 p L_2$ where
  - L_1 is the list of elements of L that are smaller than p and

Quicksort

A sorting algorithm proposed by C.A.R.Hoare. It is extensively used in practice.

- Pick a *pivot* p from the list L
- Rearrange the list as $L_1 p L_2$ where
  - L_1 is the list of elements of L that are smaller than p and
  - L_2 is the list of elements of L that are at least as big as p.

Quicksort

A sorting algorithm proposed by C.A.R.Hoare. It is extensively used in practice.

- Pick a *pivot* p from the list L
- Rearrange the list as $L_1 p L_2$ where
  - L_1 is the list of elements of L that are smaller than p and
  - L_2 is the list of elements of L that are at least as big as p.
- Sort L_1 and L_2 recursively.

Quicksort ...

265 319 389 345 159 267 348 365 128

Quicksort ...

265	319	389	345	159	267	348	365	128
159	128	265	319	389	345	267	348	365

Quicksort ...

265	319	389	345	159	267	348	365	128
159	128	265	319	389	345	267	348	365
128	159	265	267	319	345	365	389	348

Quicksort ...

265	319	389	345	159	267	348	365	128
159	128	265	319	389	345	267	348	365
128	159	265	267	319	345	365	389	348
128	159	265	267	319	345	365	389	348

Quicksort ...

265	319	389	345	159	267	348	365	128
159	128	265	319	389	345	267	348	365
128	159	265	267	319	345	365	389	348
128	159	265	267	319	345	365	389	348
128	159	265	267	319	345	348	365	389

Quicksort ...

265	319	389	345	159	267	348	365	128
159	128	265	319	389	345	267	348	365
128	159	265	267	319	345	365	389	348
128	159	265	267	319	345	365	389	348
128	159	265	267	319	345	348	365	389
128	159	265	267	319	345	348	365	389

Quicksort ...

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

Quicksort ...

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

- In the worst case, `lower` or `upper` is empty

`Quicksort` ...

```haskell
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

- In the worst case, `lower` or `upper` is empty
  - For our choice of splitter, worst case input is any sorted list

Measuring efficiency in Haskell

- Computation in Haskell is rewriting
  - Using a definition to rewrite an expression = reduction step

Measuring efficiency in Haskell

- Computation in Haskell is rewriting
  - Using a definition to rewrite an expression = reduction step
- Count number of reduction steps $T(n)$ for input of size n

Measuring efficiency in Haskell

- Computation in Haskell is rewriting
  - Using a definition to rewrite an expression = reduction step
- Count number of reduction steps $T(n)$ for input of size n
- What is the complexity of `++`?

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

Measuring efficiency in Haskell

- Computation in Haskell is rewriting
  - Using a definition to rewrite an expression = reduction step
- Count number of reduction steps $T(n)$ for input of size n
- What is the complexity of ++?

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

```
[1,2,3] ++ [4,5,6] ->
1:([2,3] ++ [4,5,6]) ->
1:(2:([3] ++ [4,5,6]))  ->
1:(2:(3:([] ++ [4,5,6])))  ->
1:(2:(3:([4,5,6])))
```

Measuring efficiency in Haskell

- Computation in Haskell is rewriting
  - Using a definition to rewrite an expression = reduction step
- Count number of reduction steps $T(n)$ for input of size n
- What is the complexity of `++`?

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)

[1,2,3] ++ [4,5,6] ->
1:([2,3] ++ [4,5,6]) ->
1:(2:([3] ++ [4,5,6]))  ->
1:(2:(3:([] ++ [4,5,6])))  ->
1:(2:(3:([4,5,6])))
```

- In `l1 ++ l2` we use the second rule `length l1` times and the first rule once
- It takes as many steps as `length l1 + 1`

Efficiency ...

- We would like to define the complexity of a program to be a function from the size of the input to the number of steps.

Efficiency ...

- We would like to define the complexity of a program to be a function from the size of the input to the number of steps.

  $T(n) = n$ for ++ where n is the length of the left argument.

Efficiency ...

- We would like to define the complexity of a program to be a function from the size of the input to the number of steps.

  $T(n) = n$ for ++ where n is the length of the left argument.

- The function ++ takes the same number of steps on inputs of the same length.

  This need not always be the case.

Efficiency ...

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- The number of steps taken depends on the input (not just its length)

Efficiency ...

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- The number of steps taken depends on the input (not just its length)

```
elem 3 [3,7,8,9]  ->
  True
```

Efficiency ...

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- The number of steps taken depends on the input (not just its length)

```
elem 3 [3,7,8,9]  ->
  True
```

```
elem 3 [4,7,8,9] ->
  elem 3 [7,8,9] ->
    elem 3 [8,9] ->
      elem 3 [9] ->
        elem 3 [] ->
          False
```

Efficiency ...

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- The number of steps taken depends on the input (not just its length)

```
elem 3 [3,7,8,9]  ->
  True
```

```
elem 3 [4,7,8,9] ->
  elem 3 [7,8,9] ->
    elem 3 [8,9] ->
      elem 3 [9] ->
        elem 3 [] ->
          False
```

- What do we take the value of $T(n)$ to be?

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity
  - The minimum among all inputs of length `n`. Best-case complexity

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity
  - The minimum among all inputs of length `n`. Best-case complexity
  - The average among all inputs of length `n`. Average-case complexity

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity
  - The minimum among all inputs of length `n`. Best-case complexity
  - The average among all inputs of length `n`. Average-case complexity
- Best-case complexity is useless. For eg. it suggests that `elem` returns the answer in one step on any inputs of any length `n`.

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity
  - The minimum among all inputs of length `n`. Best-case complexity
  - The average among all inputs of length `n`. Average-case complexity
- Best-case complexity is useless. For eg. it suggests that `elem` returns the answer in one step on any inputs of any length `n`.
- Worst-case is pessimistic. But it assures us that on any input of length `n` the program does NOT take more than `T(n)` steps.

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity
  - The minimum among all inputs of length `n`. Best-case complexity
  - The average among all inputs of length `n`. Average-case complexity
- Best-case complexity is useless. For eg. it suggests that `elem` returns the answer in one step on any inputs of any length `n`.
- Worst-case is pessimistic. But it assures us that on any input of length `n` the program does NOT take more than `T(n)` steps.
- Average-case is perhaps a more accurate description. But is usually very difficult to calculate.

Efficiency ...

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`. Worst-case complexity
  - The minimum among all inputs of length `n`. Best-case complexity
  - The average among all inputs of length `n`. Average-case complexity
- Best-case complexity is useless. For eg. it suggests that `elem` returns the answer in one step on any inputs of any length `n`.
- Worst-case is pessimistic. But it assures us that on any input of length `n` the program does NOT take more than `T(n)` steps.
- Average-case is perhaps a more accurate description. But is usually very difficult to calculate.
- For the purposes of this course, we stick to worst-case analysis.

Measuring efficiency in Haskell ...

- What is the complexity of `reverse`?

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

Measuring efficiency in Haskell ...

- What is the complexity of `reverse`?

  ```
  reverse [] = []
  reverse (x:xs) = (reverse xs) ++ [x]
  ```

- Write a recurrence for $T(n)$

$$
\begin{aligned}
T(0) &= 1 \\
T(n) &= T(n-1) + n
\end{aligned}
$$

Measuring efficiency in Haskell ...

- What is the complexity of `reverse`?

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

- Write a recurrence for $T(n)$

$$
\begin{aligned}
T(0) &= 1 \\
T(n) &= T(n-1) + n
\end{aligned}
$$

- Expand and solve

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= (T(n-2) + n-1) + n \\
&= (T(n-3) + n-2) + n-1 + n \\
&= \cdots \\
&= T(0) + 1 + 2 + \cdots + n \\
&= 1 + 1 + 2 + \cdots + n \\
&= n(n+1)/2 + 1
\end{aligned}
$$

Measuring efficiency in Haskell . . .

- Can we calculate the `reverse` faster?

Measuring efficiency in Haskell . . .

- Can we calculate the `reverse` faster?
- Build up the reverse list one item at a time

Measuring efficiency in Haskell . . .

- Can we calculate the `reverse` faster?
- Build up the reverse list one item at a time
  - Inverting a stack of books into a second stack

Measuring efficiency in Haskell ...

- Can we calculate the `reverse` faster?
- Build up the reverse list one item at a time
  - Inverting a stack of books into a second stack

```
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

Measuring efficiency in Haskell ...

- Can we calculate the `reverse` faster?
- Build up the reverse list one item at a time
  - Inverting a stack of books into a second stack

```
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Clearly `transfer l1 l2 = (reverse l1)++l2`

Measuring efficiency in Haskell ...

- Can we calculate the `reverse` faster?
- Build up the reverse list one item at a time
  - Inverting a stack of books into a second stack

```
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Clearly `transfer l1 l2 = (reverse l1)++l2`
- Therefore `reverse l = transfer l []`

Measuring efficiency in Haskell . . .

- Can we calculate the `reverse` faster?
- Build up the reverse list one item at a time
  - Inverting a stack of books into a second stack

```
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Clearly `transfer l1 l2 = (reverse l1)++l2`
- Therefore `reverse l = transfer l []`
- For `transfer`, input size is `length l1`

$$\begin{aligned} T(0) &= 1 \\ T(n) &= T(n-1) + 1 \end{aligned}$$

- Thus $T(n) = n + 1$