Computer Science 312

Haskell Lists

List Literals

Similar to Python, but all items must be of the same type

```
Prelude> []
[]

Prelude> [1, 2, 3]
[1,2,3]

Prelude> [1..10]
{1,2,3,4,5,6,7,8,9,10]
```

Basic List Operations

`null list`	Returns **`true`** if **`list`** is empty or **`false`** otherwise.
`head list`	Returns the first item (at position 0) in **`list`**.
`tail list`	Returns a list of the items after the first item in **`list`**.
`length list`	Returns the number of items in **`list`**.
`item : list`	Returns a list whose head is **`item`** and whose tail is **`list`**.
`list1 ++ list2`	Returns a list containing the items in **`list1`** followed by the items in **`list2`**.
`list !! index`	Returns the item at position **`index`** in **`list`**. Positions are counted from 0.

Many other list operations are in the **`Data.List`** module

The Essential Operations

With these four operations, you can build almost all of the others

```
Prelude> numbers = [ 1..4]

Prelude> head numbers
1

Prelude> tail numbers
[2,3,4]

Prelude> 0:numbers
[0,1,2,3,4]

Prelude> numbers
[1,2,3,4]
```

The Essential Operations

With these four operations, you can build almost all of the others

```
Prelude> :type [1,2,3,4]
[1,2,3,4] :: Num t => [t]

Prelude> :type head
head :: [a] -> a

Prelude> :type tail
Tail :: [a] -> [a]

Prelude> :type (:)
(:) :: a -> [a] -> [a]
```

`head` and **`tail`** are ***selectors***, and **`(:)`** is a ***constructor***

A Recursive Definition of a List

A list is either

empty (that is, ***null*** is true)

or

an item (called the ***head***) followed by another list (called the ***tail***)

Recursive List Processing: Length

The length of a list is

- 0 if the list is empty
- 1 + the length of the tail of the list, otherwise

```
myLength :: [a] -> Int
myLength list
    | null list = 0
    | otherwise = 1 + myLength (tail list)
```

Recursive List Processing: *i*th Item

The *i*th item of a list is

- The list's head if i = 0
- The *i*th item at *i* - 1 in the list's tail, otherwise

```
-- Assumes that list is nonempty!
ith :: [a] -> Int -> a
ith list i
    | i == 0 = head list
    | otherwise = ith (tail list) (i - 1)
```

Recursive List Processing: remove *i*th

The remove *i*th of a list is

- The list's tail if $i = 0$
- The list's head, followed by the remove *i*th of the list's tail at $i - 1$

```
-- Assumes that list is nonempty!
removeIth :: Int -> [a] -> [a]
removeIth index list
    | index == 0 = tail list
    | otherwise = head list : removeIth (index – 1) (tail list)
```


Adds head of list to result of recursive call

Trace of Remove *i*th

The remove *i*th of a list is

- The list's tail if $i = 0$
- The list's head, followed by the remove *i*th of the list's tail at $i - 1$

```
removeIth 2 [20, 30, 40, 50] ->        -- index /= 0
    20 : removeIth 1 [30, 40, 50] ->   -- index /= 0
        30 : removeIth 0 [40, 50] ->   -- index == 0, tail == [50]
            <- [50]                    -- Returns [50]
        <- [30, 50]                    -- Returns 30:[50]
    <- [20, 30, 50]                    -- Returns 20:[30, 50]
```

Using `(:)` for Pattern Matching

```
Prelude> 1:[2,3,4] --Use (:) as constructor
[1,2,3,4]

Prelude> (x:xs) = [1,2,3,4] -- Use (:) as selector

Prelude> x
1

Prelude> xs
[2,3,4]
```

`x` extracts the head, and `xs` extracts the tail

Pattern Matching with Lists

```
myLength :: [a] -> Int
myLength list
    | null list = 0
    | otherwise = 1 + myLength (tail list)
```

```
myLength :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs
```

Back to pure clausal form for list processing functions!

Pattern Matching with Lists

```
-- Assumes that list is nonempty!
removeIth :: Int -> [a] -> [a]
removeIth index list
    | index == 0 = tail list
    | otherwise = head list : removeIth (index – 1) (tail list)
```

```
-- Assumes that list is nonempty!
removeIth :: Int -> [a] -> [a]
removeIth 0 (x:xs) = xs
removeIth i (x:xs) = x : removeIth (index – 1) xs
```

To Mutate or Not to Mutate

```
# Python list
>>> lyst = [1,2,3,4]

>>> lyst.pop(1)
2

>>> lyst
[1,3,4]
```

```
-- Haskell list
Prelude> list = [1,2,3,4]

Prelude> removeIth 1 list
[1,3,4]

Prelude> list
[1,2,3,4]
```

`removeIth` is a pure function, which transforms a list into another list

No Mutations, Can Share Structure

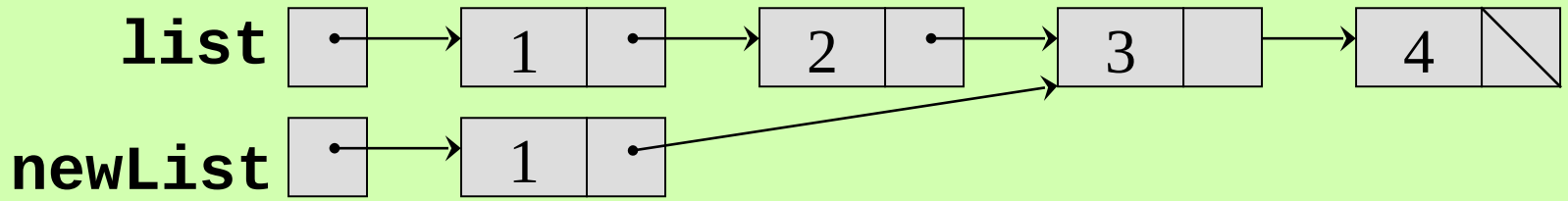
```
-- Assumes that list is nonempty!
removeIth :: Int -> [a] -> [a]
removeIth 0 (x:xs) = xs
removeIth i (x:xs) = x : removeIth (index – 1) xs
```

```
Prelude> list = [1,2,3,4]

Prelude> newList = removeIth 1 list

Prelude> newList
[1,3,4]

Prelude> list
[1,2,3,4]
```



One More List Operation: Insertion Sort

```
insert :: Ord a => a -> [a] ->[a]
insert item [] = [item]
insert item (x:xs)
    | item <= x = item : x : xs
    | otherwise = x : insert item xs
```

Sort the list's tail, and then insert the list's head into the result

```
insertionSort :: Ord a => [a] -> [a]
insertionSort [] = []
insertionSort (x:xs) = insert x (insertionSort xs)
```

String Literals and The String Type

Strings are just lists of characters

```
Prelude> "Hi there!"
"Hi there!"

Prelude> :type "Hi there!"          -- A list of characters
"Hi there!" :: [Char]

Prelude> putStrLn "Hi there!"       -- Like Python's print
Hi there!

Prelude> putStrLn "Hi\nthere!"
Hi
there!
```

For next time

Introduction to strings and tuples