Lists in Funnel

- Lists are sequences of values
- They are one of the most important data type for functional programming
- A lot of functional programming languages have lists as a built-in data type
- In Funnel, lists are not primitive and have to be encoded explicitly
- Funnel offers three ways for encoding compound data types
  - tupels
  - functions
  - records

Implementing Lists with Records

- Lists are represented as linked data structures
- We need two constructors:
  - `Nil`, for creating empty lists
  - `Cons(x, xs)`, for creating a list with head `x` and tail `xs`
- We want every list to have the functions `isEmpty`, `head` and `tail`:

```
def Nil = {
    def isEmpty = true
    def head = error "Nil.head"
    def tail = error "Nil.tail"
}
def Cons(x, xs) = {
    def isEmpty = false
    def head = x
    def tail = xs
}
```


Creating Lists

Here's a transcript of a *funny* session:

```
> val l = Cons(1, Cons(2, Cons(3, Nil)))
'val l = (<record id=10, adr=129, type=(isEmpty, head, tail)>)'
> (l.head, l.tail.head)
(1, 2)
> val l' = l.tail.tail
'val l' = (<record id=10, adr=129, type=(isEmpty, head, tail)>)'
> l'.head
3
> l'.tail.head
user abortion: "Nil.head"
```


Clients of the List Abstraction

Functions operating on lists use

- `isEmpty` for distinguishing empty lists from non-empty lists,and
- the projections `head` and `tail` for accessing the first element and the rest of the list for non-empty lists

Let's write a function `length` that computes the length of a given list:

```
def length(xs) = if (xs.isEmpty) 0 else 1 + length(xs.tail)
```

The implementation of `append`, which concatenates two lists is similar:

```
def append(xs, ys) = if (xs.isEmpty) ys
                     else Cons(xs.head, append(xs.tail, ys))
```

Example

Suppose we want to sort a list of numbers into ascending order:

- One way to sort the list `[7, 3, 9, 2]` is to sort the tail `[3, 9, 2]` to give `[2, 3, 9]`.
- It is then a matter of inserting the head `7` in the right place to give the result `[2, 3, 7, 9]`

This idea describes *Insertion Sort*:

```
def isort(xs) = if (xs.isEmpty) Nil
                else ins(xs.head, isort(xs.tail))
```

How does an implementation of the missing function `ins` look like?

Patterns of Computation

- The examples show that functions over lists often have similar structures
- We can identify several patterns of computation like
  - Transform every element of a list in some way
  - Combine the elements of a list using some operator
- Functional programming languages enable programmers to write general functions which implement patterns like this
- These functions are *higher-order functions* which get a transformation or an operator as one argument

Combining Lists

- We introduced already the function `append` for list concatenation
- Function `concat` concatenates all lists contained in another list:

  ```
  def concat(xss) = if (xss.isEmpty) Nil
                    else append(xss.head, concat(xss.tail))
  ```

  Example: `concat[[1,2], [], [3]] = [1, 2, 3]`
- `zip` combines two lists into a list of pairs:

  ```
  def zip(xs, ys) = if (xs.isEmpty || ys.isEmpty) Nil
                    else Cons((xs.head, ys.head),
                              zip(xs.tail, ys.tail))
  ```

  Example:

  `zip(["Frank", "Bill"], [1, 2]) = [("Frank", 1), ("Bill", 2)]`

Combining Lists

- A more general form of `zip` is function `zipwith`. It applies a function `f` to corresponding elements from two lists:

  ```
  def zipwith(f, xs, ys) = if (xs.isEmpty || ys.isEmpty) Nil
                           else Cons(f(xs.head, ys.head),
                                     zipWith(xs.tail, ys.tail))
  ```

- Example: you have a list of first names `fn` and a list of surnames `sn`. You can create a list of full names easily using `zipwith`:

  ```
  zipwith((first, last| first + " " + last), fn, sn)
  ```

Applying to All (mapping)

Many functions call for all of the elements of a list to be transformed in some way – this we call *mapping*.

Example: Suppose we have a list of tupels (Name, Age) and we want to convert this list into a list of names only:

```
def names(xs) = {
   if (xs.isEmpty) Nil
   else { val (name, age) = xs.head; Cons(name, names(xs.tail)) }
}
```

Instead of implementing this scheme with different transformations over and over again, we can write a *single* `map` function, which applies a function `f` to all elements of a list:

```
def map(f, xs) = if (xs.isEmpty) Nil
                 else Cons(f(xs.head), map(f, xs.tail))
```


Selecting Elements (filtering)

Selecting all the elements of a list with a given property is also common:

```
def odds(xs) = if (xs.isEmpty) Nil
               else if ((xs.head % 2) == 0) odds(xs.tail)
               else Cons(xs.head, odds(xs.tail))
```

The general function `filter` takes a property and a list and returns those elements of the list having the property.

Properties are modelled as predicates; i.e. functions over element types that return a boolean value.

```
def filter(p, xs) = if (xs.isEmpty) Nil
                    else if (!p(xs.head)) filter(p, xs.tail)
                    else Cons(xs.head, filter(p, xs.tail))
```

With `filter`, function `odds` can be rewritten in the following way:

```
def odds(xs) = filter((x| (x % 2) == 1), xs)
```


Combining Items (folding)

- Most list operations we saw return lists as their result
- The operation of *folding* an operator or function into a list of values is more general, since it can transform lists into other types
- There are two ways of folding a function into a list:

$$foldr(f, a, [x_1, x_2, \ldots, x_n]) = f(x_1, f(x_2, \ldots f(x_n, a)))$$
$$foldl(f, a, [x_1, x_2, \ldots, x_n]) = f(\ldots f(f(a, x_1), x_2), \ldots, x_n)$$

- Here's a Funnel implementation:

```
def foldr(f, a, xs) = if (xs.isEmpty) a
                      else f(xs.head, foldr(f, a, xs.tail))
def foldl(f, a, xs) = if (xs.isEmpty) a
                      else foldl(f, f(a, xs.head), xs.tail)
```


Applying Fold

Let's implement a function that calculates the sum of all numbers of a list using the `fold` combinator:

```
def sum(xs) = foldr((x, y| x + y), 0, xs)
```

Encoding the function `append` is simple as well:

```
def append(xs, ys) = foldr(Cons, ys, xs)
```

Is it possible to use `foldl` for both examples? What about efficiency?

A More Complicated Example

This is an obvious solution for reversing a list:

```
def reverse(xs) = if (xs.isEmpty) Nil
                  else append(reverse(xs.tail), Cons(xs.head, Nil))
```

Can you implement a more efficient version using the `fold` combinator?

Breaking up Lists

Another common pattern is to take or drop itemiz from a list while they have some property.

`take` returns the first `n` elements of a list, `drop` returns the list without the first `n` elements:

```
def take(n,xs) = if (n>0) Cons(xs.head, take(n–1, xs.tail)) else Nil
def drop(n,xs) = if (n>0) drop(n – 1, xs.tail) else xs
```

`takewhile` returns the longest prefix of a list, where every argument satisfies a predicate `p`:

```
def takewhile(p, xs) = if (xs.isEmpty || !p(xs.head)) Nil
                       else Cons(xs.head, takewhile(p, xs.tail))
```

`dropwhile` is implemented similarly.

Exercises

In functional programming languages, matrices are often implemented as lists of rows, where each row is itself a list of values. The matrix

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{pmatrix}$$

would be encoded like this:

$$[[x_{1,1}, x_{1,2}, x_{1,3}], [x_{2,1}, x_{2,2}, x_{2,3}]]$$

Implement the following functions operating on matrices and vectors:

- `scalarprod` computes the scalar product of two vectors
- `transpose` transposes a given matrix
- `add` adds two matrices
- `mult` multiplies two matrices

A list type is supplied, so all important list operations can be used.

Modules in Funnel

- To avoid name space conflicts, new data types are best implemented inside of a module.
- A module is a record consisting of all the constructors
- Example:

```
val List = {
  def Nil = {
    def isEmpty = true
    def head = error "Nil.head"
    def tail = error "Nil.tail"
  }
  def Cons = {
    def isEmpty = false
    def head = x
    def tail = xs
  }
}
```

Modules in Funnel

- Access to the constructors has to be qualified with `List`:

  ```
  List.Cons(1, List.Cons(2, List.Cons(3, List.Nil)))
  ```