ML & Data Abstraction

Overview

- Functions and Operators
- Lists
  - Pattern matching
- Data types
  - Tree
- Data abstraction through Modules
  - Structures
  - Signatures
  - Functors
- Higher order functions
- Infinite data structures

Operators and Functions

- Fact
  - ML introduce operators in the language for each operation we need (+,-,*,/, ...)
- Question
  - Is that the right way to deal with it ?

Operators and Functions

- Answer
  - That does not scale
  - We must change the language all the time
- Solution
  - Operators should be functions
    - Function *instance* are outside the language definition
    - We can add new ones without changing the core language
    - Better scalability
- Outstanding issue
  - Operators like "+" or "*" are used with infix notation
    - Operator appears in between the operations
  - Functions are prefixed
    - function name appears before the operands

Infix to Prefix

- Solutions
  - ML operators are not really operators...
  - ML operators are functions using infix notation

```
- op +;
val it = fn : int * int -> int
- val add = op +;
val add = fn : int * int -> int
- add(3,5);
val it = 8 : int
```

Prefix to Infix

- The other direction
  - Two keywords
  - Syntax

```
Declaration     ::=  infix  [<priority>]  <identifier>
                ::=  infixr [<priority>]  <identifier>
```

- Turns a function into either
  - A left associative or
  - A right associative operators
  - With a specific precedence

Defaults

- Priorities for builtin operators

```
infix 0 before
infix 3 o :=
infix 4 = <> > < >= <=
infixr 5 :: @
infix 6 + - ^
infix 7 * / div mod rem
```

Example

- Disjunction done manually

```
infix V;
fun false V a = a
    | true V a = true;

(* test program *)

4=5 V true;
val it = true : bool
```

Lists

- Builtin in most functional languages
- Use to represent a collection of elements
  - It is a container
  - It is typed
  - It is *polymorphic*
- Inductive definition
  - Base case
    - The empty list: nil, []
  - Inductive case
    - The list *constructor:* ::

List Example

- A list of integers

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16,
2002
- 3::nil;
val it = [3] : int list
- 2::3::nil;
val it = [2,3] : int list
- val x = [2,3];
val x = [2,3] : int list
- val y = 1::x;
val y = [1,2,3] : int list
```

- Questions
  - Is the "::" operator left or right associative ?

Polymorphism

- List can contain *anything*
- However
  - Content must be *homogeneous*
- List functions are polymorphic as well
- ML notation
  - Polymorphic types are written 'x
  - The place holder “x” stands for any type
- Lists are composite type
  - List of “something”
  - ML writes this type as
    - 'x list

Some predefined functions

- Get the first element
  - Function: hd: 'a list -> 'a
- Get the rest of the list
  - Function: tl: 'a list -> 'a list
- Check if the list is empty
  - Function: null : 'a list -> bool
- Concatenate two lists
  - Operator: op @: 'a list * 'a list -> 'a list

Exercise

- Write a function that computes the length of a list

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16,
2002
- fun length l = if l = nil then 0 else 1 + length (tl l);
stdIn:29.6 Warning: calling polyEqual
val length = fn : ''a list -> int

- fun length l = if null l then 0 else 1 + length (tl l);
val length = fn : 'a list -> int
```

Pattern Matching

- A wonderful way to improve function readability
- Remember that we can use pattern to
  - Match tuples
  - Match records
- We can also match with lists!

```
- val l = [2,3];
val l = [2,3] : int list

- val (a::b) = l;
val a = 2: int
val b = [3] : int list
```

Matching in Function

- Functions can use pattern matching
  - Program by case analysis
- Example

```
- fun length nil     = 0
  |   length (a::b) = 1 + length b;
val length = fn : 'a list -> int
```

Loops

- Is this necessary ?
  - No, not really
- Do everything with recursion

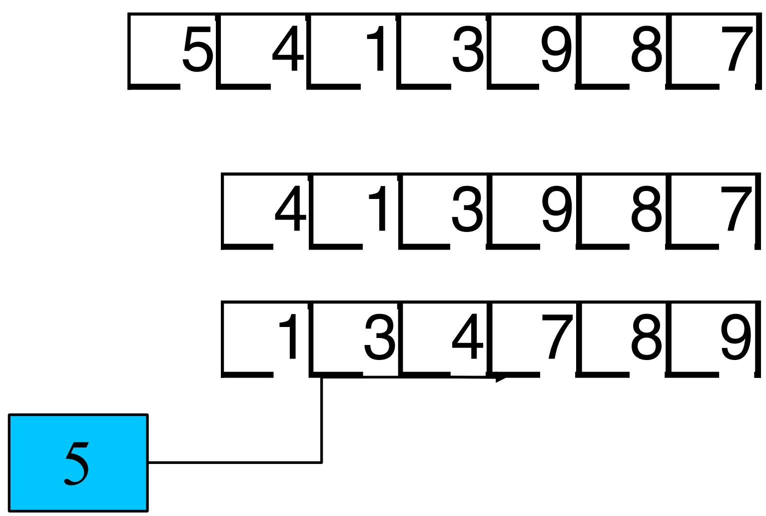
Example 1

- Compute the sum of all the elements in a list

```
fun sum nil      = 0
 |  sum (a::b) = a + sum b;
val sum = fn : int list -> int

sum [1,2,3];
val it = 6 : int
```

Example 2: Sorting

- Insertion Sort Algorithm
  - Reminder
    - Basic idea is to
      - Pick one element
      - Sort the remainder
      - Insert the picked element in the sorted remainder



Insertion Sort

- The Code

```
fun insert e nil = [e]
  | insert e (a::b) = if e>a
                       then a::(insert e b)
                       else e::a::b;

val insert = fn : int -> int list -> int list

fun iSort nil     = nil
  | iSort (a::b) = insert a (iSort b);

val iSort = fn : int list -> int list
```

Suggested Exercise

- Try Quicksort!
  - My version
    - 12 lines of ML
    - 5 lines of ML
    - …. depending on how hard I try ;-)

Overview

- Functions and Operators
- Lists
  - Pattern matching
- Data types
  - Tree
- Data abstraction through Modules
  - Structures
  - Signatures
  - Functors
- Higher order functions
- Infinite data structures

Data Types

- Objective
  - Improvement over records
  - Polymorphic types
- Motivating example
  - Manipulation of symbolic expression
    - Represent expression as trees
    - Each node can store different data
    - Each node is susceptible to a different treatment
    - Want to easily implement
      - Evaluation
      - Derivation
      - Integration

First Data type

- Let's start with a binary tree
  - We want
    - Node that store a piece of data
    - Node that stores references to two sub trees
    - Operations to
      - Add an element to a tree
      - Remove an element
      - Print the tree

```
- datatype  'a Tree = Leaf
                    | Node of 'a * 'a Tree * 'a Tree;

- Leaf;
val it = Leaf : 'a Tree
- Node(3,Leaf,Leaf);
val it = Node(3,Leaf,Leaf) : int Tree
```

Data types

- Note that
  - Data types have 1 or more constructors
  - Each constructor can store different information
    - Leaf stores nothing
    - Node stores a triplet
  - The datatype can be recursive
    - Fields of type 'a Tree
  - The datatype can be polymorphic
    - 'a Tree denotes a tree of 'a
    - Write code once *and reuse!*

Insertion in a Tree

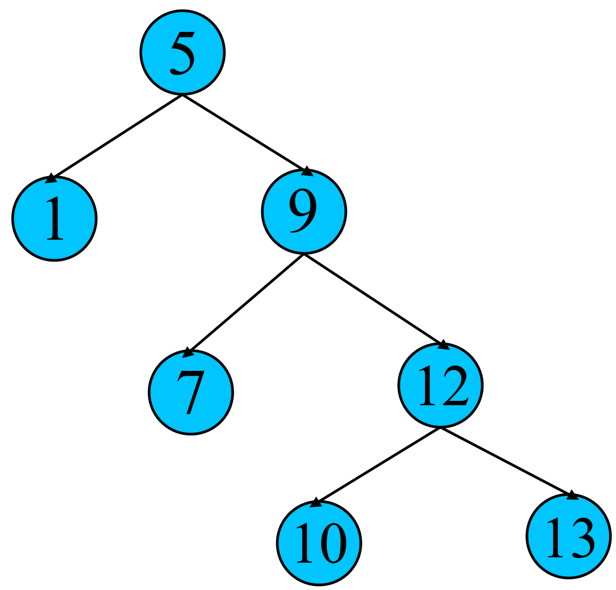
```
fun emptyTree () = Leaf;

fun addTree e Leaf              =  Node(e,Leaf,Leaf)
  | addTree e (Node(v,l,r)) = if (e < v)
                               then Node(v,addTree e l,r)
                               else if (e > v)
                                    then Node(v,l,addTree e r)
                                    else (print "oops!\n";
                                          Node(v,l,r));
```

- Function with no argument
  - Don't really exist.
  - Use the Unit value “()” instead
- Insertion
  - A new tree is constructed. The old one is untouched.
  - Sharing does happen

Functional Structures

- Key is
  - Create new structure when the old and new differ
  - Share what is untouched
- Benefits
  - Old copy is still usable
  - Side effect free

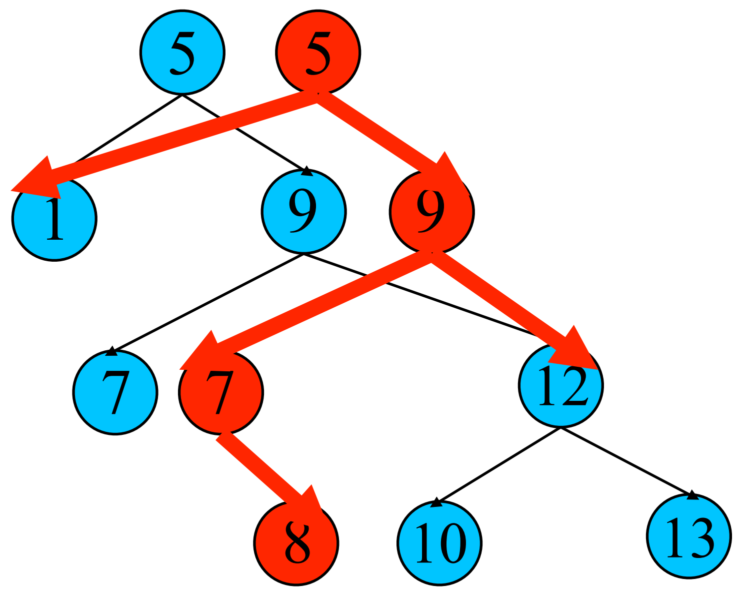
Insert(8)



Functional Structures

- Key is
  - Create new structure when the old and new differ
  - Share what is untouched
- Benefits
  - Old copy is still usable
  - Side effect free

Insert(8)



Error Handling

- Observation
  - The addTree code must deal with incorrect input
    - Attempt to add two pairs with the same key
    - What should we do ?
- Usual solution
  - Exceptions
    - Change the standard control flow
    - Exception are *raised* when error is detected
    - Exception are *handled* at the appropriate call site

Exception Example

- Raising an exception

```
Exception AlreadyPresent;

fun addTree e Leaf              =   Node(e,Leaf,Leaf)
  | addTree e (Node(v,l,r)) =   if (e < v)
                                then Node(v,addTree e l,r)
                                else if (e > v)
                                     then Node(v,l,addTree e r)
                                     else raise AlreadyPresent;
```

Exception Handling

- Add a clause to handle potential exception
  - Handler is selected based on pattern matching
- Syntax

```
<expr> handle <pattern1> => <expr1>
       |      <pattern2> => <expr2>
       | ...
       |      <patternn> => <exprn>
```

```
- addTree 1 (addTree 3 (addTree  2 (addTree 3 (leaf))))
  handle AlreadyPresent
        => (print "Attempt to add an element twice\n";Leaf);
```

Symbolic Expression

- What do we need ?
  - A Data type for the expression
    - One constructor for constant
      - Content:
    - One constructor for variables
      - Content:
    - One constructor for each arithmetic operator
      - Content:
  - A way to associate values with variables
    - Store

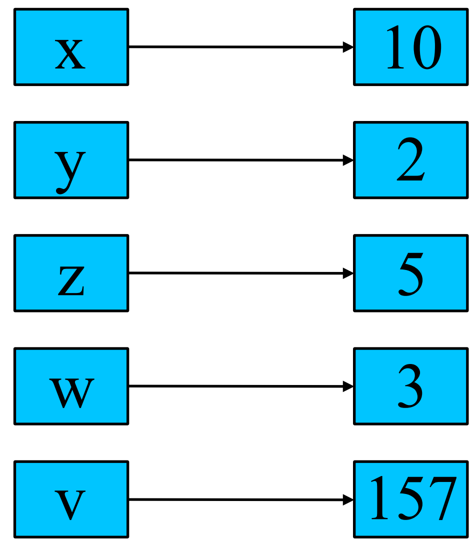
The Expression Data Type

```
- datatype  Expr = IntLit of int
                 | Var of string
                 | Plus of Expr * Expr
                 | Times of Expr * Expr
                 | Opposite of Expr;


val e0 = Plus(Var(“x”),Times(Var(“y”),IntLit(3)));
```

The Store I

- What is a store ?
  - An association between
    - A variable name
    - A value
  - Variable can be reassigned
  - Store offers permanence
- How to represent the store ?
  - First alternative
    - Suggestion ?

A Store



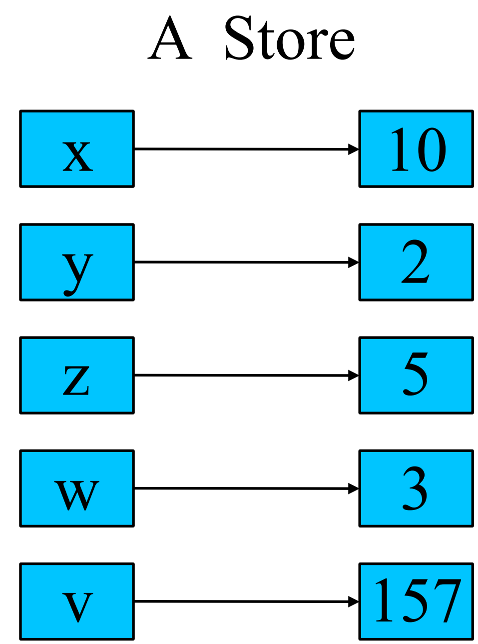
List Oriented Store

- Simple Idea
  - Store pair of string and values in a list
- Operations required
  - Create an empty store
    - empty list
  - Add a pair to a store
    - return a new list with new pair added
- Downside ?

```
val store = [("x",10),("y",5)];
exception InvalidVar;
fun makeStore () = nil;
fun bindStore x v s = ((x,v)::s);
fun findInStore x [] = raise InvalidVar
  | findInStore x ((a,v)::b) = if x=a then v else findInStore x b;
```

Functional Store

- The functional way
  - Take a look at the picture again
  - What is the device that transforms an input into an output ?



- Write a first class function generator!

```
exception InvalidVar;
fun makeStore () = fn x => raise InvalidVar;
fun bindStore x v s = fn y => if x=y then v else s y;
val store = bindStore “x”  10 (bindStore “y” 5 (makeStore ()));

val store = fn : string -> int
```

Expression Evaluation

- How to write a function that
  - Takes as input
    - An expression
    - A store
  - And evaluates the expression with respect to the store ?
- Use pattern matching

```
fun evalExpr (IntLit(c)) s = c
    | evalExpr  (Var(x))    s   = s x
    | evalExpr  (Plus(a,b)) s = (evalExpr a s) + (evalExpr b s)
    | evalExpr  (Times(a,b)) s = (evalExpr a s) * (evalExpr b s)
    | evalExpr  (Opposite(a)) s = ~ (evalExpr a s);
```

Symbolic Derivation

- Is a transformation from expression to expression
  - Use pattern matching

```
fun derive (IntLit c) x    = IntLit 0
  | derive (Var y)     x    = if x=y then IntLit 1 else IntLit 0
  | derive (Times(a,b)) x = Plus(Times(derive a x,b),

                                      Times(a,derive b x))
  | derive (Plus(a,b))  x = Plus(derive a x,derive b x)
  | derive (Opposite a) x = Opposite(derive a x);


  (* e0 = x + y * 3             *)

val e0 = Plus(Var("x"),Times(Var("y"),IntLit(3)));
val e1 = derive e0 "y";

  (*   e1 = 0 + 1 * 3 + y * 0       *)
```

Suggested Exercise

- Write a simplification routine that
  - Takes advantage of absorbing element 0 for mult.
  - Takes advantage of neutral element 1 for mult.
  - Takes advantage of neutral element 0 for addition.
- Icing on the cake
  - Also simplify terms like:
    - 3 * y * 5 to...
    - 15 * y

Overview

- Functions and Operators
- Lists
  - Pattern matching
- Data types
- Tree
- Data abstraction through Modules
  - Structures
  - Signatures
  - Functors
- Higher order functions
- Infinite data structures

Data Abstraction

- So far
  - The representation of data structures is exposed
    - Tuples
    - Records
    - Datatypes (including lists)
- What is desirable
  - A mechanism to
    - Group related functions into modules
    - Specify interfaces
  - A separation between
    - Interface **specification**
    - Interface **implementation**

Modules

- Objective
  - Group related functions, types and values
  - Form a “Data type” combining
    - State
    - Behavior
- Examples ?

ML And Modules

- ML offers a *structure*
  - Syntax

```
structure <Identifier> =
struct
    <declarations>
end;
```

- Usual issues
  - Name visibility
  - Scope
- Structure offers a *namespace*

Example

- Defining a structure for Stacks of anything

```
structure Stack =
struct
    exception EmptyStack;
    val empty       = nil;
    fun push x s    = (x::s);
    fun top nil     = raise EmptyStack
      | top (x::s) = x;
    fun pop nil     = raise EmptyStack
      | pop (x::s) = s;
end;
```

Example

- Using a structure for Stacks

```
structure Stack =
struct
    exception EmptyStack;
    val empty       = nil;
    fun push x s    = (x::s);
    fun top nil     = raise EmptyStack
      | top (x::s) = x;
    fun pop nil     = raise EmptyStack
      | pop (x::s) = s;
end;
val s = Stack.empty;
val s = [] : 'a list

val s = Stack.push “Hello” s;
val s = ["Hello"] : string list
```

The Achilles' heel

- We did not achieve data *abstraction*
- We only have *modularization*
  - Structures can be abused

```
val s = Stack.push “Hello” Stack.empty;
val s = ["Hello"] : string list

let val (h::t) = s
in  print (“Head is: ” ^ h ^ “\n”)
end;

      head is hello
      val it = () : unit
```

Why is this so bad ?

Interfaces

- First step
  - Disconnect the contract from the implementation
- ML solution
  - Signatures
    - A signature is a specification of the names and types that a structure should support if it wants to implement the interface.
    - A signature is the *type* of the structure
  - Syntax
    - Details of specs are in syntax charts (pp. 458-459)

```
signature <Identifier> =
sig
    <specifications>
end;
```

Example

- A Signature for Stacks

```
signature StackSig =
sig
    exception EmptyStack;
    val empty : 'a list;
    val push  : 'a -> 'a list -> 'a list;
    fun top   : 'a list -> 'a;
    fun pop   : 'a list -> 'a list;
end;
```

Signatures are Just Types!

- We need to
  - Declare a structure
  - Impose that the structure's type is the signature

```
signature StackSig =
sig
    exception EmptyStack;
    val empty : 'a list;
    val push  : 'a -> 'a list -> 'a list;
    fun top   : 'a list -> 'a;
    fun pop   : 'a list -> 'a list;
end;
structure MyStack : StackSig = Stack;
val s = MyStack.push “Hello” MyStack.empty;
```

Are We Done ?

- Does this solve the problem ?
- What we gained
- What remains

Abstracting Over Types

- We must disconnect
  - The type used in the implementation
  - From the type used in the signature
- Solution
  - Abstract over type and include type spec in signature

Type Abstraction

```
signature StackSig = sig
    exception EmptyStack;
    type 'a Stack;
    val empty : 'a Stack;
    val push  : 'a -> 'a Stack -> 'a Stack;
    fun top   : 'a Stack -> 'a;
    fun pop   : 'a Stack -> 'a Stack;
end;
structure Stack = struct
    exception EmptyStack;
    type 'a Stack   = 'a list;
    val empty       = nil;
    fun push x s    = (x::s);
    fun top nil     = (* as before *)
    fun pop nil     = (* as before *)
end;
```

Constraining Structures

- What is left
  - Declare a structure of the new type
    - Normal type constraint
    - Opaque type constraint

```
structure S1 :  StackSig = Stack;
structure S2 :> StackSig = Stack;

val s = S1.push "Hello" S1.empty;
val s = ["Hello"] : string Stack.Stack


val s = S2.push "Hello" S2.empty;
val s = - : string S2.Stack
```

Decoupling

- ML Solution
  - Functors
    - Functors *look* like functions over structures.
    - Functors
      - Take structures as argument
      - Produce a structure as ouptut
      - The output is a *specialized* structure

Example

- A functor to build a stack testing structure

```
functor StackTest (aStack : StackSig) =
struct
   fun pushAlot n s =
          if n=0
          then s
          else pushAlot (n-1) (aStack.push n s);
   fun testIt n = let val s = aStack.empty
                      in pushAlot n s
                  end
end;

structure Program = StackTest(Stack);
Program.testIt 10;
val it = [1,2,3,4,5,6,7,8,9,10] : int Stack.Stack
structure Program = StackTest(S2);
```

Overview

- Functions and Operators
- Lists
  - Pattern matching
- Data types
- Tree
- Data abstraction with
  - Structures
  - Signatures
  - Functors
- Higher order functions
- Infinite data structures

A Puzzle

- What is common to all this ?

```
fun sum nil     = 0
   |sum (a::b) = a + sum b;

fun prod nil    = 1
   |prod (a::b)= a * prod b;

fun rev nil l   = l
   |rev (a::b) l = rev b (a::l);

fun member x nil = false
   |member x (a::b) = x=a orelse member x b;
```

Commonality

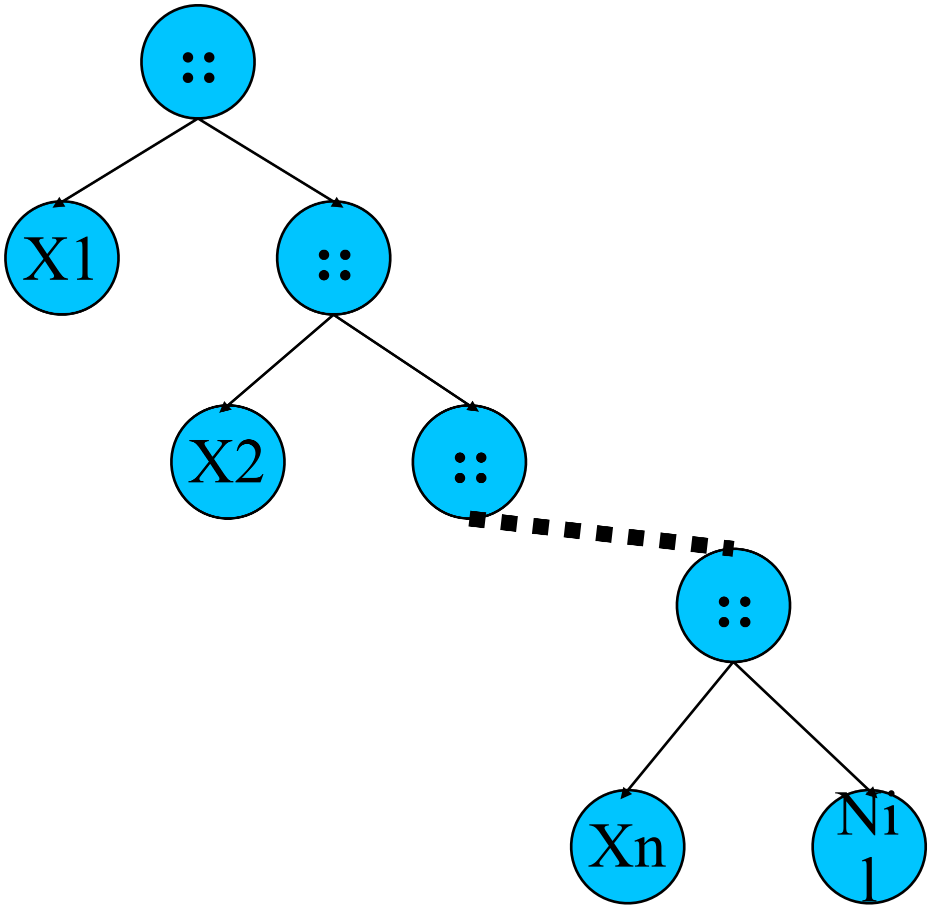
- All routines are
  - Induction on some list structure
  - Compute a new value
  - New value is function of the whole structure.
- Why reinvent the wheel ?
  - Write the inductive code once and for all
  - Apply the inductive code to a combination function
- This *scheme* is a higher order function
  - It needs
    - A base case for the induction
    - A combination function
    - A list to apply it on.

Folding

- Abstract view

$$foldr(f, e, [x_1, x_2, \cdots, x_n]) = f(x_1, f(x_2, \cdots, f(x_n, e)) \cdots)$$

$$foldl(f, e, [x_1, x_2, \cdots, x_n]) = f(x_n, f(x_{n-1}, \cdots, f(x_1, e)) \cdots)$$

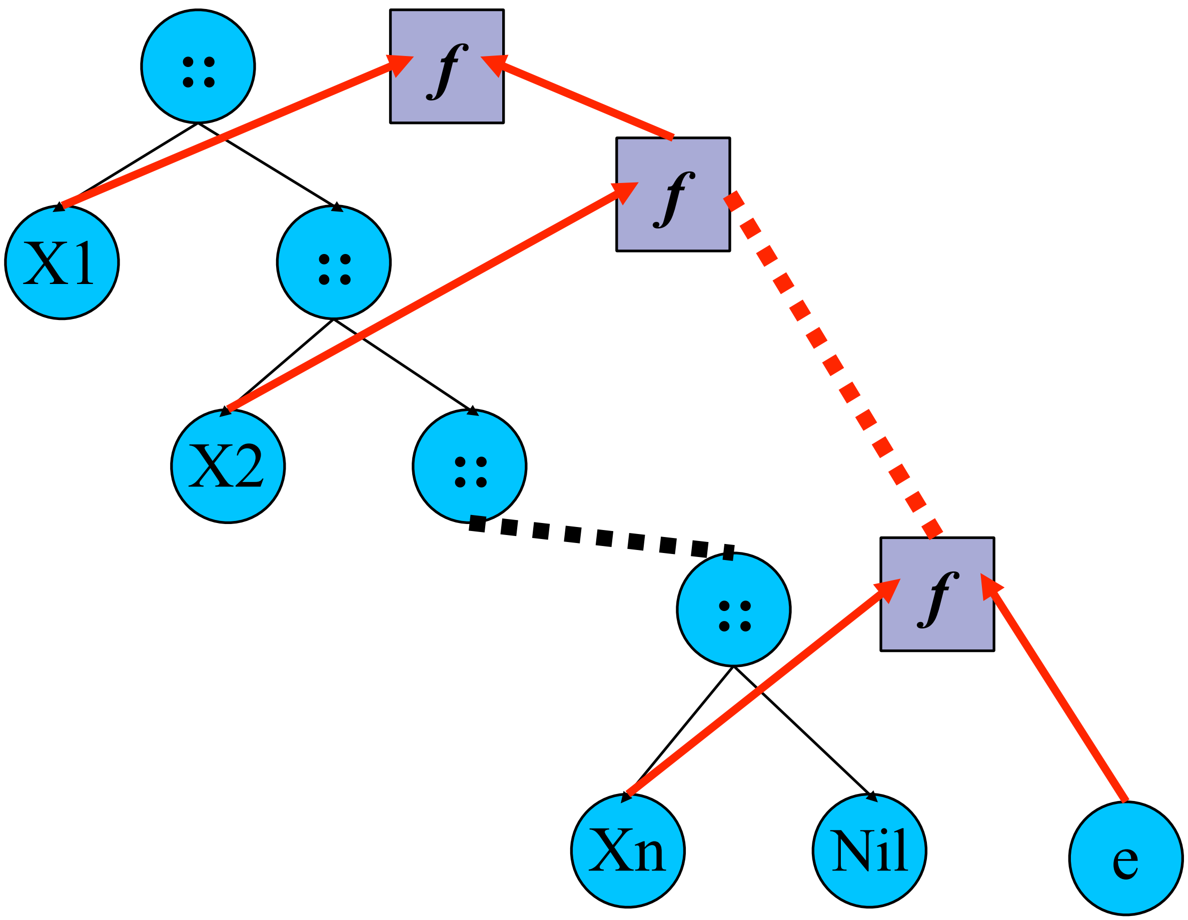


Folding

- Abstract view

$$foldr(f, e, [x_1, x_2, \cdots, x_n]) = f(x_1, f(x_2, \cdots, f(x_n, e))\cdots)$$

$$foldl(f, e, [x_1, x_2, \cdots, x_n]) = f(x_n, f(x_{n-1}, \cdots, f(x_1, e))\cdots)$$



Folding in ML

- And here is the code

```
fun foldr f e nil     = e
   |foldr f e (a::b) = f a (foldr f e b);

fun foldl f e nil     = e
   |foldl f e (a::b) = foldl f (f e a) b;
```

Revisiting Sum

- How to express sum with folding...

```
(* Take 1 *)
fun sum l = foldr (op +) 0 l;

(* Take 2 *)
fun sum l = foldl (op +) 0 l;
```

- What is the difference ?

Another One

- What are these two functions doing ?

```
(* Take 1 *)
fun ??? l = foldr (op ::) nil l;

(* Take 2 *)
fun ??? l = foldl (op ::) nil l;
```

Other Higher Order Functions

- Mappings
  - Takes
    - A list of 'a
    - A function from 'a -> 'b
  - Produces
    - A list of 'b
- Example
  - Convert a string to uppercase
    - A string is a list of characters
      - Convert with convenience functions explode/implode
    - Need a function to turn a character into its uppercase.

More Examples

```
fun map f nil     = nil
   |map f (a::b) = (f a)::(map f b);

fun toUpper s =
     implode (map Char.toUpper (explode s));
```

More Higher Order

- Filtering a list
  - Takes
    - A predicate function *p*: 'a -> bool
    - A list of 'a
  - Returns
    - A list of 'a that satisfy *p*
- Example

```
fun filter p nil = nil
   |filter p (a::b) = if p a
                      then a::(filter p b)
                      else filter p b;

fun odd l = filter (fn x => x mod 2=1) l;
```

Overview

- Functions and Operators
- Lists
  - Pattern matching
- Data types
- Tree
- Data abstraction with
  - Structures
  - Signatures
  - Functors
- Higher order functions
- Infinite data structures

Infinite structures

- Often called streams or sequence
- Example
  - The natural numbers
    - [0,1,2,3,4,5,6,7,....
- Objective
  - Represent the infinite sequence so that
    - It produces values only when needed
    - It can be used like finite sequences (list)
      - We want higher order functions on infinite streams

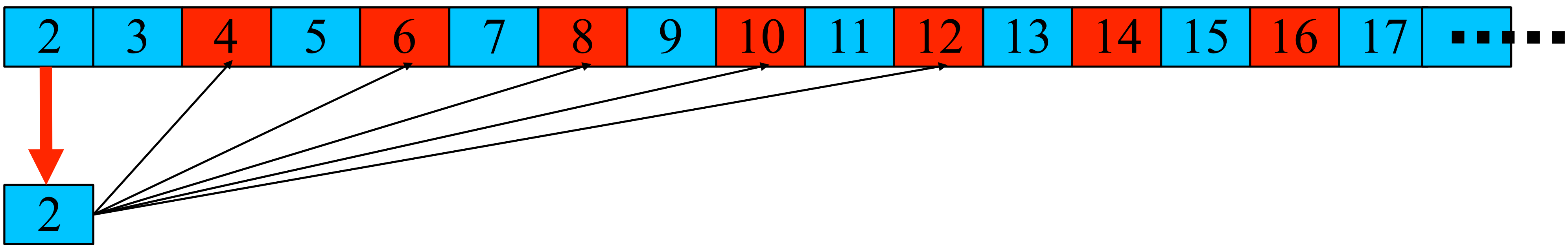
Motivating Example

- Computing primes
  - With the Sieves of Eratosthenes
- Eratosthenes algorithm

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	......

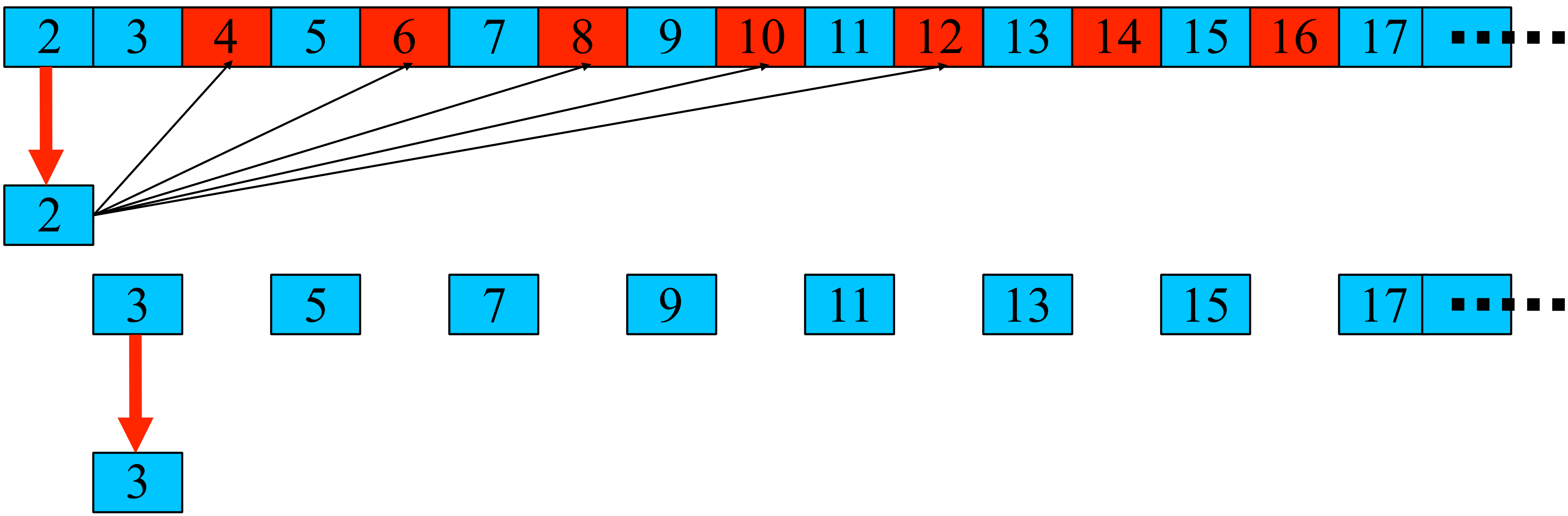
Motivating Example

- Computing primes
  - With the Sieves of Eratosthenes
- Eratosthenes algorithm



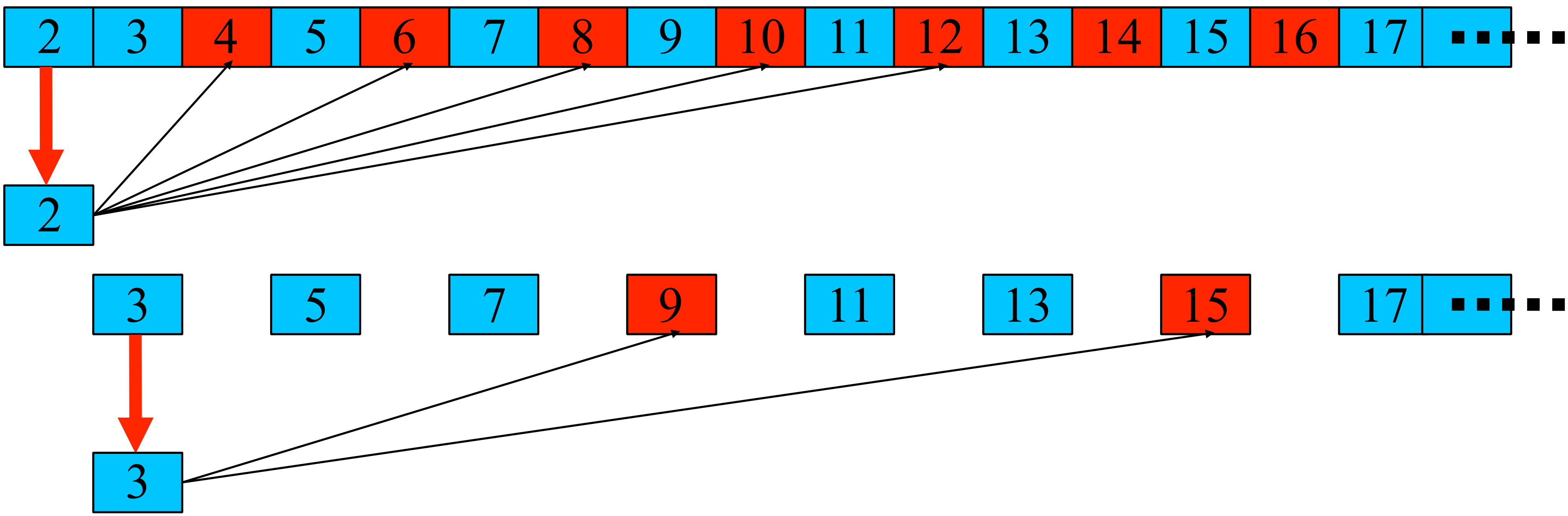
Motivating Example

- Computing primes
  - With the Sieves of Eratosthenes
- Eratosthenes algorithm



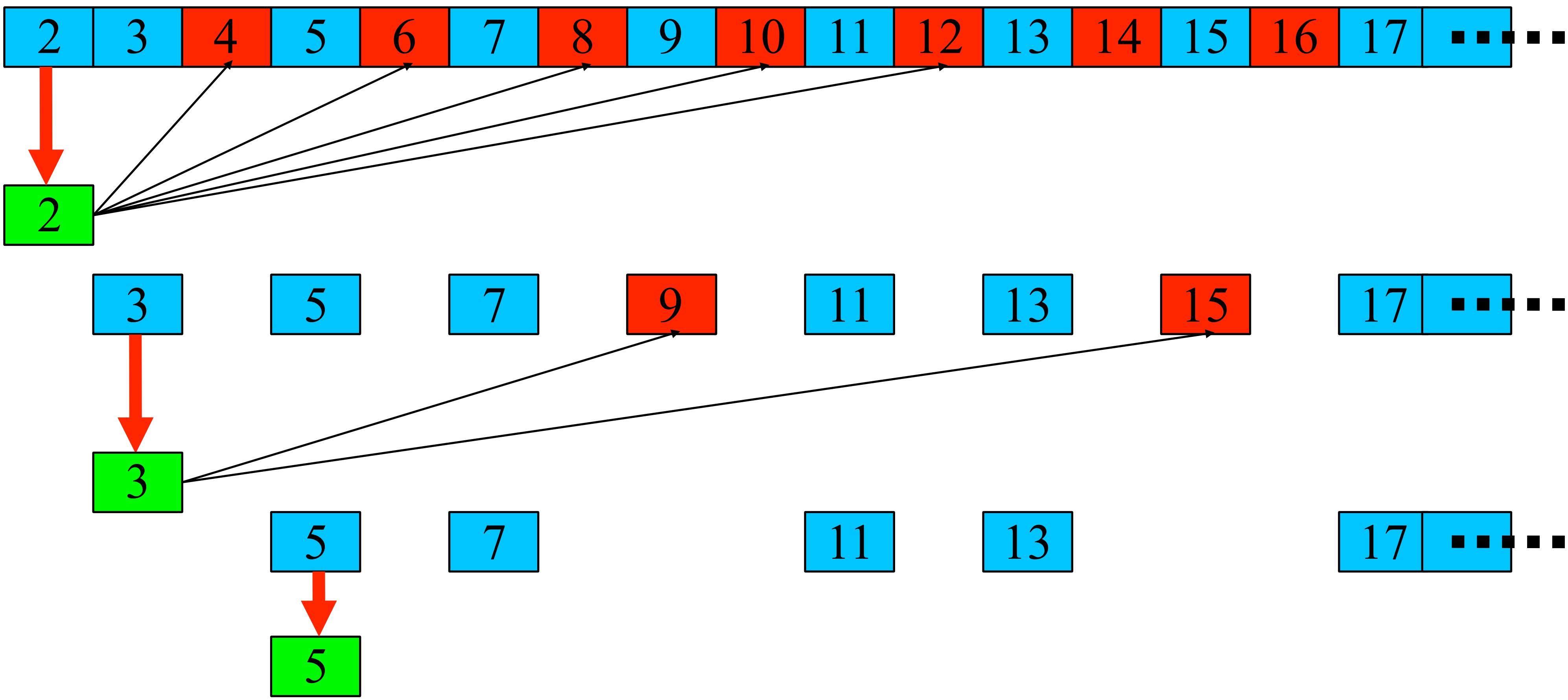
Motivating Example

- Computing primes
  - With the Sieves of Eratosthenes
- Eratosthenes algorithm



Motivating Example

- Computing primes
  - With the Sieves of Eratosthenes
- Eratosthenes algorithm



In A Nutshell

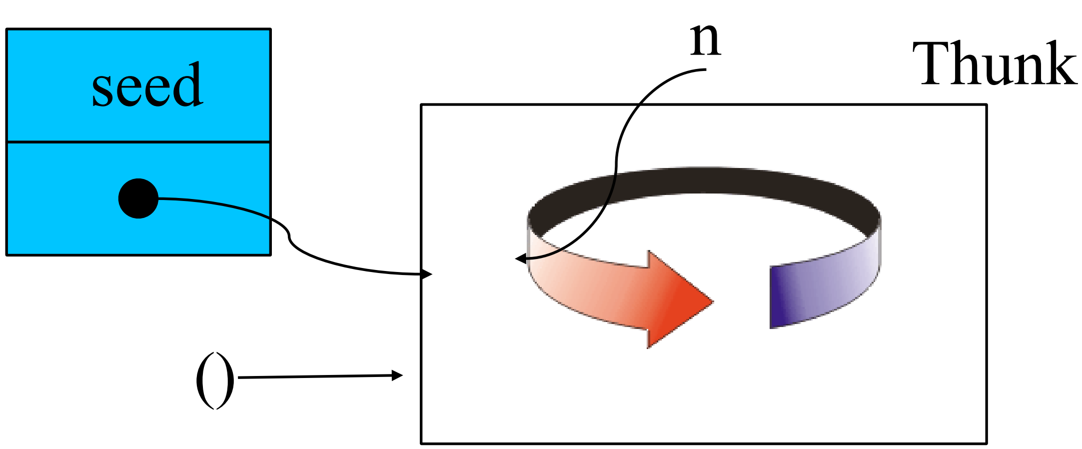
- The primes are
  - The first elements of a stream of streams.
- How to turn this into a runnable algorithm ?
  - Use a stream to represent the first sequence
  - Use a higher order function to filter the stream based on its first element
  - The first element is a prime
  - The filtered stream is the input of the next round

First Step

- First Issue
  - Represent a stream
- Solution
  - Use a data type with two constructors
    - One for the empty stream
    - The other To build a stream out of
      - A root element
      - A higher order function that will build the tail on demand
  - Provide functions to
    - Seed the stream
    - Pick up its first element
    - Pick up a leading sub sequence

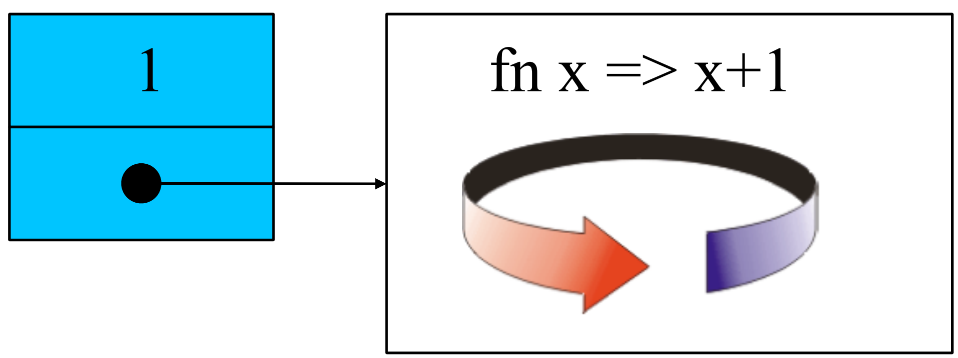
Sequences

```
datatype 'a S = Empty
              | Cons of 'a * (unit-> 'a S);
exception Error;
fun from s n = Cons(s,fn()=>from (n s) n);
```



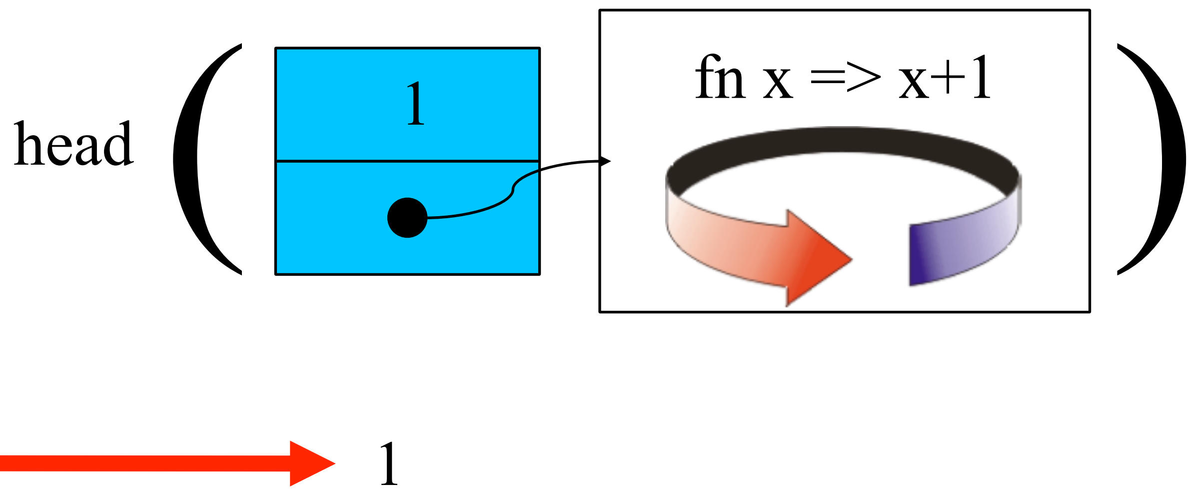
Sequences

```
datatype 'a S = Empty
              | Cons of 'a * (unit-> 'a S);
exception Error;
fun from s n = Cons(s,fn()=>from (n s) n);

val z = from 1 (fn x=> x+1);
```



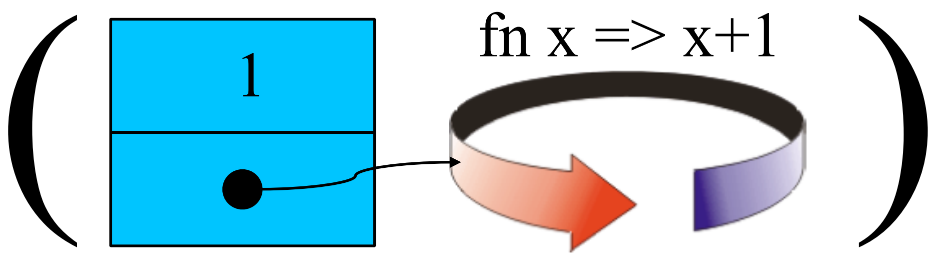
Picking up the head

```
datatype 'a S = Empty
              | Cons of 'a * (unit-> 'a S);
exception Error;
fun from s n = Cons(s,fn()=>from (n s) n);
fun head Empty          = raise Error
   |head (Cons(a,b)) = a;
```



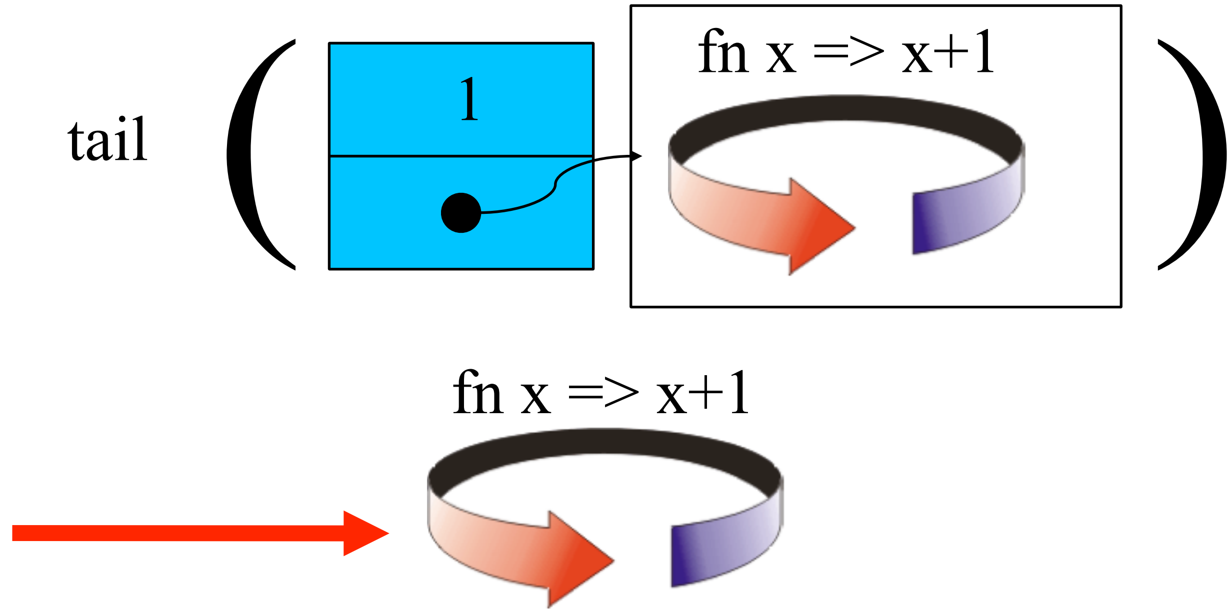
Picking up the tail ?

tail

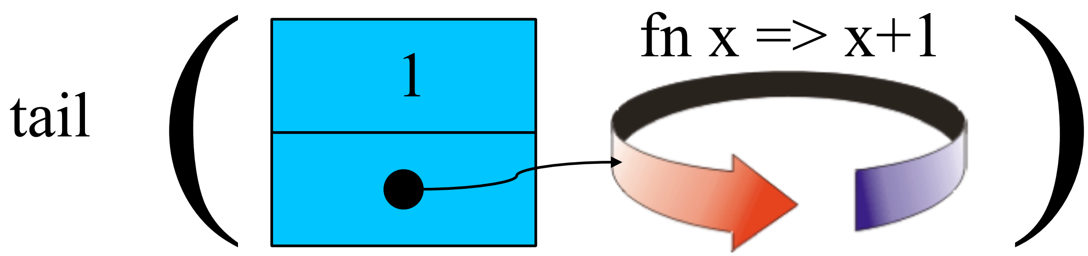


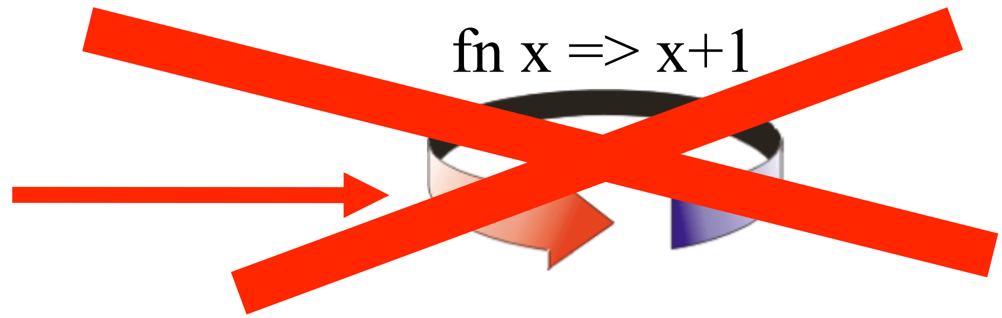
???????

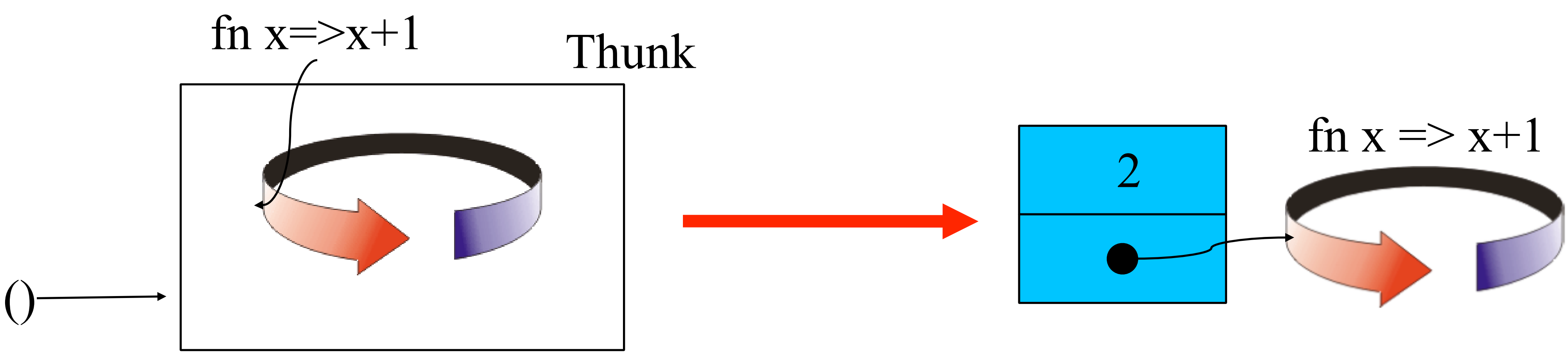
Picking up the tail ?



Picking up the tail ?







Picking up the tail

```
datatype 'a S = Empty
              | Cons of 'a * (unit-> 'a S);
exception Error;
fun from s n = Cons(s,fn()=>from (n s) n);
fun head Empty        = raise Error
   |head (Cons(a,b)) = a;
fun tail Empty        = raise Error
   |tail (Cons(a,b)) = b();
```

Fixed Length Prefix

- Objective
  - Find a prefix of length *n*

```
datatype 'a S = Empty
              | Cons of 'a * (unit-> 'a S);
exception Error;
fun from s n = Cons(s,fn()=>from (n s) n);
fun head Empty        = raise Error
   |head (Cons(a,b)) = a;
fun tail Empty        = raise Error
   |tail (Cons(a,b)) = b();
fun take n Empty       = raise Error
   |take 0 l           = nil
   |take n (Cons(a,b))= a::(take (n-1) b())
```

Higher Order and Streams

- We still need to filter a stream
  - We need a function that
    - Takes a stream and a predicate as input
    - Returns a filtered stream

```
datatype 'a S = Empty
              | Cons of 'a * (unit-> 'a S);
exception Error;

fun from s n = Cons(s,fn()=>from (n s) n);

fun filter Empty p        = Empty
   |filter (Cons(a,b)) p = if p a
                           then Cons(a,fn()=>filter (b()) p)
                           else filter (b()) p;
```

Eratosthenes Algorithm

- Three ingredients
  - The Natural stream
  - A function to filter a stream with a value
  - A function to filter the primes

```
val nat      = from 2 (fn x => x+1);
fun sift s v = filter s (fn x=>x mod v<>0);
fun sieve s  = let    val h = head s
                      val t = tail s

        inCons(h,fn()=>sieve (sift t h))
              end;
val tenP   = take 10 (sieve nat);
val tenP   = [2,3,5,7,11,13,17,19,23,29]:int list
```