Programming Languages and Techniques (CIS120)

Lecture 3

August 31st, 2015

Lists and Recursion

Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Tuesday 9/8 at midnight
- Clickers: attendance grades start Friday
- Reading: Please read Chapter 3 of the course notes, available from the course web pages
  - And chapters 1 and 2, if you haven't yet!
- Questions?
  - Post to Piazza (privately if you need to include code!)
- TA office hours: on course Calendar webpage

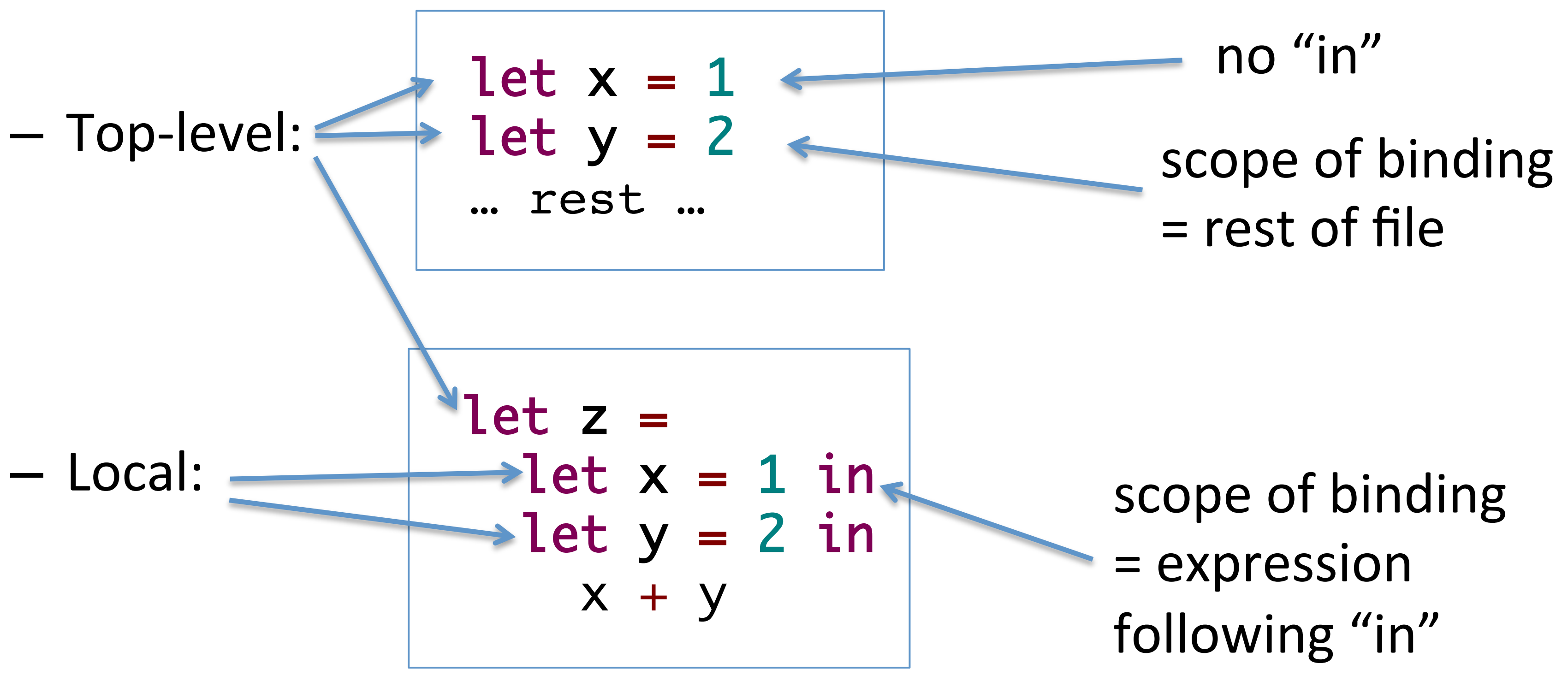
Have you started working on HW 01?

1) Yes

2) No

Summary of 'let' Syntax

- OCaml offers two forms of 'let' declarations:



– Top-level:

```
let x = 1
let y = 2
… rest …
```

no "in"

scope of binding
= rest of file

– Local:

```
let z =
  let x = 1 in
  let y = 2 in
    x + y
```

scope of binding
= expression
following "in"

Summary of ‘let’ Syntax

- Each let-binding declares a name for either a *value*…

```
let x = 2 + 4 in
    x + x
```

or a *function*…

```
let f (y:int) : int = y + y  in
  f 2
```

Note that local function bindings are also allowed (as illustrated here)!

What is the value computed for 'answer' in the following program? (0 .. 9)

```
let answer : int =
  let x = 1 in
  let y = x + x in
  x + y
```

```
let answer : int =
  let y = 1 + 1 in
  1 + y
```

```
let answer : int =
  let y = 2 in
  1 + y
```

```
let answer : int =
  1 + 2
```

```
let answer : int =
  3
```

What is the value computed for ‘answer’ in the following program? (0 .. 9)

```
let answer : int =
  let x = 3 in
  let f (y : int) = y + x in
  let x = 1 in
  f x
```

Answer: 4

Commands

Commands

```
;; run_test "Attendees at $5.00" test

;; print_endline "Attendees at $5.00"
;; print_int (attendees 500)
```

- Top-level commands run tests and print to the console
  - They affect the state of the machine but do not yield useful values
- Such commands are the *only* places that semicolons should appear in your programs (so far)
  - Many languages use ';' as statement terminators…not OCaml

A Design Problem / Situation

Suppose we are asked by Penn to design a new email system for notifying instructors and students of emergencies or unusual events.

What should we be able to do with this system?

Subscribe students to the list, query the size of the list, check if a particular email is enrolled, compose messages for all the list, filter the list to just students, etc.

Design Pattern

1. Understand the problem

   What are the relevant concepts and how do they relate?

2. Formalize the interface

   How should the program interact with its environment?

3. Write test cases

   How does the program behave on typical inputs? On unusual ones? On erroneous ones?

4. Implement the behavior

   Often by decomposing the problem into simpler ones and applying the same recipe to each

1. Understand the problem

How do we store and query information about email addresses?

Important concepts are:

1. An email list (collection of email addresses)
2. A fixed collection of *instructor_emails*
3. Being able to *subscribe* students & instructors to the list
4. Counting the *number_of_emails* in a list
5. Determining whether a list *contains* a particular address
6. Given a message to send, *compose* messages for all the email addresses in the list
7. *remove_instructors*, leaving an email list just containing the list of enrolled students

2. Formalize the interface

- Represent an email by a *string* (the email address itself)
- Represent an email list using an *immutable list of strings*
- Represent the collection of instructor emails using a *toplevel definition*

```
let instructor_emails : string list = …
```

- Define the interface to the functions:

```
let subscribe (email : string)
              (l : string list) : string list = …

let length (l : string list) : int = …

let contains (l : string list) (email : string) : bool = …
```

3. Write test cases

```
let l1 : string list = [ “stevez@cis.upenn.edu”;
                         “lankas@seas.upenn.edu”;
                         “maxmcc@sas.upenn.edu” ]
let l2 : string list = [ “lankas@seas.upenn.edu” ]
let l3 : string list = []

let test () : bool =
  (length l1) = 3
;; run_test “length l1” test

let test () : bool =
  (length l2) = 1
;; run_test “length l2” test

let test () : bool =
  (length l3) = 0
;; run_test “length p3” test
```

Define email lists for testing. Include a variety of lists of different sizes and incl. some instructor and non-instructor emails as well.

Interactive Interlude

email.ml

Lists

A Value-Oriented Approach

What is a list?

A list value is either:

`[]`	the *empty* list, sometimes called *nil*

or

`v :: tail`	a *head* value v, followed by a list of the remaining elements, the *tail*

- Here, the '`::`' operator *constructs* a new list from a head element and a shorter list.
  - This operator is pronounced "cons" (for "construct")
- Importantly, *there are no other kinds of lists*.
- Lists are an example of an *inductive datatype*.

Example Lists

To build a list, cons together elements, ending with the empty list:

`1::2::3::4::[]`	a list of four numbers
`“abc”::”xyz”::[]`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

Convenient List Syntax

Much simpler notation: enclose a list of elements in `[` and `]` separated by `;`

`[1;2;3;4]`	a list of four numbers
`[“abc”;”xyz”]`	a list of two strings
`[true]`	a list of one boolean
`[]`	the empty list

Calculating With Lists

- Calculating with lists is just as easy as calculating with arithmetic expressions:

(2+3)::(12 / 5)::[]

$\longmapsto$ 5::(12 / 5)::[] because 2+3 $\Rightarrow$ 5

$\longmapsto$ 5::2::[] because 12/5 $\Rightarrow$ 2

A list is a value whenever all of its elements are values.

List Types*

The type of lists of integers is written

```
int list
```

The type of lists of strings is written

```
string list
```

The type of lists of booleans is written

```
bool list
```

The type of lists of lists of strings is written

```
(string list) list
```

etc.

*Note that lists in OCaml are *homogeneous* – all of the list elements must be of the same type. If you try to create a list like [1; “hello”; 3; true] you will get a type error.

Clickers, please…

Which of the following expressions has the type

`int list ?`

```
1) [3; true]

2) [1;2;3]::[1;2]

3) []::[1;2]::[]

4) (1::2)::(3::4)::[]

5) [1;2;3;4]
```

Answer: 5

Which of the following expressions has the type
`(int list) list  ?`

```
1) [3; true]

2) [1;2;3]::[1;2]

3) []::[1;2]::[]

4) (1::2)::(3::4)::[]

5) [1;2;3;4]
```

Answer: 3

What can we do with lists?

What operations can we do on lists?

1. Access the elements
2. Create new lists by adding an element
3. Calculate its length
4. Search the list
5. Transform the list
6. Filter the list
7. …

Value oriented programming:

We can *name* the sub-components of a list.

We can construct new values using those names.

Pattern Matching

OCaml provides a single expression called *pattern matching* for inspecting a list and naming its subcomponents.

```
let mylist : int list = [1; 2; 3; 5]

let y : int =
  begin match mylist with
  | [] -> 42
  | first::rest -> first+10
  end
```

case branches

match expression syntax is:

```
begin match … with
  | … -> …
  | … -> …
end
```

Case analysis is justified because there are only *two* shapes a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch

- `first` names the head of the list; its type is the element type.
- `rest` names the tail of the list; its type is the list type

The type of the match expression is the (one) type shared by its braches.

Calculating with Matches

- Consider how to run a match expression:

```
begin match [1;2;3] with
   | [] -> 42
   | first::rest -> first + 10
end
```

$\longmapsto$

1 + 10

$\longmapsto$

11

Note: `[1;2;3]` equals `1::(2::(3::[]))`

It doesn't match the pattern [] so the first branch is skipped, but it *does* match the pattern `first::rest` when `first` is 1 and `rest` is `(2::(3::[]))`.
So, substitute 1 for `first` in the second branch.

The Inductive Nature of Lists

A list value is either:

`[]` the *empty* list, sometimes called *nil*

or

`v :: tail` a *head* value v, followed by a list of the remaining elements, the *tail*

- Why is this well-defined? The definition of list mentions 'list'!
- Solution: 'list' is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of (1+n) elements, add a head element to an *existing* list of n elements
  - The set of list values contains all and only values constructed this way
- Corresponding computation principle: *recursion*

Recursion

Recursion principle:

Compute a function value for a given input by combining the results for strictly smaller subcomponents of the input.

– The structure of the computation follows the inductive structure of the input.

- Example:

```
length 1::2::3::[]  =  1 + (length 2::3::[])
length 2::3::[]     =  1 + (length 3::[])
length 3::[]        =  1 + (length [])
length []           =  0
```

Recursion Over Lists in Code

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty. *Pattern matching* determines which.

```
let rec length (l : string list) : int =
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

If the list is non-empty, then "x" is the first string in the list and "rest" is the remainder of the list.

Patterns specify the **structure** of the value and (optionally) give **names** to parts of it.

Calculating with Recursion

```
    length ["a"; "b"]
↦       (substitute the list for l in the function body)
         begin match "a"::"b"::[] with
         | [] -> 0
         | ( x :: rest ) -> 1 + length rest
         end
↦       (second case matches with rest = "b"::[])
    1 + (length "b"::[])
↦       (substitute the list for l in the function body)
    1 + (begin match "b"::[] with
           | [] -> 0
           | ( x :: rest ) -> 1 + length rest
           end )
↦       (second case matches again, with rest = [])
    1 + (1 + length [])
↦       (substitute [] for l in the function body)
    …
↦   1 + 1 + 0 ⇒ 2
```

```
let rec length (l:string list) : int=
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec length (l : string list) : int =
  begin match l with
  | [] -> 0
  | ( x :: rest ) -> 1 + length rest
  end
```

```
let rec contains (l:string list) (s:string) : bool =
  begin match l with
  | [] -> false
  | ( x :: rest ) -> s = x || contains rest s
  end
```

Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

The branch for `[ ]` calculates the value (`f [ ]`) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates
(`f(hd::rest)`) given `hd` and (`f rest`).

- this is the *inductive case* of the recursion

Design Pattern for Recursion

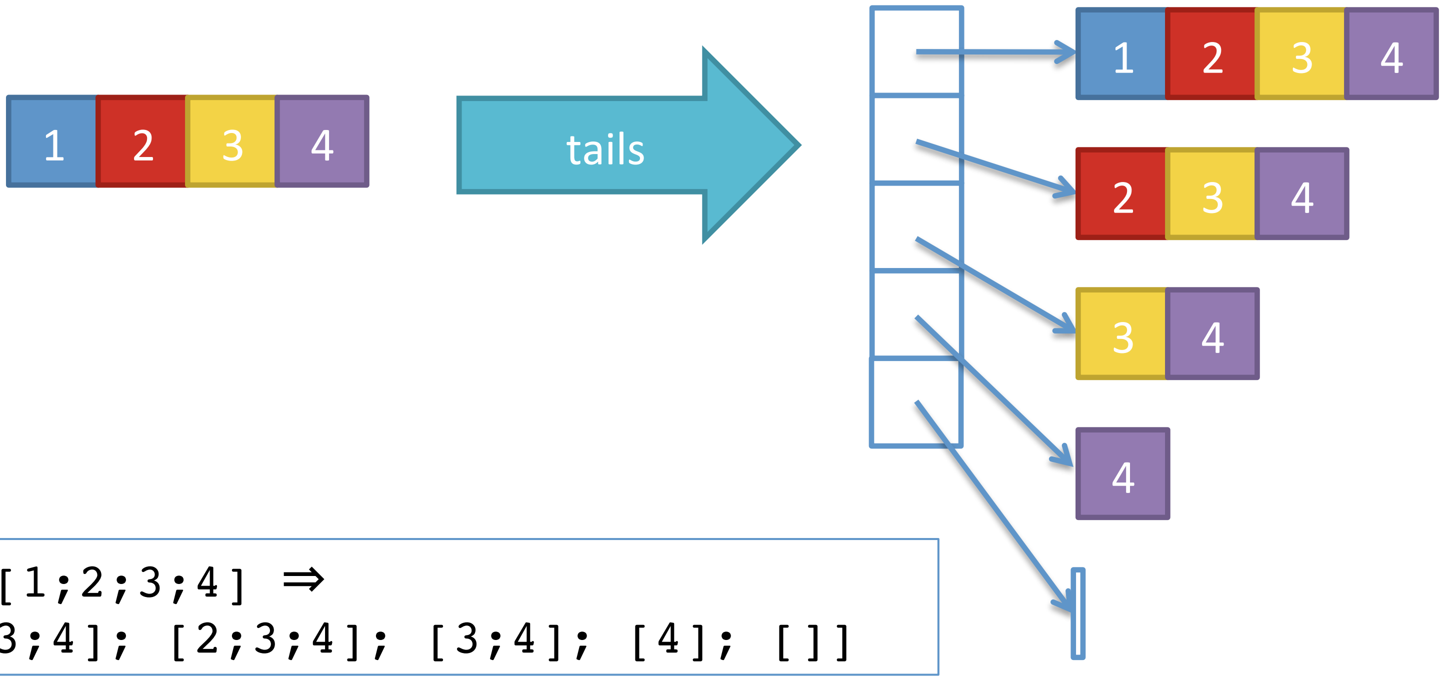
1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
   - If the main input to the program is an immutable list, look for a recursive solution…
     - Is there a direct solution for the empty list?
     - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?

Interactive Interlude

email.ml

tails

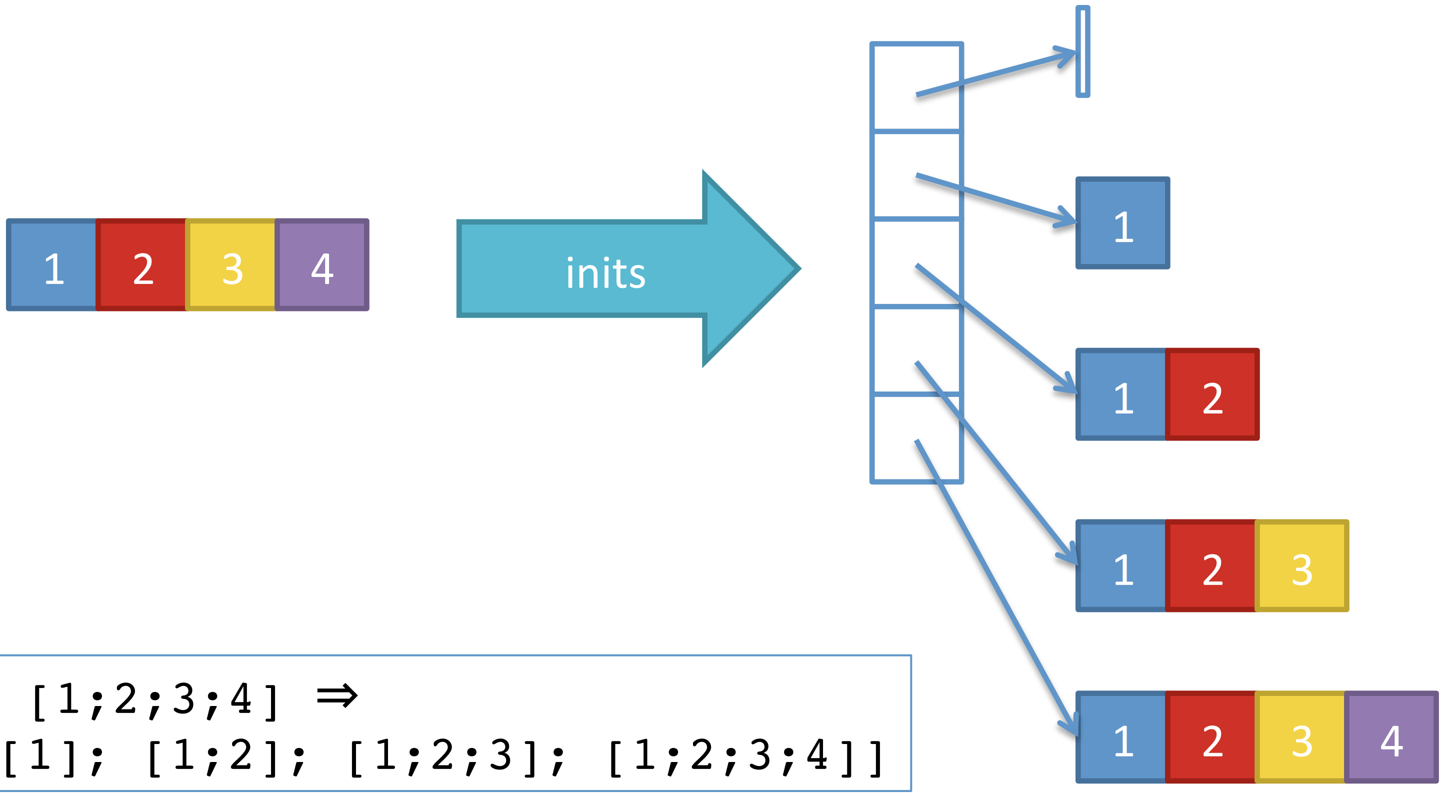
- Design problem: Given a list of integers, produce all suffixes of a given list, starting with the full list and removing the first element at each step



```
tails [1;2;3;4] ⇒
[[1;2;3;4]; [2;3;4]; [3;4]; [4]; []]
```

inits

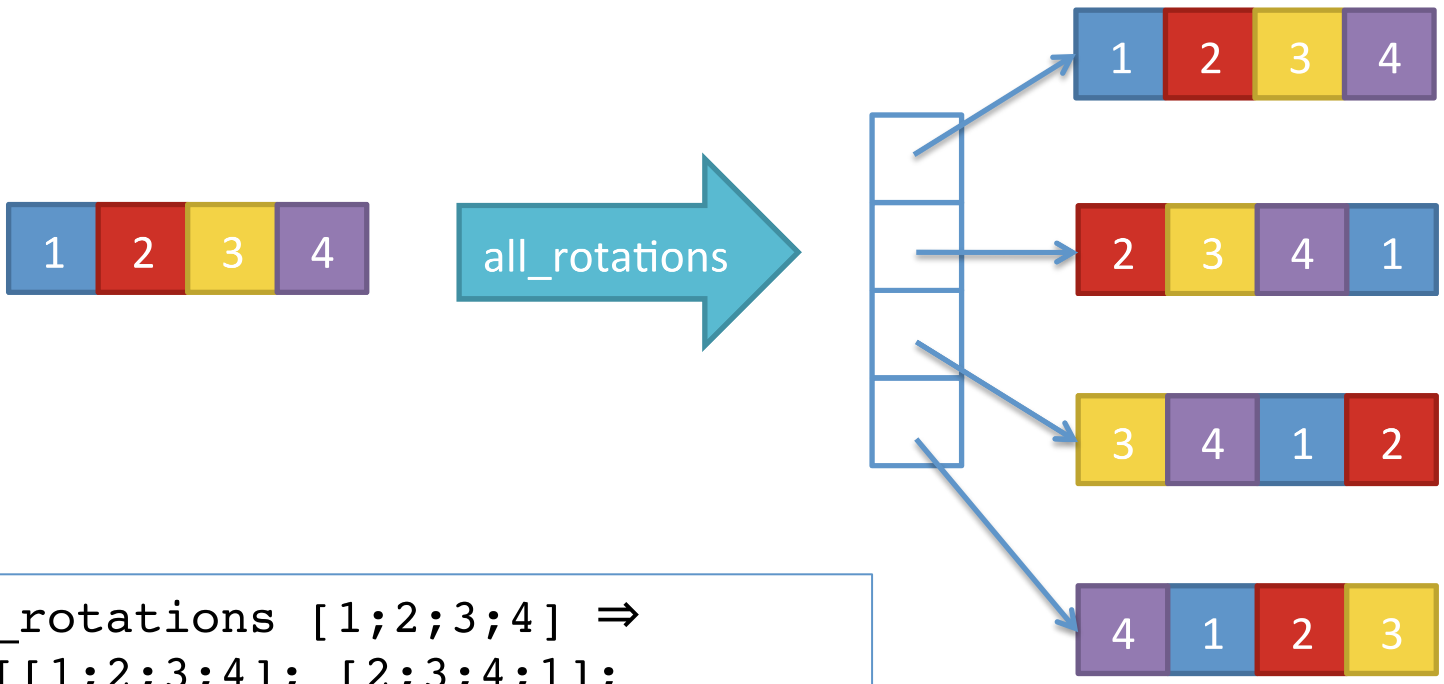
- Design problem: Given a list, produce all *initial prefixes* of the list.



```
inits [1;2;3;4] ⇒
[[]; [1]; [1;2]; [1;2;3]; [1;2;3;4]]
```

Challenge: All rotations

- Design problem: Given a list, produce all *rotations* of the list.



```
all_rotations [1;2;3;4] ⇒
   [[1;2;3;4]; [2;3;4;1];
    [3;4;1;2]; [4;1;2;3]]; ]
```