Programming Languages and Techniques (CIS120)

Lecture 3

January 24, 2014

Lists and Recursion

Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Tuesday, January 28th at midnight
  - ~~Labs~~ Extended office hours ~~start~~ this week!
- 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

A Design Problem / Situation

Suppose we are asked by Penn to design a new email system for notifying students of emergencies such as Snowpocalpses.

We should be able to subscribe students to the list, query the size of the list, check if a particular email is enrolled, compose messages for all the lists, filter the list to just enrolled instructors, 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 emails?

Important concepts are:

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

2. Formalize the interface

- Represent an email by a *string* (the email itself)
- Represent an email list using an *immutable list of strings*
- Represent the collection of instructor emails using a *toplevel definition*
- Define the interface to the functions:

```
let subscribe (email : string)
              (l : string list) : string list =
let number_of_emails (l : string list) : int =
let contains (l : string list) (email : string) : bool =
let compose (msg : string)
            (l : string list) : string list =
let filter_instructors (l : string list) : string list =
```

3. Write test cases

```
let l1 : string list = [ "posera@cis.upenn.edu"
                       ; "tgarsys@seas.upenn.edu"
                       ; "harmoli@seas.upenn.edu ]
let l2 : string list = [ "tgarsys@seas.upenn.edu" ]
let l3 : string list = []

let test () : bool =
  (number_of_emails l1) = 3
;; run_test "number_of_emails l2" test

let test () : bool =
  (number_of_emails l2) = 1
;; run_test "number_of_emails l2" test

let test () : bool =
  (number_of_emails l3) = 0
;; run_test "number_of_emails 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

What is a list?

- A list 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*.

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

Explicitly parenthesized

‘`::`’ is an ordinary operator like + or ^, except it takes an element and a *list* of elements as inputs:

`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 are values.

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.

Pattern Matching

OCaml provides a single expression for inspecting lists, called *pattern matching*.

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

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

match expression syntax is:

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

case branches

This case analysis is justified because there are only *two* shapes that a list can have.

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

Calculating with Matches

- Consider how to run a match expression:

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

⟼

1 + 10

⟼

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

Using Recursion Over Lists

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 number_of_emails (pl : string list) : int =
  begin match l with
  | [] -> 0
  | ( email :: rest ) -> 1 + number_of_emails rest
  end
```

If the lists is non-empty, then "email" is the first email of 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

```
number_of_emails ["posera@cis.upenn.edu";"harmoli@seas.upenn.edu"]
```

⟼ *(substitute the list for l in the function body)*

```
begin match "posera@cis.upenn.edu"::
            ("harmoli@seas.upenn.edu"::[]) with
  | [] -> 0
  | (email :: rest) -> 1 + (number_of_emails rest)
end
```

⟼ *(second case matches with rest = "harmoli@seas.upenn.edu"::[])*

```
1 + (number_of_songs "harmoli@seas.upenn.edu"::[])
```

⟼ *(substitute the list for l in the function body)*

```
1 + (begin match "harmoli@seas.upenn.edu"::[] with
       | [] -> 0
       | (email :: rest) -> 1 + (number_of_emails rest)
     end
```

⟼ *(second case matches again, with rest = [])*

```
1 + (1 + number_of_emails [])
```

⟼ *(substitute [] for l in the function body)*

…

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

Recursive function patterns

Recursive functions over lists follow a general pattern:

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

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

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

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…
     - 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?
     - Is there a direct solution for the empty list?