Programming Languages and Techniques (CIS120)

Lecture 3
January 19\textsuperscript{th} 2016

Lists and Recursion
Announcements

• Recitations start today!

• Homework 1: OCaml Finger Exercises
  – Due: Tuesday 1/26 at midnight

• Clickers: attendance grades start Friday
  – Quizzes: TP160116

• 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!)
  – Look at HW1 FAQ

• TA office hours: on course Calendar webpage
Have you successfully installed OCaml on your laptop?

1) Yes
2) No
Have you started working on HW 01?

1) Yes
2) No
What is an OCaml module?

```ocaml
;; open Assert

let attendees (price:int) :int =
  (-15 * price) / 10 + 870

let test () : bool =
  attendees 500 = 120

;; run_test "attendees at 5.00" test

let x : int = attendees 500

;; print_int x
;; print_endline "end of demo"
```

Toplevel items:

- Declarations (start with let)
  - Identifiers
  - Functions
- Commands (start with ;;)
  - module imports
  - run_test
  - printing
OCaml offers two forms of ‘let’ declarations:

- **Top-level:**
  ```ocaml
  let x : int = 1
  let y : int = 2
  ... rest ...
  ```
  - no “in”
  - scope of binding = rest of file

- **Local:**
  ```ocaml
  let z : int =
  let x = 1 in
  let y = 2 in
  x + y
  ```
  - scope of binding = expression following “in”
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)

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

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

let answer : int =
    let y = 2 in
    let x = 2 in

let answer : int =
    let x = 2 in
    x + 2

let answer : int =
    2 + 3
```
let total_secs (hours:int) (minutes:int) (seconds:int) : int =
(hours * 60 + minutes) * 60 + seconds
Once a function has been declared, it can be invoked by writing the function name followed by a list of arguments. This is a *function application* expression.

```
| total_secs | 5 30 22 |
```

(Note that the list of arguments is *not* parenthesized.)
Calculating With Functions

• To calculate the value of a function application, first calculate values for its arguments and then substitute them for the parameters in the body of the functions.

```plaintext
let total_secs (hours:int) (minutes:int) (seconds:int): int =
  (hours * 60 + minutes) * 60 + seconds
```

```
total_secs (2 + 3) 12 17

→ total_secs 5 12 17
→ (5 * 60 + 12) * 60 + 17
→ (300 + 12) * 60 + 17
→ 312 * 60 + 17
→ 18720 + 17
→ 18737
```
Working with lists
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
  ```ml
  let instructor_emails : string list = ...
  ```

- Define the interface to the functions:
  ```ml
  let subscribe (email : string) (lst : string list) : string list = ...
  let length (lst : string list) : int = ...
  let contains (lst : string list) (email : string) : bool = ...
  ```
3. Write test cases

```ocaml
let l1 : string list = [ "sweirich@cis.upenn.edu";
                       "mattch@seas.upenn.edu";
                       "maxmcc@sas.upenn.edu" ]
let l2 : string list = [ "mattch@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

e-mail.ml