Value-Oriented Programming (continued)

Lists and Recursion
CIS 120 Announcements

• Homework 1: OCaml Finger Exercises
  – Due: Tuesday 9/12 at midnight
  – Safari automatically unzips submit.zip: see Piazza for how to disable that

• Reading: Please read up to Chapter 3

• Questions?
  – Post to Piazza (privately if you need to include code!)
  – Look at HW1 FAQ

• TA office hours: on course Calendar webpage

• Recitations start today!
Have you started working on HW 1?
Review: Value-Oriented Programming

• Ocaml promotes a **value-oriented** style
  – We’ve seen that there are a few *commands*...
    - `print_endline`, `run_test`
  ... but these are used rarely
  – Most of what we write is *expressions* denoting *values*

• We can think of an OCaml expression as just a way of writing down a *value*

• We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to this value
• The meaning (“denotation”) of a pure expression doesn’t change over time:

```
let atts : int = (attendees 500)
let atts2 : int = atts + atts
```

```
let atts2 : int =
  (attendees 500)
  + (attendees 500)
```

Both ways of computing atts2 yield the same value (in contrast with an imperative language, which runs attendees’ effects twice.)
let total_snds (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.

(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.

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

```
total_snds (2 + 3) 12 17
⇒ total_snds 5 12 17
⇒ (5 * 60 + 12) * 60 + 17  subst. the args in the body
⇒ (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 mailing 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 email
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*

```plaintext
let instructor_emails : string list = ...
```

- Define the interface to the functions:

```plaintext
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 = [ "stevez@cis.upenn.edu";
                         "mattch@seas.upenn.edu";
                         "davisbet@seas.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 l3" test
```

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

A Value-Oriented Approach
What is a list?

A list value is either:

- \([\ ]\) the *empty* list, sometimes called *nil*
- \(v :: \text{tail}\) 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) ints

“abc”::”xyz”::[ ]  a list of (two) strings

(false::[ ])::(true::[ ])::[ ]  a list of lists that each contain booleans

[ ]  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

ture::[ ]              # a list of one boolean

[ ]                   # the empty list
```
Convenient Syntax

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

- \[1;2;3;4\] a list of (four) ints
- \[“abc”;”xyz”\] a list of (two) strings
- \[[false];[true]\] a list of lists that each contain booleans
- [ ] the empty list
NOT Lists

These are not lists:

- \([1;true;3;4]\)  
  different element types
- 1::2  
  2 is not a list
- 3::[ ];::[]  
  different element types
Calculating With Lists

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

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

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

\[\rightarrow 5 :: 2 :: []\] 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

```
type int_list = int list
```

The type of lists of strings is written

```
type string_list = string list
```

The type of lists of booleans is written

```
type bool_list = bool list
```

The type of lists of lists of strings is written

```
type (string_list 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.
Interactive Interlude

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

```ocaml
let mylist : int list = [1; 2; 3; 5]
let y : int = begin
match mylist with
| [] -> 42
| first::rest -> first+10
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 branches.
Calculating with Matches

- Consider how to evaluate a match expression:

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

  \[ \rightarrow 1 + 10 \]

  \[ \rightarrow 11 \]

  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:

  \[
  \text{length } 1::2::3::[] = 1 + (\text{length } 2::3::[])
  \]

  \[
  \text{length } 2::3::[] = 1 + (\text{length } 3::[])
  \]

  \[
  \text{length } 3::[] = 1 + (\text{length } [])
  \]

  \[
  \text{length } [] = 0
  \]
Interactive Interlude

email.ml
Recursion Over Lists in Code

The function calls itself \textit{recursively} so the function declaration must be marked with \textit{rec}.

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

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

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 \textit{structure} of the value and (optionally) give \textit{names} to parts of it.
Calculating with Recursion

\[
\text{length } [\text{"a"}; \text{"b"}]
\]

\[
\rightarrow
(substitute \text{the list for } l \text{ in the function body})
\begin{align*}
\text{begin match } \text{"a"::\text{"b"}::[]} \text{ with} \\
| [] & \rightarrow 0 \\
| (x :: \text{rest}) & \rightarrow 1 + \text{length rest}
end
\end{align*}
\]

\[
\rightarrow
(\text{second case matches with } \text{rest } = \text{"b"::[]})
\]
\[
1 + (\text{length } \text{"b"::[]})
\]

\[
\rightarrow
(substitute \text{the list for } l \text{ in the function body})
\]
\[
1 + (1 + \text{length } [\text{}])
\]

\[
\rightarrow
(\text{second case matches again, with } \text{rest } = [\text{}])
\]
\[
1 + (1 + \text{length } [\text{}])
\]

\[
\rightarrow
(substitute [\text{}] for \text{l in the function body})
\]
\[
\ldots
\]

\[
\rightarrow
1 + 1 + 0 \Rightarrow 2
\]
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:

```ocaml
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
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?