Lists II, Tuples, and Patterns
Announcements

• Homework 1: OCaml Finger Exercises
  – Due: Monday, Jan 23rd at 11:59:59pm (midnight)
  – Don’t use ‘@’ for Problem 7. 😊

• TA office hours: on website and Piazza

• Questions?
  – Post to Piazza (privately if need to include code)
  – Office hours (Weirich: 4-5PM today, none on Monday)
Who is CIS 120?

- College
- SEAS
- Wharton

- freshman
- sophomore
- junior
- senior
- other

- male
- female
Prior experience with programming

- Work experience involving significant programming
- CIS110
- An AP or IB Computer Science course or the equivalent
- Multiple courses/camps/experiences that involved programming
- A course/camp/experience that taught me how to do some programming
- Very little (e.g. a little html code)

CIS120 / Spring 2012
Size of largest program

- Java or C#
- C or C++
- (Visual)Basic
- Python or Ruby
- Javascript
- Scheme or Lisp
- ML or Haskell
- ML or Haskell
- Other

CIS120 / Spring 2012
Lists
The type of lists of integers is written \texttt{int list}.

The type of lists of strings is written \texttt{string list}.

The type of lists of booleans is written \texttt{bool list}.

The type of lists of lists of strings is written \texttt{(string list) list}.

etc.

*Note that lists in OCaml are \textit{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.
What is a list?

• A list is either:
  [-] the empty list, sometimes called nil, or

  \( v::\text{tail} \) 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 is no other way to create a list.
Example Lists

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

1 :: 2 :: 3 :: 4 :: []  

“abc” :: “xyz” :: []  

true :: []  

[]

a list of four numbers

a list of two strings

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

ture::[
)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
The function calls itself *recursively* so the function declaration must be marked with rec.

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

- If the lists is non-empty, then “song” is the first song 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 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.
Calculating with Cases

• Consider how to run a match expression:

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

\[ \Rightarrow 1 + 10 \]

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.
Calculating with Recursion

number_of_songs ["Monster";"Teeth"]

 begins match "Monster"::("Teeth"::[]) with
  | [] -> 0
  | (song :: rest) -> 1 + (number_of_songs rest)
 end

 begins match "Teeth"::[] with
  | [] -> 0
  | (song :: rest) -> 1 + (number_of_songs rest)
 end

 begins match [] with
  | [] -> 0
  | (song :: rest) -> 1 + (number_of_songs rest)
 end

let rec number_of_songs (pl : string list) : int =
  begin match pl with
    | [] -> 0
    | (song :: rest) -> 1 + number_of_songs rest
  end
Recursive function patterns

Recursive functions over lists follow a general pattern:

```ml
let rec number_of_songs (pl : string list) : int =
    begin match pl with
    | [] -> 0
    | (song :: rest) -> 1 + number_of_songs rest
    end

let rec contains (pl:string list) (s:string) : bool =
    begin match pl with
    | [] -> false
    | (song :: rest) -> s = song || contains rest s
    end
```
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?
More List Examples

see lists.ml
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]; []]
• Design problem: Given a list, produce all *initial prefixes* of the list.

\[
inits \[1;2;3;4\] \Rightarrow \[
[[]]; [1]; [1;2]; [1;2;3]; [1;2;3;4]
\]
Challenge: All rotations

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

\[
\text{all_rotations} \ [1;2;3;4] \Rightarrow \n [1;2;3;4]; [2;3;4;1]; [3;4;1;2]; [4;1;2;3]
\]