CSCI 334:
Principles of Programming Languages

Lecture 12: ML and F#

Instructor: Dan Barowy

Williams

Announcements

Announcements

Lab machines: see email for dotnet fix

Announcements

Lab machines: see email for dotnet fix

Also, clarified zip3 example in HW6 PDF

Compound Types:

Records,
Lists,
Tuples,
ADTs

Records

Records

Records

- Like tuple, but with labeled elements:

```
> type Point = { X: float; Y: float; Z:
  float; }
> let mypoint = { X = 1.0; Y = 1.0; Z =
  -1.0 };
```

Records

- Like tuple, but with labeled elements:

```
> type Point = { X: float; Y: float; Z:
  float; }
> let mypoint = { X = 1.0; Y = 1.0; Z =
  -1.0 };
```

- Selector operator:

```
> mypoint.X;;
val it : float = 1.0
- mypoint.Z;
val it : float = -1.0
```

Lists

Lists

Lists

- Examples

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list
  - all elements of list must be same type

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list
  - all elements of list must be same type
- Operations

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list
  - all elements of list must be same type
- Operations
  - length `length [1;2;3]` ⇒ `3`

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list
  - all elements of list must be same type
- Operations
  - length `length [1;2;3]` ⇒ `3`
  - append `[1;2]@[3;4]` ⇒ `[1; 2; 3; 4]`

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list
  - all elements of list must be same type
- Operations
  - length `length [1;2;3]` ⇒ `3`
  - append `[1;2]@[3;4]` ⇒ `[1; 2; 3; 4]`
  - cons `1::[2;3]` ⇒ `[1; 2; 3]`

Lists

- Examples
  - `[1; 2; 3; 4], ["wombat"; "numbat"]`
  - `[]` is empty list
  - all elements of list must be same type
- Operations
  - length `length [1;2;3]` ⇒ `3`
  - append `[1;2]@[3;4]` ⇒ `[1; 2; 3; 4]`
  - cons `1::[2;3]` ⇒ `[1; 2; 3]`
  - map `List.map succ [1;2;3]` ⇒ `[2;3;4]`

Many Types Of Lists

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  `- [];`

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  ```
  - [];
  val it : 'a list
  ```


Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  ```
  - [];
  val it : 'a list
  ```

- Polymorphic type

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  ```
  - [];
  val it : 'a list
  ```
- Polymorphic type

  - `'a` is a type variable that represents any type

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  ```
  - [];
  val it : 'a list
  ```
- Polymorphic type
  - `'a` is a type variable that represents any type

  - `1::[] : int list`

Many Types Of Lists

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  `- [];`
  `val it : 'a list`
- Polymorphic type
  - `'a` is a type variable that represents any type
  - `1::[] : int list`
  - `"a"::[] : string list`

Lists

- Functions on Lists (usually recursive)

```
> let rec product nums =
    if (nums = [])
      then 1
      else
       (List.head nums)
       * product(List.tail nums);;

val product : int list -> int

- product [5; 2; 3];;
val it : int = 30
```

Pattern Matching

pattern matching

A pattern is built from

- values,
- constructors,
- and variables

- Tests whether value(s) have shape defined by pattern
- If matches, binds variable(s) in pattern to value(s)

Pattern Matching on Integers

Pattern Matching on Integers

- Patterns on integers

```
let rec listInts n =
  match n with
  | 0 -> [0]
  | n -> n :: listInts (n-1);;

> listInts 3;;
val it : int list = [3; 2; 1; 0]
```

Pattern Matching on Integers

- Patterns on integers

```
let rec listInts n =
  match n with
  | 0 -> [0]
  | n -> n :: listInts (n-1);;

> listInts 3;;
val it : int list = [3; 2; 1; 0]
```

- Let's flip this list around

Revisiting Local Declarations

Revisiting Local Declarations

- You can define anything almost anywhere.

Revisiting Local Declarations

- You can define anything almost anywhere.
- E.g., a function inside a function. This is very useful.

Revisiting Local Declarations

- You can define anything almost anywhere.
- E.g., a function inside a function. This is very useful.

```
let listInts n =
  let rec li n =
    match n with
    | 0 -> [0]
    | n -> n :: li (n-1)
  List.rev (li n)
```

Revisiting Local Declarations

- You can define anything almost anywhere.
- E.g., a function inside a function. This is very useful.

```
let listInts n =
  let rec li n =
    match n with
    | 0 -> [0]
    | n -> n :: li (n-1)
  List.rev (li n)

> listInts 3;;
val it : int list = [0; 1; 2; 3]
```

Pattern Matching on Lists

- List is one of two things:
  - `[]`
  - `"first elem" :: "rest of elems"`
  - `E.g., [1; 2; 3] = 1::[2,3] = 1::2::[3] = 1::2::3::[]`

- Can define function by cases

```
let rec product xs =
  match xs with
  | []    -> 1
  | x::xs -> x * product (xs);;
```

The Length Function

- Another Example

```
let rec length xs =
  match xs with
  | []    -> 0
  | x::xs -> 1 + length xs;;
```

- What is the type of length?

Pattern Matching on Tuples

```
let rec cartesianProduct xs ys =
  match xs,ys with
  | [],_      -> []
  |  _,[]     -> []
  | x::xs',_ ->
    let zs = List.map (fun y -> (x,y)) ys
    zs @ cartesianProduct xs' ys
```

Patterns and Other Declarations

- Patterns can be used in place of variables
- Most basic pattern form
  - `let <pattern> = <exp>;`
- Examples
  - `let x = 3;;`
  - `let tuple = ("moo", "cow");;`
  - `let (x,y) = tuple;;`
  - `let myList = [1; 2; 3];;`
  - `let w::rest = myList;;`
  - `let v::_ = myList;;`

Activity

Write a function `is_older` that takes two dates (where a date is `int*int*int`) and returns `true` or `false`. It evaluates to `true` if and only if the first argument is a date that comes before the second argument. If the two dates are the same, return false.

E.g.,
`is_older (2018,2,21) (2018,2,22)` returns `true`

Activity

Activity

- Write a function `number_in_month` that takes a list of dates (where a date is `int*int*int`) and an `int month` and returns how many dates are in `month`