CSCI 334:
Principles of Programming Languages

Lecture 4: ML, part 2

Instructor: Dan Barowy
Williams

1

Topics

Pattern matching

Algebraic data types

Option type

2

Your to-dos

1. Read *Advanced F#* **by Thursday**.
2. Lab 2, **due Sunday 9/24 by 10pm** (partner lab).

3

Announcements

- CS Colloquium this **Friday, Sept 22 @ 2:35pm in Wege Auditorium (TCL 123)**

Your classmates

What I Did Last Summer, Industry Edition

Short presentations by your fellow CS students about internship experiences in industry. CS Colloquium credit awarded for attendance.

4

Free your mind

5

Freeing your mind is difficult

6

Remember how I asked you to “be like Neo” and free your mind? Freeing your mind is difficult. If you found the last assignment to be a bit of a challenge, that’s OK. Even Neo tanked it the first time. But keep at it.

Pattern Matching

7

Here's the first feature that is likely VERY different from something you've seen before. Once you get used to this feature, you will miss it in other languages. In fact, some non-functional languages have started to incorporate this feature, like TypeScript.

Pattern matching

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

Using **patterns**…

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

8

Suppose somebody asks you to write a program in F# to multiply together all the elements of a list. Since we don't have loops, we will need to use recursion. Remember how recursion works: we need a base case and a recursive case. The base case is to return 1 so that our multiplication problem is grounded. Then we multiple each element one at a time in the recursive case. To do so, we need to remove the head of the list and multiply it by the product of the rest of the list. However, there is a much cleaner way to express this problem using patterns. I'll explain the difference in a minute, but first, just appreciate how much nicer this looks.

Pattern matching

A **pattern** is built from

- **values**,
- (de)**constructors**,
- and **variables**

Tests whether values match "pattern"

If yes, values bound to variables in pattern

9

A pattern is made from values, deconstructors, and variables. A deconstructor is like a constructor, but the inverse. When the value of a variable matches a pattern, we can deconstruct its values and execute a line of code.

Pattern matching

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

Using **patterns**…

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

10

The pattern in the code below has two cases. Either the list is empty or is is not. If it is empty, return one. If it is not, deconstruct the list in a head and a tail, then multiply the head by the product of the tail.

Activity: Pattern matching on integers

Write a function `listOfInts` that returns a list of integers from **zero** to `n`.

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

Oops! This returns the list backward.

Let's flip it around.

11

Spend a minute writing this function. If you are at home, cover up the solution until you are ready.

This solution is *almost* correct. The list is backward.

Revisiting local declarations

Let's fix our code the lazy way…

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

… by defining a function inside our function.

12

We'll use pipe forward and the built-in List.rev function.

Sidebar: breakpoint debugging

13

Debugging programs is a pain. If you've never used a breakpoint debugger, now is the time to learn.

Pattern matching on lists

14

- Remember, a 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 length xs =
  match xs with
  | []    -> 0
  | x::xs -> 1 + length xs
```

Patterns in declarations

15

- 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`

16

Algebraic Data Types*

*not to be confused with Abstract Data Types!

17

Algebraic Data Type

An **algebraic data type** is a composite data type, made by combining other types in one of two different ways:

- by **product**, or
- by **sum**.

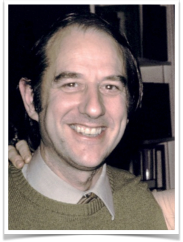
You've already seen **product types**: tuples and records.

So-called b/c the set of all possible values of such a type is the cartesian product of its component types.

We'll focus on **sum types**.

18

Algebraic Data Types



- Invented by Rod Burstall at University of Edinburgh in '70s.
- Part of the HOPE programming language.
- Not useful without pattern matching.
- Like peanut butter and chocolate, they are "better together."

19

In case you've never heard the "better together" reference, he's some pop culture trivia.

A "move" function in a game

20

Suppose we want to model moving a character in one of four directions.

A "move" function in a game (Java)

```
public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public … move(int x, int y, int dir) {
  switch (dir) {
    case NORTH: ...
    case ...
  }
}
```

21

We might do it like this in Java. It works, but it sure is a lot of typing!

A “move” function in a game (Java)

Discriminated Union (sum type)

```
type Direction =
     North | South | East | West;
let move coords dir =
  match coords,dir with
   |(x,y),North -> (x,y - 1)
   |(x,y),South -> (x,y + 1)
```

- Above is an “incomplete pattern”
- ML will warn you when you’ve missed a case!
  - “proof by exhaustion”

22

We can do it much more concisely in F# using patterns. Importantly, F# will tell you when you’ve missed a case.

Parameters

```
type Shape =
    | Rectangle of float * float
    | Circle of float
```

- Pattern match to extract parameters

```
let s = Rectangle(1.0,4.0)
match s with
| Rectangle(w,h) -> …
| Circle(r) -> …
```

23

So, stepping back a little, an algebraic data type is a way of defining a piece of data by cases. The key thing to remember is that the type here is Shape. However, a shape can have cases. The names of those cases are constructors for each kind of Shape. When we match a Shape in a pattern, we can deconstruct each case into its component values.

Named parameters

```
type Shape =
    | Rectangle of width: float * height: float
    | Circle of radius: float
```

- Names are really only useful for initialization, though.

```
let s = Rectangle(height = 1.0, width = 4.0)
```

24

You can also name the pieces of each case, which helps with initialization.

ADTs can be recursive and generic

25

```
type MyList<'a> =
      | Empty
      | NonEmpty of head: 'a * tail: MyList<'a>
```

```
> NonEmpty(2, Empty);;
    val it : MyList<int> = NonEmpty (2,Empty)
```

You can also make an ADT recursive, and you can also make it generic. Recall that a linked list is both recursive and generic.

Avoiding errors with patterns

26

- Another example: handling errors.
- SML has exceptions (like Java)
- But an alternative, easy way to handle many errors is to use the option type:

```
type option<'a> =
| None
| Some of 'a
```

Here's another nice way to use patterns: avoiding errors. Unlike throwing exceptions in Java (F# also has exceptions, BTW), F# has a convenience, *type safe* method for handling errors that can be used *to guarantee* that the user of a function handles the error condition.

Avoiding errors with patterns

27

```
let divide quot div =
  match div with
  | 0 -> None
  | _ -> Some (float quot/float div)
```

For example.

Avoiding errors with patterns

28

```
> divide 6 7;;
val it : float option = Some 0.8571428571

> divide 6 0;;
val it : float option = None

>
```

option type

29

- Why `option`?
- `option` is a data type;

  not handling errors is a static type error!

Recap & Next Class

30

Today:

Pattern matching
Algebraic data types
Option type

Next class:

Higher order functions