CSCI 334: Principles of Programming Languages

Lecture 5: Algebraic Data Types

Instructor: Dan Barowy

Williams

1

Announcements

- No announcements.

2

Topics

Algebraic data types

Avoiding errors

3

Your to-dos

1. Lab 2, **due Monday 2/19 by 10pm** (partner lab).
2. Read *Syntax* and *Introduction to the Lambda Calculus, Part 1* by **next Thursday, 2/22**.

4

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
```

5

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.

Pattern matching on lists

- 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
```

6

Quiz

7

Algebraic Data Types*

*not to be confused with Abstract Data Types!

8

Algebraic Data Type

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

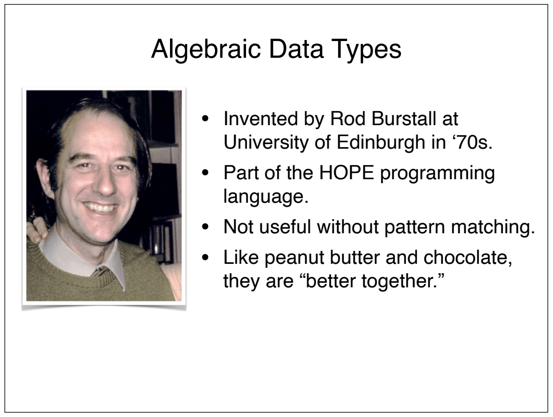
- by **product**, or
- by **sum**.

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

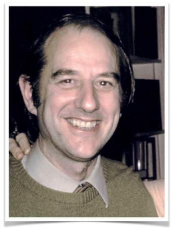
So-called b/c the set of all possible values of such a type is the cartesian product of its component types.

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

9



Algebraic Data Types

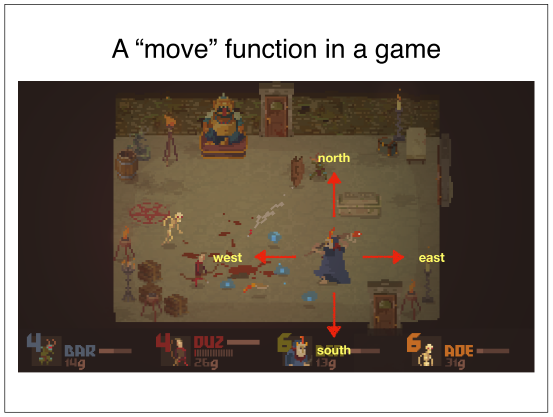


- Invented by Rod Burstall at University of Edinburgh in ‘70s.
- Part of the HOPE programming language.
- Not useful without pattern matching.
- Like peanut butter and chocolate, they are “better together.”

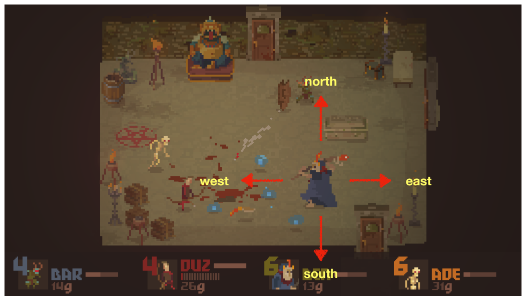
10

11

In case you’ve never heard the “better together” reference, he’s some pop culture trivia.

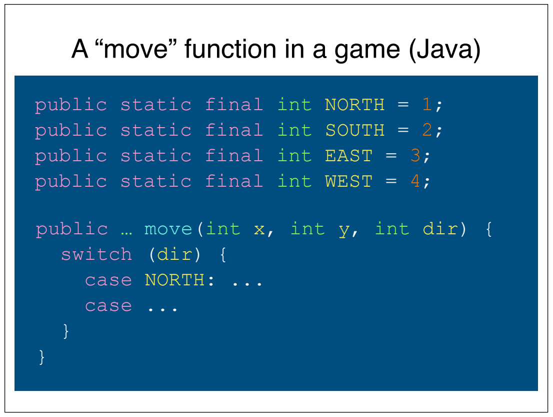


A “move” function in a game



12

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

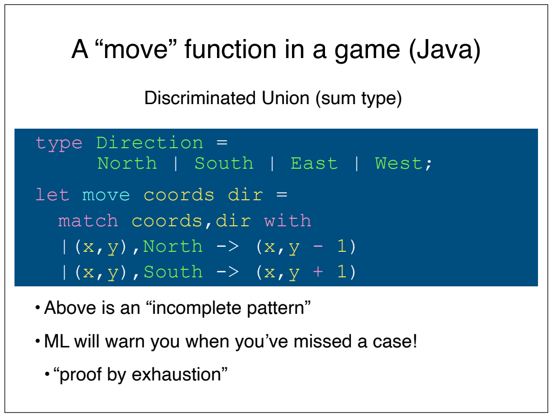


A “move” function in a game (Java)

```
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 ...
  }
}
```

13

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)

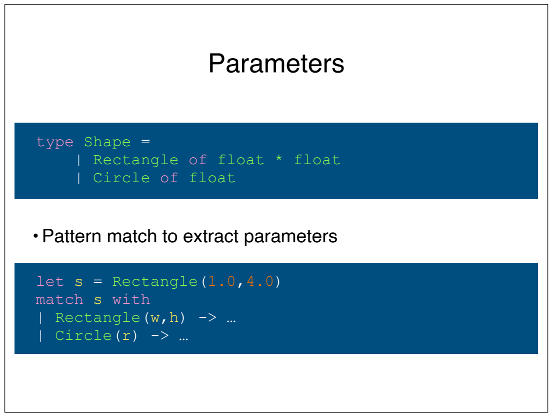
Discriminated Union (sum type)

```
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”

14

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) -> …
```

15

So, stepping back a little, an algebraic data type is a way of defining a piece of data by cases. The key thing to observe 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)
```

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

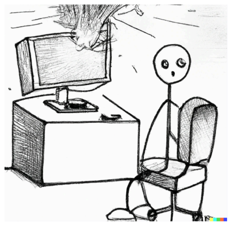
ADTs can be recursive and generic

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

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

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

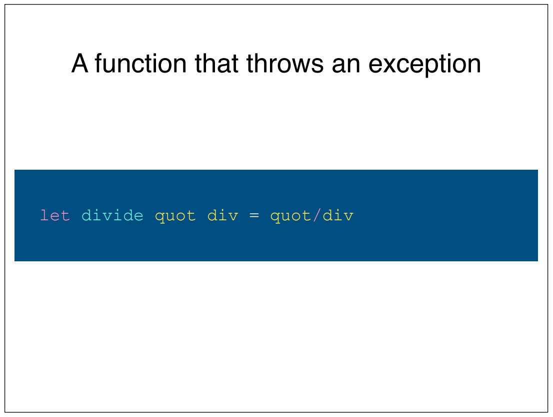
Avoiding errors



Exploding programs is **no fun**.

Validate input so that users don't get hit by shrapnel.

18 I'm going to discuss two approaches.



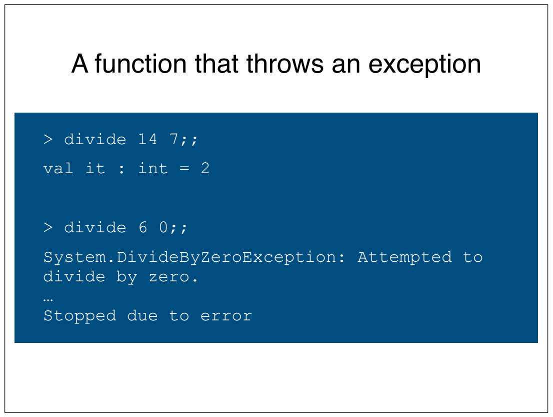
A function that throws an exception

```
let divide quot div = quot/div
```

19

Here's a toy example of a function that can fail (with an exception). Although the failure mode of this function may seem obvious for this example, in general, it is often hard to see which inputs may cause a function to fail, especially if you did not write the function.

I am choosing a function that does integer division here because floating point divide by zero is infinity (not an exception).

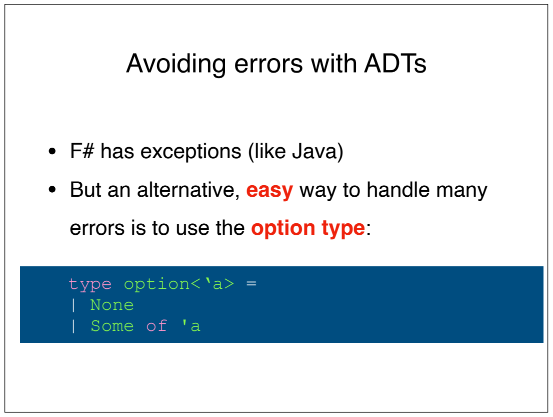


A function that throws an exception

```
> divide 14 7;;
val it : int = 2

> divide 6 0;;
System.DivideByZeroException: Attempted to
divide by zero.
…
Stopped due to error
```

20

In case it wasn't clear, here's the function failing.

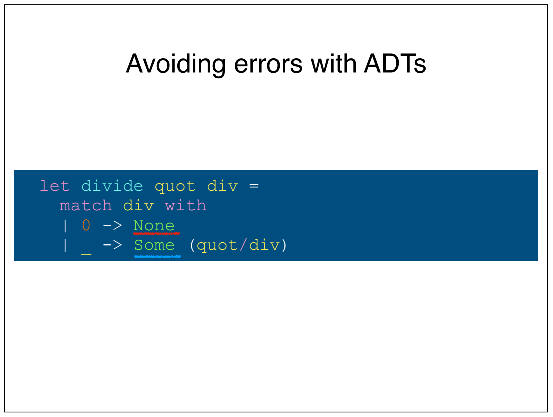


Avoiding errors with ADTs

- F# 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
```

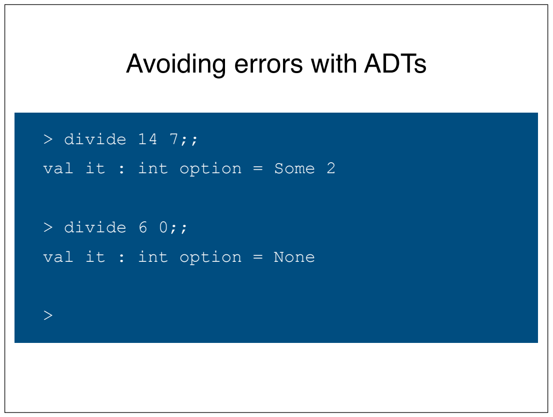
21


Avoiding errors with ADTs

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

22

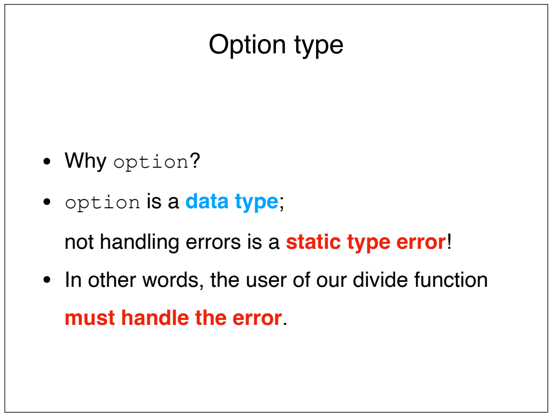
For example.


Avoiding errors with ADTs

```
> divide 14 7;;
val it : int option = Some 2

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

>
```

23

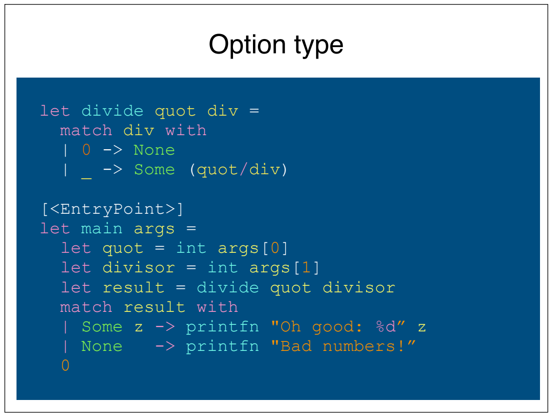
Think of the Some case as the number 2 being inside a box of type Option. When that box is labeled “Some” there is something inside it. When it is labeled “None” there is nothing inside it.


Option type

- Why `option`?
- `option` is a **data type**;
  not handling errors is a **static type error**!
- In other words, the user of our divide function **must handle the error**.

24

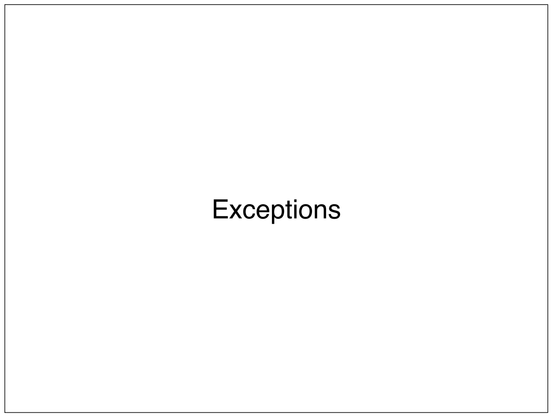
So why is this a good idea? In short, it forces the user of the function to acknowledge the failure mode of the program, and to write program logic to handle it. Failing to handle the error is a type error, which means that their program will not compile. Now, we cannot guarantee that the user does the “right thing” with the failure, but at least we can guarantee that they must do something. Moreover, we make the user’s life easier because they do not need to understand the domain of the function deeply— they just need to think of a corrective action.



Option type

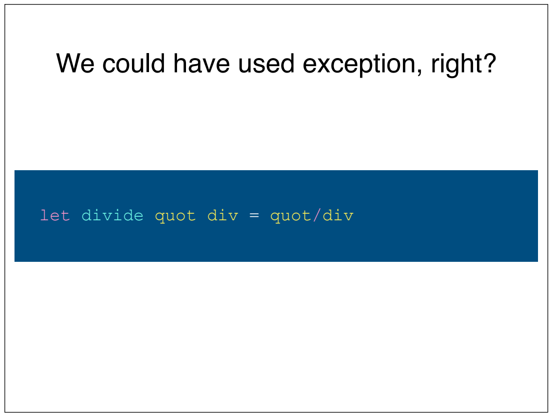
```
let divide quot div =
  match div with
  | 0 -> None
  | _ -> Some (quot/div)

[<EntryPoint>]
let main args =
  let quot = int args[0]
  let divisor = int args[1]
  let result = divide quot divisor
  match result with
  | Some z -> printfn "Oh good: %d" z
  | None   -> printfn "Bad numbers!"
  0
```

25 Here is a complete example, with a main method. Try it yourself!



Exceptions

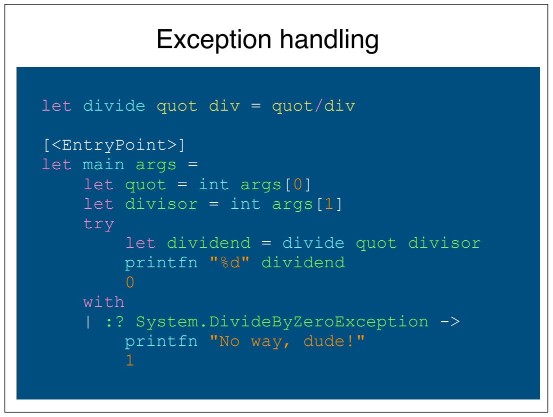
26 Of course, F# also has exceptions.



We could have used exception, right?

```
let divide quot div = quot/div
```

27 This function naturally uses exceptions, since integer division by zero is undefined and throws a floating point exception.

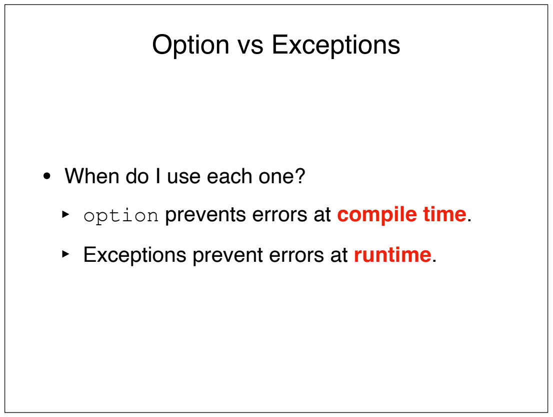


Exception handling

```
let divide quot div = quot/div

[<EntryPoint>]
let main args =
    let quot = int args[0]
    let divisor = int args[1]
    try
        let dividend = divide quot divisor
        printfn "%d" dividend
        0
    with
    | :? System.DivideByZeroException ->
        printfn "No way, dude!"
        1
```

28

Here's a complete example. Observe that the burden is shifted entirely to the user of the divide function. Also, F# does not force users to handle exceptions, so if they do not actively anticipate errors, they are likely to miss the fact that they need to do this. Still, it is a simple mechanism and can work reliably when the domain is communicated clearly to the user of the function (e.g., through comments).

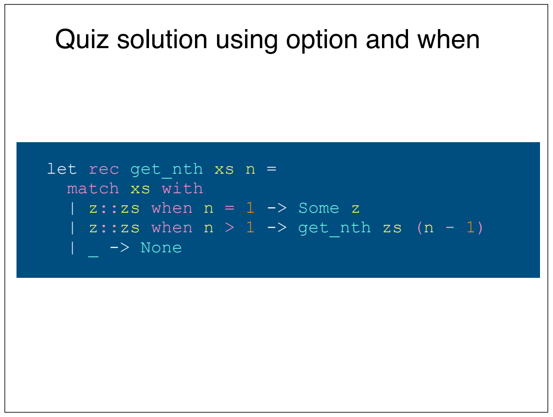


Option vs Exceptions

- When do I use each one?
  - `option` prevents errors at **compile time**.
  - Exceptions prevent errors at **runtime**.

29

Why might you want to use option vs exceptions? The question comes down to when you want errors in program logic handled. Option ensures that logic errors are handled at compile time. Exception handlers ensure that logic errors are handled at runtime. I have a personal preference for the former because I think coding is hard and that we need all the help that we can get.

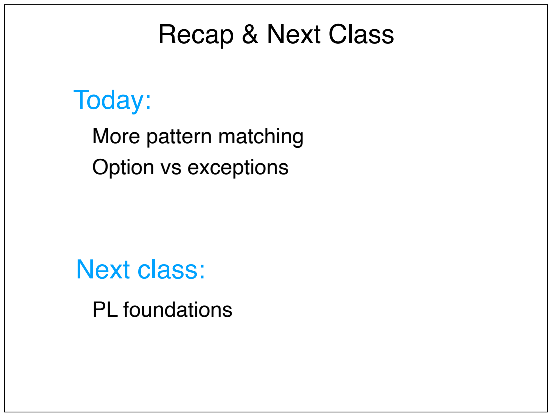


Quiz solution using option and when

```
let rec get_nth xs n =
  match xs with
  | z::zs when n = 1 -> Some z
  | z::zs when n > 1 -> get_nth zs (n - 1)
  | _ -> None
```

30

We can use Option and "when" syntax in our patterns to make our quiz solution pretty.



Recap & Next Class

Today:

More pattern matching
Option vs exceptions

Next class:

PL foundations