CSCI 334:
Principles of Programming Languages

Lecture 4: Algebraic Data Types

Instructor: Dan Barowy

Williams

Announcements

- No announcements.

Topics

Pattern matching

Algebraic data types

Avoiding errors

Your to-dos

1. Lab 2, **due Wednesday 2/19 by midnight**
   1. **partner lab**: make sure you hand in `collaborators.txt` whether you work with a partner or not
   2. **No copy/paste** of code.
   3. `pushcheck` will remind you
2. Read *F#: The Cool Stuff*
3. First real quiz on Thursday.

Pattern Matching

Pattern matching

A **pattern** is built from

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

Tests whether values match "pattern"

If match, **binds** values to variables in pattern

Patterns in declarations

- Most basic pattern form
  - `let <pattern> = <exp>`
- Examples
  - **without** pattern
    - `let tuple = ("moo", "cow")`
  - **with** pattern
    - `let (x,y) = ("moo", "cow")`
  - **without** pattern
    - `let myList = [1; 2; 3]`
  - **with** pattern
    - `let w::rest = [1; 2; 3]`
    - `let v::_ = myList`

Pattern matching

```
let rec sum (xs: int list) : int =
    if xs = [] then
        0
    else
        (List.head xs) + sum (List.tail xs)
```

Using **patterns**…

```
let rec sum (xs: int list) : int =
    match xs with
    | [] -> 0
    | y::ys -> y + sum ys
```

Another example

- 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
```

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.

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 :: li (i - 1)
  li n |> List.rev
```

… by defining a function inside our function.

Patterns can be used for any data structure

- literal values
- lists
- tuples
- records
- ADTs
- and combinations of these!

Algebraic Data Types*

*not to be confused with Abstract Data Types!

Algebraic Data Type

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

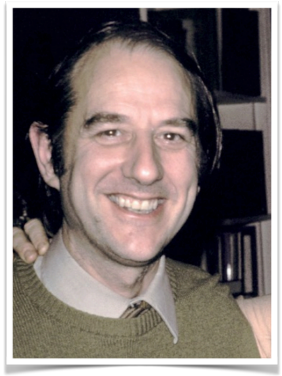
- by **product**, or
- by **sum**.

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**.

Algebraic Data Types



- Invented by Rod Burstall at University of Edinburgh in '70s.
- Part of the HOPE programming language.
- **Pattern matching** and **ADTs** are **better together**.

A "move" function in a game

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 ...
  }
}
```

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”

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) -> …
```

Named parameters

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

- Names are really only useful for initialization, though.

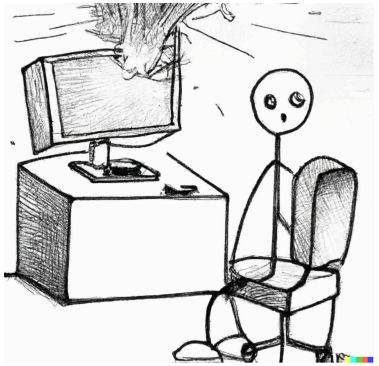
```
let s = Rectangle(height = 1.0, width = 4.0)
```

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)
```

Avoiding errors



Exploding programs is **no fun**.

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

A function that throws an exception

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

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
```

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
```

Avoiding errors with ADTs

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

Avoiding errors with ADTs

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

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

>
```

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**.

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
```

Exceptions

We could have used exception, right?

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

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
```

Option vs Exceptions

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

Recap & Next Class

Today:

Pattern matching
Algebraic data types
Option vs exceptions

Next class:

Higher-order functions