CIS 1904: Haskell

Logistics

- HW 01 due yesterday
- HW 02 will be released tomorrow

Algebraic Data Types

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

Sum type: values can be any **one** of the listed types.

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Tuples
  - `(True, 6) :: (Bool, Int)`
  - `(11, False, “abc”) :: (Int, Bool, String)`

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

keyword that tells Haskell we're defining a type

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

↖ keyword that tells Haskell we're defining a type

`type` is also a keyword, but it is for defining type *synonyms*.

```
type PennID' = PennID
```

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

↰ name of our new type

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

↰ name of our new type

Note: This is actually called a type **constructor**. We will come back to this when we talk about polymorphism.

By default, we will use “constructor” to mean “data constructor”.

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

↰ (data) *constructor* for the type

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

↰ (data) *constructor* for the type

Note: for single-constructor types, it’s common in Haskell to use the same name for the type itself and the constructor.

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

↑

like a tuple, but the order of elements is flexible

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

Syntax note: type and constructor names must start with a capital letter.

Variable names should start with a lower case letter.

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}
```

In OCaml:

```
type PennID = { name : string; year : int; idNum : int; }
```

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Records

```
data PennID = PennID {name :: String, year :: Int, idNum :: Int}

{idNum=12345678, year=2026, name=“Real Person”} :: PennID

{year=2028, name=“Human Being”, idNum=00000000} :: PennID
```

What is an ADT?

Main idea: making a type out of other types.

Product type: values have one value of each of the listed types.

- Tuples: elements are in fixed order and not tagged, e.g. `(True, 6)`
- Records: elements are in any order and tagged, e.g. `{year=2028, name="A"}`

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Untagged Unions

```
union grade {
    int percent;
    char letter;
};
```

* C example because Haskell does not allow these

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Untagged Unions

```
union grade {
    int percent;
    char letter;
};

union grade x;
x.letter = ‘A’;
x.percent + 5;
```

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Untagged Unions

```
union temperature {
    int celsius;
    int fahrenheit;
};

union temperature x;
x.fahrenheit = 32;
printf( "Temp in Celsius: %d\n", x.celsius);
```

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Tagged unions (aka variants)

```
data Temperature
= Celsius Int
| Fahrenheit Int
```

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Tagged unions (aka variants)

```
data Temperature
= Celsius Int
| Fahrenheit Int
```

`Celsius` and `Fahrenheit` are data constructors.
We *destruct* this type by pattern matching, so we always know what case we're in.

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Tagged unions (aka variants)

```
data Temperature
= Celsius Int
| Fahrenheit Int
```

```
whichOne :: Temperature -> String
whichOne (Celsius x) = “Celsius”
whichOne (Fahrenheit x) = “Fahrenheit”
```

What is an ADT?

Main idea: making a type out of other types.

Sum type: values can be any **one** of the listed types.

- Untagged Unions

```
union temperature {
    int celsius;
    int fahrenheit;
};

union temperature x;
x.fahrenheit = 32;
printf( "Temp in Celsius: %d\n", x.celsius);
```

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
deriving(Show, Eq)
```

Anatomy of An ADT

constructors

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

The constructors are the *only* way to build something of this type; even functions have to use these internally.
This is why pattern matching on all the constructors can be exhaustive.

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

types for arguments
to the constructors

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

types for arguments
to the constructors

Q: What is the type of `Rectangle`?

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

types for arguments
to the constructors

Q: What is the type of `Rectangle`?
A: `Int -> Int -> GeometricObject`

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

types for arguments to the constructors

Q: What is the type of `Rectangle`?
A: `Int -> Int -> GeometricObject`
Constructors in Haskell are first-class values, like functions.

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

Q: Is this a product type? A sum type?

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

Q: Is this a product type? A sum type?
A: Yes

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

We can build this type in several different ways, like a sum type.
It can be a container for 0, 1, or 2 `Int`s.

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

The last option contains 2 `Int`s, like a tuple might, so we have a product.

Anatomy of An ADT

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int
```

A sum type is when we might have *any* of the listed types.
A product type is when we have one of *each* of the listed types.
Here we have elements of both!

Aside: why are they called **algebraic** data types?

Aside: why algebraic?

```
data GeometricObject
= Point
    +
| Line Int
    +
| Rectangle Int Int
```

A **sum** type is when we might have *any* of the listed types.
This is like a **disjoint union** $\uplus$ in math.

Aside: why algebraic?

```
data GeometricObject
= Point
| Line Int
| Rectangle (Int × Int)
```

A **product** type is when we have one of *each* of the listed types.
This is like a **Cartesian product** × in math.

Aside: why algebraic?

- Sum types ≈ +
- Product types ≈ ×

Aside: why algebraic?

- Sum types ≈ +
- Product types ≈ ×
- 0?
- 1?

Aside: why algebraic?

- Sum types ≈ +
- Product types ≈ ×
- 0 ≈ Empty
- 1 ≈ Unit

```
data Empty

data Unit = Unit
```

Pattern Matching

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

dimension :: GeometricObject → Int
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

dimension :: GeometricObject → Int
dimension Point = 0
dimension (Line _) = 1
dimension (Rectangle _ _) = 2
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int


dimension :: GeometricObject → Int
dimension s = case s of
    Point -> 0
    Line _ -> 1
    Rectangle _ _ -> 2
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

getDimensions :: GeometricObject -> (GeometricObject, [Int])
getDimensions Point = (Point, [])
getDimensions s@(Line len) = (s, [len])
getDimensions s@(Rectangle len width) = (s, [len, width])
```

Pattern Matching

```
data Pair = Pair Int Int

data GeometricObject
= Point
| Line Int
| Rectangle Pair


getDimensions :: GeometricObject -> (GeometricObject, [Int])
getDimensions Point = (Point, [])
getDimensions (Line len) = (s, [len])
getDimensions (Rectangle (Pair len width)) = (s, [len, width])
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

isPoint :: GeometricObject -> Bool
isPoint Point = True
isPoint (Line 0) = True
isPoint (Rectangle 0 0) = True
isPoint _ = False
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

x = 0

isPoint :: GeometricObject -> Bool
isPoint Point = True
isPoint (Line x) = True
isPoint (Rectangle x x) = True
isPoint _ = False
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

x = 0

isPoint :: GeometricObject -> Bool
isPoint Point = True
isPoint (Line x) = True
isPoint (Rectangle x x) = True
isPoint _ = False
```

Pattern Matching

```
data GeometricObject
= Point
| Line Int
| Rectangle Int Int

bothPoint :: GeometricObject -> Bool
bothPoint s1 s2 = case (s1, s2) of
    (Point, Point) -> True
    _ -> False
```

Pattern Matching

```
data IntList
= Nil
| Cons Int IntList
```

Pattern Matching

```
data IntList
= Nil
| Cons Int IntList
```

```
len :: IntList -> Int
len Nil = 0
len Cons _ tl = 1 + (len tl)
```

Expression Problem

```
data Shape
= Rectangle Int Int
| Circle Int


area :: Shape -> Double
area (Rectangle x y) = …
area (Circle x y) = …
```

```
interface Shape
    double area();
    int perimeter();

class Rectangle implements Shape
    int x, y;
    double area() { … }

class Circle implements Shape
    int r;
    double area() { … }
```

Expression Problem

```haskell
data Shape
= Rectangle Int Int
| Circle Int


area :: Shape -> Double
area (Rectangle x y) = …
area (Circle x y) = …
```

```java
interface Shape
    double area();
    int perimeter();

class Rectangle implements Shape
    int x, y;
    double area() { … }

class Circle implements Shape
    int r;
    double area() { … }
```

Which is easier to add a new function for?
Which is easier to add a new shape for?