CIS 1904: Haskell

Algebraic Dataypes

Logistics

- HW 02 due yesterday
- HW 03 will be released tonight

Algebraic Datatypes

Main idea: we can combine types to make other types

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Records

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

Elements of a record have variable order and are tagged.

Algebraic Datatypes

Main idea: we can combine types to make 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

Algebraic Datatypes

Main idea: we can combine types to make 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
```

Algebraic Datatypes

Main idea: we can combine types to make 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

Algebraic Datatypes

Main idea: we can combine types to make 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

Algebraic Datatypes

Main idea: we can combine types to make 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

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

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Records

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

↖ field/element

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Records

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

↰ field/element type

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Records

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


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

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

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Records

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

We can *destruct* a record with pattern matching.

```
foo :: PennID -> Int
foo (PennID {idNum=id, year=y, name=s}) = …
```

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Tuples

```
data IntStringPair = IntStringPair Int String
```

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Tuples

```
data IntStringPair = IntStringPair Int String

foo :: IntStringPair -> Int
foo (IntStringPair x s) = …
```

Algebraic Datatypes

Main idea: we can combine types to make 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. `IntStringPair 6 “xyz”`
- Records: elements are in any order and tagged
  - e.g. `PennID {id = 00000000, year=2028, name=“A”}`

Algebraic Datatypes

Main idea: we can combine types to make other types

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

Algebraic Datatypes

Main idea: we can combine types to make other types

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

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

* C example because Haskell does not allow these

Algebraic Datatypes

Main idea: we can combine types to make other types

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

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

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

Algebraic Datatypes

Main idea: we can combine types to make other types

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

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

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

Algebraic Datatypes

Main idea: we can combine types to make other types

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

- Tagged unions (aka variants)

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

Algebraic Datatypes

Main idea: we can combine types to make 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.

Algebraic Datatypes

Main idea: we can combine types to make 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”
```

Algebraic Datatypes

We can combine sum and product types.

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

Algebraic Datatypes

We can combine sum and product types.

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

1 + |Int| + |Int| x |Int|

Algebraic Datatypes

We can combine sum and product types.

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

Our three constructors are the *only* way to build something of this type.
Even functions have to use these internally.

This is how pattern matching on all the constructors can be exhaustive.

Algebraic Datatypes

We can combine sum and product types.

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

What is the type of `Rectangle`?

Algebraic Datatypes

We can combine sum and product types.

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

What is the type of `Rectangle`?

```
Int -> Int -> GeometricObject
```

Algebraic Datatypes

We can combine sum and product types.

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

What is the type of `Rectangle`?

`Int -> Int -> GeometricObject`

Constructors in Haskell are first-class values, just like functions.

Algebraic Datatypes

We can combine sum and product types.

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

Why is this both a product and a sum type?

Algebraic Datatypes

We can combine sum and product types.

```
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!

Algebraic Datatypes

We can combine sum and product types.

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

Algebraic Datatypes

We can combine sum and product types.

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

This type has several constructors, i.e., it can represent several different varieties of structure.

Algebraic Datatypes

We can combine sum and product types.

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

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

Algebraic Datatypes

Why are they called **algebraic** datatypes?

```
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.

Algebraic Datatypes

Why are they called **algebraic** datatypes?

```
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.

Algebraic Datatypes

Why are they called **algebraic** datatypes?

Sum types ≈ + ≈ logical OR

Product types ≈ × ≈ logical AND

Algebraic Datatypes

Why are they called **algebraic** datatypes?

Sum types ≈ +

Product types ≈ ×

0?

1?

Algebraic Datatypes

Why are they called **algebraic** datatypes?

Sum types ≈ +

Product types ≈ ×

0 ≈ Empty

1 ≈ Unit

```
data Empty
data Unit = Unit
```

Algebraic Datatypes

Why are they called **algebraic** datatypes?

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

How many `GeometricObject`s are there, if we say |`Int`| is the number of `Int`s?

Algebraic Datatypes

Why are they called **algebraic** datatypes?

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

|`GeometricObject`| = 1 + |`Int`| + |`Int`| x |`Int`|

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

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

Pattern Matching

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

dims :: GeometricObject → [Int]
dims Point = []
dims (Line len) = [len]
dims (Rectangle len width) = [len,width]
```

Pattern Matching

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

dimsPair :: GeometricObject → (GeometricObject, [Int])
dimsPair Point = (Point, [])
dimsPair s@(Line len) = (s, [len])
dimsPair s@(Rectangle len width) = (s, [len, width])
```

Pattern Matching

```haskell
data Pair = Pair Int Int

data GeometricObject
    = Point
    | Line Int
    | Rectangle Pair

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

Pattern Matching

```
x = 0

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

This is called *variable shadowing*, because the later binding sites overshadow the previous ones.

Pattern Matching

```
x = 0

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

The `Rectangle` case gives a compilation error – we can only name one of the arguments `x`, to avoid ambiguity

Pattern Matching

```
isSquare :: GeometricObject -> Bool
isSquare (Rectangle x y) | x == y = True
isSquare _ = False
```

Expression Problem

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

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

```
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) = …
```

```java
interface Shape
     double area();

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?

Expression Problem

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

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

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

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

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

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

Expression Problem

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

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

perimeter = Shape -> Double
perimeter (Rectangle x y) = …
perimeter (Circle x) = …
perimeter (Triangle x y) = …
```

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

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

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

class Triangle implements Shape
     int x, y;
     double area() { … }
     double perimeter();
```

Exercises

Higher-order functions

Recall: functions in Haskell are *first-class*

- They can be passed around as inputs to other functions
- They can be returned as outputs of other functions

Higher-order functions

Recall: functions in Haskell are *first-class*

- They can be passed around as inputs to other functions
- They can be returned as outputs of other functions

A function that takes in OR returns another function is called *higher-order*.

Higher-order functions

Examples:

- `map :: (Int -> Int) -> [Int] -> [Int]`
- `filter :: (Int -> Bool) -> [Int] -> [Int]`

These take in functions and, if partially applied, return functions.

Higher-order functions

Example:

```
compose :: (Int -> Int) -> (Int -> Int) -> Int -> Int
compose f g = \x -> f (g x)
```

`\x ->` syntax is used for anonymous functions, like `fun x =>` in OCaml

Higher-order functions

Example:

```
compose :: (Int -> Int) -> (Int -> Int) -> Int -> Int
compose f g = f . g
```

The standard library provides function composition, written as `(.)` .

Higher-order functions

Composition helps write code that is:

- Concise
- Understandable
- In keeping with the “wholemeal” programming style, which operates on entire data structures at once

```
foo :: String -> String
foo s = length (filter (== ‘C’) (toUpper s))
```

Higher-order functions

Composition helps write code that is:

- Concise
- Understandable
- In keeping with the “wholemeal” programming style, which operates on entire data structures at once

```
foo :: String -> String
foo s = length (filter (== ‘C’) (toUpper s))

foo = length . filter (== ‘C') . toUpper
```

Higher-order functions

Composition helps write code that is:

- Concise
- Understandable
- In keeping with the “wholemeal” programming style

```
foo :: String -> String
foo s = length (filter (== ‘C’) (toUpper s))

foo = length . filter (== ‘C') . toUpper
```

Partial application

```
add :: Int -> Int -> Int
add x y = x + y
```

All functions in Haskell take one argument*!

Partial application

```
add :: Int -> Int -> Int
add x y = x + y
```

All functions in Haskell take one argument*!

```
add :: Int -> (Int -> Int)
```

```
add 0 :: Int -> Int
```

Partial application

```
add :: Int -> Int -> Int
add x y = x + y
```

All functions in Haskell take one argument*!

```
add :: Int -> (Int -> Int)
```

```
add 0 :: Int -> Int
```

Note: `(Int -> Int) -> Int` is **not** the same as `Int -> Int -> Int`.
`->` is *right-associative*.

Partial application

`->` is right-associative.

`Int -> Int -> Int` is the same as `Int -> (Int -> Int)`

Function *application* is left-associative.

`add 0 1` is the same as `(add 0) 1`

Partial application

Note: Haskell lets us write anonymous functions like `\x y z -> …`

This is just syntactic sugar for `\x -> (\y -> (\z -> …))`.

Similarly, `add x y = …` is just syntactic sugar for `add = \x -> (\y -> …)` .

Currying

```
add :: (Int, Int) -> Int
add (x,y) = x + y

add :: Int -> Int -> Int
add x y = x + y
```

These are equivalent! We call going from the first to the second *currying*, after Haskell Curry, and the reverse *uncurrying*.

Currying makes partial application easier.

Partial application

When writing functions, consider:

Which argument are you most likely to want to partially apply with?

`filter f xs` – we often want to filter multiple lists using the same criterion

`filter xs f` – we rarely want to filter the same original list using multiple criteria

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

```
foo :: String -> String
foo xs = map toUpper xs
```

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

```
foo :: String -> String
foo xs = map toUpper xs

foo = \xs -> map toUpper xs
```

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

```
foo :: String -> String
foo xs = map toUpper xs

foo = \xs -> map toUpper xs

foo = map toUpper
```

Prefix Operators

```
add :: Int -> Int -> Int
```

- `add 0 1` – used as a prefix
- ``0 `add` 1`` – used as an infix

Infix Operators

`(+) :: Int -> Int -> Int`

- `(+) 0 1` – used as a prefix
- `0 + 1` – used as an infix

Haskell lets us use *operator sections*, i.e., partially apply these on either side:

- `map (+ 1) xs`
- `filter (0 <) xs`

Operator sections

```
div :: Int -> Int -> Int

(-) :: Int -> Int -> Int

(>) :: Int -> Int -> Bool
```

`(0 -)` – argument is on the left, so we know it's the first argument

`(> 100)` – argument is on the right, so we know it's the second argument

`div 8` – no sides to distinguish; we have to interpret it as the first argument