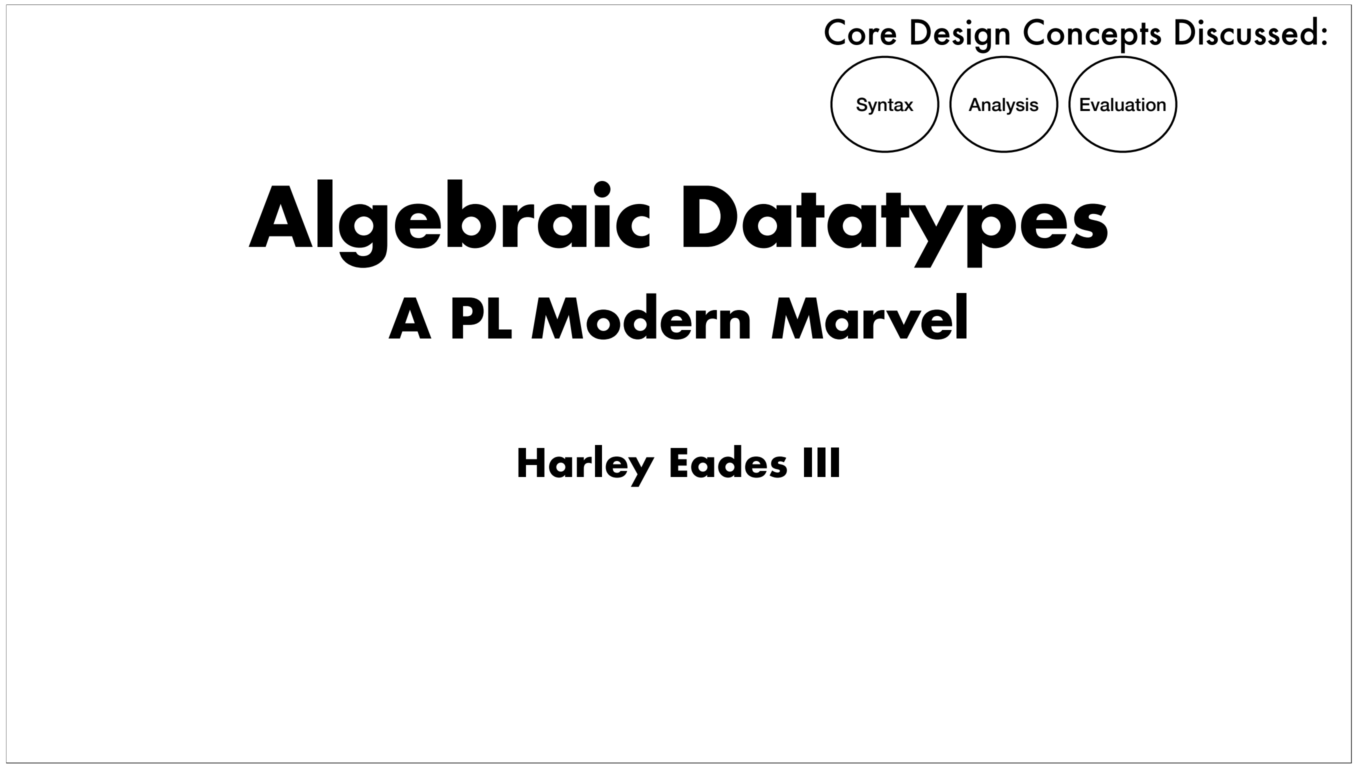


Core Design Concepts Discussed:

Syntax Analysis Evaluation

Algebraic Datatypes

A PL Modern Marvel

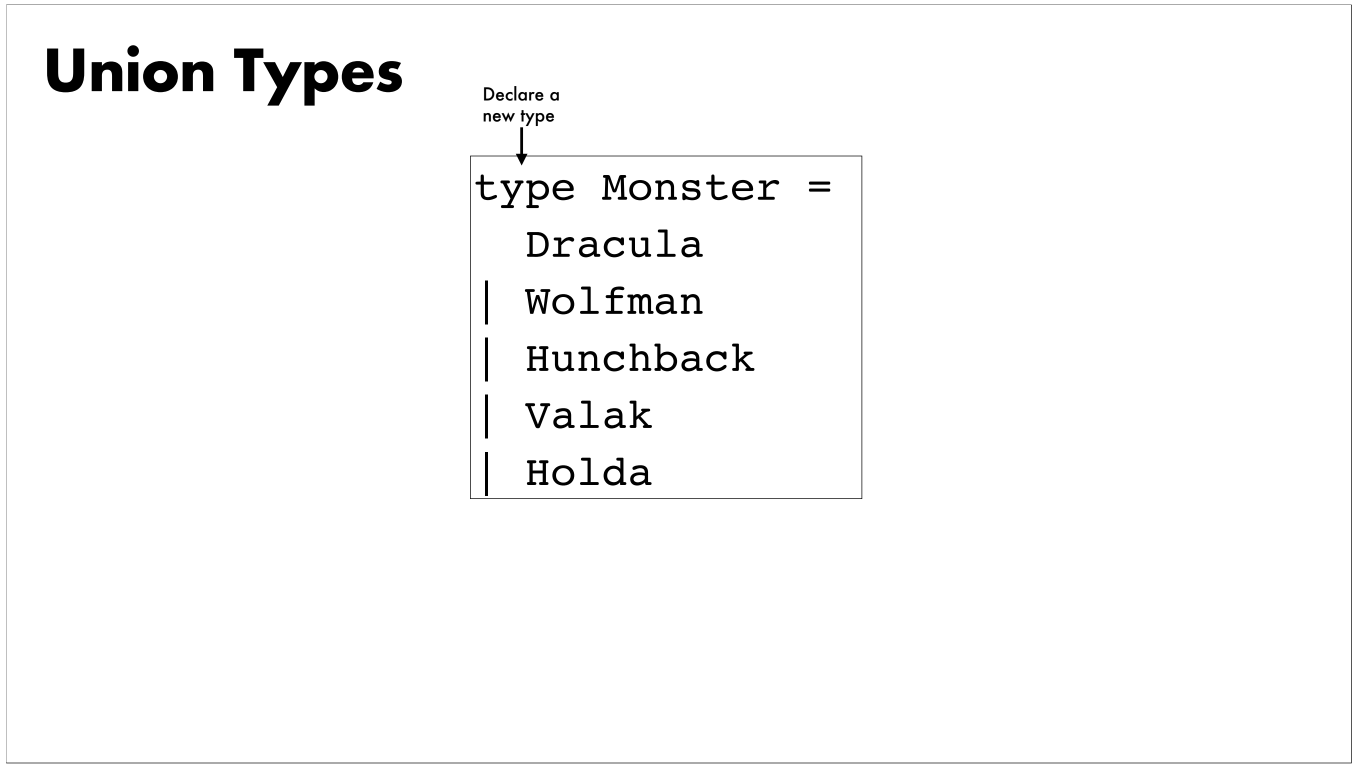
Harley Eades III

We now turn out attention to one of the coolest features of modern programming languages: Algebraic Datatypes. They allow us to define our own types in an elegant, simple, but powerful way. First, let's consider the simplest form...

Union Types

```
type Monsters =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
```

Here I have defined a new type...

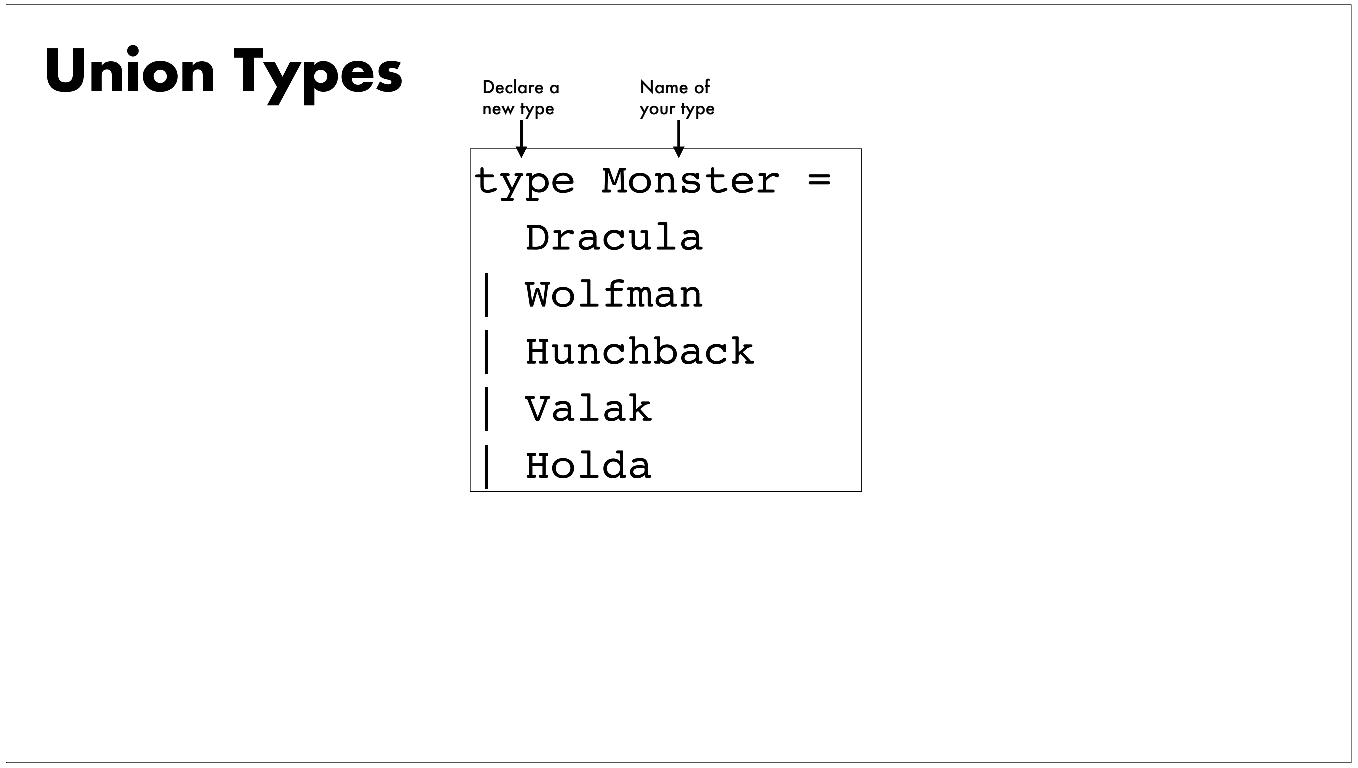


Union Types

Declare a new type

```
type Monster =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
```

named Monster...



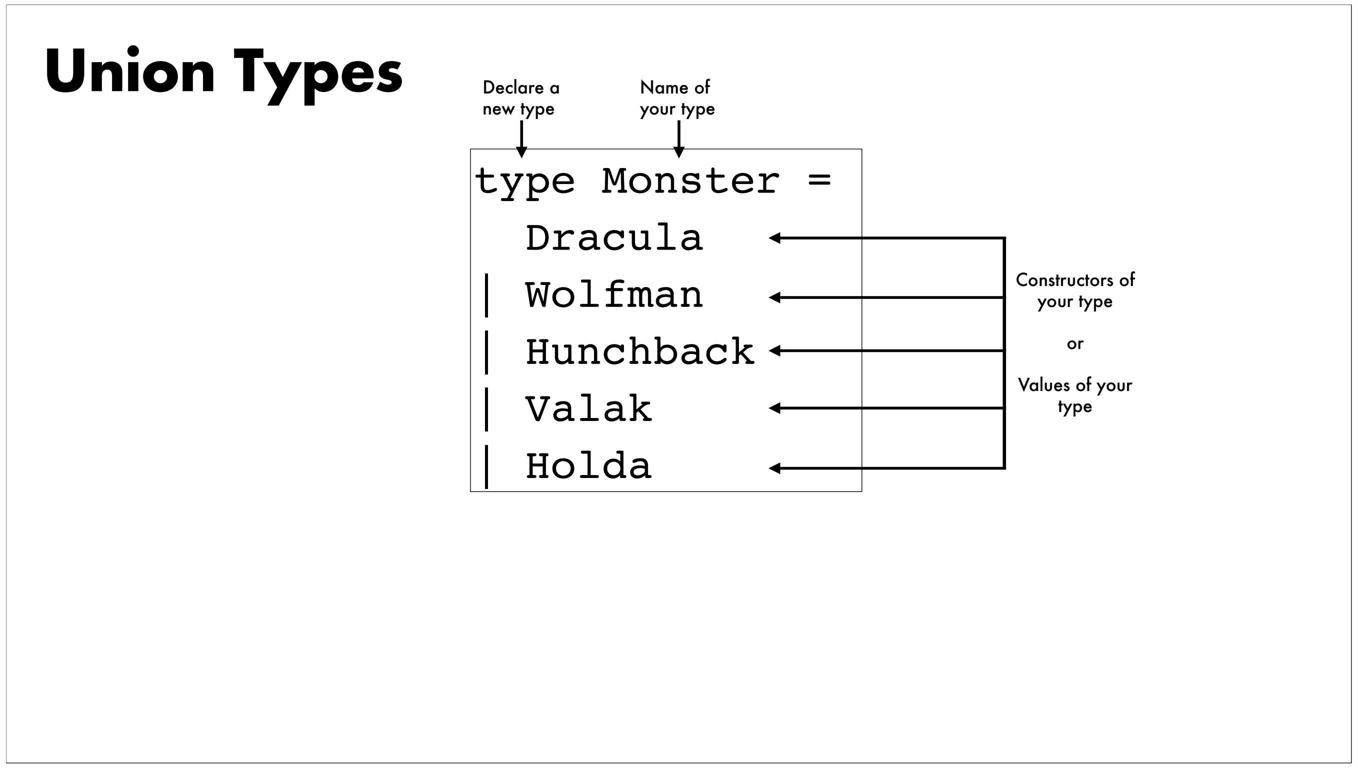
Union Types

Declare a new type

Name of your type

```
type Monster =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
```

Then the type is defined to have five values we call...

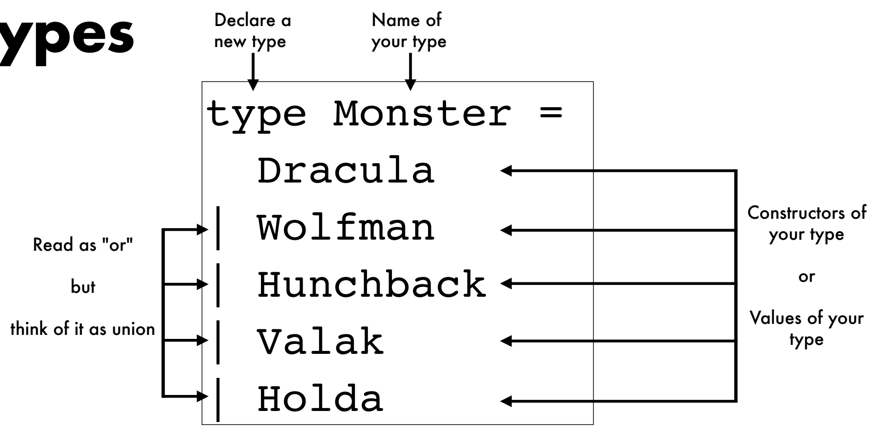


Union Types

```
type Monster =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
```

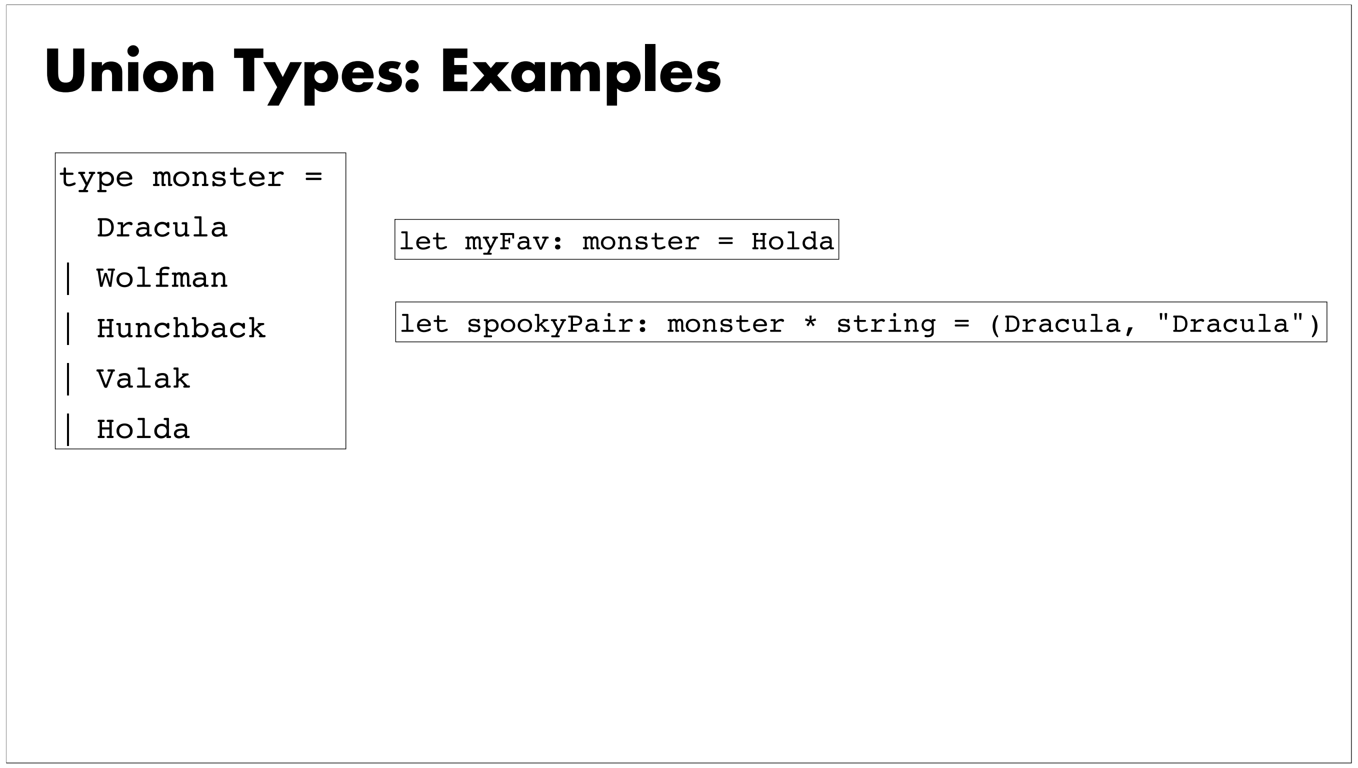
constructors. These are the only values with type Monster. We define each constructor using an upper-case first letter and separate them using....

Union Types



```
type Monster =
   Dracula
 | Wolfman
 | Hunchback
 | Valak
 | Holda
```

An "or" (the pipe), but you should semantically think of the pipe as a union. This is where the name "Union Types" comes from. Informally, we can think of Monster as a set with five elements in the set: Dracula, Wolfman, Hunchback, Valak, and Holda...



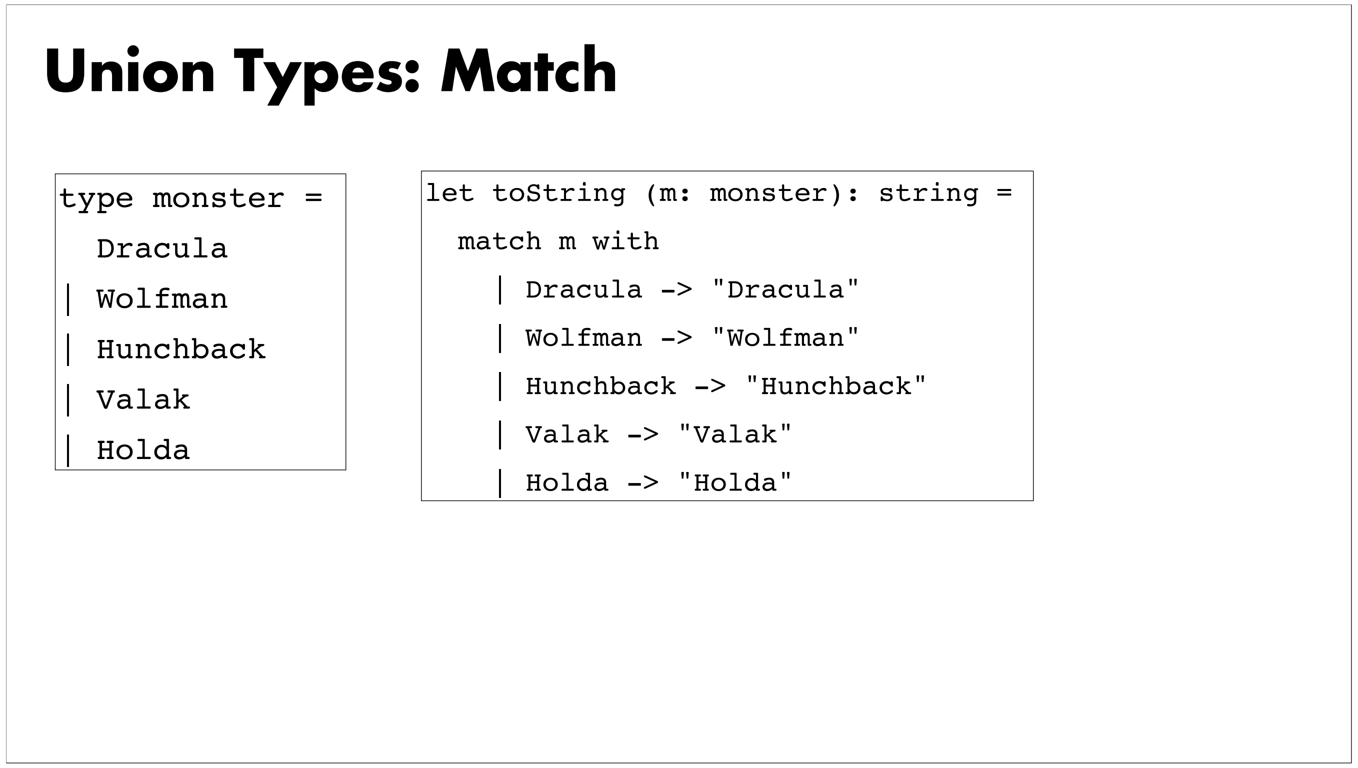
Union Types: Examples

```
type monster =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
```

```
let myFav: monster = Holda
```

```
let spookyPair: monster * string = (Dracula, "Dracula")
```

Consider a few examples. Here I define myFav of type monster whose value is Holda. Then I define spookyPair as a tuple of a monster and a string. But, what if I want to inspect a value of type Monster?...



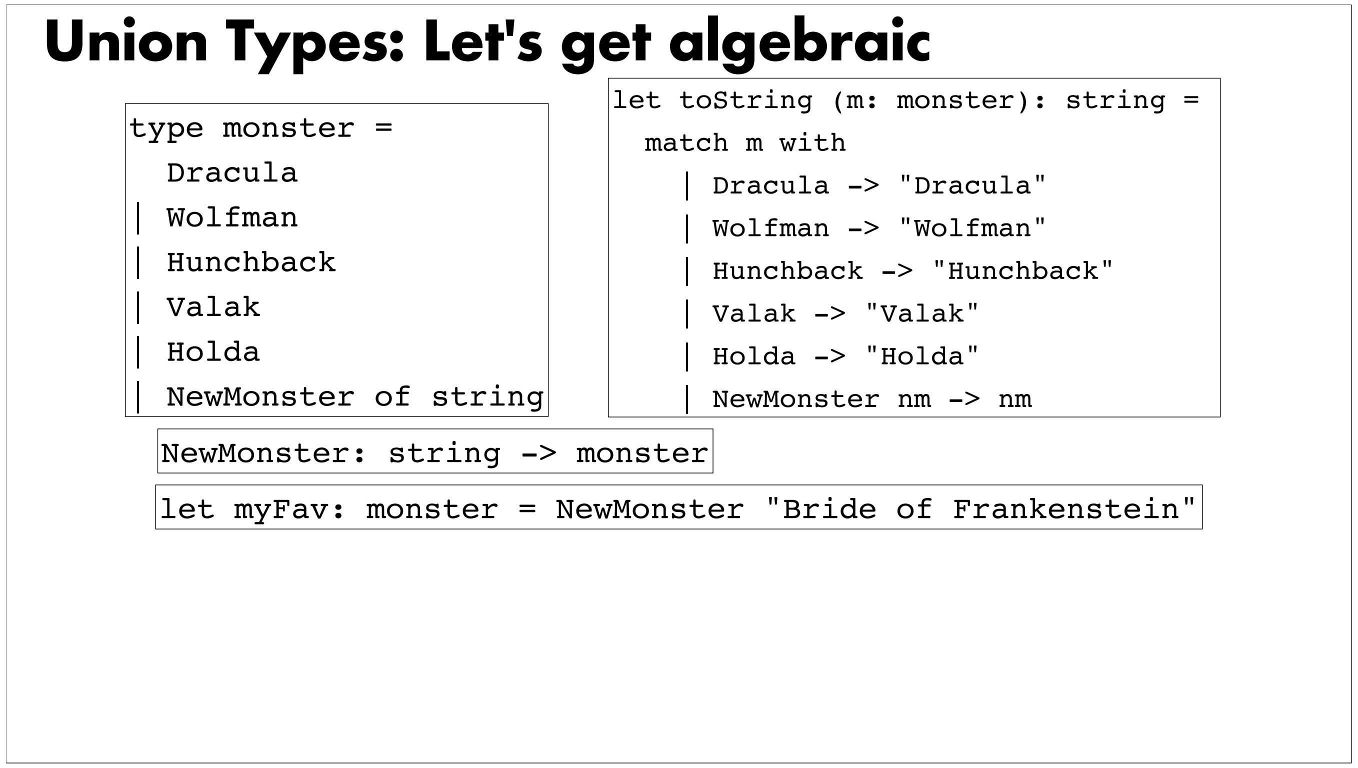
Union Types: Match

```
type monster =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
```

```
let toString (m: monster): string =
  match m with
    | Dracula -> "Dracula"
    | Wolfman -> "Wolfman"
    | Hunchback -> "Hunchback"
    | Valak -> "Valak"
    | Holda -> "Holda"
```

We can do this with the match-statement. It's name is short for pattern match. We simply match on the input m and for each possible constructor we define how the function should operate.

So far, what we've introduced are what we call enums in most C-based languages like C#, but algebraic datatypes are SO MUCH MORE POWERFUL!



Union Types: Let's get algebraic

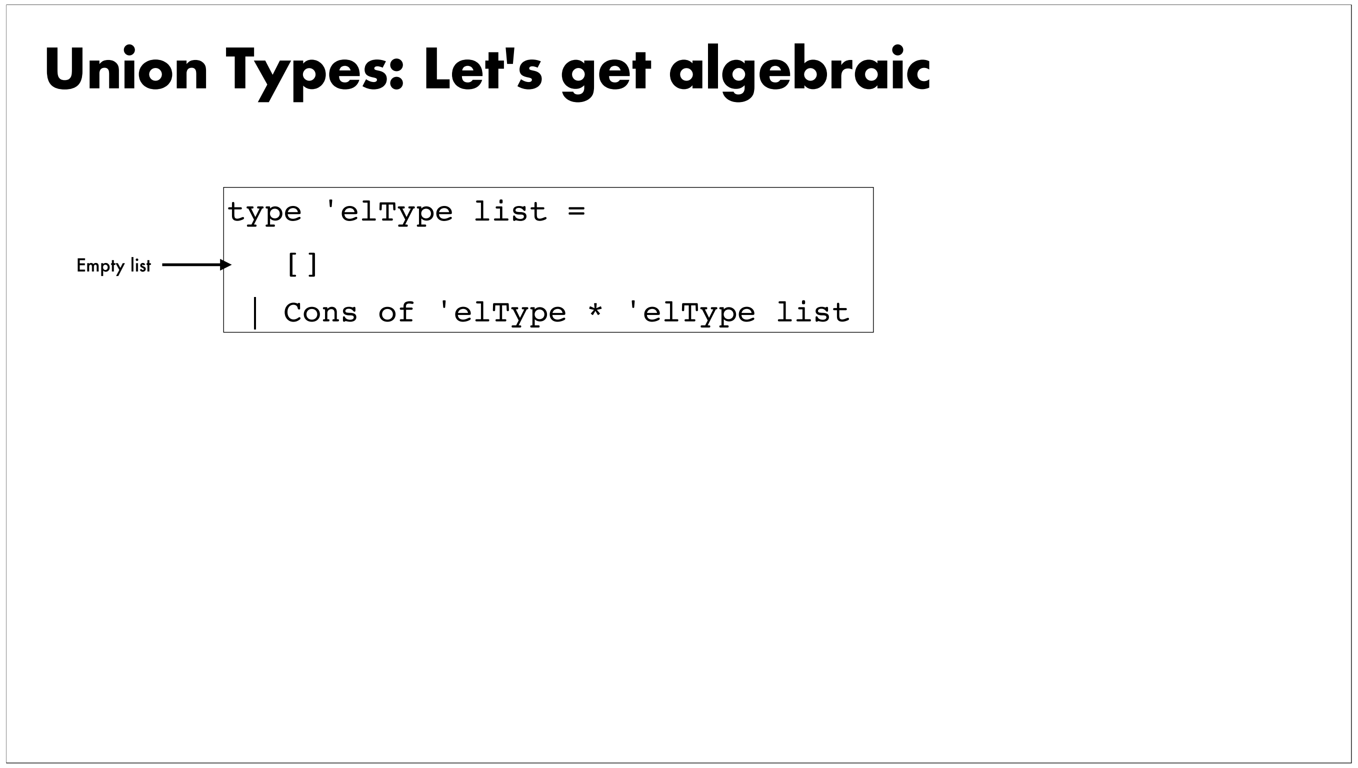
```
type monster =
  Dracula
| Wolfman
| Hunchback
| Valak
| Holda
| NewMonster of string
```

```
let toString (m: monster): string =
  match m with
    | Dracula -> "Dracula"
    | Wolfman -> "Wolfman"
    | Hunchback -> "Hunchback"
    | Valak -> "Valak"
    | Holda -> "Holda"
    | NewMonster nm -> nm
```

```
NewMonster: string -> monster
```

```
let myFav: monster = NewMonster "Bride of Frankenstein"
```

Here I've extended monster with a new constructor called NewMonster who takes an argument of type string. This makes it so we don't have to know every monster right away, and allow us to extend the monster type programmatically. As you can see the variable myFav is now a NewMonster which is applied to "Bride of Frankenstein". When we pattern match on a monster we now get a variable for the name of the new monster called nm. Here I simply output that name...

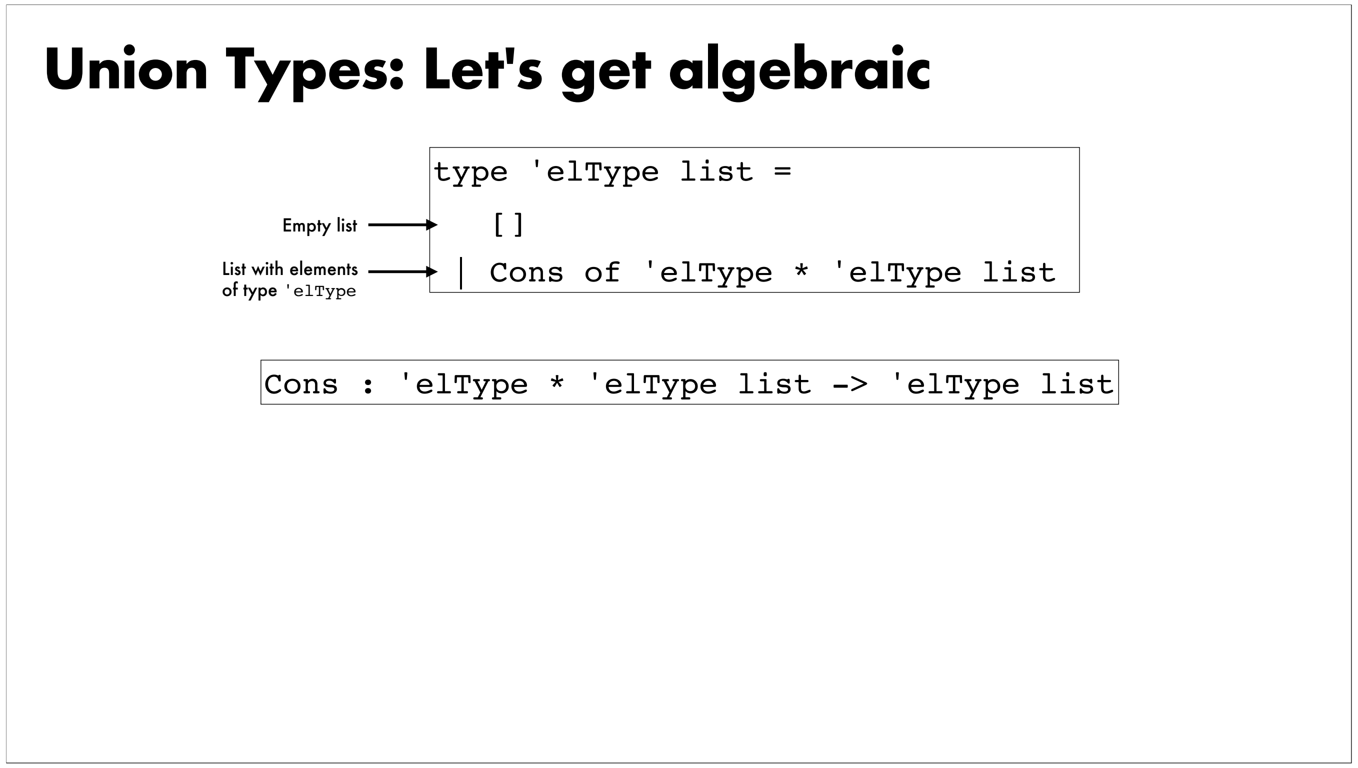


Union Types: Let's get algebraic

Empty list

```
type 'elType list =
   []
 | Cons of 'elType * 'elType list
```

Let's take it a bit further. Here I have the type of lists whose empty list is open and close square brackets. Then to add elements to some list we use the Cons....



Union Types: Let's get algebraic

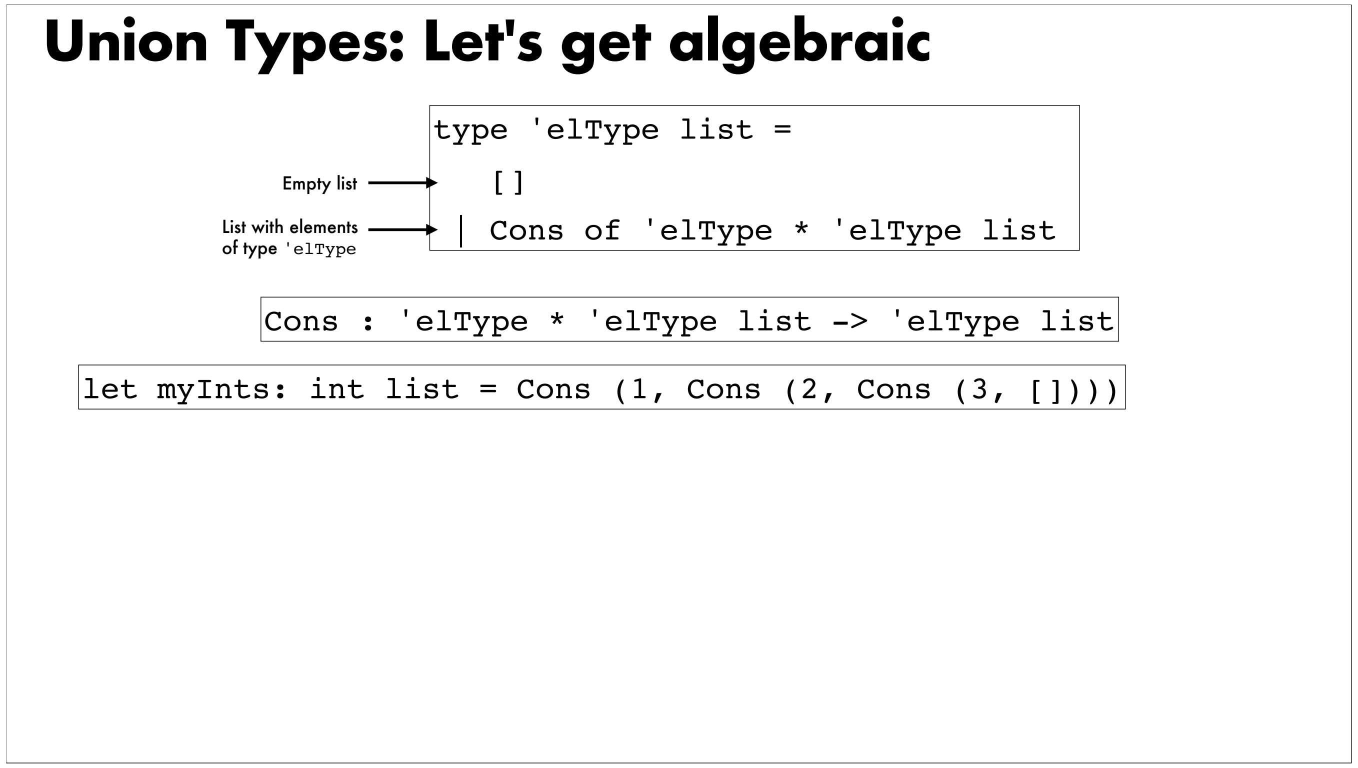
Empty list

List with elements of type `'elType`

```
type 'elType list =
   []
 | Cons of 'elType * 'elType list
```

```
Cons : 'elType * 'elType list -> 'elType list
```

constructor. This constructor takes two parameters one of type `elType which is the type of the elements. Notice this type is a parameter to the list type itself. This is polymorphic, and `elType is indeed a type variable. We use cons to add elements to a list we already have, for example...



Union Types: Let's get algebraic

Empty list

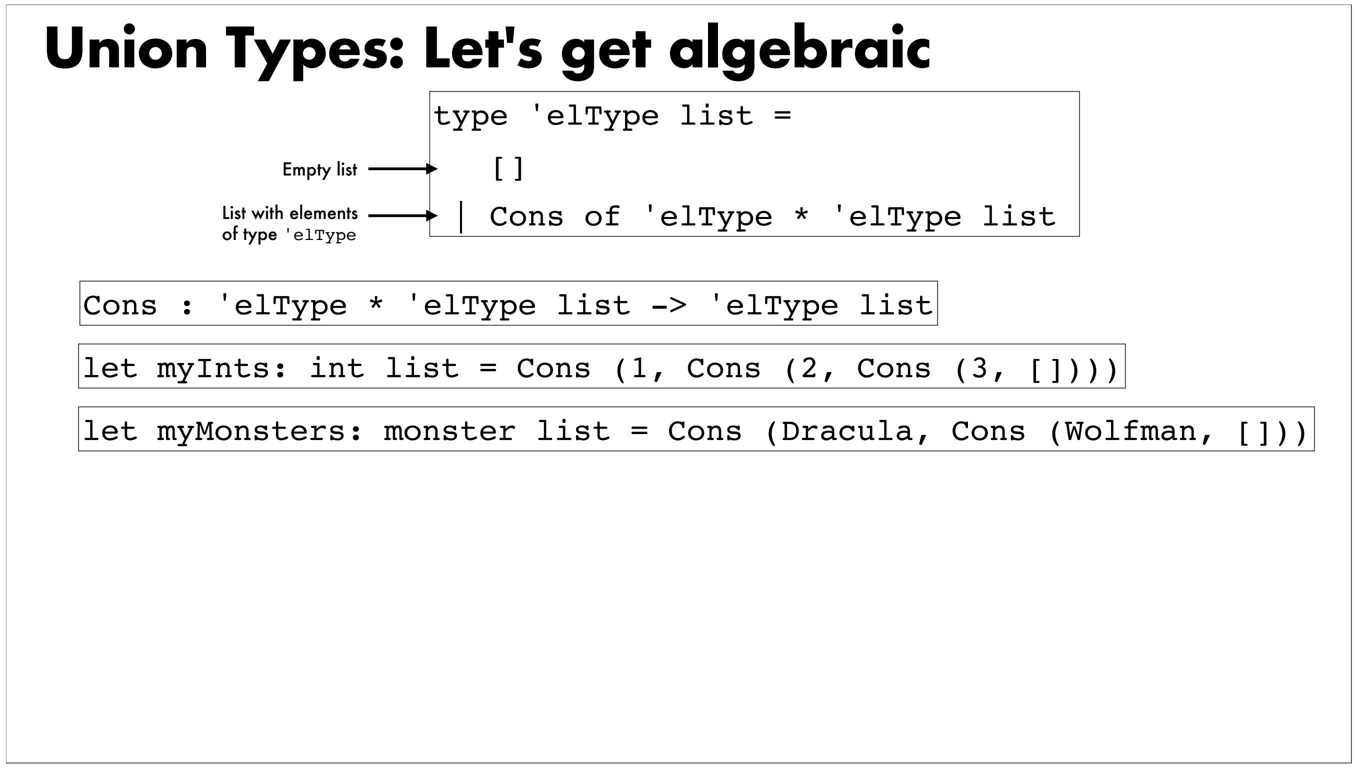
List with elements of type `'elType`

```
type 'elType list =
   []
 | Cons of 'elType * 'elType list
```

```
Cons : 'elType * 'elType list -> 'elType list
```

```
let myInts: int list = Cons (1, Cons (2, Cons (3, [])))
```

We can construct the list of ints called MyInts. Which consists of the list containing 1, 2, and 3. Notice that the third application of Cons is applied to 3 and the empty list....



Union Types: Let's get algebraic

Empty list

List with elements of type `'elType`

```
type 'elType list =
   []
 | Cons of 'elType * 'elType list
```

```
Cons : 'elType * 'elType list -> 'elType list
```

```
let myInts: int list = Cons (1, Cons (2, Cons (3, [])))
```

```
let myMonsters: monster list = Cons (Dracula, Cons (Wolfman, []))
```

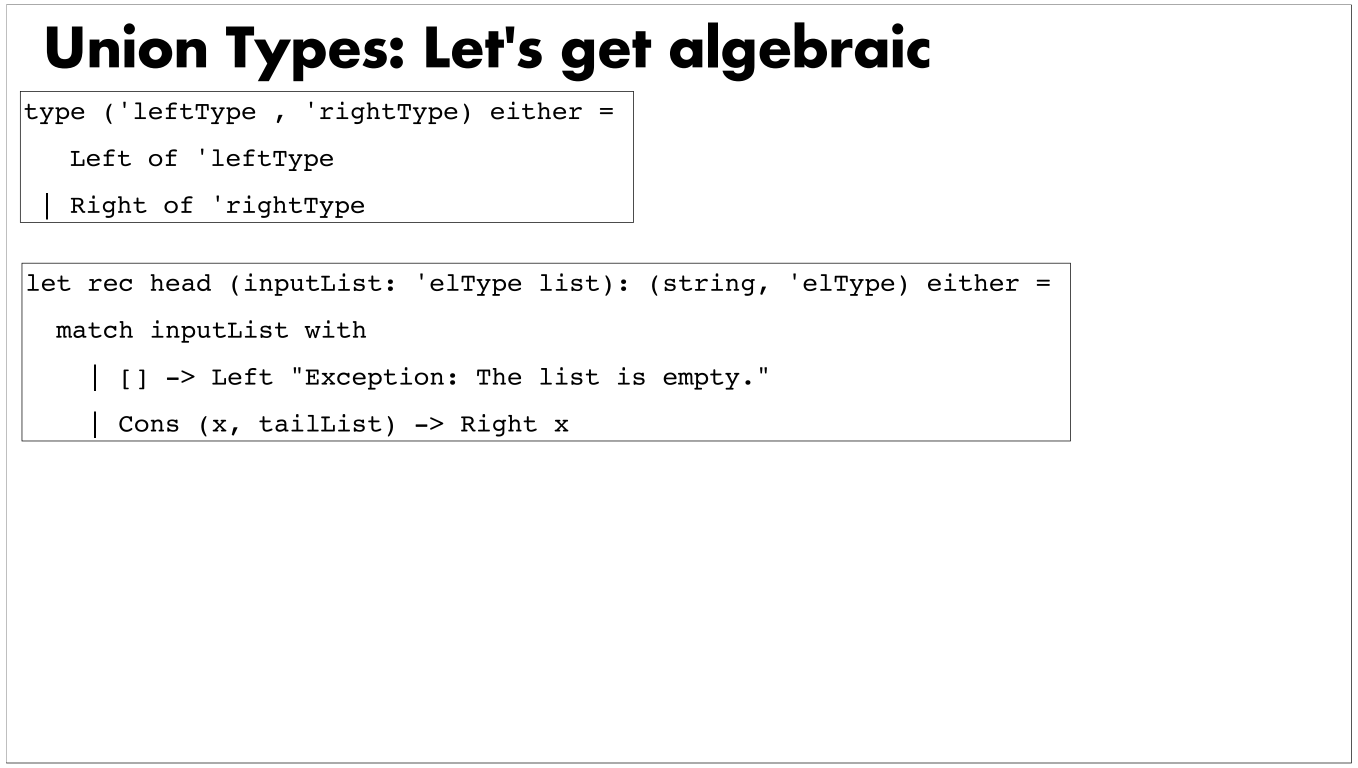
A second example is the list of myMonsters which has elements Dracula and Wolfman. Notice that in both examples, the type parameter to list is different based on what type of elements we want to have in the list....

Union Types: Let's get algebraic

```
type 'elType list =
   []
 | Cons of 'elType * 'elType list
```

```
let rec length (inputList: 'elType list): int =
  match inputList with
    | [] -> 0
    | Cons (x, tailList) -> 1 + length tailList
```

When we pattern match on a list we have cases for empty and for cons, and in the case of cons, we get a bound variable, here I named it x, for the head of the list and a bound variable containing the tail of the list, here I called it tailList. In this example, I calculate the length of a list which simply counts the number of elements in the list...



Union Types: Let's get algebraic

```
type ('leftType , 'rightType) either =
   Left of 'leftType
 | Right of 'rightType
```

```
let rec head (inputList: 'elType list): (string, 'elType) either =
  match inputList with
    | [] -> Left "Exception: The list is empty."
    | Cons (x, tailList) -> Right x
```

Here is an interesting example of using two type parameters. The type either is the disjoint union of the types 'leftType and 'rightType. We have one constructor, Left, who has a parameter of 'leftType and one constructor, Right, who has a parameter of type 'rightType. This is a useful type of signally an error, for example, in this example I define head which simple returns the first element of a list, but in the case of the empty list we have no such element, and thus, we have nothing to return, so in this case, I return Left signaling an error with a string stating the type of exception. So that's algebraic data types. We will do some examples during class. Read chapter 10.