Functions with Records

In this lesson, we will see how functions can interact with the record data structure.

Extracting Values from Records

A function can be used to retrieve or modify field values from a record. As long as the implementation of our function is correct, it will automatically detect the type of the record and access the specified field.

Let's write a function which retrieves the *name* and *age* from the `wizardInfo` record type we created earlier.

```
type wizardInfo = {
  name: string,
  age: int,
  school: string,
  house: string
};

let wizard = {
  name: "Harry",
  school: "Hogwarts",
  house: "Gryffindor",
  age: 14
};

let getName = (wizard: wizardInfo) => (wizard.name, wizard.age);

Js.log(getName(wizard));
```

For documentation purposes, we annotated the type in the `wizard` argument. The function simply returns the `name` and `age` properties as a tuple!

Nested Records

Functions can also work with nested records. All we have to do is access the nested field using pattern matching or the `.` operator.

Here's an example which retrieves the *house* from a `wizard` record:

```
type schoolInfo = {
  school: string,
  house: string
};

let schoolInfo: schoolInfo = {
  school: "Hogwarts",
  house: "Gryffindor"
};

type wizardInfo = {
  name: string,
  age: int,
  schoolInfo: schoolInfo
}

/* A nested record */
let wizard: wizardInfo = {
  name: "Harry",
  schoolInfo,
  age: 14
};


/* Accessing the house field through a function */
let getHouse = (wizard: wizardInfo) => {
  let {schoolInfo: {house}} = wizard;
  house;
}

Js.log(getHouse(wizard));
```

To make the function above even more concise, we can simply pass the pattern as the argument to the function:

```
let getHouse = ({schoolInfo: {house}}) => house; /* Compiler automatically infers type */

Js.log(getHouse(wizard));
```

Creating Records

This shouldn’t sound like a very complex process now. We can pass a tuple as the argument to a function, and convert it to a record by giving field names to its components:

```
type superhero = {
  realName: string,
  heroName: string
};

let createHero = (real, hero) => {
  realName: real,
  heroName: hero
};

let clark = "Clark Kent";
let superman = "Superman";

let superman = createHero(clark, superman);

Js.log(superman);
```

We could further simplify this with **punning**. Additionally, creating a tuple for the two arguments above would reduce the number of arguments in our function:

```
type superhero = {
  realName: string,
  heroName: string
};

let createHero = ((realName, heroName)) => {
  /* Punning */
  realName,
  heroName
};


let hero = ("Clark Kent", "Superman");

let superman = createHero(hero);

Js.log(superman);
```

In the next lesson, we'll see how functions can act as values in Reason.