

# CS 51

## CODE REVIEW 1

SAMUEL GREEN AND GABBI MERZ

### 1. TOPICS

- Record Types and Algebraic Data Types
- Currying and Partial Application\*
- Option Types and Error Handling\*
- From last week:
  - Higher order functions
  - Anonymous functions
  - General syntax: `match`, `let`, `in`

### 2. RECORDS

Records allow us to combine related pieces of data into one unified structure. Imagine we wanted to model data for a student. A record could be a good choice, because there are multiple different views of different types, and they don't have a natural ordering:

```
# type student = {  
  name : string;  
  email : string ;  
  huid : int ;  
  enrolled : bool  
} ;;
```

Here's an example of a value of this type:

```
# let sam = {  
  name = "samuel";  
  email = "s@muel.green";  
  huid = 51515151;  
  enrolled = false  
} ;;  
  
val sam : student =
```

```
{name = "samuel"; email = "s@muel.green"; huid = 51515151;
  enrolled = false}
```

To pull a field out of a record, we can use pattern matching, field punning, or dot notation. There isn't a "right" choice for which of these syntaxes should be preferred – I'd recommend relying on the one that makes your code the most readable.

### 2.1. Pattern Matching.

```
let get_huid1 (stu : student) =
  let { huid = h; _ } = stu in h ;;
```

```
get_huid1 sam ;;
```

In this case, the identifier `h` is bound to the value in the `huid` field of the record.

**2.2. Field Punning.** With a dryly appropriate name, field punning allows for direct access of record fields by name in pattern matches. For example, this is equivalent to the function above:

```
# let get_huid2 (stu : student) =
  let { huid ; _ } = stu in huid ;;
val get_huid2 : student -> int = <fun>
# get_huid2 sam ;;
- : int = 51515151
```

As with tuples, we can move this destructuring into the argument:

```
# let get_huid3 ({ huid; _ } : student) =
  huid ;;
val get_huid3 : student -> int = <fun>
# get_huid3 sam ;;
- : int = 51515151
```

**2.3. Dot Notation.** Finally, there's standard dot notation, which is probably most appropriate when accessing only 1 field of a record. In the case of our running example:

```
# let get_huid4 (s : student) = s.huid ;;
val get_huid4 : student -> int = <fun>
# get_huid4 sam ;;
- : int = 51515151
```

## 3. RECORD TYPES

Algebraic data types are great, but what if we want to model a system or some data that we need to constraint to a fixed set of different values? Imagine, for

example, that we wanted to add a field to the student record to record the student's enrollment type.

We could start by storing this data as a string, like this:

```
# type student_with_type = {
  email : string ;
  enrollment_type : string
} ;;
type student_with_type = { email : string; enrollment_type : string; }
```

By convention, we could then say that a student is either "FAS", "DCE", or "None". We can construct an valid value like this:

```
# let new_sam = { email = "HUID";
                  enrollment_type = "HEAD TF, NOT STUDENT" };;
val new_sam : student_with_type =
  {email = "HUID"; enrollment_type = "HEAD TF, NOT STUDENT"}
```

Wait a second! This doesn't meet our convention for values of the student\_with\_type type, but it's still valid, according to OCaml.

```
# new_sam ;;
- : student_with_type =
{email = "HUID"; enrollment_type = "HEAD TF, NOT STUDENT"}
```

We solve this problem by with Algebraic Data Types (ADTs), which limit the space of possible values. There are two types of ADTs: Cartesian products (i.e. tuples) and sum types (called variants). Here's an example that uses both:

```
# type enrollment = Faculty of string * int |
                  FAS | DCE | Not ;;
type enrollment = Faculty of string * int | FAS | DCE | Not
```

In this case, the Faculty variant is used to construct named tuples of a string and an int. The other variant create values that represent those values distinct, such as:

```
# let shieb : enrollment = Faculty ("Prof. Shieber", 1) ;;
val shieb : enrollment = Faculty ("Prof. Shieber", 1)
# let sam : enrollment = FAS ;;
val sam : enrollment = FAS
# let reagan : enrollment = DCE ;;
val reagan : enrollment = DCE
# let goodell : enrollment = Not ;;
val goodell : enrollment = Not
```

We can embed ADTs in records, name records in ADTs, and take advantage of all the other value manipulation syntax (`match`, `let`, type signatures) that we've see so far. Example:

```
# let is_shieber (enr : enrollment) : bool =
  match enr with
  | Faculty (name, _) -> name = "Prof. Shieber"
  | _ -> false ;;
val is_shieber : enrollment -> bool = <fun>
# is_shieber shieb ;;
- : bool = true
```

3.1. **Recursive ADTs.** A final note is that ADTs can be recursive. For example:

```
# type partner = Two of student * partner | One of student ;;
type partner = Two of student * partner | One of student
```

This would allow us to represent students partnerships and also demonstrates how to use records in ADTs.

#### 4. EXERCISES

**Exercise 1.** *Improve the original definition of `student_with_type` data type. Why is the old version bad and the new version better?* □

**Exercise 2.** *Write a data type `counts` to represent counting by one. Then write a function that takes a number and returns its `int` representation.* □

**Exercise 3.** *Here's the solution to `curry`. Re-write it without using the syntactic sugar of multiple arguments, and give the complete type of the identifier `curry`.* □

```
let curry f x y = f (x, y) ;;
```

**Exercise 4.** *Write a function `mapfilter` that takes a list and two functions, `pred` and `transf`. `mapfilter` should return a list containing the result of applying `transf` to every element for which `pred` is true, and removing the elements for which `pred` is false.*

- (1) *write the type of `mapfilter`.*
- (2) *write `mapfilter` without using the `List` module.*
- (3) *re-write `mapfilter` using the `List` module.*
- (4) *just for practice, write `mapfilter` so that it returns `None` if the resulting list is empty. Is this a necessary way to handle errors here?*

□