

CS 51

CODE REVIEW 3

SAMUEL GREEN AND GABBI MERZ

CONTENTS

1. Invariants	1
2. Modeling Choices	2
3. Concise and Field Punning: <code>valid_date</code>	3
4. Abstraction: <code>count_people</code> and the <code>List</code> module.	4
5. Nested Recursion: <code>find_parents</code>	5
6. Problem Set 3: Tips ‘n’ Tricks	6

Note that there are *no* new syntax here – we’re just putting previous concepts and syntax together and thinking about modeling tradeoffs and other abstract concepts!

1. INVARIANTS

Definition 1. An *invariant* is a condition that is assumed to true of any value of a particular type, but that is not necessarily enforced by the compiler.

Lab 3 and Problem Set 3 focus significantly on the design and enforcement of invariants as a useful mechanism for programming in OCaml.

What was the invariant of the RGB variant of the `color` type in Lab 3?

Did the choice of the type help us enforce the invariant? What burden does this place on us as programmers? Could a different type definition have helped? Does the inclusion of `color_label` help for any specific reason?

2. MODELING CHOICES

Because we have to autograde the labs and problem sets, there doesn't end up being a lot of space to exercise your modeling muscles when it comes to representing data in OCaml. Did you agree with the modeling choices we made in the lab?

Consider for example the following type for color:

```
# type color_label = Orange | Brown | Green ;;
type color_label = Orange | Brown | Green
# type color = Simple of color_label | RGB of int * int * int ;;
type color = Simple of color_label | RGB of int * int * int
```

Why did choose this definition, particularly of RGB? What do you think of this redefinition?

```
# type rgb = { r : int ; g : int ; b : int } ;;
type rgb = { r : int; g : int; b : int; }
# type color = { simple : color_label option; rgb : rgb option } ;;
type color = { simple : color_label option; rgb : rgb option; }
```

Is this example better or worse? Is there something better or worse? Is there an example that's similarly "good" to the first version but that uses a record type instead?

3. CONCISE AND FIELD PUNNING: VALID_DATE

Here's a long version of the `valid_date` function. What do you think of its design and style?

```
# let valid_date (d : date) : date =
  if d.year <= 0 then raise (Invalid_Date "only positive years") else
  if d.month = 1 || d.month = 3 || d.month = 5 || d.month = 7 || d.month = 8 ||
    d.month = 10 || d.month = 12 then
    (if d.day > 31 then raise (Invalid_Date "too many days") else
     if d.day < 1 then raise (Invalid_Date "days must be >1") else
     d) else
  if d.month = 4 || d.month = 6 || d.month = 9 || d.month = 11 then
    (if d.day > 30 then raise (Invalid_Date "too many days") else
     if d.day < 1 then raise (Invalid_Date "days must be >1") else
     d) else
  if d.month = 2 then
    (if d.year mod 4 = 0 && d.year mod 100 <> 0 || d.year mod 400 = 0 then
      if d.day > 29 then raise (Invalid_Date "too many days") else
      if d.day < 1 then raise (Invalid_Date "days must be >1") else
      d
    else
      if d.day > 28 then raise (Invalid_Date "too many days") else
      if d.day < 1 then raise (Invalid_Date "days must be >1") else
      d) else
  raise (Invalid_Date "bad month") ;;
```

What are some first observations about this code, perhaps some easy simplifications that could be made?

4. ABSTRACTION: COUNT_PEOPLE AND THE LIST MODULE.

Recall the family tree modeling problem from the Lab. (Aside: did you like the model? Did you agree with the type definitions?)

Here's a definition of the function `count_people` from the lab:

```
let rec count_people (fam : family) : int =  
  match fam with  
  | Single _ -> 1  
  | Family (_, _, fl) -> 2 + count_people fl ;;
```

Great. Right? (What's the error here?)

How can we improve this implementation?

5. NESTED RECURSION: `FIND_PARENTS`

6. PROBLEM SET 3: TIPS ‘N’ TRICKS

- Keep your eyes open for folding opportunities. They can appear where you least expect them and make for very slick code!
- Built in operators are powerful, and OCaml knows a lot about how to manipulate values. The `Pervasives` module has all of the secrets. For example, do you know the type of the `(=)` function? How about other comparators?

```
# ( = ) ;;
```

```
- : 'a -> 'a -> bool = <fun>
```

- Make conscious choices about what method of record field accessing you use. Using the “easiest” can make your code much more complicated than necessary!
- Be deliberate about understanding the `times` function and the algorithm **before** you start to code: the code will only get in the way.
- Appropriately scope helper functions. *If you write a helper function that is only ever called within one other function it should be scoped as such.*
- Make sure to read and understand the style guide.