CSE 341
Section 5

HW2 Debrief, Currying, Modules

Agenda

- HW2 Debrief
- Currying
- Modules
- Q&A

Homework 2 Recap

- If-then-else vs. case expression
  - If-then-else is preferred:
    ```
    match x with
    0 => "zero"
    | _ => "not-zero"
    ```
  - Case statement is preferred:
    ```
    if null xs
    then "empty"
    else if null (tl xs)
    then "one elt"
    else "more than one elt"
    ```

Homework 2 Recap

- Wildcards
  - Use wildcards when we don’t use the value in the pattern
    ```
    match arith with
    Const x => Const 1 (* we don’t use x! *)
    | Mult(x, y) => Const x (* we don’t use y! *)
    ```

Key Concepts Review

- Currying
  - Have a function take the first conceptual argument and return another function that takes the second conceptual argument and so on.

- Modules
  - A powerful tool for enforcing abstraction and safety
  - Keep type representation opaque to outside client => guaranteed that invariants are protected

Currying

Recall every ML function takes exactly one argument
Currying

Currying is particularly convenient for creating similar functions with iterators. Here is a curried version of a fold function for lists:

```ml
fun fold f t = fold (fn (x,y) => x+y) 0 xs
```

Now we could use this fold to define a function that sums a list elements like this:

```ml
fun sum1 xs = fold (fn (x,y) => x+y) 0 xs
```

But that is unnecessarily complicated compared to just using partial application:

```ml
val sum2 = fold (fn (x,y) => x+y) 0
```

Let's practice! (a), (b), (e), (i) on Worksheet

Modules

- Can group bindings into separate modules
- Good for maintaining invariants by hiding implementation details from client
  ```ml
  structure MyModule = struct
  bindings
  end
  ```
- Inside a module, can use earlier bindings as usual
  - Can have any kind of binding (val, datatype, exception, ...)
- Outside a module, refer to earlier modules' bindings via `ModuleName.bindingName`
- Just like `List.foldl` and `Char.toLower`, now you can define your own modules

Remember: `structure Foo :> BAR` is allowed if Foo provides:
- every non-abstract type in BAR (as specified)
- every abstract type in BAR (in some way)
- every val-binding in BAR (can have more general types)
- every exception in BAR

Foo can also define things that are not defined in BAR!

Let's practice! (a) on Worksheet

Question Time

Feel free to ask questions about material, review questions, etc.