CSE305 Week 4: Structural Types (in OCaml) (plus some material on compilation)

Patterns in OCaml (carried over from last week, minus a couple lines)

Let's dive right into the grammar rules before showing how examples conform to them. Again we give just a subset of the rules at https://v2.ocaml.org/manual/patterns.html#start-section:

```
PAT ::=  <vname>  |  _  |  <constant>  |  PAT as <vname>  |  (PAT [: TYPEXP])
      |  PAT | PAT
      |  <cname> PAT
      |  PAT {, PAT }+                     (tuple pattern)
      |  [ PAT {; PAT}[;] ]                (pattern for fixed-size list)
      |  PAT :: PAT                        (pattern to handle general-size list)
      |  [| PAT {; PAT}[;] |]              (pattern for fixed-size "value array")
      |  <char> .. <char>
```

The first line of options are like those we had with ordinary expressions: constant and variable (lowercase) are options, but now also _ for wildcard. The second line says a pattern can have internal BNF-like alternatives and they are the outermost/loosest/highest "operators" in any pattern.

In the third line, <cname> mostly means a ***capitalized*** identifier name. Those come from user-defined type constructors, which includes classes but more primitive stuff first (next week).

The next four lines are the bread-and-butter tuple and list patterns, plus one for arrays. One technical note: The empty list `[]` is classed as a `<constant>`. So is the empty array `[||]`. Note that if we had written the rule for list patterns as `[ {PAT ;} ]` it would have suggested that you could put space between the brackets. *(Voiceover: you can...)* It would also require a final `;` in a nonempty list pattern. The last line says a range of literal characters is also a pattern.

Functions and Patterns

Function definitions in OCaml turn right back to patterns in several ways. Recall we had:

```
                DEF  ::=  let [rec] VPAT = EXP { and VPAT = EXP }
                       |  (other stuff)
```

Now here is the rule for `VPAT`:

```
     VPAT  ::=  PAT
        |  <vname> { PARAM } [: TYPEXP] [:> TYPEXP]
        |  <vname> : POLYTYPEXP
```

where we will see the **:> type coercion** (which is like `extends` or `implements` in OOP languages) and **polymorphic type expressions** later.

The official OCaml grammar actually does the following---which is equivalent:

```
            DEF   ::=  let [rec] LET_BINDING { and LET_BINDING }
    LET_BINDING ::= PAT = EXP
                 |  <vname> { PARAM } [: TYPEXP] [:> TYPEXP] = EXP
                 |  <vname> : POLYTYPEXP = EXP
```

This merely moved the eventually-required "**= EXP**" part downstream. Either way, it comes down to **PAT**. So does **PARAM**, incidentally:

```
           PARAM  ::=  PAT  |  other-stuff-with-labels...
```

Pattern Matching in OCaml

To show how patterns are used, we need only mention two more lines of the rules for expressions:

```
     EXP ::=  match EXP with PATMATCH
          |   EXP { ARG }+

  PATMATCH ::=  [ | ] PAT [when EXP] -> EXP { | PAT [when EXP] -> EXP }
```

I could have included the line with `ARG` last week; `ARG` goes right back to `EXP` but with **label** options too. So it is mainly the pattern for expressions that consist of a function (which itself can be returned in an expression from another function) being applied to some arguments. (Well, it is literally applied to the first **ARG**; any arguments after it get carried along for future rides.)

If we ignore the optional `when` feature for now (until we cover the "guarded `if`" idea in more-conventional programming languages), and ignore that OCaml doesn't care if you put an unnecessary bar `|` before the first pattern in your **`match`** body, we can **condense** this into one simplified rule---also showing possible indentation:

```
EXP ::=  match EXP with
            PAT -> EXP
        { | PAT -> EXP }
```

This says that you do need bars to separate multiple patterns used in your match. Let's derive a whole example function that does pattern matching. The example is

```
let rec sumList ell = match ell with
     [] -> 0
   | x :: rest -> x + sumList rest;;
```

As noted before, the syntactic category for this is `DEF` (which comes as a simple case of `COMPUNIT`, which is the start symbol for the whole programming language grammar). Now we are going to illustrate two processes that are different---and work at different levels---but have affinity:

I. **Parsing** the sumList program to show how it is derived in the OCaml BNF grammar.
II. "Parsing" a given list to show how we get the sum.

In both case it's intuitively a **top-down parse/derivation**, picking apart what is given rather than trying to build it up from scratch.

I.

```
DEF ⟹ let rec VPAT = EXP

⟹ let rec <vname> PARAM = EXP              (taking one param from { PARAM }^+)

⟹ let rec sumList PARAM = EXP

⟹* let rec sumList ell = EXP

⟹ let rec sumList ell = match EXP with
        PAT -> EXP
      | PAT -> EXP                           (taking one extra pattern from { | PAT -> EXP })

⟹^3 let rec sumList ell = match ell with  (via EXP ⟹ <value-path> ⟹ <vname>⟹ ell)
        PAT -> EXP
      | PAT -> EXP

⟹ let rec sumList ell = match ell with
        <constant> -> EXP
      | PAT -> EXP

⟹ let rec sumList ell = match ell with
        [] -> EXP
      | PAT -> EXP

⟹* let rec sumList ell = match ell with
        [] -> 0
```

```
       | PAT -> EXP

⟹ let rec sumList ell = match ell with
         [] -> 0
       | PAT :: PAT -> EXP

⟹^2 let rec sumList ell = match ell with
         [] -> 0
       | <vname> :: <vname> -> EXP

⟹^2 let rec sumList ell = match ell with
         [] -> 0
       | x :: rest -> EXP

⟹ let rec sumList ell = match ell with
         [] -> 0
       | x :: rest -> EXP + EXP

⟹ let rec sumList ell = match ell with
         [] -> 0
       | x :: rest -> EXP + EXP ARG

⟹^* let rec sumList ell = match ell with
         [] -> 0
       | x :: rest -> x + sumList rest
```

II.

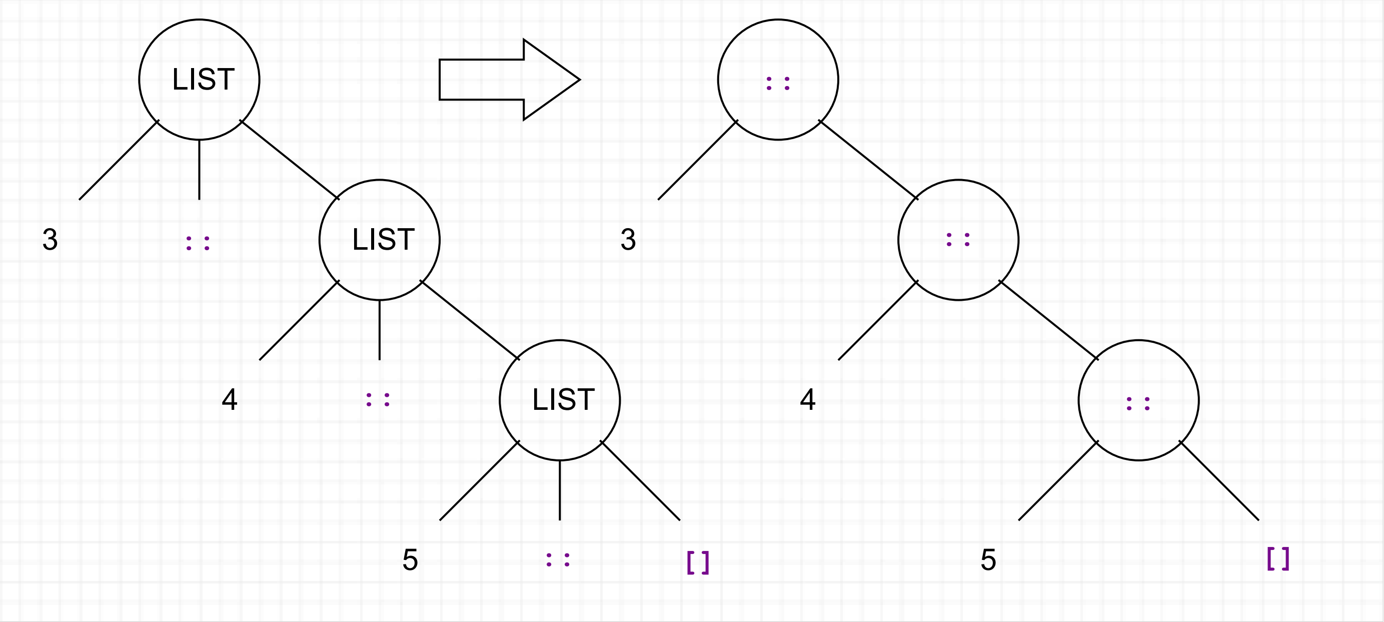
Now let's run it on the given list `[3;4;5]`. In "**infix cons form**", this list is

```
3 :: 4 :: 5 :: []
```

The **`list`** type is recursively defined in "BNF style" as follows:

```
        LIST  ::=   []  |  <element> :: LIST
```

We can give both a simple parse tree and an expression tree for our list:



Viewing 3 :: 4 :: 5 :: [] as a "list expression", note that it associates to the right (like powering) rather than to the left. Thus, the outermost operator (also called the *highest* operator or the "top-level" operator) in our list "ell" = 3 :: 4 :: 5 :: [] is the :: between the 3 and the 4. That is, we would group it as 3 :: (4 :: 5 :: []) if we had to. This is what we match on in the first call of our recursive program, whose body again is:

```
match ell with
      [] -> 0
    | x :: rest -> x + sumList rest
```

What happens on execution is this:

- Our list ell does not match the first case with [].
- So we try the second case. It matches. The oucome of the match is that
  - the (user-chosen) name **x** is temporarily **bound** to the value 3
  - the name **rest** is temporarily **bound** to the rest of the list, which is 4 :: 5 :: [].
- The body after the -> of that case uses the bound values. It has a recursive call to sumList. That gets the argument 4 :: 5 :: [] and we re-enter the **match** to parse it anew.

There is nothing special about the names x and rest here. They are ordinary user variables, **x** of type int and **rest** of type int list. The choices **hd** and **tl** for "head" and "tail" are more common and are being used in recitations, or more simply **h** and **t**. [The one reason I (KWR) shy away is that in Standard ML they are predefined library functions at top level, so you want to avoid clashing with them. Whereas, in OCaml they are inside the **List** module: List.hd ell gives 3 and List.tl ell gives [4;5]. This is an example of *guarding names inside namespaces* and will be talked about more in terms of "Encapsulation Constructs" in Sebesta chapter 11.]

What we will cover *soon*, however, is that the body of the -> is a **scope**, as happens also with the in part of a let-in construct, and the bindings of **x** to 3 and **rest** to 4::5::[] are confined to that time-instance of the scope. That's good, because the recursive call `sumList [4;5]` opens up a new scope (in the time dimension) in which a new instance of **`x`** is bound to 4 and **`rest`** to the **singleton list** [5], which is semantically (but not syntactically) the same as `5 :: []`. The third call `sumList [5]` binds a new **`x`** to 5 and **`rest`** to `[]`.

That sets up the final recursive call `sumList []`. This matches *the base case* and just returns 0. The whole evaluation can be traced like so:

```
sumList [3;4;5]
    ⟹ 3 + sumList [4;5]
         ⟹ 3 + 4 + sumList[5]
               ⟹ 3+4+5+sumList[]
                     ⟹ 3+4+5+0
                          = 12.
```

I've written this trace to look like a derivation---and fact is, in the **operational semantics** of OCaml, it would be analyzed as a derivation, with steps for evaluating the + sign too. (Purple bold means here: the term is in the text but we only skimmed it in section 3.5 to get a general idea of how this kind of recursive thinking gets parsed into an iteration so we can rigorously trace what it does.)

FAFO Interlude

Why didn't the professor write `3 + sumList 4::5::[]` when tracing the recursion? We can literally try it:

```
# sumList [4;5];;
- : int = 9
# sumList 4 :: 5 :: [];;
Error: This expression has type int but an expression was expected of type
         int list
# sumList 4::5::[];;
Error: This expression has type int but an expression was expected of type
         int list
```

Closing up whitespace did not help. What is going on, isn't `4::5::[]` the argument? Well, it is if we put it in parentheses:

```
# sumList (4::5::[]);;
- : int = 9
```

What's going on is that there is an invisible binary operation in OCaml, one of higher precedence than ::

for lists. It is called the **application** operator. It is the only operator in the pure form of Alonzo Church's **λ-calculus** and he made it invisible too. To confirm what is happening, note that putting the parentheses around it instead gives the same error message.

```
# (sumList 4) :: 5 :: [];;
Error: This expression has type int but an expression was expected of type
         int list
```

But if we use an int → int function, then OK. The only built-in one I can find is `~-` for unary minus:

```
# ~- 4::5::[];;
- : int list = [-4; 5]
# ~- [4;5];;
Error: This expression has type 'a list
       but an expression was expected of type int
```

Here's another instructive error. Suppose we wanted to make the base case add one-half:

```
# let rec sumList2 ell = match ell with
     [] -> 0.5
     | x::rest -> x + sumList2 ell;;
Error: This expression has type float but an expression was expected of type
         int
```

It is interesting that OCaml doesn't complain about the base case 0.5 ruining what the recursion case represents as an integer function by using `+` not `+.` as the sum operator. It has already decided from the base case that sumList2 is supposed to return a float. Whereas, if the cases are written in the other order---

```
# let rec sumList3 ell = match ell with
     x::rest -> x + sumList3 ell
     | [] -> 0.5;;
Error: This expression has type float but an expression was expected of type
         int
```

---what happened is that it decided from the first-written case that the return type of `sumList3` was going to be `int`, so the return of 0.5 in the base case is the mismatch. The error message is the same either way, but the point of error is different. By the way, it is fine to write cases in any order:

```
# let rec sumList4 ell = match ell with
     x::rest -> x +. sumList4 rest
        | [] -> 0.5;;
```

```
val sumList4 : float list -> float = <fun>
```

[**Added**: Lecture had a stupid typo, "`ell`" in place of the green "`rest`" above. That's what caused the infinite recursion. Of course the order should not cause it here, because the empty list `[]` does not match the recursive case.] There are often precedence ~~and stack-overflow~~ reasons to put base cases first, however. One more riff:

```
# let rec sumList5 = function
     [] -> 0.5
    | x::rest -> x +. sumList5 rest;;
val sumList5 : float list -> float = <fun>
```

This defines an "anonymous function" (which Scala also gives when you write the keyword `lambda` instead of `function`) but then assigns it to have the name `sumList5` anyway.

Here are the relevant syntax with more lines of the BNF grammar for expressions, and several more takeaways:

```
EXP ::=  match EXP with PATMATCH
      |  function PATMATCH
      |  fun { PARAM }+  [: TYPEXP] -> EXP
      |  try EXP with PATMATCH
```

- Defining a function can be entirely based on pattern matching its possible cases.
- An anonymous function definition---which is all body---counts as an expression.
- An anonymous function can also be given in the more-usual parameter form but using `fun` as the keyword. (The **`[: TYPEXP]`** part is optionally giving a return type for the function, which the final expression must then type out to.)
- Exceptions use pattern-matching as well---on exception patterns that follow the grammar rule PAT ::= `exception` PAT (that and the PAT ::= `lazy` PAT rule I had at the end of the Thu. 2/16 lecture but we won't see them until much later).

A List-to-List Example

Let's write a function **`rev`** to reverse a list, so e.g. `rev [2;3;4;5] = [5;4;3;2]`. We can take the given list one element at a time, starting with the 2 in front. Thinking recursively, here's what we want to do:

- Grab the front element (here, the `2`; then the rest is `[3;4;5]`).
- Reverse the rest recursively (so getting `[5;4;3]`).
- Then stick the grabbed element onto the end (giving `[5;4;3;2]`)..

The reversal of the empty list is just itself. So `[] -> []` can be our base case. It already makes the output type be a list. Our first impulse is to try:

```
let rec rev = function
     [] -> []
     | x::rest -> (rev rest) :: x   ;;
```

but this gives an error. The `::` constructor only takes a list on the right and an element on the left, not vice-versa. There is alas no "backward cons" operator, but we can make do with the built-in `@` operator for concatenating two lists. We have to make x into a singleton list for this to work:

```
let rec rev = function
     [] -> []
     | x::rest -> (rev rest) @ [x]   ;;
val rev : 'a list -> 'a list = <fun>
```

Then `rev [2;3;4;5]` duly gives `[5;4;3;2]`. (And `rev 2::3::4::5::[]` gives the same precedence error as above, but `rev (2::3::4::5::[])` is fine.) Moreover, this code isn't limited to lists of integers---the `'a list` notation means a list of any type. This is like **`list<A>`** in C++ or **`List<A>`** in Java where the template/generic parameter **`A`** can be filled in by any type. E.g.:

```
rev ["Never"; "odd"; "or"; "even"]  gives  ["even"; "or"; "odd"; "Never"]
rev ['N' ; 'e' ; 'v' ; 'e' ; 'r' ; 'o' ; 'd' ; 'd' ; 'o' ; 'r'; 'e' ; 'v' ; 'e' ; 'n']
=   ['n'; 'e'; 'v'; 'e'; 'r'; 'o'; 'd'; 'd'; 'o'; 'r'; 'e'; 'v'; 'e'; 'N']
```

The former is a **`string list`**, the latter a **`char list`**.

FAFO Again

Suppose we left off the square brackets on the [x] with the append function:

```
let rec revv (ell: 'a list) : 'a list = match ell with
   [] -> []
   | x :: rest -> (revv rest) @ x   ;;
```

Is this an error? OCaml goes into "Sherlock Holmes" mode: The empty list [] can be a list of any type. So I have to make my judgment entirely on the recursive case. Now the **`@ x`** part means that x has to be a list of something, say **`List<A>`** to borrow Java notation.. But the **`x ::`** part should mean that **`x`** is an element. Ah, that's not necessarily a contradiction: a list can be an element of a *list-of-lists*, which is **`List<List<A>>`** in Java notation. (Voiceover: in Java unlike C++ it is OK to put two angle brackets together---they aren't automatically **lexed** as a shift operator.) That makes **`x: List<A>`** and **`rest:`**

`List<List<A>>` an OK combo for the `::` operator. This also means that the argument **`rest`** in the recursive call (**`revv rest`**) is of type **`List<List<A>>`**. But the value of that call has to be the same type as **`x`** in order to use the **@**. Actually, that squares up: **`revv`** becomes a function from lists-of-lists to simple lists. OCaml says

```
val revv : 'a list list -> 'a list = <fun>
```

`revv [ 3::4::5::[]; 2::1::[]; 0::7::8::[] ]` gives `[0; 7; 8; 2; 1; 3; 4; 5]`

So it both reverses the list-of-lists and joins them up into one list. Clever---maybe too clever?? Putting a type specifier **`let rec revv (ell: 'a list) = match...`** in the function header would have caught this as a typo, which it probably was. [Don't believe everything the professor tells you, try it for yourself...]

The upshot is that OCaml bends over backwards to try to accommodate programs when possible. The algorithm it uses to recitfy types is strongly related to what is called **unification** in Prolog. [So: Specifying both the input and output types of functions is good advice if you have a specific purpose and safety-first is the need. But in this course the principle is FAFO-first and we will leave types out with OCaml. That OCaml is able to be smarter at **type inference** than Scala helps.]

[The Tue. 2/21 lecture ended here, as planned.]

User-defined Type Construction in OCaml

This is the last major piece of the "pure" language before we get to classes and objects---and it already gives a different take on some features of OOP. If you've ever thought it would be useful for **`enum`**erations in C/C++/Java or Javascript etc. to take parameters, this is the place to start.

Here are the syntax rules, starting as another option for DEF. It cuts out some advanced stuff from the official grammar; for instance, type parameters (which are called template paramaters in C++ and **generics** in Java) can be marked **+** or **–** to allow subclass or superclass compatibility, and/or **!** to promise that concretely instantiated types will not leave loose ends. The compatibility issue goes under the buzzwords **covariance** / **contravariance** and will be touched on in April. Rather than angle brackets, or square brackets as in Scala, generics in OCaml use an apostrophe ' (which can be hard to notice in the grammar) on an identifier that must begin with a letter or underscore.

```
DEF     ::=  type [nonrec] TYPEDEF {and TYPEDEF}
TYPEDEF ::=  [GENID] <lcalphaid> [= TYPEXP] { constraint TYPEXP = TYPEXP }
          |  [GENID] <lcalphaid> = TYPEREP { constraint TYPEXP = TYPEXP }

GENID   ::=  '<alphaid> | ('<alphaid> {, '<alphaid>})
TYPEREP ::=  <Ucalphaid> [of TYPEXP]  { | <Ucalphaid> [of TYPEXP] }
```

The first line of TYPEDEF merely makes a new name for an existing type, just like a `typedef` in C/C++. The second line is new stuff. Ignoring type constraints and tuples of generics (which are like having multiple templates as in `Map<K,V>` in Java or `Map[K,V]` in Scala), we can put the most frequently used syntax just on one line:

```
TYPEDEF ::=
  [ '<alphaid> ] <lcalphaid> = <Ucalphaid> [of TYPEXP] { | <Ucalphaid> [of TYPEXP] }
```

[**Footnotes:** The interplay of any-case, lower-case, and uppercase is stricter than in Standard ML and bears noting: lowercase on the left-hand side of the = sign names the type, and uppercase gives its **constructors**. I've put that in red because of a nomenclature issue. The official OCaml grammar has the rules ***typexpr*** ::= [***typexpr***] ***typeconstr***, which I've written as TYPEXP ::= [TYPEXP] TYCON, and derives ***typeconstr*** to ***typeconstr_name*** optionally prefixed by a dotted module path, and ***typeconstr_name*** only to a lowercase identifier. I conveyed the intent of that lowercase identifier by paraphrasing the rule as:

```
  TYCON ::= the basic types  |   list   |   lots of other stuff.
```

One reason I'm doing that is to emphasize---right away---that **list** is not a keyword in OCaml. End-users can (almost) entirely replicate what it does. It certainly "constructs" types like with **`int`** going to **`int list`**. However, the *uppercase* stuff is what behaves more like the kind of OOP *constructor* we're familiar with---especially close to value-class constructors in Scala. Standard ML didn't have so strict a divide, so I used "constructor" for both, and the OCaml data types page assigned in this week's reading uses **constructor** for the uppercase stuff too. The real thing to emphasize, though, is that **matchable structure** *comes directly from the uppercase stuff* (and from `::` and `[]` which are treated specially in OCaml), but only from *concrete values* of the lowercase types.]

For the first **example**, here is how `list` might actually be implemented in OCaml---or rather, how it is implemented in Standard ML (except for the first keyword being `datatype`):

```
type 'a list  =  []  |  :: of 'a * 'a list
```

The [] and :: are now in orange because OCaml treats them specially. The OCaml rule actually allows them as alternatives where I've put `<Ucalphaid>`, but they seem to give "Syntax error" anyway. Moreover, :: *is the only infix constructor allowed in OCaml*. So to emulate lists, we have to "roll our own" in prefix notation. This is the standard way to do it:

```
type 'a list = Nil  |  Cons of 'a * 'a list
```

Amazingly, this is allowed---so **`list`** is not a keyword but rather in the user-(re-)definable domain. But It doesn't wipe out the other form of list---instead you may get weird error messages like "`Error: This` expression has type int list/1019 but an expression was expected of type int list/9." So let's not go there; let's roll our own type name too:

```
type 'a lyst = Nil  |  Cons of 'a * 'a lyst
```

Now to define [3;4;5] as a "lyst" we need to write

```
let ell = Cons(3, Cons(4, Cons(5,Nil)));;
```

Notice that the arguments to `Cons` really are tuples---with the comma where the * went in the type definition. `Cons` and `Nil` have the capital letters needed to work as structure elements. Now we can craft our sumList function to match on this structure instead:

```
let rec sumList ell = match ell with
     Nil -> 0
   | Cons(x,rest) -> x + sumList rest;;
```

Then `sumList ell` gives 12 just like before. Note that in the body, all the major items are user-defined.

Simpler Examples: Option Types and More

The *option type* comes preloaded in OCaml but is completely user-definable:

```
type 'a option = None | Some of 'a
```

An example I could use is that chess players in their first few tournaments are *unrated*, until they get their first integer rating. If you define

```
let printedRating rating = match rating with
    None -> "Unrated"
    | Some x -> string_of_int x;;
```

then you can print unrated players as "Unrated", someone like me as "2372". But you can also make this type-design pattern more customized:

```
type rating = Unrated | Rated of int
```

Now let's make a more descriptive function of how a US chess rating is classified:

```
let playerClass = function
     Unrated -> "Unrated"
   | Rated r -> if r >= 2400 then "Senior Master"
                else if r >= 2200 then "Master"
                else if r >= 2000 then "Expert"
                else if r >= 1800 then "Class A"
                else if r >= 1600 then "Class B"
                else if r >= 1400 then "Class C"
                else if r >= 1200 then "Class D"
                else if r >= 1000 then "Class E"
                else "Novice";;
```

Notice how using the "`function`" form is a little neater than an initial `match` was. The long `else if` branching is clumsily indented, however. Incidentally, OCaml does not have a separate keyword like `elif` in Python or `elsif` in Perl and Ruby. Condensing "`else if`" into one keyword still would leave the "dangling else" ambiguity in force---except that this one is not ambiguous anyway because it does not have two consecutive ifs (i.e., no if ... then if ...) and every if is here matched by an else. But in practice one doesn't case: the rule that it goes with the most recent `if` is exactly the reading the above suggests, which is what we want. There is, however, a neat way to bring this all under one `PATMATCH`:

```
let playerClass = function
     Unrated -> "Unrated"
   | Rated r when r >= 2400 -> "Senior Master"
   | Rated r when r >= 2200 -> "Master"
   | Rated r when r >= 2000 -> "Expert"
   | Rated r when r >= 1800 -> "Class A"
   | Rated r when r >= 1600 -> "Class B"
   | Rated r when r >= 140 -> "Class C"
   | Rated r when r >= 1200 -> "Class D"
   | Rated r when r >= 1000 -> "Class E"
   | _                      -> "Novice";;
```

For example:

`playerClass (Rated 1500)`   gives   `"Class C"`
`playerClass (Rated 150)`   gives   `"Novice"`

If you blipped the red 0 to make the line "`| Rated r when r >= 140`" then it would short-circuit the last two lines, so that `playerClass (Rated 150)` would give "`Class C`". OCaml does *not* warn about the redundancy, because:

- `if-then-else` and `when` clauses, and inequality comparisons more generally, *do not provide matchable structure.*

Equality to literal constants *does* count as matchable structure. So you can imitate classic switch statements this way:

```
let answer charRead = match charRead with
   | 'y' | 'Y' -> Some "Yes"
   | 'n' | 'N' -> Some "No"
   |  _        -> None
```

This also illustrates patterns having an internal **|** (the magenta ones), which in C/C++/Java[script] **`switch`** blocks is handled by writing consecutive labels **`case 'y': case 'Y':`** with no body in between. (This is part of the reason why **`switch`** has its infamous fall-through behavior when you forget to insert a **`break`**. This is also an example where having the optional initial **|** before `'y'` could be less confusing as well as neater---the book by Schieber likes that style.)

```
let answer = function
   | 'y' | 'Y' -> Some "Yes"
   | 'n' | 'N' -> Some "No"
   |  _        -> None
```

Structured Enumerations

A simple example (parallel to the "color" examples in the readings and recitations, and like an example in the Lewis & Lacher Scala text used in CSE250):

```
type trafficColor = Green | Yellow | Red

let cycle = function
    Green -> Yellow
  | Yellow -> Red
  | Red -> Green
```

OCaml and Scala do not make "enum constants" equivalent to integers beginning with 0 like in C/C++. So you don't have the shortcut of defining `cycle(x)` to return `(x+1)%3` and casting that back to the enumeration. Working with the structure directly is safer, albeit longwinded.

Let's see something else before coming back to colors:

```
type weekday = Monday | Tuesday | Wednesday | Thursday | Friday;;

let nextDay = function
  | Monday -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday ->  ??
```

If we put Saturday then we get the error that Saturday is not part of the weekday type. We could imitate the cycle function and put Monday. That would be right if we really want to cycle to the next business day. But suppose we really want to say that this case is undefined?

- Scala might actually let you leave the ?? alone---you get an error if you try to execute it. Well, it does so with whole function bodies, but you could fill in such a "stub function".
- You can throw an exception on that branch. The syntax to define an exception is another case of a top-level definition, basically:

```
DEF ::= exception <Ucalphaid> [of TYPEXP]
```

So one can do things like:

```
exception Cain;;
let nextDay = function
  | Monday -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday -> raise Cain;;
```

OCaml lets that slide as a function definition.

[This is exactly where a hard freeze of the console PC ended the lecture with 10 minutes left. I will pick up from here on Tuesday.]

But if you try to use it in a context where you catch the exception, you have to provide a type-matching value. Putting e.g. an int instead won't fly:

```
      let nextDayE day =
         try
            nextDay day
         with
            Cain ->  0;;
Error: This expression has type int but an expression was expected of type
   weekday.
```

- For the same reason, filling in None for the ?? is a type error. But we can make that OK by having *every* value be an option:

```
let nextDayOpt = function
  | Monday -> Some Tuesday
  | Tuesday -> Some Wednesday
  | Wednesday -> Some Thursday
  | Thursday -> Some Friday
  | Friday ->  None
```

Ocaml says `val nextDayOpt : weekday -> weekday option = <fun>`. But is it fun? We now have to do a case-match to "unpack" the value of the `nextDayOpt` function. That unpacking allows client code to customize how it wants to deal with the `Friday` giving `None` case. But it's a little clunky, and the OCaml website admits that you can get the cruft of multiple `Some` levels having to be "`join`"-ed down to one level.

One nice power of OCaml enumerations is associating data to the alternatives. We could do:

```
type appointment = weekday * float;;
```

intending a float like 13.45 to mean an appointment at 1:45pm on a given day. Then we have to write it as a tuple, e.g. `(Monday, 15.00)` is a valid appointment. But it might be considered neater to integrate the time into the type:

```
type appointment =
   | Monday of float
   | Tuesday of float
   | Wednesday of float
   | Thursday of float
   | Friday of float;;
```

Now we can write `Monday 15.00`, or `Monday(15.00)`, to denote the appointment. The latter especially reveals that the constructors are really functions---using "of" the way we say "f of x" in English---taking a float as argument and outputting an appointment object. And much like you can have multiple constructors for a class taking different types of arguments, the types after `of` don't have to agree. If you want to legislate that Friday appointments are only on-the-hour, you could end with `Friday` **of int** instead.

We could use the same idea to denote the integer values associated to RGB intensities on a computer screen:

```
type rgbHue = Red of int | Green of int | Blue of int;;
```

The hitch is that you only get one component of an RGB triad that way. You can make a triple of hues:

```
type rbgColor = rgbHue * rgbHue * rgbHue;;
```

The problem is that this doesn't enforce that the first component is the red intensity, then the green value, then the blue value. Indeed, the contradictory all-red triad `(Red 0, Red 127, Red 255)` is a valid value. You could just use `type rgbColor = int * int * int` as the representation, leaving it implicit that in a triple such as `(53,17,242)`, the numbers are in the order red-green-blue. This loses all the benefits of strong typing, however, and could lead to colors being confused with other integer triples. The ultimate proper way to handle this is with a **record** or **class** object. OCaml has both, but the former is (IMPHO) a legacy from Standard ML; the whole point of teaching OCaml with the O is to cover *objects*. So we will mostly ignore OCaml records and do thsi when we get to classes and objects later.

Extension as Generalization Versus Specialization

One object-oriented aspect bears noting now, also as a lead-in to the final main example of modeling arithmetical expressions within OCaml. Suppose we want to make a full week including Saturday and Sunday, but want to re-use the `weekday` type we have. We can't do

```
type day = Saturday | Sunday | weekday
```

because "`weekday`" is not an uppercase constructor. But we can do:

```
type day = Saturday | Sunday | Weekday of weekday
```

The "clunk" however is we can't simply define `nextDay(Sunday) = Monday` even now. We have to do:

```
let nextDay3 = function
  | Weekday Monday -> Weekday Tuesday
  | Weekday Tuesday -> Weekday Wednesday
  | Weekday Wednesday -> Weekday Thursday
  | Weekday Thursday -> Weekday Friday
  | Weekday Friday -> Saturday
  | Saturday -> Sunday
  | Sunday -> Weekday Monday;;
```

This is valid. But clunky. We would like `day` to "inherit" the weekdays at top level. If we just rewrite `Monday | ... | Friday` as the other five alternatives for `type day`, we will **shadow** the same names in our previous `weekday` type, which OCaml allows (!) but IMPHO it's a mess. Another issue is that even if we inherited the `nextDay` function, it would have the wrong meaning for the `day` type. But let's pose a philosophical question:

- Suppose we could seamlessly make `day` inherit the `weekday` constructors `Monday` through `Friday` at top level. Which would be the logical base class, `day` or `weekday`?

We arrive at the most-featured example of OCaml data types in the readings with this question also in mind. [The actual lecture would have stopped here.]

Modeling Arithmetical Expressions

We can bring things full-circle by modeling arithmetical expressions *within* OCaml. One thing we might want to do is allow the same expressions not only to work for both **`int`** and **`float`** arguments, but possibly to be instantiated for adding and/or multiplying matrices and vectors and anything for which we can interpret those operations. Let's start with $+$ and $-$ for now, since multiplying vectors gets a little funky. Because we can't define our own infix constructors in OCaml, we have to use prefix notation for all binary operations.

```
type 'a addExp = Const of 'a | Var of string | Parens of 'a addExp
   | Neg of 'a addExp
   | Plus of 'a addExp * 'a addExp
   | Minus of 'a addExp * 'a addExp;;
```

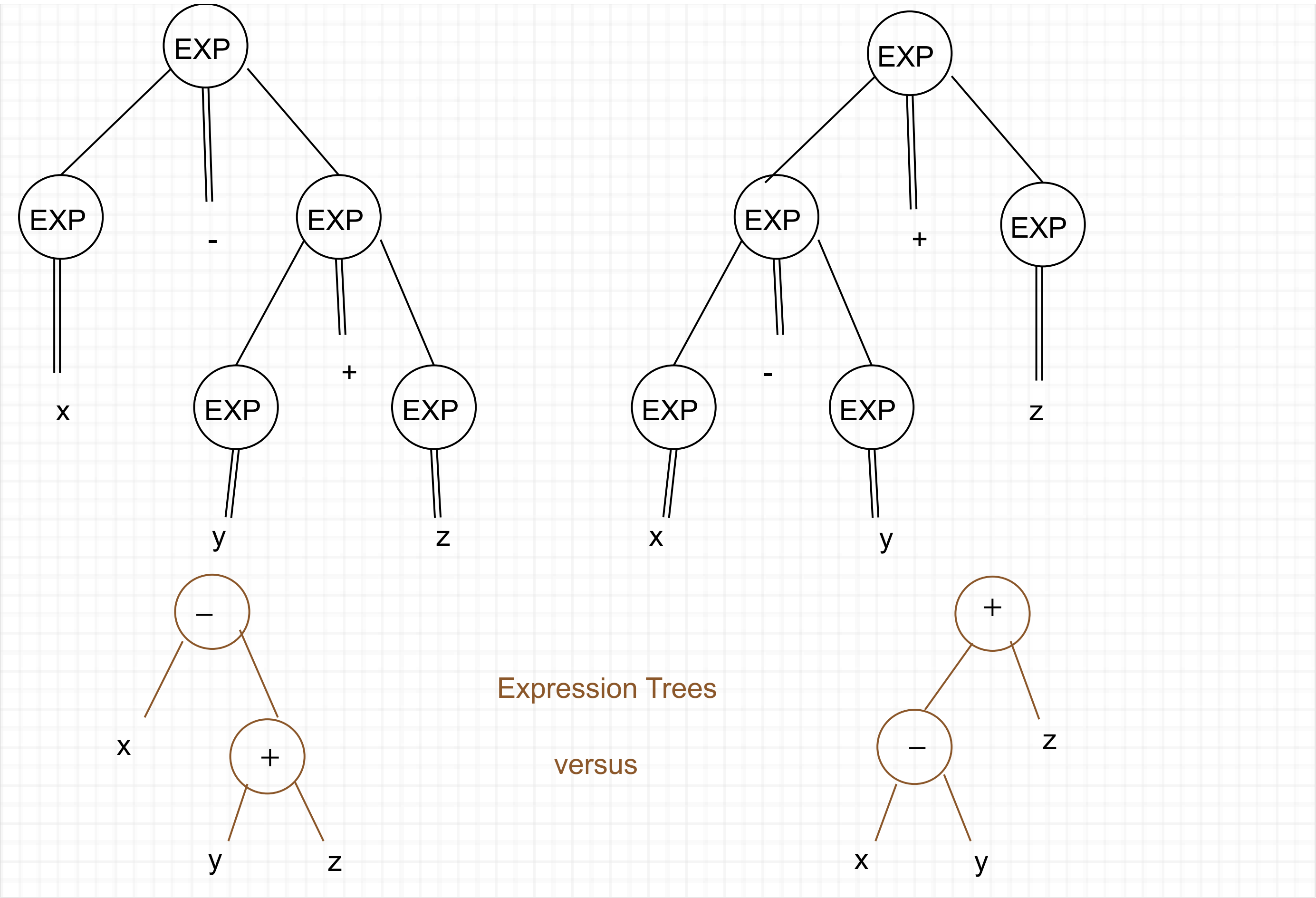
Except for not modeling times and divides (yet), this is richer than the examples in the readings with the `Parens` and `Neg` options. It does, however, model the "simple ambiguous grammar"

$$E \ ::= \ \langle constant\rangle \mid \langle variable\rangle \mid (E) \mid \ -E \mid E\,\langle binop\rangle\, E$$

where we're only allowing $\langle binop\rangle \ ::= \ + \mid \ -$ for now (on your HW, allow $*$ too). It doesn't specify precedence or associative grouping rules---they will, however, come out with how we build expression objects in this grammar (where we might see that the `Parens` rule is unnecessary).

```
let xmypz = Plus(Minus(Var "x", Var "y"), Var "z");;
let xmqypz = Minus(Var "x", Plus(Var "y", Var "z"));;
```

This calls up both the parse trees and the expression trees from last week's lectures:



If we specify the generic type `'a` as `int` then we can write a recursive evaluator for additive integer expressions. We need to design a method `fetch` to read the value of a variable---this takes us already into the area of the text's chapter 5---but let's just throw in a "stub function" that always returns 7.

```
let fetch (x:string) = 7;;

let rec eval(exp: int addExp) = match exp with
     Const a -> a
   | Var x -> fetch x
   | Parens exp -> eval exp
   | Neg exp -> -(eval exp)
   | Plus(exp1,exp2) -> (eval exp1) + (eval exp2)
   | Minus(exp1,exp2) -> (eval exp1) - (eval exp2);;
```

OCaml duly reports `val eval: int addExp -> int = <fun>`. Then:

`eval xmypz`   gives 7
`eval xmqypz` gives -7.

Now, however, suppose we want to *re-use* this code while supporting multiplication and division too. Following the "`Weekday of weekday`" idea, we could do:

```
type 'a mulExp =
     Times of 'a mulExp * 'a mulExp
   | Div of 'a mulExp * 'a mulExp
   | AddExp of 'a addExp;;
```

We can extend our evaluator to this code and even re-use the previous method via composition (rather than inheritance):

```
let rec eval2 = function
   | Times(exp1,exp2) -> (eval2 exp1)*(eval2 exp2)
   | Div(exp1,exp2) -> (eval2 exp1)/(eval2 exp2)
   | AddExp aexp -> (eval aexp);;
```

The problem, however, is that our ability to build up expressions is stunted. We can do $a*(b+c)$ in problem (2)(ii) on Assignment 1 simply enough:

```
Times(AddExp (Var "a"), AddExp(Plus(Var "b", Var "c")));;
```

But $a*b+c$ without the parentheses is a problem. The outer operator is $+$, so it needs to effectively start with `Plus`. Since it needs to be a `mulExp` it must start `AddExp(Plus(..., ...))` just like we had to do with the second part of `Times` before. The second ... can be filled in with `Var "c"` again. But what about the first ... ? Since it is representing $a*b$, it has to begin with `Times`. But to be allowed with `Plus`, it has to be already an `addExp` on the outside. There is no way to fix it.

Other things are out of whack. To reflect the precedence of $*, /$ over $+, -$, the `addExp` type should feed into `mulExp`, but the above is the other way around. The "chair lift" `Parens of 'a addExp` only goes as high as `addExp` because `mulExp` wasn't yet in the picture when we made it part of the `addExp` type.

Pause for another question: Suppose we could do this smoothly with (these or someother kind of) *objects*. That is, suppose we can start with a class **`AddExp`** and then generalize it to a class **`MulExp`** which includes additive expressions as special cases (where no * or / operation happens to be used). The question is:

- ***Then which would be the base class, `AddExp` or `MulExp`?***

[This or the similar question in the "day" types above were the potential stopping points in the original plan, depending on how time worked out. The actual timing would have had me stop at the question above if the hard freeze hadn't happened---with maybe a minute to "flash" the first 'a addExp example and then say that expression and binary tree examples would be covered in next week's recitations. But now with the above re-starting on Tuesday 2/18, it's an "in for a dime, in for a dollar" situation---so I've extended the notes below, and the whole thing will fill 50-60 minutes. Then I'll go into an abbreviated form of my notes from previous years on "The Stages of Compilation"---which in turn abbreviated what Sebesta says about lexing and parsing in chapter 4 (and type-checking in the skimmed part of chapter 3), with a few words on the further stages of generating object code, linking it, and where things like the "Java Virtual Machine" fits in. The VM idea leads into my giving a "mini assembly language" with a stack mode of evaluating that rides shotgun to the text's chapters 5 and 6, which I will begin covering on Thursday.]

[But carrying on for Tuesday, to illustrate a couple of more sophisticated match-type and recursion ideas and also fully expand the big programming-language design issue behind these questions...]

Short of answering the question, we actually can fix the previous problems by imitating the "ETF" grammar and programming the types all together, with a "chairlift" all the way back up:

```
type 'a exp = Plus of 'a exp * 'a term | Minus of 'a exp * 'a term | Term of 'a term
and 'a term = Times of 'a term * 'a factor | Div of 'a term * 'a factor | Factor of 'a factor
and 'a factor = Const of 'a | Var of string | Parens of 'a exp;;
```

[Note: if you mouse-copy this, be sure it gives a space before each "and".] The `and` form was needed to handle the **mutual recursion** back to `'a exp`. A corresponding evaluation function has to have three similar mutually interlocking parts, one to handle each of the three types:

```
let rec evalExp = function
   | Plus(e,t) -> (evalExp e) + (evalTerm t)
   | Minus(e,t) -> (evalExp e) - (evalTerm t)
   | Term t -> (evalTerm t)
and evalTerm = function
   | Times(t,f) -> (evalTerm t)*(evalFactor f)
   | Div(t,f) -> (evalTerm t)/(evalFactor f)
   | Factor f -> (evalFactor f)
and evalFactor = function
   | Const c -> c
   | Var str -> (fetch str)
   | Parens e -> (evalExp e);;
```

[This presumes the definition of `fetch` above is still in scope.] Because we used the `int` forms of the arithmetic operators, OCaml reports the types as

```
val evalExp : int exp -> int = <fun>
val evalTerm : int term -> int = <fun>
val evalFactor : int factor -> int = <fun>
```

We can also (re-)write a single eval function. Here's a funny wrinkle: Beginning with the most specific case of factors, the first impulse would be to start:

```
let eval = function
   | Factor f -> (evalFactor f)
   | Term t -> (evalTerm t)
   | ... ??
```

The return type will clearly be `int`. But what is the argument type? The first line says the argument is **`Factor f`**, which is a case of the type `'a term` (with `'a` necessarily instantiated as `int`, so really `int term`). But the second line gives a **`Term t`**, which is an instance of the type `'a exp`. That is going to be a clash, even before we get to the problem that nowhere in the above is there a structure marker `Exp of 'a exp`. Well, we can just make the last line read `| e -> (evalExp e)` and OCaml would realize that `e` has to have type `'a exp` (once again, actually `int exp` in this instantiation).

But how to resolve the clash? We could make a "**union type**"

```
    type 'a allExp = Exp of 'a exp | TLOTerm of 'a term | TLOFactor of 'a factor.
```

Here "`TLO`" means "top-level only" which happily would be true: To model a big expression tree rooted as, say, a factor going to `(Parens e)`, we would only have to use `TLOFactor` once.

However, we can recognize that when rooted from the E nonterminal in our original grammar, such a tree would begin like the derivation $E \implies T \implies F \implies (E)$... That is to say,as an **expression** it would begin with `Term(Factor(Parens(...)))` Or in case it were just a variable whose value we would then fetch, it would be `Term(Factor(Var str))`. This is needed anyway if we do `Plus(x,t)` where we just want `x` to be a variable but by the definition of the Plus constructor it has to be an expression. For example, $x + y{*}z$ gets modeled by:

```
      let xpytz = Plus(Term(Factor(Var "x")), Times(Factor(Var "y"), Var "z"));;
```

OCaml responds "`val xpytz : 'a exp = ...`" and repeats what you typed in place of the ... So to treat a factor as an expression, you need to begin with `Term(Factor(...))`, which of itself has type `'a exp`. Although nested, `Term(Factor(...))` is **matchable structure**. So we can write the evaluator without needing another layer of type construction, provided we agree that it can only be given an `int exp` to evaluate:

```
let eval = function
   | Term(Factor f) -> (evalFactor f)
   | Term t -> (evalTerm t)
   | e -> (evalExp e);;
```

OCaml rogers this with "`val eval: int exp -> int = <fun>`". *Do take time to key this in and try some example expressions---just realize that for now, every variable you have will give a 7.* So if you do `eval xptyz;;` you will get 7 + 7*7 = 56. To get a evaluator for floating-point expressions, you'd have to go all the way back to writing a `float` version of `evalExp` using `+.`, `-.`, `*.`, and `/.` But you could at least re-use the `'a exp`, `'a term`, and `'a factor` types. ***Well***, if you think about it a little more, *this is no more or less than what the* `evalExp` *function already does automatically.*

Are we good now? *Yes* if we only care about arithmetical expressions with `+,−,*,/`. This is an example of a **closed world assumption**, one that the above OCaml code is implicitly making (by how it limited its mutual recursion). *But as soon as we try to open up this world to include, say, a shift operator << with lower precedence than + and -, the scheme cannot adapt.* We could make an abstraction `int shiftExp` with a whole new tier (shifting presumes integers), imitating the grammar in the homework solution. But we can't actually re-use and extend the `int exp`, `int term`, and `int factor` types, for one thing because they were coded with a "chairlift" that goes only up to `'a exp`.

This is an arm of a major conundrum called the "Expression Problem" and named for Philip Wadler, who first articulated it best. This goes a little beyond the Sebesta text, though from the linked Wikipedia page's OOP (in C#) example you can see the similarity to what we were trying in OCaml above. The nub of the problem is that when we try to "extend by generalizing", the base class / subclass relation *works in the wrong direction*. A buzzword for this nub in classical OOP languages is "retroactive" followed by "generalization" or "abstraction" or "superclassing"---while the term "Open Classes" referenced in the Wikipedia article focuses this on the Expression Problem itself. I cooked up such a scheme---as a "Swiss Army knife" that also handles the "Square-Versus-Rectangle Problem" and the "Binary Methods (Co/Contra-Variance) Problem"---but it quickly gets hideously complicated in practice.

[Whew. Back to earth on Thursday. But along for the ride, we've gone through much of 8 of 12 sections of the OCaml reference https://v2.ocaml.org/manual/language.html, leaving mainly the *long* section 9 on Classes to come. That will come when we hit OOP in the textbook after spring break, so until then we'll focus on more usages of the core OCaml language vis-a-vis the procedural and expressions part of more-standard languages.]