CSE305 Week 5: Using OCaml Types; Compilation; then Sebesta Ch. 5

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]
```

Exception names are uppercase and are treated with much the same syntax as (other) type constructors. 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. To use the exception, the syntax is much the same: `try EXPR with PATMATCH`. 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 int;;
```

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 pass by OCaml records and do this 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" 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 a featured example of OCaml data types in the readings with this question also in mind.

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;;
```

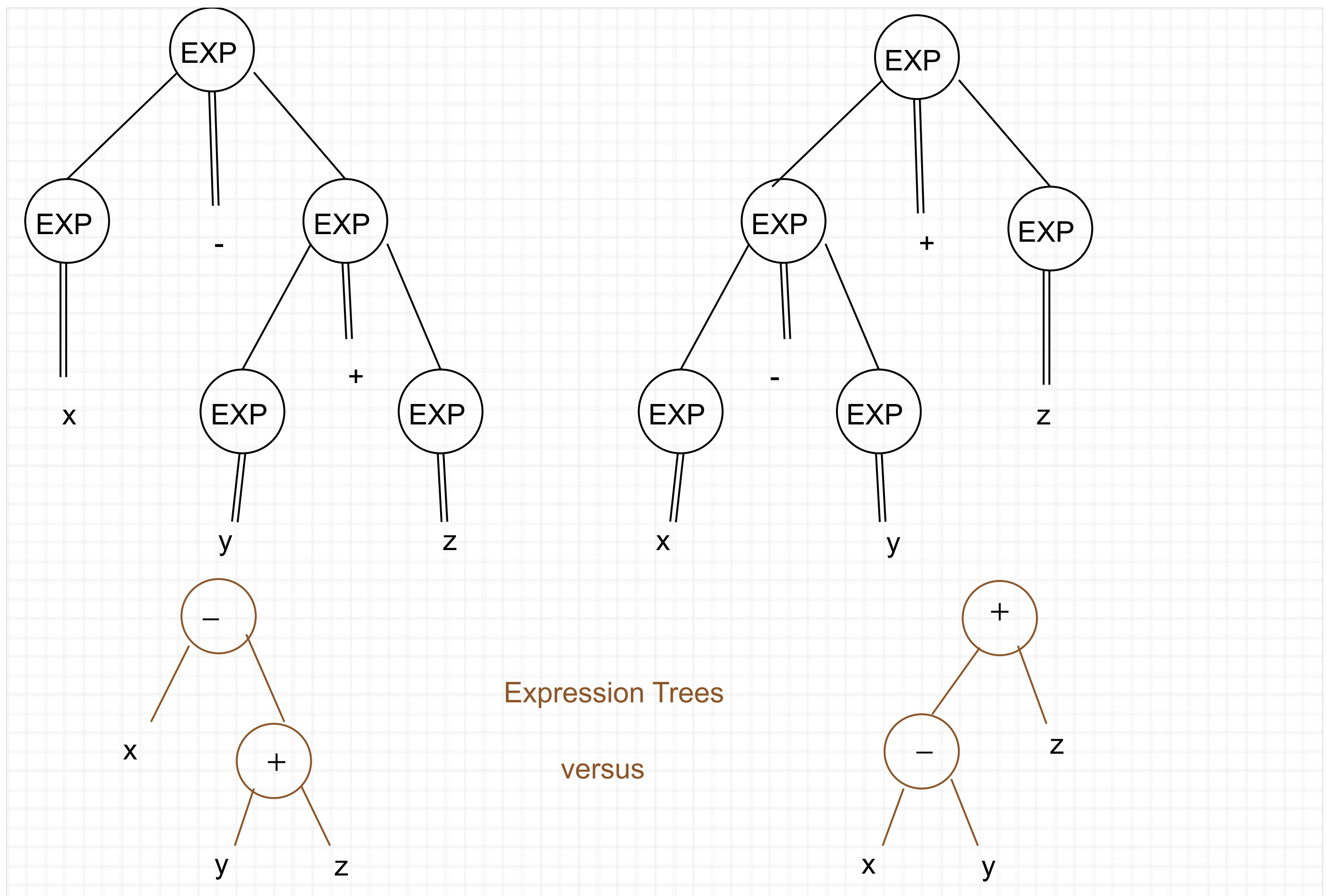
Except for not modeling times and divides (yet), this is richer than the examples in the readings with the `Parens` and `Neg` options. It does, however, model the "simple ambiguous grammar"

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

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 = 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 there 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`?***

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:

E ::= T | $E + T$ | $E - T$
T ::= F | $T{*}F$ | T / F
F ::= (E) | <var> | <const>

```
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 xpytz;;` 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.]

Interlude: Assignment 1 Key; Q&A on Assignment 2

The Assignment 1 key is a plain-text file at https://cse.buffalo.edu/~regan/cse305/CSE305S23ps1key.txt
[go over key]

For a little extra relating to the above theme, here's how we could add a shift-expression to
If $\gg$ is left-associative:

$S \;::=\; S \gg E \mid E$

$E \;::=\; T \mid E+T \mid E-T$

$T \;::=\; F \mid T*F \mid T/F \mid T\%F$

$F \ ::= \ (E) \mid$ <var> $\mid$ <const>

```
type 'a shiftexp = RightShift of 'a shiftexp * 'a exp | Exp of 'a exp
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;;
```

If $\gg$ is right-associative: $S \ ::= \ E \gg S \mid S$
$E \ ::= \ T \mid E + T \mid E - T$
$T \ ::= \ F \mid T{*}F \mid T/F \mid T\%F$
$F \ ::= \ (E) \mid$ <var> $\mid$ <const>

Topic Parallel to Sebesta Chapter 4: The Stages of Compilation

By and large, compilers work in the following stages:

1. **Lexing** and **Symbol Table Construction**
2. **Parsing**
3. **Type-Checking**, **Scope Resolution**, and other "**semantic analysis**"
4. **Object Code Generation**
5. **Linking** (if needed)
6. **Native Code Generation** and/or (further) **Optimization**, if requested.

Lexing determines the individual **tokens** in the program. For example, the string on problem (3), **`r%2*n>>d-1`**, has all its chars rammed together but is not a single token.

- A simple lexing rule distinguishes alphanumeric identifier strings from symbols that denote operators and from various kinds of parentheses/brackets.
- For symbolic chars, when do two or more of them combine to form a single token, such as **>>** for right shift? When and why are they separate for nested Java generics, but not in C++?
- Java limits operators to a predefined set (and only those can be overloaded). Whereas, OCaml allows user-defined operator symbols, which can be made infix by placing theminside parentheses.

A possible (simplified) way of representing tokens within OCaml could be:

```
type token = Ucid of string | Lcid of string | Op of string
           | ParenL | ParenR | SqbrL | SqbrR | Comma | Semicolon | ...
           | Int of int | Float of float  | Qchar of char | Qstring of string | ...
```

Then the lexer could convert the string on problem (3) into a list like this:

(Lcid "r") :: (Op "%") :: (Int 2) :: (Op "*") :: (Lcid "n") :: (Op ">>") :: (Lcid "d") :: (Op "-") :: (Int 1) :: []

This counts as a homogeneous `token list` even though the tokens mix integers and strings. The lines between stages are not completely sharp. For example:

- Lexing may be prefaced by a **preprocessing** step. The main example is that C/C++ allows textual **macros** that allow you to abbreviate textual constructs. Those macros are **expanded** (meaning substituted by their literal text) before lexing begins. A simple example is **`#define PI 3.14159`**. A typical caveat is that if your program has an all-caps identifier like **`SPIN`** then the **`PI`** might result in the nonsense **`S3.14159N`**, whereupon the only thing worse than getting a syntax error because of that is *not* getting a syntax error... Other preprocessing is stripping out comments and resolving local font mappings.
- The OCaml grammar defines parsing rules for different kinds of identifiers and constants.
- **Attribute grammars** (skimmed in section 3.5) are a way to bring parts of stage 3 under the same kind of tools as parsing.
- Portable object code might be regarded as the end goal of compilation, especially in the form of a **virtual machine** like the Java VM.
- Or a compiler might begin by taking VM code as input.
- Optimizations can come into play at any stage. There are also **code transformations** that may or may not be regarded as optimizations.
- There can even be some "run-time compilation" when a new **stack frame** (also called an **activation** frame or record) for a function or method call is allocated.

Linking comes into play when a program has separate compilation units when regarded in its entirety---such as using standard library modules that are already compiled. In OCaml, the object code files have suffix **`.cmo`**, and there are also **`.cmi`** "interface" files to help the linker. On our system, the native code is called "**`a.out`**" unless you use the **`-o <exe>`** compiler option to give the executable another name.

Examples of Code Transformations

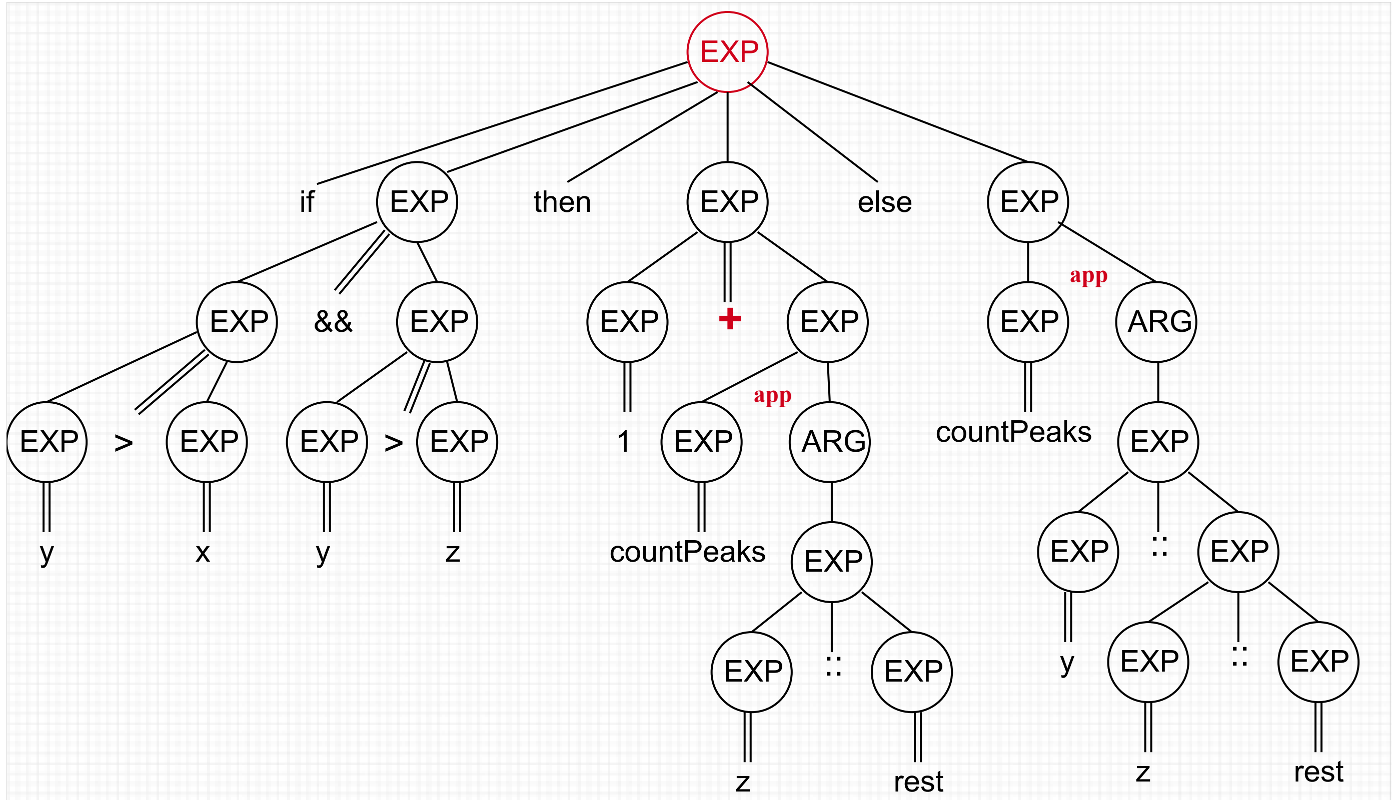
This doubles as an example of "deeper recursion" when working with lists in OCaml. We want to write a function **`countPeaks`** that counts the number of elements in a numerical list that are greater than both the element before and the element after it. The ends of lists cannot count as peaks---so lists of up to two elements have zero peaks. Here is a first stab at the needed recursive code:

```
let rec countPeaks ell = match ell with
     [] -> 0
   | [_] -> 0
   | [_; _] -> 0
   | x::y::z::rest -> if   y > x && y > z
                      then 1 + countPeaks(rest)
                      else countPeaks(y::z::rest);;
```

Is this code correct? Well, it compiles. The OCaml compiler is so "strong" you can deceive yourself by thinking, "if it compiles it must be correct." Consider the list `[2; 7; 4; 5; 1; 7]`. It has two peaks, the 7 and the 5. But the above code will miss the 5. Fix:

```
let rec countPeaks ell = match ell with
      [] -> 0
   | [_] -> 0
   | [_; _] -> 0
   | x::y::z::rest -> if   y > x && y > z
                      then 1 + countPeaks(z::rest)
                      else countPeaks (y::z::rest);;
```

Here is the parse tree for the main body after the last `->`. [Full tree added at end.]



The **`else`** branch has the "invisible application operator" as its highest operator. Whereas, the **`then`** branch has **+** as its highest operator. There is a standard code transformation that **hoists** the recursive call to `countPeaks` to be highest in that branch as well. It involves making the output value a second parameter that is **accumulated**, in what is called **accumulator passing style**:

```
let rec countPeaks (ell,n) = match ell with
      [] -> n   (* output not 0 but the running count n *)
   | [_] -> n
   | [_; _] -> n
```

```
  | x::y::z::rest -> if   y > x && y > z
                     then countPeaks(z::rest, n+1)
                     else countPeaks(y::z::rest, n);;
```

The base cases now change to output n. This works the same overall if you call the function initially with `n=0`. Often the recursive function is renamed to be a "helper function" and the public function would be `let rec countPeaks(ell) = countPeaksHelper(ell,0)`.
We can further hoist the recursive call in both branches combined:

```
let rec countPeaks (ell,n) = match ell with
     [] -> n
  | [_] -> n
  | [_; _] -> n
  | x::y::z::rest -> countPeaks(if   y > x && y > z
                                then (z::rest, n+1)
                                else (y::z::rest, n));;
```

This makes the program **tail recursive**, which fosters implementation by iteration rather than actual recursive calls. (Tail recursion comes up in chapter 15 of the text, so it is black bold, but both this and accumulator passing style may have been mentioned in CSE250 if not before.)

There is a further code transformation that we have already been doing interchangeably, which is the `function` PATMATCH form of defining a function. Some of these transformations are done automatically by compilers, but I've not been able to prove that this one is on our local system:

```
let rec countPeaks = function
     ([],n) -> n
  | ([_],n) -> n
  | ([_; _],n) -> n
  | (x::y::z::rest,n) -> countPeaks(if   y > x && y > z
                                    then (z::rest, n+1)
                                    else (y::z::rest, n));;
```

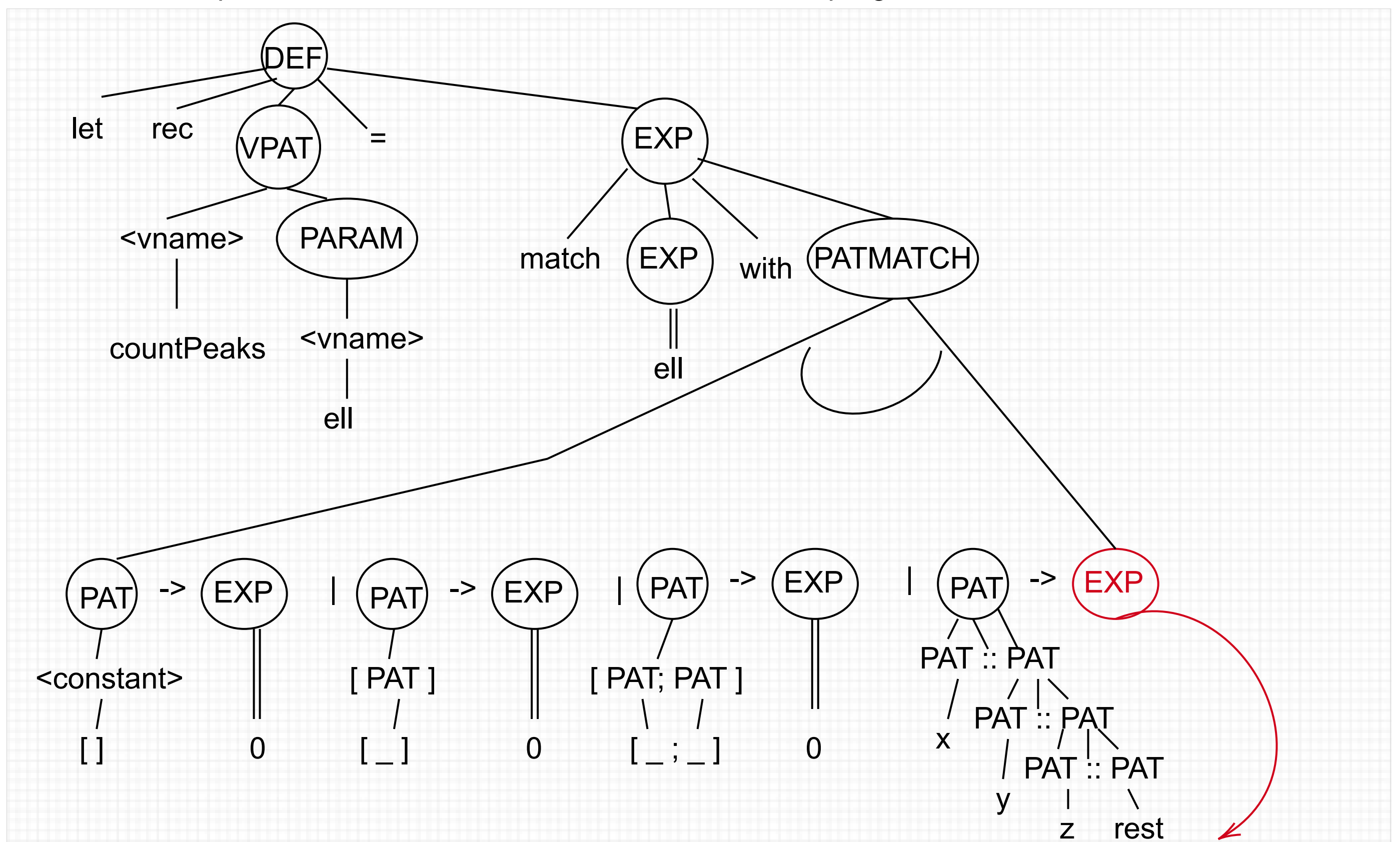
This also goes as an example of matching multiple structure in tuples at once. One final edit, which ought to be totally equivalent, is to combine the three cases that just return n:

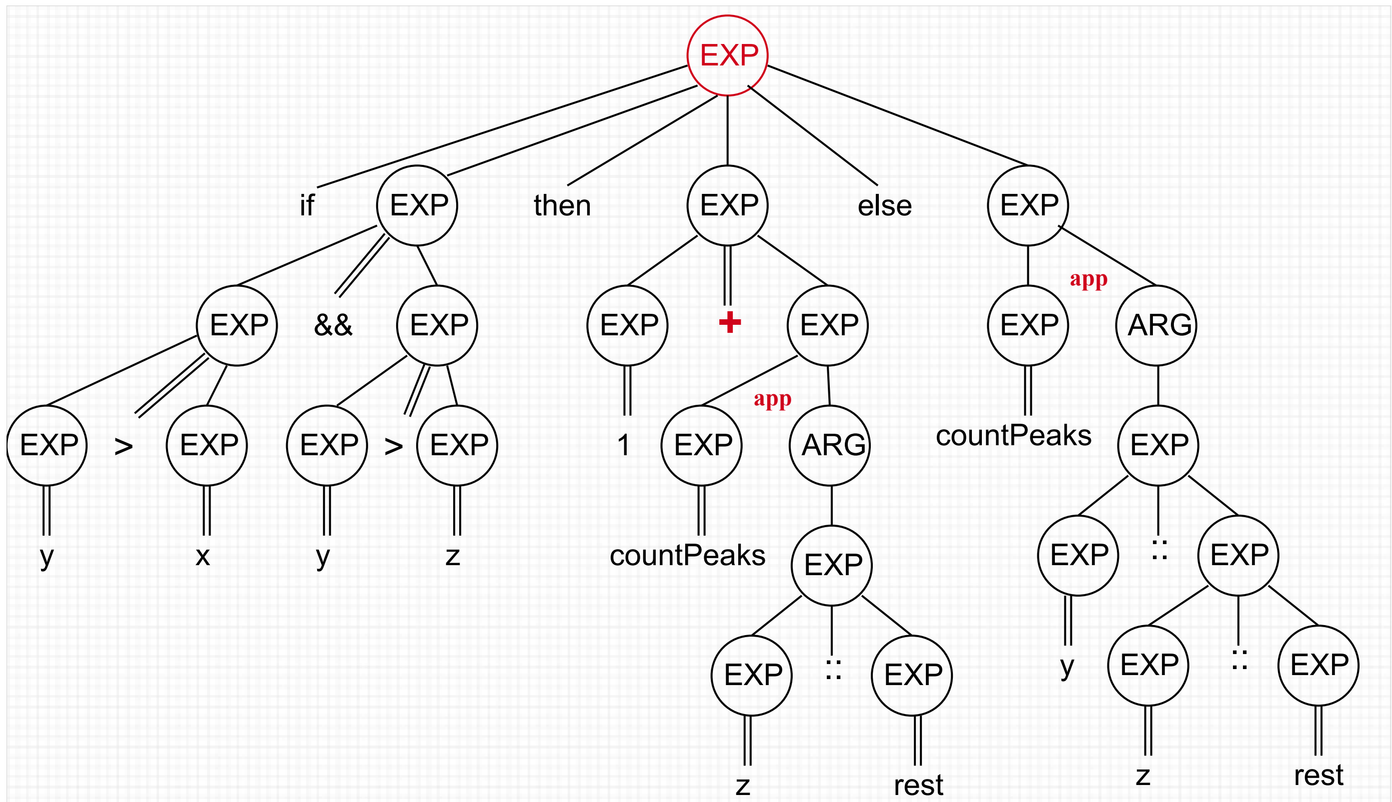
```
let rec countPeaks = function
     ([],n) | ([_],n) | ([_; _],n) -> n
  | (x::y::z::rest,n) -> countPeaks(if   y > x && y > z
                                    then (z::rest, n+1)
                                    else (y::z::rest, n));;
```

More-significant optimizations have to do with editing expressions when some of their values are known **at compile time** or **at activation time**---the latter means when a stack frame for a function/method call is allocated---especially when their value is 0. We will cover this after chapters 5--7.

```
let rec countPeaks ell = match ell with
     [] -> 0
   | [_] -> 0
   | [_; _] -> 0
   | x::y::z::rest -> if   y > x && y > z
                      then 1 + countPeaks(z::rest)
                      else countPeaks (y::z::rest);;
```

Added---whole parse tree of first version of the `countPeaks` program, red EXP nodes coincide:





CSE305 Week 5B, Ch. 5 on Variables: Their Names, Bindings, Values, and Scopes

These notes parallel Sebesta's official PowerPoint slides, which are now copyright (c) 2022 by Pearson, Inc. Mine include minimal literal material from those slides and give mostly different examples, so to be within the bounds of *fair use*. Original content is is copyright (c) K.W. Regan, 1999--2023.

For example, the textbook (12 ed.) says: "A *name* is a string of characters used to identify some entity in a program"---after having said that names are associated not only with variables but "also to subprograms, formal parameters, and other program constructs. The term identifier is often used interchangeably with name." My notes paraphrase this as:

Name = a lexical token that can be bound to an object (e.g., to a procedure, function, variable, type, etc.) Usually synonymous with "**Identifier**."

My notes were oriignally paralleling Sebesta's slides as of 1999; the current slides have nothing like this. Going on in my words: Even the "+" sign may be treated as a name. In C++, its formal name is "**`operator+`**", and one can define new functions with this name, e.g. for vector addition. (This is called "operator overloading.")

Sebesta's older slides had literally:
Names—Design issues:

Maximum length?
Are connector characters allowed?
Are names case sensitive?

The new slides postpone the first two and add "Are the special words of the language **keywords** or **reserved words**?" IMHO those two terms are interchangeable, but the difference is that a **reserved word** may never be used as a user-defined name, whereas a **keyword** is prohibited only in certain contexts. To counterpoint the latter, OCaml allows you to say not only "`let list = 5;;`" but even "`type list = int`". Then if you enter `let ell =[3;4;5];;` at the console, it replies (something like) "`val ell : int list/2 = [3;4;5]`". My advice: Ignore the distinction and **Just Don't Do It**.

The main connector char choice is between _ (underscore) and - (hyphen). The latter can confuse with a binary infix minus sign, but is fine in Lisp because **`x-y`** in Lisp is **(- x y)**. Older languages tended to be non-case-sensitive (or to require `ALL CAPS`), but now mixed-case (especially camelCase) names are commonplace. OCaml assigns different type roles to uppercase and lowercase identifiers. The biggest (IMPHO) issue with names is the inclusion of **module paths** (or **class paths**, or **package paths**) which are usually prefixed via dots. We will address it and the issue of **namespace guarding** versus "polluting the top-level namespace" when we hit chapter 11, in particular section 11.7.

Variables and Storage Objects

Here Sebesta's new slides begin: "A **variable** is an abstraction of a memory cell" (the text adds, "...or collection of cells") and goes on to list six attributes of a variable. Well, OCaml purists may disagree. In mathematical formulas, from which the invention of the idea of a variable sprang, they are abstractions of *values*. What I believe more helpful is to focus on the memory cell(s) themselves and what they represent in a program. The term "storage object" is not in Sebesta; it comes via my colleague Professor Stuart Shapiro from an older textbook.

Definition: A **storage object** is a memory cell or collection of cells used to represent an entity in a program. Its primary attributes are:

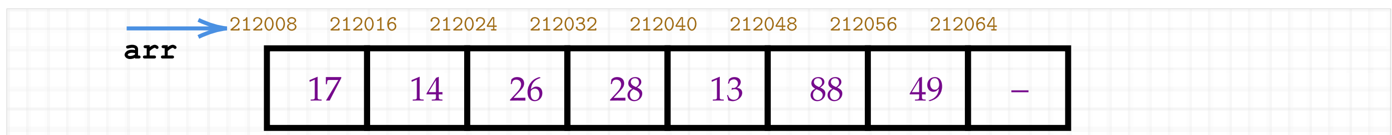
- Durable: its **address** and (machine-based) **type**.
- Transient: its **value** at any particular time---or for its **lifetime**, in the case of a **stored constant** or other **immutable object**).

Program constants assigned values at compile time can be implemented without creating storage objects, but this is often dependent on implementation details and the requested optimization level. In C/C++, if you write `#define PI 3.14159` then you will get the literal number wherever you pretended `PI` was a variable. But if you write `const PI = 3.14159`, even if you add that it is `static`, it will usually be treated as a constant storage object. We will see why even in (the pure core of) OCaml, immutable variables are generally still implemented via storage objects.

Here is a diagram of a simple "atomic" storage object (holding `int` or `float` or `double` or `char`, say---in this case `int`), together with another object denoting a pointer (or **`ref`**) to that cell:

- The **program type** of the storage object is "integer" (`int` or `Integer` or `Integer` depending on the language). The machine type is probably 32-bit int---since the **address** 582684 is divisible by 4 but not by 8, it cannot be *word-aligned* as a 64-bit int.
- The **value** of the storage object (as shown) is the integer 3.
- The **program type** of the pointer object is **`int*`** in C/C++, **`int ref`** if we cover the pointers part of OCaml. In Scala it would be **`unsafe.Cint`**. The **machine type** of the pointer is again just an integer.
- The **value** of the pointer object (as shown) is `582684`, which is the address of the integer storage object it points to.
- Pointer values and integers can be added and otherwise mixed together freely in C/C++. Note, however, if the pointer is named `p` in the program and you do `p+3`, the effect will be to add 3 times the *width* in bytes of the type pointed at. The byte width of `int32` is 4, so the address value of `p+3` will be `582696`, not 582687.

Here is an illustration of a compound object. The implementation layout might be the same for an `int list` as for an `int array` in Ocaml---and no different in the machine code from an integer array or vector in C/C++/Java/etc. I've made them 64-bit ints this time:



To illustrate some low-level details as FYI: If this is a raw C/C++ array named `arr`, then `arr` is implemented as an `int` pointer whose address value is the same as the address of the lead element of the array. The array-lookup **`operator[]`** is formally the same as adding the index multiplied by the element width to the pointer, so here:

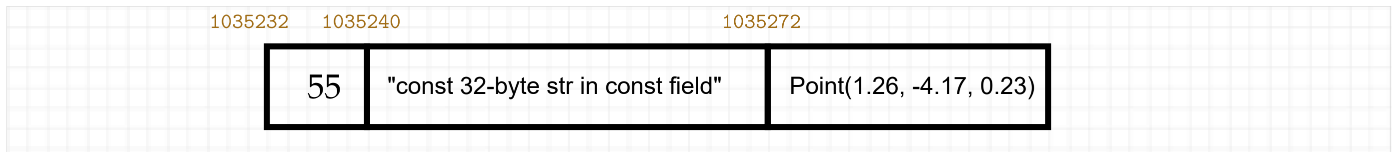
```
arr[i] = arr + i*(width of int64) = arr + 8*i
```

For example, `arr[5]` first computes the address `212008 + 5*8 = 212048`. The object code then automatically **fetches** the value at that address, which is 88. Whenever it happens, this is called **automatic dereferencing**.

The bigger picture important to us now is that the entire collection of cells is a storage object composed of atomic storage objects all of the same type. It has a name in the program, `arr`. The individual

elements do not have lexical names---they are accessed by *indexing* which does a "pointer trace." Note that the array as executed in the program has size 7, but especially if the array is a C++ `vector` or Scala `ArrayBuffer` (which are resizable), the implementation will probably commandeer a power-of-2 number of memory cells, leaving unused ones untouched.

Here is one more diagram of a storage object, compound but **inhomogeneous**:



In OCaml, this could be a tuple of type `int * string * point`, with the last type possibly giving several options for specifying points in space, one being `Point of (float * float * float)`. But this could equally well be a tuple in Python or a class object with three fields in a bunch of programming languages---with the third field being from a `Point` class.

In the case of a class object---or alternately, a **record** with named fields (a feature introduced by COBOL way back in 1958, and which ML and OCaml have too)---the three individual components have names in the program (if the whole object has a name). If the fields are named `serial`, `description`, and `coordinates`, and the whole object is named `item`, then the components have the dotted names `item.serial`, `item.description`, and `item.coordinates`. If this is just a tuple, however, the components might not have names. OCaml has the build-in f*unctions* **`fst`** and **`snd`** for the left and right part of a ***pair***, but they don't work on tuples of size 3 or higher, and there is no **`thd`** function or any indexing function for tuples. (You have to use **`match`** to access their contents.)

This leads to a new observation that refines what Sebesta says about variables and names.

Storage Objects and Names

- Every name in a program (well, the name of a variable, i.e., the kind of identifier OCaml calls a *value-name*) references a storage object that is created when the name is *declared*.
- Not every storage object created during execution has a name, however.

Definition: Generally (says the text), a **binding** is an association between an entity and an attribute, such as between a variable and its type or value, or between an operation and a symbol. The most important use of **binding** (IMPHO) is between a name and an associated storage object (this is what the text means when it says being **bound to an address** in memory).

The text talks about different binding times: (i) *Language* design time, when e.g. operator names are chosen; (ii) *Language implementation time*, when e.g. **`int`** is chosen to mean 32-bit or 64-bit integer; (iii) *Compile time*, when the type of a variable name is determined; (iv) *Load time*, when **static** variables

can be laid out in memory, and (v) *Runtime*, when non-static variables are allocated.

The text defines a binding to be **static** if it is made before any of the program's operations get going and lasts throughout the execution of the program, and **dynamic** if it is made during execution and can possibly change. IMPHO, the latter usage can get confused with "dynamically scoped variables", so I prefer to avoid and simply talk about *regularly bound* variables.

I diagram the binding of a variable **x** to a storage object by a short down-arrow between **x** and the address. It is not a pointer arrow. For a pointer (or explicit reference) variable **p**, the short arrow goes to the pointer's own addrss in memory. Long arrows show the indirect effect of the pointer's own *value*, which is another address.

[Coverage of chapter 5 into 6 will continue next Tuesday.]