# CSE341: Programming Languages Winter 2017
# Unit 2 Summary

*Standard Description: This summary covers* **roughly** *the same material as class and recitation section. It can help to read about the material in a narrative style and to have the material for an entire unit of the course in a single document, especially when reviewing the material later. Please report errors in these notes, even typos. This summary is not a sufficient substitute for attending class, reading the associated code, etc.*

## Contents

## The Pieces of a Programming Language

Now that we have learned enough ML to write some simple functions and programs with it, we can list the essential "pieces" necessary for defining and learning *any* programming language:

- Syntax: How do you write the various parts of the language?
- Semantics: What do the various language features mean? For example, how are expressions evaluated?
- Idioms: What are the common approaches to using the language features to express computations?
- Libraries: What has already been written for you? How do you do things you could not do without library support (like access files)?
- Tools: What is available for manipulating programs in the language (compilers, read-eval-print loops, debuggers, ...)

While libraries and tools are essential for being an effective programmer (to avoid reinventing available solutions or unnecessarily doing things manually), this course does not focus on them much. That can leave the wrong impression that we are using "silly" or "impractical" languages, but libraries and tools are just less relevant in a course on the conceptual similarities and differences of programming languages.

## Conceptual Ways to Build New Types

Programming languages have *base types*, like `int`, `bool`, and `unit` and *compound types*, which are types that contain other types in their definition. We have already seen ways to make compound types in ML, namely by using tuple types, list types, and option types. We will soon learn new ways to make even more flexible compound types and to give names to our new types. To create a compound type, there are really only three essential building blocks. Any decent programming language provides these building blocks in some way:[1]

- "Each-of": A compound type `t` describes values that contain *each of* values of type `t1`, `t2`, ..., ***and*** `tn`.

[1] As a matter of jargon you do not need to know, the terms "each-of types," "one-of types," and "self-reference types" are not standard – they are just good ways to think about the concepts. Usually people just use constructs from a particular language like "tuples" when they are talking about the ideas. Programming-language researchers use the terms "product types," "sum types," and "recursive types." Why product and sum? It is related to the fact that in Boolean algebra where 0 is false and 1 is true, *and* works like multiply and *or* works like addition.

- "One-of": A compound type `t` describes values that contain a value of *one of* the types `t1`, `t2`, ..., ***or*** `tn`.
- "Self-reference": A compound type `t` may refer to itself in its definition in order to describe recursive data structures like lists and trees.

Each-of types are the most familiar to most programmers. Tuples are an example: `int * bool` describes values that contain an `int` *and* a `bool`. A Java class with fields is also an each-of sort of thing.

One-of types are also very common but unfortunately are not emphasized as much in many introductory programming courses. `int option` is a simple example: A value of this type contains an `int` *or* it does not. For a type that contains an `int` *or* a `bool` in ML, we need datatype bindings, which are the main focus of this section of the course. In object-oriented languages with classes like Java, one-of types are achieved with subclassing, but that is a topic for much later in the course.

Self-reference allows types to describe recursive data structures. This is useful in combination with each-of and one-of types. For example, `int list` describes values that either contain nothing *or* contain an `int` *and* another `int list`. A list of integers in any programming language would be described in terms of *or*, *and*, and *self-reference* because that is what it means to be a list of integers.

Naturally, since compound types can nest, we can have any nesting of each-of, one-of, and self-reference. For example, consider the type `(int * bool) list list * (int option) list * bool`.

## Records: Another Approach to "Each-of" Types

Record types are "each-of" types where each component is a *named field*. For example, the type `{foo : int, bar : int*bool, baz : bool*int}` describes records with three fields named `foo`, `bar`, and `baz`. This is just a new sort of type, just like tuple types were new when we learned them.

A *record expression* builds a *record value*. For example, the expression `{bar = (1+2,true andalso true), foo = 3+4, baz = (false,9) }` would evaluate to the record value `{bar = (3,true), foo = 7, baz = (false,9)}`, which can have type `{foo : int, bar : int*bool, baz : bool*int}` because the order of fields never matters (we use the field names instead). In general the syntax for a record expression is `{f1 = e1, ..., fn = en}` where, as always, each `ei` can be any expression. Here each `f` can be any field name (though each must be different). A field name is basically any sequence of letters or numbers.

In ML, we do not have to declare that we want a record type with particular field names and field types — we just write down a record expression and the type-checker gives it the right type. The type-checking rules for record expressions are not surprising: Type-check each expression to get some type `ti` and then build the record type that has all the right fields with the right types. *Because the order of field names never matters, the REPL always alphabetizes them when printing just for consistency.*

The evaluation rules for record expressions are analogous: Evaluate each expression to a value and create the corresponding record value.

Now that we know how to build record values, we need a way to access their pieces. For now, we will use `#foo e` where `foo` is a field name. Type-checking requires `e` has a record type with a field named `foo`, and if this field has type `t`, then that is the type of `#foo e`. Evaluation evaluates `e` to a record value and then produces the contents of the `foo` field.

## By Name vs. By Position, Syntactic Sugar, and The Truth About Tuples

Records and tuples are *very* similar. They are both "each-of" constructs that allow any number of components. The only real difference is that records are "by name" and tuples are "by position." This means with records we build them and access their pieces by using field names, so the order we write the fields in a record expression does not matter. But tuples do not have field names, so we use the position (first, second, third, ...) to distinguish the components.

By name versus by position is a classic decision when designing a language construct or choosing which one to use, with each being more convenient in certain situations. As a rough guide, by position is simpler for a small number of components, but for larger compound types it becomes too difficult to remember which position is which.

Java method arguments (and ML function arguments as we have described them so far) actually take a hybrid approach: The method body uses variable *names* to refer to the different arguments, but the caller passes arguments by *position*. There are other languages where callers pass arguments by name.[2]

Despite "by name vs. by position," records and tuples are still so similar that we can define tuples entirely in terms of records. Here is how:

- When you write `(e1,...,en)`, it is another way of writing `{1=e1,...,n=en}`, i.e., a tuple expression is a record expression with field names 1, 2, ..., $n$.
- The type `t1 * ... * tn` is just another way of writing `{1:t1, ..., n:tn}`.
- Notice that `#1 e`, `#2 e`, etc. now already mean the right thing: get the contents of the field named 1, 2, etc.

In fact, this is how ML actually defines tuples: A tuple *is* a record. That is, all the syntax for tuples is just a convenient way to write down and use records. The REPL just always uses the tuple syntax where possible, so if you evaluate `{2=1+2, 1=3+4}` it will print the result as `(7,3)`. Using the tuple *syntax* is better style, but we did not need to give tuples their own *semantics*: we can instead use the "another way of writing" rules above and then reuse the semantics for records.

This is the first of many examples we will see of ***syntactic sugar***. We say, "tuples are just syntactic sugar for records with fields named 1, 2, ..., $n$." It is *syntactic* because we can describe everything about tuples in terms of equivalent record syntax. It is *sugar* because it makes the language sweeter. The term *syntactic sugar* is widely used. Syntactic sugar is a great way to keep the key ideas in a programming-language small (making it easier to implement) while giving programmers convenient ways to write things. Indeed, in Homework 1 we used tuples without knowing records existed even though tuples are records.

## Datatype Bindings: Our Own "One-of" Types

We now introduce *datatype bindings*, our third kind of binding after variable bindings and function bindings. We start with a silly but simple example because it will help us see the many different aspects of a datatype binding. We can write:

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

[2]The phrase "call by name" actually means something else in relation to function arguments. It is a different topic.

Roughly, this defines a new type where values have an `int * int` *or* a `string` *or* nothing. Any value will also be "tagged" with information that lets us know which *variant* it is: These "tags," which we will call *constructors*, are `TwoInts`, `Str`, and `Pizza`. Two constructors could be used to tag the same type of underlying data; in fact this is common even though our example uses different types for each variant.

More precisely, the example above adds four things to the environment:

- A new type `mytype` that we can now use just like any other type
- Three *constructors* `TwoInts`, `Str`, and `Pizza`

A *constructor* is two different things. First, it is either a function for creating values of the new type (if the variant has `of t` for some type `t`) or it is actually a value of the new type (otherwise). In our example, `TwoInts` is a function of type `int*int -> mytype`, `Str` is a function of type `string->mytype`, and `Pizza` is a value of type `mytype`. Second, we use constructors in case-expressions as described further below.

So we know how to build values of type `mytype`: call the constructors (they are functions) with expressions of the right types (or just use the `Pizza` value). The result of these function calls are values that "know which variant they are" (they store a "tag") and have the underlying data passed to the constructor. The REPL represents these values like `TwoInts(3,4)` or `Str "hi"`.

What remains is a way to retrieve the pieces...

## How ML Does *Not* Provide Access to Datatype Values

Given a value of type `mytype`, how can we access the data stored in it? First, *we need to find out which variant it is* since a value of type `mytype` might have been made from `TwoInts`, `Str`, or `Pizza` and this affects what data is available. Once we know what variant we have, then we can access the pieces, if any, that variant carries.

Recall how we have done this so for lists and options, which are also one-of types: We had functions for testing which variant we had (`null` or `isSome`) and functions for getting the pieces (`hd`, `tl`, or `valOf`), which raised exceptions if given arguments of the wrong variant.

ML could have taken the same approach for datatype bindings. For example, it could have taken our datatype definition above and added to the environment functions `isTwoInts`, `isStr`, and `isPizza` all of type `mytype -> bool`. And it could have added functions like `getTwoInts` of type `mytype -> int*int` and `getStr` of type `mytype -> string`, which might raise exceptions.

But ML does not take this approach. Instead it does something better. You could write these functions yourself using the better thing, though it is usually poor style to do so. In fact, after learning the better thing, we will no longer use the functions for lists and options the way we have been — we just started with these functions so we could learn one thing at a time.

## How ML Provides Access to Datatype Values: Case Expressions

The better thing is a *case expression*. Here is a basic example for our example datatype binding:

```
fun f x =  (* f has type mytype -> int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1 + i2
      | Str s => String.size s
```

In one sense, a case-expression is like a more powerful if-then-else expression: Like a conditional expression, it evaluates two of its subexpressions: first the expression between the `case` and `of` keywords and second the expression in the *first branch that matches*. But instead of having two branches (one for `true` and one for `false`), we can have one branch for each variant of our datatype (and we will generalize this further below). Like conditional expressions, each branch's expression must have the same type (`int` in the example above) because the type-checker cannot know what branch will be used.

Each branch has the form `p => e` where `p` is a *pattern* and `e` is an expression, and we separate the branches with the `|` character. Patterns look like expressions, but do not think of them as expressions. Instead they are used to *match* against the result of evaluating the case's first expression (the part after `case`). This is why evaluating a case-expression is called *pattern-matching*.

For now (to be significantly generalized soon), we keep pattern-matching simple: Each pattern uses a different constructor and pattern-matching picks the branch with the "right one" given the expression after the word `case`. The result of evaluating that branch is the overall answer; no other branches are evaluated. For example, if `TwoInts(7,9)` is passed to `f`, then the second branch will be chosen.

That takes care of the "check the variant" part of using the one-of type, but pattern matching *also* takes care of the "get out the underlying data" part. Since `TwoInts` has two values it "carries", a pattern for it can (and, for now, must) use two variables (the `(i1,i2)`). As part of matching, the corresponding parts of the value (continuing our example, the 7 and the 9) are bound to `i1` and `i2` in the environment used to evaluate the corresponding right-hand side (the `i1+i2`). In this sense, pattern-matching is like a let-expression: It binds variables in a local scope. The type-checker knows what types these variables have because they were specified in the datatype binding that created the constructor used in the pattern.

Why are case-expressions better than functions for testing variants and extracting pieces?

- We can never "mess up" and try to extract something from the wrong variant. That is, we will not get exceptions like we get with `hd []`.
- If a case expression forgets a variant, then the type-checker will give a warning message. This indicates that evaluating the case-expression could find no matching branch, in which case it will raise an exception. If you have no such warnings, then you know this does not occur.
- If a case expression uses a variant twice, then the type-checker will give an error message since one of the branches could never possibly be used.
- If you still want functions like `null` and `hd`, you can easily write them yourself (but do not do so for your homework).
- Pattern-matching is much more general and powerful than we have indicated so far. We give the "whole truth" about pattern-matching below.

## Useful Examples of "One-of" Types

Let us now consider several examples where "one-of" types are useful, since so far we considered only a silly example.

First, they are good for enumerating a fixed set of options – and much better style than using, say, small integers. For example:

```
datatype suit = Club | Diamond | Heart | Spade
```

Many languages have support for this sort of *enumeration* including Java and C, but ML takes the next step of letting variants carry data, so we can do things like this:

```
datatype rank = Jack | Queen | King | Ace | Num of int
```

We can then combine the two pieces with an each-of type: `suit * rank`

One-of types are also useful when you have different data in different situations. For example, suppose you want to identify students by their id-numbers, but in case there are students that do not have one (perhaps they are new to the university), then you will use their full name instead (with first name, optional middle name, and last name). This datatype binding captures the idea directly:

```
datatype id = StudentNum of int
            | Name of string * (string option) * string
```

Unfortunately, this sort of example is one where programmers often show a profound lack of understanding of one-of types and insist on using each-of types, which is like using a saw as a hammer (it works, but you are doing the wrong thing). Consider BAD code like this:

```
(* If student_num is -1, then use the other fields, otherwise ignore other fields *)
 {student_num : int, first : string, middle : string option, last : string}
```

This approach requires all the code to follow the rules in the comment, with no help from the type-checker. It also wastes space, having fields in every record that should not be used.

On the other hand, each-of types are exactly the right approach if we want to store for each student their id-number (if they have one) *and* their full name:

```
{ student_num : int option,
  first       : string,
  middle      : string option,
  last        : string }
```

Our last example is a data definition for arithmetic expressions containing constants, negations, additions, and multiplications.

```
datatype exp = Constant of int
             | Negate of exp
             | Add of exp * exp
             | Multiply of exp * exp
```

Thanks to the self-reference, what this data definition really describes is *trees* where the leaves are integers and the internal nodes are either negations with one child, additions with two children or multiplications with two children. We can write a function that takes an `exp` and evaluates it:

```
fun eval e =
    case e of
        Constant i => i
      | Negate e2  => ~ (eval e2)
      | Add(e1,e2) => (eval e1) + (eval e2)
      | Multiply(e1,e2) => (eval e1) * (eval e2)
```

So this function call evaluates to 15:

```
eval (Add (Constant 19, Negate (Constant 4)))
```

Notice how constructors are just functions that we call with other expressions (often other values built from constructors).

There are many functions we might write over values of type `exp` and most of them will use pattern-matching and recursion in a similar way. Here are other functions you could write that process an `exp` argument:

- The largest constant in an expression
- A list of all the constants in an expression (use list append)
- A `bool` indicating whether there is at least one multiplication in the expression
- The number of addition expressions in an expression

Here is the last one:

```
fun number_of_adds e =
    case e of
        Constant i      => 0
      | Negate e2       => number_of_adds e2
      | Add(e1,e2)      => 1 + number_of_adds e1 + number_of_adds e2
      | Multiply(e1,e2) => number_of_adds e1 + number_of_adds e2
```

## Datatype Bindings and Case Expressions So Far, Precisely

We can summarize what we know about datatypes and pattern matching so far as follows: The binding

```
datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

introduces a new type `t` and each constructor `Ci` is a function of type `ti->t`. One omits the "`of ti`" for a variant that "carries nothing" and such a constructor just has type `t`. To "get at the pieces" of a `t` we use a case expression:

```
case e of p1 => e1 | p2 => e2 | ... | pn => en
```

A case expression evaluates `e` to a value `v`, finds the first pattern `pi` that *matches* `v`, and evaluates `ei` to produce the result for the whole case expression. So far, patterns have looked like `Ci(x1,...,xn)` where `Ci` is a constructor of type `t1 * ... * tn -> t` (or just `Ci` if `Ci` carries nothing). Such a pattern matches a value of the form `Ci(v1,...,vn)` and binds each `xi` to `vi` for evaluating the corresponding `ei`.

## Type Synonyms

Before continuing our discussion of datatypes, let's contrast them with another useful kind of binding that also introduces a new type name. A *type synonym* simply creates another name for an existing type that is entirely interchangeable with the existing type.

For example, if we write:

```
type foo = int
```

then we can write `foo` wherever we write `int` and vice-versa. So given a function of type `foo->foo` we could call the function with 3 and add the result to 4. The REPL will sometimes print `foo` and sometimes print `int` depending on the situation; the details are unimportant and up to the language implementation. For a type like `int`, such a synonym is not very useful (though later when we study ML's module system we will build on this feature).

But for more complicated types, it can be convenient to create type synonyms. Here are some examples for types we created above:

```
type card = suit * rank

type name_record = { student_num : int option,
                     first       : string,
                     middle      : string option,
                     last        : string }
```

Just remember these synonyms are fully interchangeable. For example, if a homework question requires a function of type `card -> int` and the REPL reports your solution has type `suit * rank -> int`, this is okay because the types are "the same."

In contrast, datatype bindings introduce a type that is not the same as any existing type. It creates constructors that produces values of this new type. So, for example, the only type that is the same as `suit` is `suit` unless we later introduce a synonym for it.

## Lists and Options are Datatypes

Because datatype definitions can be recursive, we can use them to create our own types for lists. For example, this binding works well for a linked list of integers:

```
datatype my_int_list = Empty
                     | Cons of int * my_int_list
```

We can use the constructors `Empty` and `Cons` to make values of `my_int_list` and we can use case expressions to use such values:[3]

```
val one_two_three = Cons(1,Cons(2,Cons(3,Empty)))

fun append_mylist (xs,ys) =
    case xs of
        Empty => ys
      | Cons(x,xs') => Cons(x, append_mylist(xs',ys))
```

It turns out the lists and options "built in" (i.e., predefined with some special syntactic support) are just datatypes. As a matter of style, it is better to use the built-in widely-known feature than to invent your own.

[3] In this example, we use a variable `xs'`. Many languages do not allow the character `'` in variable names, but ML does and it is common in mathematics to use it and pronounce such a variable "exes prime."

More importantly, it is better style to use pattern-matching for accessing list and option values, ***not*** the functions `null`, `hd`, `tl`, `isSome`, and `valOf` we saw previously. (We used them because we had not learned pattern-matching yet and we did not want to delay practicing our functional-programming skills.)

For options, all you need to know is `SOME` and `NONE` are constructors, which we use to create values (just like before) and in patterns to access the values. Here is a short example of the latter:

```
fun inc_or_zero intoption =
    case intoption of
        NONE => 0
      | SOME i => i+1
```

The story for lists is similar with a few convenient syntactic peculiarities: `[]` really is a constructor that carries nothing and `::` really is a constructor that carries two things, but `::` is unusual because it is an infix operator (it is placed between its two operands), both when creating things and in patterns:

```
fun sum_list xs =
    case xs of
        [] => 0
      | x::xs' => x + sum_list xs'

fun append (xs,ys) =
    case xs of
        [] => ys
      | x::xs' => x :: append(xs',ys)
```

Notice here `x` and `xs'` are nothing but local variables introduced via pattern-matching. We can use any names for the variables we want. We could even use `hd` and `tl` — doing so would simply shadow the functions predefined in the outer environment.

The reasons why you should usually prefer pattern-matching for accessing lists and options instead of functions like `null` and `hd` is the same as for datatype bindings in general: you cannot forget cases, you cannot apply the wrong function, etc. So why does the ML environment predefine these functions if the approach is inferior? In part, because they are useful for passing as arguments to other functions, a major topic for the next section of the course.

## Polymorphic Datatypes

Other than the strange syntax of `[]` and `::`, the only thing that distinguishes the built-in lists and options from our example datatype bindings is that the built-in ones are *polymorphic* – they can be used for carrying values of *any* type, as we have seen with `int list`, `int list list`, `(bool * int) list`, etc. You can do this for your own datatype bindings too, and indeed it is very useful for building "generic" data structures. While we will not focus on using this feature here (i.e., you are not responsible for knowing how to do it), there is nothing very complicated about it. For example, this is *exactly* how options are pre-defined in the environment:

```
datatype 'a option = NONE | SOME of 'a
```

Such a binding does *not* introduce a *type* `option`. Rather, it makes it so that if `t` is a type, then `t option` is type. You can also define polymorphic datatypes that take multiple types. For example, here is a binary tree where internal nodes hold values of type `'a` and leaves hold values of type `'b`

```
datatype ('a,'b) tree = Node of 'a * ('a,'b) tree * ('a,'b) tree
                      | Leaf of 'b
```

We then have types like `(int,int) tree` (in which every node and leaf holds an `int`) and `(string,bool) tree` (in which every node holds a `string` and every leaf holds a `bool`). The way you use constructors and pattern-matching is the same for regular datatypes and polymorphic datatypes.

## Pattern-Matching for Each-Of Types: The Truth About Val-Bindings

So far we have used pattern-matching for one-of types, but we can use it for each-of types also. Given a record value `{f1=v1,...,fn=vn}`, the pattern `{f1=x1,...,fn=xn}` matches and binds `xi` to `vi`. As you might expect, the order of fields in the pattern does not matter. As before, tuples are syntactic sugar for records: the pattern `(x1,...,xn)` is the same as `{1=x1,...,n=xn}` and matches the tuple value `(v1,...,vn)`, which is the same as `{1=v1,...,n=vn}`. So we could write this function for summing the three parts of an `int * int * int`:

```
fun sum_triple (triple : int * int * int) =
    case triple of
      (x,y,z) => z + y + x
```

And a similar example with records (and ML's string-concatenation operator) could look like this:

```
fun full_name (r : {first:string,middle:string,last:string}) =
    case r of
        {first=x,middle=y,last=z} => x ^ " " ^ y ^ " " ^z
```

However, a case-expression with one branch is poor style — it looks strange because the purpose of such expressions is to distinguish *cases*, plural. So how should we use pattern-matching for each-of types, when we know that a single pattern will definitely match so we are using pattern-matching just for the convenient extraction of values? It turns out you can use patterns in val-bindings too! So this approach is better style:

```
fun full_name (r : {first:string,middle:string,last:string}) =
    let val {first=x,middle=y,last=z} = r
    in
        x ^ " " ^ y ^ " " ^z
    end
fun sum_triple (triple : int*int*int) =
    let val (x,y,z) = triple
    in
        x + y + z
    end
```

Actually we can do even better: Just like a pattern can be used in a val-binding to bind variables (e.g., `x`, `y`, and `z`) to the various pieces of the expression (e.g., `triple`), we can use a pattern when defining a function binding and the pattern will be used to introduce bindings by matching against the value passed when the function is called. So here is the third and best approach for our example functions:

```
fun full_name {first=x,middle=y,last=z} =
    x ^ " " ^ y ^ " " ^z
```

```
fun sum_triple (x,y,z) =
    x + y + z
```

This version of `sum_triple` should intrigue you: It takes a triple as an argument and uses pattern-matching to bind three variables to the three pieces for use in the function body. But it looks exactly like a function that takes three arguments of type `int`. Indeed, is the type `int*int*int->int` for three-argument functions or for one argument functions that take triples?

It turns out we have been basically lying: There is no such thing as a multi-argument function in ML: ***Every function in ML takes exactly one argument!*** Every time we write a multi-argument function, we are really writing a one-argument function that takes a tuple as an argument and uses pattern-matching to extract the pieces. This is such a common idiom that it is easy to forget about and it is totally fine to talk about "multi-argument functions" when discussing your ML code with friends. But in terms of the actual language definition, it really is a one-argument function: syntactic sugar for expanding out to the first version of `sum_triple` with a one-arm case expression.

This flexibility is sometimes useful. In languages like C and Java, you cannot have one function/method compute the results that are immediately passed to another multi-argument function/method. But with one-argument functions that are tuples, this works fine. Here is a silly example where we "rotate a triple to the right" by "rotating it to the left twice":

```
fun rotate_left (x,y,z) = (y,z,x)
fun rotate_right triple = rotate_left(rotate_left triple)
```

More generally, you can compute tuples and then pass them to functions even if the writer of that function was thinking in terms of multiple arguments.

What about zero-argument functions? They do not exist either. The binding `fun f () = e` is using the unit-pattern `()` to match against calls that pass the unit value `()`, which is the only value of type `unit`. The type unit is just a datatype with only one constructor, which takes no arguments and uses the unusual syntax `()`. Basically, `datatype unit = ()` comes pre-defined.

## Digression: Type inference

By using patterns to access values of tuples and records rather than `#foo`, you will find it is no longer necessary to write types on your function arguments. In fact, it is conventional in ML to leave them off — you can always use the REPL to find out a function's type. The reason we needed them before is that `#foo` does not give enough information to type-check the function because the type-checker does not know what other fields the record is supposed to have, but the record/tuple patterns introduced above provide this information. In ML, every variable and function has a type (or your program fails to type-check) — type inference *only* means you do not need to write down the type.

So none of our examples above that used pattern-matching instead of `#middle` or `#2` need argument types. It is often better style to write these less cluttered versions, where again the last one is the best:

```
fun sum_triple triple =
    case triple of
      (x,y,z) => z + y + x
fun sum_triple triple =
    let val (x,y,z) = triple
    in
```

```
        x + y + z
    end
fun sum_triple (x,y,z) =
    x + y + z
```

This version needs an explicit type on the argument:

```
fun sum_triple (triple : int * int * int) =
    #1 triple + #2 triple + #3 triple
```

The reason is the type-checker cannot take

```
fun sum_triple triple =
    #1 triple + #2 triple + #3 triple
```

and infer that the argument must have type `int*int*int`, since it could also have type `int*int*int*int` or `int*int*int*string` or `int*int*int*bool*string` or an infinite number of other types. If you do not use `#`, ML almost never requires explicit type annotations thanks to the convenience of type inference.

In fact, type inference sometimes reveals that functions are more general than you might have thought. Consider this code, which does use part of a tuple/record:

```
fun partial_sum (x,y,z) = x + z
fun partial_name {first=x, middle=y, last=z} = x ^ " " ^ z
```

In both cases, the inferred function types reveal that the type of `y` can be *any* type, so we can call `partial_sum (3,4,5)` or `partial_sum (3,false,5)`.

We will discuss these *polymorphic functions* as well as how *type inference* works in future sections because they are major course topics in their own right. For now, just stop using `#`, stop writing argument types, and do not be confused if you see the occasional type like `'a` or `'b` due to type inference, as discussed a bit more next...

## Digression: Polymorphic Types and Equality Types

We now encourage you to leave explicit type annotations out of your program, but as seen above that can lead to surprisingly general types. Suppose you are asked to write a function of type `int*int*int -> int` that behaves like `partial_sum` above, but the REPL indicates, correctly, that `partial_sum` has type `int*'a*int->int`. *This is okay* because the *polymorphism* indicates that `partial_sum` has a *more general* type. If you can take a type containing `'a`, `'b`, `'c`, etc. and replace each of these *type variables* consistently to get the type you "want," then you have a more general type than the one you want.

As another example, `append` as we have written it has type `'a list * 'a list -> 'a list`, so by consistently replacing `'a` with `string`, we can use `append` as though it has the type `string list * string list -> string list`. We can do this with any type, not just `string`. And we do not actually *do* anything: this is just a mental exercise to check that a type is more general than the one we need. Note that type variables like `'a` must be replaced *consistently*, meaning the type of `append` is *not* more general than `string list * int list -> string list`.

You may also see type variables with two leading apostrophes, like `''a`. These are called *equality types* and they are a fairly strange feature of ML not relevant to our current studies. Basically, the `=` operator in ML

(for comparing things) works for many types, not just `int`, but its two operands must have the same type. For example, it works for `string` as well as tuple types for which all types in the tuple support equality (e.g., `int * (string * bool)`. But it does not work for every type.[4] A type like `''a` can only have an "equality type" substituted for it.

```
fun same_thing(x,y) = if x=y then "yes" else "no" (* has type ''a * ''a -> string *)
fun is_three x = if x=3 then "yes" else "no" (* has type int -> string *)
```

Again, we will discuss polymorphic types and type inference more later, but this digression is helpful for avoiding confusion on Homework 2: if you write a function that the REPL gives a more general type to than you need, that is okay. Also remember, as discussed above, that it is also okay if the REPL uses different type synonyms than you expect.

## Nested Patterns

It turns out the definition of patterns is recursive: anywhere we have been putting a variable in our patterns, we can instead put another pattern. Roughly speaking, the semantics of pattern-matching is that the value being matched must have the same "shape" as the pattern and variables are bound to the "right pieces." (This is very hand-wavy explanation which is why a precise definition is described below.) For example, the pattern `a::(b::(c::d))` would match any list with at least 3 elements and it would bind `a` to the first element, `b` to the second, `c` to the third, and `d` to the list holding all the other elements (if any). The pattern `a::(b::(c::[]))` on the other hand, would match only lists with exactly three elements. Another nested patterns is `(a,b,c)::d`, which matches any non-empty list of triples, binding `a` to the first component of the head, `b` to the second component of the head, `c` to the third component of the head, and `d` to the tail of the list.

In general, pattern-matching is about taking a value and a pattern and (1) deciding if the pattern matches the value and (2) if so, binding variables to the right parts of the value. Here are some key parts to the elegant recursive definition of pattern matching:

- A variable pattern (`x`) matches any value `v` and introduces one binding (from `x` to `v`).
- The pattern `C` matches the value `C`, if `C` is a constructor that carries no data.
- The pattern `C p` where `C` is a constructor and `p` is a pattern matches a value of the form `C v` (notice the constructors are the same) if `p` matches `v` (i.e., the nested pattern matches the carried value). It introduces the bindings that `p` matching `v` introduces.
- The pattern `(p1,p2,...,pn)` matches a tuple value `(v1,v2,...,vn)` if `p1` matches `v1` and `p2` matches `v2`, ..., and `pn` matches `vn`. It introduces all the bindings that the recursive matches introduce.
- (A similar case for record patterns of the form `{f1=p1,...,fn=pn}` ...)

This recursive definition extends our previous understanding in two interesting ways. First, for a constructor `C` that carries multiple arguments, we do not have to write patterns like `C(x1,...,xn)` though we often do. We could also write `C x`; this would bind `x` to the tuple that the value `C(v1,...,vn)` carries. What is really going on is that all constructors take 0 or 1 arguments, but the 1 argument can itself be a tuple. So `C(x1,...,xn)` is really a nested pattern where the `(x1,...,xn)` part is just a pattern that matches all tuples

[4] It does not work for functions since it is impossible to tell if two functions always do the same thing. It also does not work for type `real` to enforce the rule that, due to rounding of floating-point values, comparing them is almost always wrong algorithmically.

with $n$ parts. Second, and more importantly, we can use nested patterns instead of nested case expressions when we want to match only values that have a certain "shape."

There are additional kinds of patterns as well. Sometimes we do not need to bind a variable to part of a value. For example, consider this function for computing a list's length:

```
fun len xs =
    case xs of
       [] => 0
     | x::xs' => 1 + len xs'
```

We do not use the variable `x`. In such cases, it is better style not to introduce a variable. Instead, the *wildcard pattern* `_` matches everything (just like a variable pattern matches everything), but does not introduce a binding. So we should write:

```
fun len xs =
    case xs of
       [] => 0
      | _::xs' => 1 + len xs'
```

In terms of our general definition, wildcard patterns are straightforward:

- A wildcard pattern (`_`) matches any value `v` and introduces no bindings.

Lastly, you can use integer constants in patterns. For example, the pattern `37` matches the value `37` and introduces no bindings.

## Useful Examples of Nested Patterns

An elegant example of using nested patterns rather than an ugly mess of nested case-expressions is "zipping" or "unzipping" lists (three of them in this example):[5]

```
exception BadTriple

fun zip3 list_triple =
    case list_triple of
        ([],[],[]) => []
      | (hd1::tl1,hd2::tl2,hd3::tl3) => (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
      | _ => raise BadTriple

fun unzip3 lst =
    case lst of
        [] => ([],[],[])
      | (a,b,c)::tl => let val (l1,l2,l3) = unzip3 tl
                       in
                           (a::l1,b::l2,c::l3)
                       end
```

[5]Exceptions are discussed below but are not the important part of this example.

This example checks that a list of integers is sorted:

```
fun nondecreasing intlist =
    case intlist of
        [] => true
      | _::[] => true
      | head::(neck::rest) => (head <= neck andalso nondecreasing (neck::rest))
```

It is also sometimes elegant to compare two values by matching against a pair of them. This example, for determining the sign that a multiplication would have without performing the multiplication, is a bit silly but demonstrates the idea:

```
datatype sgn = P | N | Z

fun multsign (x1,x2) =
  let fun sign x = if x=0 then Z else if x>0 then P else N
  in
      case (sign x1,sign x2) of
          (Z,_) => Z
        | (_,Z) => Z
        | (P,P) => P
        | (N,N) => P
        | _     => N (* many say bad style; I am okay with it *)
  end
```

The style of this last case deserves discussion: When you include a "catch-all" case at the bottom like this, you are giving up any checking that you did not forget any cases: after all, it matches anything the earlier cases did not, so the type-checker will certainly not think you forgot any cases. So you need to be extra careful if using this sort of technique and it is probably less error-prone to enumerate the remaining cases (in this case `(N,P)` and `(P,N)`). That the type-checker will then still determine that no cases are missing is useful and non-trivial since it has to reason about the use `(Z,_)` and `(_,Z)` to figure out that there are no missing possibilities of type `sgn * sgn`.

## *Optional:* Multiple Cases in a Function Binding

So far, we have seen pattern-matching on one-of types in case expressions. We also have seen the good style of pattern-matching each-of types in val or function bindings and that this is what a "multi-argument function" really is. But is there a way to match against one-of types in val/function bindings? This seems like a bad idea since we need multiple possibilities. But it turns out ML has special syntax for doing this in function definitions. Here are two examples, one for our own datatype and one for lists:

```
datatype exp = Constant of int | Negate of exp | Add of exp * exp | Multiply of exp * exp

fun eval (Constant i) = i
  | eval (Negate e2) = ~ (eval e2)
  | eval (Add(e1,e2)) = (eval e1) + (eval e2)
  | eval (Multiply(e1,e2)) = (eval e1) * (eval e2)

fun append ([],ys) = ys
  | append (x::xs',ys) = x :: append(xs',ys)
```

As a matter of *taste*, your instructor has never liked this style very much, and you have to get parentheses in the right places. But it is common among ML programmers, so you are welcome to as well. As a matter of *semantics*, it is just syntactic sugar for a single function body that is a case expression:

```
fun eval e =
    case e of
        Constant i => i
      | Negate e2  => ~ (eval e2)
      | Add(e1,e2) => (eval e1) + (eval e2)
      | Multiply(e1,e2) => (eval e1) * (eval e2)

fun append e =
    case e of
       ([],ys) => ys
     | (x::xs',ys) => x :: append(xs',ys)
```

In general, the syntax

```
fun f p1 = e1
|   f p2 = e2
...
|   f pn = en
```

is just syntactic sugar for:[6]

```
fun f x =
   case x of
     p1 => e1
   | p2 => e2
...
   | pn => en
```

Notice the `append` example uses nested patterns: each branch matches a pair of lists, by putting patterns (e.g., `[]` or `x::xs'`) inside other patterns.

## Exceptions

ML has a built-in notion of exception. You can *raise* (also known as *throw*) an exception with the `raise` primitive. For example, the `hd` function in the standard library raises the `List.Empty` exception when called with `[]`:

```
fun hd xs =
    case xs of
        []   => raise List.Empty
      | x::_ => x
```

[6] As a technicality, `x` must be some variable not already defined in the outer environment and used by one of the expressions in the function.

You can create your own kinds of exceptions with an exception binding. Exceptions can optionally carry values with them, which let the code raising the exception provide more information:

```
exception MyUndesirableCondition
exception MyOtherException of int * int
```

Kinds of exceptions are a *lot* like constructors of a datatype binding. Indeed, they are functions (if they carry values) or values (if they don't) that create values of type `exn` rather than the type of a datatype. So `Empty`, `MyUndesirableCondition`, and `MyOtherException(3,9)` are all values of type `exn`, whereas `MyOtherException` has type `int*int->exn`.

Usually we just use exception constructors as arguments to `raise`, such as `raise MyOtherException(3,9)`, but we can use them more generally to create values of type `exn`. For example, here is a version of a function that returns the maximum element in a list of integers. Rather than return an option or raise a particular exception like `List.Empty` if called with `[]`, it takes an argument of type `exn` and raises it. So the caller can pass in the exception of its choice. (The type-checker can infer that `ex` must have type `exn` because that is the type `raise` expects for its argument.)

```
fun maxlist (xs,ex) =
    case xs of
        [] => raise ex
      | x::[] => x
      | x::xs' => Int.max(x,maxlist(xs',ex))
```

Notice that calling `maxlist([3,4,0],List.Empty)` would not raise an exception; this call passes an exception *value* to the function, which the function then does not *raise*.

The other feature related to exceptions is *handling* (also known as *catching*) them. For this, ML has handle-expressions, which look like `e1 handle p => e2` where `e1` and `e2` are expressions and `p` is a pattern that matches an exception. The semantics is to evaluate `e1` and have the result be the answer. But if an exception matching `p` is raised by `e1`, then `e2` is evaluated and that is the answer for the whole expression. If `e1` raises an exception that does not match `p`, then the entire handle-expression also raises that exception. Similarly, if `e2` raises an exception, then the whole expression also raises an exception.

As with case-expressions, handle-expression can also have multiple branches each with a pattern and expression, syntactically separated by `|`.

## Tail Recursion and Accumulators

This topic involves new programming idioms, but no new language constructs. It defines *tail recursion*, describes how it relates to writing *efficient* recursive functions in functional languages like ML, and presents how to use *accumulators* as a technique to make some functions tail recursive.

To understand tail recursion and accumulators, consider these functions for summing the elements of a list:

```
fun sum1 xs =
    case xs of
        [] => 0
      | i::xs' => i + sum1 xs'
```

```
fun sum2 xs =
    let fun f (xs,acc) =
            case xs of
                [] => acc
              | i::xs' => f(xs',i+acc)
    in
        f(xs,0)
    end
```

Both functions compute the same results, but `sum2` is more complicated, using a local helper function that takes an extra argument, called `acc` for "accumulator." In the base case of `f` we return `acc` and the value passed for the outermost call is 0, the same value used in the base case of `sum1`. This pattern is common: The base case in the non-accumulator style becomes the initial accumulator and the base case in the accumulator style just returns the accumulator.

Why might `sum2` be preferred when it is clearly more complicated? To answer, we need to understand a little bit about how function calls are implemented. Conceptually, there is a *call stack*, which is a stack (the data structure with push and pop operations) with one element for each function call that has been started but has not yet completed. Each element stores things like the value of local variables and what part of the function has not been evaluated yet. When the evaluation of one function body calls another function, a new element is pushed on the call stack and it is popped off when the called function completes.

So for `sum1`, there will be one call-stack element (sometimes just called a "stack frame") for each recursive call to `sum1`, i.e., the stack will be as big as the list. This is necessary because after each stack frame is popped off the caller has to, "do the rest of the body" — namely add `i` to the recursive result and return.

Given the description so far, `sum2` is no better: `sum2` makes a call to `f` which then makes one recursive call for each list element. However, when `f` makes a recursive call to `f`, *there is nothing more for the caller to do after the callee returns except return the callee's result.* This situation is called a *tail call* (let's not try to figure out why it's called this) and functional languages like ML typically promise an essential optimization: When a call is a tail call, the caller's stack-frame is popped *before* the call — the callee's stack-frame just *replaces* the caller's. This makes sense: the caller was just going to return the callee's result anyway. Therefore, calls to `sum2` never use more than 1 stack frame.

Why do implementations of functional languages include this optimization? By doing so, recursion can sometimes be as efficient as a while-loop, which also does not make the call-stack bigger. The "sometimes" is exactly when calls are tail calls, something you the programmer can reason about since you can look at the code and identify which calls are tail calls.

Tail calls do not need to be to the same function (`f` can call `g`), so they are more flexible than while-loops that always have to "call" the same loop. Using an accumulator is a common way to turn a recursive function into a "tail-recursive function" (one where all recursive calls are tail calls), but not always. For example, functions that process trees (instead of lists) typically have call stacks that grow as big as the depth of a tree, but that's true in any language: while-loops are not very useful for processing trees.

## More Examples of Tail Recursion

Tail recursion is common for functions that process lists, but the concept is more general. For example, here are two implementations of the factorial function where the second one uses a tail-recursive helper function so that it needs only a small constant amount of call-stack space:

```
fun fact1 n = if n=0 then 1 else n * fact1(n-1)
```

```
fun fact2 n =
    let fun aux(n,acc) = if n=0 then acc else aux(n-1,acc*n)
    in
        aux(n,1)
    end
```

It is worth noticing that `fact1 4` and `fact2 4` produce the same answer even though the former performs $4*(3*(2*(1*1)))$ and the latter performs $(((1*4)*3)*2)*1$. We are relying on the fact that multiplication is associative ($a*(b*c) = (a*b)*c$) and that multiplying by 1 is the identity function ($1*x = x*1 = x$). The earlier `sum` example made analogous assumptions about addition. In general, converting a non-tail-recursive function to a tail-recursive function usually needs associativity, but many functions are associative.

A more interesting example is this inefficient function for reversing a list:

```
fun rev1 lst =
   case lst of
       [] => []
     | x::xs => (rev1 xs) @ [x]
```

We can recognize immediately that it is not tail-recursive since after the recursive call it remains to append the result onto the one-element list that holds the head of the list. Although this is the most natural way to reverse a list recursively, the inefficiency is caused by more than creating a call-stack of depth equal to the argument's length, which we will call $n$. The worse problem is that the total amount of work performed is proportional to $n^2$, i.e., this is a quadratic algorithm. The reason is that appending two lists takes time proportional to the length of the first list: it has to traverse the first list — see our own implementations of append discussed previously. Over all the recursive calls to `rev1`, we call `@` with first arguments of length $n-1, n-2, ..., 1$ and the sum of the integers from 1 to $n-1$ is $n*(n-1)/2$.

As you learn in a data structures and algorithms course, quadratic algorithms like this are much slower than linear algorithms for large enough $n$. That said, if you expect $n$ to always be small, it may be be worth valuing the programmer's time and sticking with a simple recursive algorithm. Else, fortunately, using the accumulator idiom leads to an almost-as-simple linear algorithm.

```
fun rev2 lst =
    let fun aux(lst,acc) =
            case lst of
                [] => acc
              | x::xs => aux(xs, x::acc)
    in
        aux(lst,[])
    end
```

The key differences are (1) tail recursion and (2) we do only a constant amount of work for each recursive call because `::` does not have to traverse either of its arguments.

# A Precise Definition of Tail Position

While most people rely on intuition for, "which calls are tail calls," we can be more precise by defining *tail position* recursively and saying a call is a tail call if it is in tail position. The definition has one part for each kind of expression; here are several parts:

- In `fun f(x) = e`, `e` is in tail position.
- If an expression is not in tail position, then none of its subexpressions are in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (but not `e1`). (Case-expressions are similar.)
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (but no expressions in the bindings are).
- Function-call arguments are not in tail position.
- ...