University of Washington CSE 341: Programming Languages
Unit 2 Reading Notes

Standard Description: This summary covers **some of** *the same material as lecture and 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.*

Last updated: July 22, 2022

Contents

The Pieces of a Programming Language

Now that we have learned enough OCaml 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.

Conversely, while syntax is essential and may indeed in practice be very important, from a programming-languages perspective, it is relatively uninteresting. For any concept in our language, we need some way to write it down. We want what we are writing to be unambiguous and we might have various preferences or want some shortcuts or conventions across languages, but we can vie syntax as just a necessary thing to learn.

That leaves semantics and idioms as the key things for this course to focus on: What do different programming constructs mean, how can we define elegant, orthogonal concepts that can then be composed in powerful ways, and what are the common ways people use fundamental constructs to accomplish their goals in software?

Conceptual Ways to Build New Types

Programming languages have *base types*, like `int` and `bool`, and `float` 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`.
- "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 OCaml, we need a *variant type definition*; variant types 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.

[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.

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`. This is a complicated type that would probably be poor style, but it shows how our language of types is in fact a compact way to compose different building blocks to describe the "shape" of data.

Records: Another Approach to "Each-of" Types

in ocaml, 这里逗号要改分号

Record types are "each-of" types where each component is a *named field*. For example, we can write `type glorp = {foo : int, bar : int*bool, baz : bool*int}` to create a new type named `glorp` for 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, with the key difference that we have to give a name to the type (in this case `glorp`). It is this name that is now part of our language, so we can create a list of `glorp` values and use the type `glorp list` to describe it.

A *record expression* builds a *record value*. For example, the expression `{bar = (1+2,true && true), foo = 3+4, baz = (false,9) }` would evaluate to the record value `{bar = (3,true), foo = 7, baz = (false,9)}`, which has type `glorp` 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, but for a record expression to type-check, the field names and the types of the corresponding expressions have to "line up" with one of the preceding record-type definitions. (The rules on whether/how different record types can use the same field names can be a distraction; let's restrict ourselves not to use the same field names in different record types.)

The evaluation rules for record expressions are similar: 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. Using similar syntax to accessing pieces of an each-of type in other languages, in OCaml, we can use `e.foo` where `e` evaluates to a record with a `foo` field to extract the contents of that field. 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 `e.foo`. Evaluation evaluates `e` to a record value and then produces the contents of the `foo` field.

Comparing Tuples and Records

Records and tuples are *very* similar. They are both "each-of" constructs that allow any number of components. In terms of "what we can do," they are equally powerful. OCaml could have provided just one or the other, but because each can be easier to use and better style in different situations (and because people simply have different taste too), it provided both.

One key difference between them 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 OCaml 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]

The other key difference between records and tuples in OCaml is that record types have to be given a name (like `glorp` in our silly example) before they can be used while tuple types do not have a name, we simply write their structure out where needed, like `int * bool`. Again, each can be convenient in different situations. Names help keep different concepts in our programs clear and separate. But for "simple things" like `int * bool`, names can clutter our programs and create coordination challenges in large programs whereas everybody can "just use" `int * bool` to mean the same thing.

These two differences are *orthogonal*. It turns out a language could have by-position each-of types that need types names or have by-field-name each-of types that do not have type names. OCaml just happens not to have these two of the four possibilities.

We will not use record types much, but we are emphasizing here how much they are (but in a couple ways are not) like tuple types because we will now discuss variant types that are not at all like tuple types.

Introducing Variant Type Definitions: Our Own "One-of" Types

We now introduce *variant types*. Like record types, we will introduce a new named type. Unlike record types, we are *not* defining an each-of type. Instead, we are defining a one-of type. We will start with simple *enumerations* to get comfortable with some of the ideas and then move on to variant types that can carry data, which makes variant types much more powerful, especially when also used to define recursive (i.e., self-reference) types. We will:

- Define new variant types with variant-type definitions.
- Build values of those types with *constructor* expressions.
- Use values of those types with *match* expressions, which are elegant forms that combine all the key aspects of using a one-of type.

Variant Types for Enumerations

Here is a first variant-type definition, to create a type where there are exactly seven different values that have that type:

```
type si_unit =
  | Second
  | Meter
  | Kilogram
  | Ampere
  | Kelvin
  | Mole
  | Candela
```

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

We have defined a new type `si_unit`. The syntax where different possibilities are separated by `|` indicates we are defining a variant type. (The first `|` is optional, but that's just a syntax detail.) In addition to defining the new type name `si_unit`, this definition introduces into the environment seven *constructors* `Second`, `Meter`, and so forth. (Do not confuse this use of "constructor" with the use in languages like Java. Both construct values of a particular type, so there is a modest connection.) In OCaml, constructors have to start with a capital letter, which distinguishes them from variables because variables cannot start with a capital letter. For type-checking, each of these constructors are expressions of type `si_unit`. For evaluation, each of these constructors are values.

So we now know how to build values of type `si_unit`, though it is simple so far: we just use the constructors. So if `x` and `y` have type `int`, we could write:

```
let foo = if x > y then Ampere else Meter
let bar = [Second; Meter; Second]
```

and `foo` would have type `si_unit` and be bound to either `Ampere` or `Meter` and `bar` would be a three-element list of type `si_unit list`. Naturally, a value of type `si_unit` could be passed to or returned from functions, put in tuples, etc.

We call `si_unit` an *enumeration* or "enum" and many languages support this simpler kind of one-of type. It is really just a finite number of constants. We could use the numbers 1–7 (or 0–6) to represent the different kinds of units, but that would be worse style — it is much less clear what we mean, we cannot keep somebody from using 37 where we do not intend, and we do not get the type system to help us keep units separate from other uses of numbers.

But like any type of data, we also need a way to *use* values of an enumeration. The key to using an enumeration, or more generally any one-of type is to *figure out which variant* a value is, which then allows you to "do different things depending on which one." For enumerations, we could rely on structural equality with expressions like:

```
if foo = Ampere then "oh, it's an ampere" else "nope, not an ampere"
```

and with enough conditional logic (e.g., nested if-then-else expressions) get our work done, but OCaml has match-expressions which are much better style: easier to read, less error-prone, and designed exactly for using one-of types. Here is a function of type `si_unit -> string` where the implementation uses a match expression to convert an `si_unit` to a `string`:

```
let string_of_si_unit (u : si_unit) =
  match u with
  | Second -> "second"
  | Meter -> "meter"
  | Kilogram -> "kilogram"
  | Ampere -> "ampere"
  | Kelvin -> "kelvin"
  | Mole -> "mole"
  | Candela -> "candela"
```

Note there is nothing special about the string results we chose being "spelled" similar to the constructor names. We could have just as well implemented this function:

```
let string_of_si_unit (u : si_unit) =
  match u with
```

```
  | Second -> "one sixtieth of a minute"
  | Meter -> "a little more than half my height"
  | Kilogram -> "kinda heavy"
  | Ampere -> "ampere"
  | Kelvin -> ""
  | Mole -> "ampere" (* oh, now that's confusing *)
  | Candela -> "brightness"
```

In any case, a match expression for an enumeration is like a more compact multi-branch conditional (so like nested if-then-else) where we can have one branch for each constructor. The syntax uses the keywords `match` and `with`, where an expression, let's call it `e`, between these words is what we are *matching against*. We then have *branches* where each branch looks like `| pi -> ei` where `pi` is a *pattern* and `ei` is an expression. (The `|` for the first branch is optional.)

The evaluation rules are to evaluate `e` to a value `v`, then find the first branch with a pattern that *matches*, then evaluate that branch's expression to produce the overall result. So we will evaluate two expressions: the expression after the `match` keyword and exactly one of the branch's expressions. For enumerations, the *pattern matching* that chooses which `pi` matches is what you would expect: each `pi` is one of the variants of some variant type and the branch that matches `v` is the one chosen.

It can be initially confusing that the `pi` look like expressions since we also use constructors to build values of a variant type. But patterns are not expression. They are a syntactic subset of expressions (so far, just constructor names), but they have a different meaning – they are used only for *pattern matching against* `v`.

Also notice that by using a different constructor for each of seven branches, (a) we cover exactly all the cases and (b) the order of the branches does not matter. (Later we will see fancier pattern matching where the order can matter, but if each branch matches against a distinct subset of the possible values `v`, then the order cannot matter.)

We have not yet discussed type-checking. While match expressions are very general and powerful, for enumerations, the type-checking rules are straightforward and helpful but there are several things to check. First, the expression after `match` should have a variant type, in our example `si_unit`. Second, each pattern (the `pi`) should be one of the constructors for that variant type. Third, there should be one branch for each constructor – forgetting a constructor produces a warning you should treat as an error (the evaluation rules would raise an exception if `v` was this constructor) and repeating a constructor is an error. Fourth, all the branches' expressions should type-check *with the same type* and this is the overall type of the match expression. Just like with the two branches of an if-then-else expression, since we do not know which branch will be taken, we need them all to have the same type.

Here is another example with a different variant type and a similar "conversion" function that produces a `float` instead of a `string`:

```
type si_prefix =
  | Giga
  | Mega
  | Kilo
  | Milli
  | Micro
  | Nano

let scale p =
  match p with
  | Giga  -> 1e9
  | Mega  -> 1e6
```

```
  | Kilo  -> 1e3
  | Milli -> 1e-3
  | Micro -> 1e-6
  | Nano  -> 1e-9
```

Putting it all together:

```
(scale Kilo, string_of_si_unit Meter)
```

produces the pair `(1e3,"meter")`, which has type `float * string`.

Before moving beyond enumerations, let's notice that you have long been familiar with one of the simplest and most widely used enumerations: booleans! The booleans are nothing more than an enumeration with two constructors, `true` and `false`. Indeed, the only thing that keeps us from simply defining them via:

```
type bool = | true | false
```

is the syntactic detail that the constructors do not start with capital letters. For whatever reason, OCaml just didn't "follow its own rule" and name them `True` and `False`.

And indeed you *can* use match expressions with booleans, e.g.,
`match x > y with | true -> z | false -> w - 1`. Doing so is poor style because an if-then-else expression better conveys your meaning, but this shows that if-then-else is syntactic sugar. We can completely define the typing and evaluation rules for `if e1 then e2 else e3` by saying it is syntactic sugar for `match e1 with | true -> e2 | false -> e3`. You can check that this approach produces the exact same typing and evaluation rules for conditional expressions that we gave at the beginning of the course. This shows that a language with variant types does not need to "build in" any support for booleans as they can be defined in a library.

Variant Types Beyond Enumerations

Variant types become much more powerful when we allow some or all of the constructors to *carry data*. Then a value of a variant type is not just one of a finite number of constants, but instead can "tag" some other type of data. That other data can even include more (recursive) values of the variant type. We will proceed through three examples to show how variant types can carry data:

- A silly example to show the syntax and semantics
- A more useful example involving shapes
- An example with a recursive definition of arithmetic expressions

Here is the variant-type definition for our silly example:

```
type silly =
  | A of int * bool * (string list)
  | Foo of string
  | Pizza
```

Here, `Pizza` is a constant (a value) of type `silly` like before but the `A` and `Foo` constructors are *not* — they do not type-check "by themselves." Instead they are constructors that have to be *applied* to

data of the right type to create something of type `silly`. For example, if `s` has type `string list`, then `A(3+7, true, "hi" :: s)` has type `silly` because the `A` constructor is applied to expressions of type `int`, `bool` and `string list` as required. Applying `A` to the wrong number of arguments or to arguments of the wrong type does not type-check. The resulting value is `A(10,true,["hi";...])` where the ... is the list bound to `s`. The way to think about this value is that it has two parts, a *tag* `A` and *underlying data* of `10`, `true`, and a particular list. The tag will let us use a match expression and pattern-matching to "know if we have an `A`" and then "get back out" the underlying data. So every value of type `silly` is built from exactly one of three constructors *and* if it is built from `A` then it has an `int`, a `bool`, and a `string list` *and* if it is built from `Foo` then it has a `string`.

In general, for any variant type, any value of that type will have exactly one of the constructors as a "tag" and the "tag" determines what other data, if any, is part of the value. Enumerations are simply the case where all tags indicate there is no other data.

Now that we have seen defining the `silly` type and how to build values of the `silly` type, we can of course pass such values around like any other values. To use them, we use match-expressions like we did with enumerations, but now with a richer form of patterns in order to *extract the underlying data*. Our silly example continues:

```
let silly_over_silly s =
  match s with
  | A(x,y,z) -> List.hd z
  | Foo s -> s ^ s
  | Pizza -> "ham and pineapple"
```

This is a function of type `silly -> string`. As before, the match expression will *pattern-match against* the result of evaluating the expression after the `match`, which in this case is `s`, which will need to have type `silly` because the patterns in the branches match against `silly` values. And like before, we have one branch for each constructor and all the expressions (to the right of each `->`) must have the same type.

The key difference is that patterns for the `A` and `Foo` constructors now have one variable for each piece of underlying data carried by that constructor: 3 for `A`, 1 for `Foo` (and 0 for `Pizza`). While we will generalize this later, think of having the correct number of variables as being required by type-checking. These variables are local variables in scope only in that branch and used only if that branch is "the taken branch."

For example, if `s` evaluates to `A(10,true,["hi";...])`, then the first branch will be taken and `List.hd z` will be evaluated in an environment where `x` is bound to 10, `y` is bound to `true`, and `z` is bound to `["hi";...]`. Thus the result of the entire match expression will be `"hi"`. If instead `s` evaluates to `Foo "bye"`, then the second branch will be taken and `s ^ s` will be evaluated in an environment where `s` is bound to `"bye"` (shadowing the outer `s` that is the function parameter) and the overall result will be `"byebye"`.

The type-checking rules correspond as we should expect. For example, we type-check `List.hd z` in a static environment where `x` has type `int`, `y` has type `bool`, and `z` has type `string list`.

As this example shows, a match expression is compound operation that combines four more basic operations at once:

- Checking what tag a value has
- Choosing a branch based on the tag
- Extracting the pieces of data (if any) carried by the constructor
- Binding local variables to those pieces of data

While a different language design could have separated these pieces into smaller primitives, combining them has some advantages. As before, the type-checker can make sure we do not forget any tags. Now it can also make sure we extract all the correct pieces of data. And overall, we generally should have one branch for "every kind of data there might be" and match expressions capture that programming pattern directly and concisely. Finally, as we will see, pattern matching is actually much more powerful.

As promised, here is a less silly example of a variant type and three functions over it:

```
type shape =
  | Circle of float * float * float (* center-x, center-y, radius *)
  | Rectangle of float * float * float * float (* x1,y1,x2,y2 (opposite corners) *)
  | Triangle of float * float * float * float * float * float (* x1,y1,x2,y2,x3,y3 *)

let area s =
  match s with
  | Circle (x,y,radius) -> Float.pi *. radius *. radius
  | Rectangle (x1,y1,x2,y2) -> Float.abs ((x2 -. x1) *. (y2 -. y1))
  | Triangle (x1,y1,x2,y2,x3,y3) ->
      let a = x1 *. (y2 -. y3) in
      let b = x2 *. (y3 -. y1) in
      let c = x3 *. (y1 -. y2) in
      Float.abs ((a +. b +. c) /. 2.0)

let epsilon = 0.0001

let well_formed s =
   area s > epsilon

let num_straight_sides s =
  (* will soon learn better style than these variable names *)
  match s with
  | Circle (x,y,r) -> 0
  | Rectangle (a,b,c,d) -> 4
  | Triangle (x1,x2,x3,x4,x5,x6) -> 3
```

Here, every value of type `shape` is a circle, a rectangle, or a triangle. Each variant of shape is represented in a different way, i.e., carries underlying data that has a particular meaning.

`area` is then a function of type `shape -> float` that can compute the area of any variant of shape. The math for each variant of shape is different but the approach of "check the tag, extract the pieces, compute with those pieces" is analogous.

The function `well_formed` of type `shape -> bool` determines if a shape "makes sense." It is tempting to jump to using a match expression and checking things like whether a triangle's points are colinear or a circle has a zero radius. But sometimes we do not need that and our computation can be expressed by using other functions that take care of considering different kinds of shapes. As shown here, we simply see if a shape has a non-zero area. (We use an `epsilon` to avoid rounding error. We also implicitly allow a circle to have a negative radius. If we wanted to prevent that, we would need a match expression to extract the radius and check it.)

Finally `num_straight_sides` of type `shape -> int` does not actually use any of the underlying data. We will learn later how to write patterns that do not need to make up variable names we will not use, but for the moment they do no harm and it does *not* work to try just to use `Circle` or similar as a pattern.

For our third example, we will define a recursive data type that describes arithmetic expressions that are integer constants, negations, additions, or multiplications:

```
type expr =
  | Constant of int
  | Negate of expr
  | Add of expr * expr
  | Mul of expr * expr
```

Before "writing any code," let's understand what values of type `expr` "look like." Any value of type `expr` is built from one of the four constructors. In the `Negate` case, the underlying data is another `expr` just as in the `Add` and `Mul` cases, the underlying data is two other `expr` values. In generally any `expr` is a *tree* with constants at the leaves and internal nodes either having one child (for negations) or two children (for additions and multiplications). It is not a binary search tree or even a binary tree at all, but rather a tree that represents an arithmetic expression. It is entirely defined by us; OCaml the language does not "know" what we "mean" by a value of type `expr` – it is just a tree data structure with tags and children and leaves tagged with `Constant` that hold numbers. It is fine — and fairly common — that two constructors (`Add` and `Mul`) happen to have the same type of underlying data. We use different constructors because we want the tags to indicate they are different.

For example, consider the OCaml expression:

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

This is a small tree with "Add" at the root, a left child of the "Constant 19" and a right chid that is a negation that itself has one child.

A natural function to want to write is one that evaluates such an expression to the intended mathematical result. Such a function has type `exp -> int` and is an elegant use of match expressions and recursion:

```
let rec eval e =
  match e with
  | Constant i -> i
  | Negate e1 -> - (eval e1)
  | Add (e1, e2) -> (eval e1) + (eval e2)
  | Mul (e1, e2) -> (eval e1) * (eval e2)
```

If we call `eval (Add (Constant 19, Negate (Constant 4)))` we get the answer of 15 we expect.

But there is nothing special about the `eval` function. We can write any function we want over the trees represented by `expr` values and they will typically have the same overall recursive structure. Here are three more examples all of which do something at least plausibly useful:

```
let rec max_const (e: expr) : int =
  let max (x,y) = if x > y then x else y in
  match e with
  | Constant i -> i
  | Negate e1 -> max_const e1
  | Add (e1, e2) -> max (max_const e1, max_const e2)
  | Mul (e1, e2) -> max (max_const e1, max_const e2)

let rec has_const_not_under_add e =
```

```
    match e with
    | Constant i -> true
    | Negate e1 -> has_const_not_under_add e1
    | Add (e1,e2) -> false
    | Mul (e1, e2) -> has_const_not_under_add e1 || has_const_not_under_add e2

let rec number_of_adds e =
  match e with
  | Constant i -> 0
  | Negate e1 -> number_of_adds e1
  | Add (e1, e2) -> 1 + number_of_adds e1 + number_of_adds e2
  | Mul (e1, e2) -> number_of_adds e1 + number_of_adds e2
```

Variant Types So Far, Precisely

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

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

introduces a new type `t` and each constructor `Ci` can be used with an expression of type `ti` to produce a value of type `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 match expression:

```
match e with | p1 -> e1 | p2 -> e2 | ... | pn -> en
```

A match 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 match expression. So far, patterns have looked like `Ci(x1,...,xn)` where `Ci` is a constructor with data `t1 * ... * tn` (or just `Ci` if `Ci` carries no data). Such a pattern matches a value of the form `Ci(v1,...,vn)` and binds each `xi` to `vi` for evaluating the corresponding `ei`.

How OCaml Does Not Support One-Of Types (But Could)

In OCaml, the approach to variant types described above is truly what is "built in" to the language: Variant type definitions, constructors, and pattern matching via match expressions are the core building blocks that support one-of types defined by programmers or defined by the standard library.

There are others ways other programming languages support one-of types and, naturally, OCaml could have taken these approaches. For a given variant type, what we need is:

- A way to build each variant
- A way to check which variant a value is
- A way to branch based on which variant a value is
- A way to extract the data from a data-carrying variant.

For the last two, a common alternative to match expressions is with functions that do exactly this, defined as part of the variant type definition. For example, for our silly variant type:

```
type silly =
  | A of int * bool * (string list)
  | Foo of string
  | Pizza
```

we could imagine a language that did not have match expressions but created functions like:

- `isA`, `isFoo`, and `isPizza` of type `silly -> bool` that evaluate to `true` for the correspondingly tagged data, so for example `isFoo (A (10,true,[]))` would evaluate to `false` and `isFoo (Foo "hi")` would evaluate to `true`.
- `a_fst`, `a_snd`, and `a_thd` of types `silly -> int`, `silly -> bool`, and `silly -> string list` respectively that given a value of type `silly` build from `A` return the corresponding piece of data and *raise an exception for other variants of* `silly`.
- `foo_fst` of type `silly -> string` that given a value of type `silly` build from `Foo` returns the underlying string and *raises an exception for other variants of* `silly`.

So far, we have basically seen lists and options work this way. `e = []` is like this "is the list empty" function and `List.hd` and `List.tl` are like the `a_fst` and `a_snd` functions but for non-empty lists. But as shown below, it turns out what is actually built into OCaml is pattern matching – this is how `List.hd` and `List.tl` are implemented. (Structural equality is its own primitive, but we do not need to use it once we have pattern matching.)

Indeed, once we have pattern matching and a way to raise exceptions (a later topic but you can get the gist in the examples here), we can implement all the functions above if we want to. Here are some examples:

```
let isA s =
  match s with
  | A (x,y,z) -> true
  | Foo x -> false
  | Pizza -> false

let a_fst s =
  match s with
  | A (x,y,z) -> x
  | Foo x -> raise (Failure "a_fst")
  | Pizza -> raise (Failure "a_fst")
```

It is rare to define and use functions like this though – match expressions are powerful enough to do so, but it is usually better to use that power directly for reasons we have been emphasizing:

- 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 `List.hd x` where `x` turns out to be `[]`.
- If a match expression forgets a variant, then the type-checker will give a warning message. This indicates that evaluating the match 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 match expression uses a variant twice, then the type-checker will give a warning since one of the branches could never possibly be used.
- Pattern-matching is much more general and powerful than we have indicated so far. We give the "whole truth" about pattern-matching below when we learn nested patterns.

So, as a matter of style, do not define functions like `isA` or `a_fst`. (There are occasional exceptions to this guideline once we have first-class functions in the next unit, but they are still rare.)

Why You May Not Have Seen Variant Types Before

You likely have never seen anything like variant-type definitions before, so it is natural to be skeptical that they are so important. If you have programmed productively in other languages without them, then can they really be so essential? We argue that whether or not you have OCaml-style variant types, some support for one-of types *is* essential, but it is rarely focused on as what it is — a way to support "I have this *or* I have that" whereas each-of types do tend to get focused on when teaching programming.

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. This variant type captures the idea directly:

```
type id =
  | StudentNum of int
  | FullName of 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 name field, otherwise the name
 field should be the empty string *)
 { student_num : int; name : 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, we use option which is a one-of type for this) *and* their full name:

```
{ student_num : int option;  name : string }
```

By composing each-of types and one-of types in *just the right way*, we can often make sure our data structures encode *by construction* the sensible values rather than relying on comments and programming conventions. Neither each-of types nor one-of types can do this alone; their power is in how we combine them.

As we see much later in the course, in object-oriented programming, one-of types are achieved via subclassing. In dynamically typed languages, one-of types are often achieved without the user-defined tags of a variant type but instead just using the implicit tags built into the language, such as being able to at run-time pass any value anywhere and then ask what kind of data it is.

Type Synonyms

Before continuing our discussion of variant types, 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 OCaml's module system we will build on this feature).

But for more complicated types, it can be convenient to create type synonyms. For example,

```
type point_2d = float * float
type point_3d = float * float * float
```

Just remember these synonyms are fully interchangeable. For example, if you need to write a function of type `point_2d -> float * float * float` and the REPL reports your solution has type `float * float -> point_3d`, this is okay because the types are "the same."

In contrast, variant type definitions and record type definitions introduce a type that is not the same as any existing type. A variant type creates constructors that produces values of this new type. So, for example, the only type that is the same as `si_unit` is `si_unit` unless we later introduce a synonym for it.

Lists and Options are Variant Types (We Just Didn't Tell You Yet)

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

```
type 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 match expressions to use such values:[3]

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

let rec append_mylist (xs,ys) =
  match xs with
  | 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 variant types. As a matter of style, it is better to use the built-in widely-known feature than to invent your own.

More importantly, it is better style to use pattern-matching for accessing list and option values, ***not*** the functions `List.hd`, `List.tl`, and `Option.get` we saw previously. Similarly, we should generaly *not* use the

[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."

`e = []` or `e = None` tests. 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:

```
let inc_or_zero intoption =
  match intoption with
  | 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:

```
let rec sum_list xs =
  match xs with
  | [] -> 0
  | x::xs' -> x + sum_list xs'

let rec append (xs,ys) =
  match xs with
  | [] -> 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.

The reasons why you should usually prefer pattern-matching for accessing lists and options instead of `e = []`, `List.hd`, and `List.tl` is the same as for variant types in general: you cannot forget cases, you cannot apply the wrong function, etc. So why does the OCaml 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 unit of the course.

Polymorphic Variant Types

Other than the strange syntax of `[]` and `::`, the only thing that distinguishes the built-in lists and options from our example variant type definitions 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 variant types too, and indeed it is very useful for building "generic" data structures. While we will not focus on using this feature, it is just a small extension of what we have already done. For example, this is *exactly* how options are predefined in the environment:

```
type '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 a type. You can also define polymorphic variant types 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`:

```
type ('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 polymorphic variant types as for other variant types.

Pattern-Matching for Each-Of Types: The Truth About Let-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. For tuples, the pattern `(x1,...,xn)` matches the tuple value `(v1,...,vn)`, and binds each variable to the corresponding tuple component where naturally the order does matter. So we could write this function for summing the three parts of an `int * int * int`:

```
let sum_triple (triple : int * int * int) =
  match triple with
  | (x,y,z) -> z + y + x
```

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

```
type three_strings = {first:string,middle:string,last:string}
let full_name (r : three_strings) =
  match r with
  | {first=x,middle=y,last=z} -> x ^ " " ^ y ^ " " ^z
```

However, a match-expression with one branch is poor style — it looks strange because the purpose of such expressions is to distinguish *variants*, plural. So we showed the examples above for sake of explaining what follows, but do *not* write one-branch match expressions.

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 let-bindings too! So this approach is better style:

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

Actually we can do even better: Just like a pattern can be used in a let-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:

```
let full_name {first=x,middle=y,last=z} =
    x ^ " " ^ y ^ " " ^z
let 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 OCaml: ***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 OCaml 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 match 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":

```
let rotate_left (x,y,z) = (y,z,x)
let 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 `let 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 variant type with only one constructor, which takes no arguments and uses the unusual syntax `()`. Basically, `type unit = | ()` comes predefined.

Digression: Type inference and Polymorphic Types

For familiarity and to make error messages more readable, we began writing explicit types down for function arguments. Some OCaml programmers choose to do so and some even write down return types for functions as well. We can choose to write types down for any variables we introduce, even local variables. But in OCaml we never need to: The type-checker is able to *infer* the types of everything for us just based on how variables are used. We will study the basics of how type inference works later.

Especially with pattern matching of function arguments, type inference sometimes reveals that functions are more general than you might have thought. Consider this code, which does use part of a tuple:

```
let partial_sum (x,y,z) = x + z
```

The inferred function type 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* later because they are a major course topic in their own right. For now, just feel free to stop writing explicit types and do not be confused if you see the occasional type like `'a` or `'b` due to type inference.

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`.

Again, we will discuss polymorphic types and type inference more later, but this digression is helpful for avoiding confusion in the meantime: 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. (The different subpatterns must use different variables, however.)
- (A similar case for record patterns of the form `{f1=p1,...,fn=pn}` ...)

This recursive definition extends our previous understanding in a way that lets us write more concise and elegant programs. We can use nested patterns instead of nested match 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:

```
let rec len xs =
  match xs with
  | [] -> 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:

```
let rec len xs =
  match xs with
  | [] -> 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 match-expressions is "zipping" or "unzipping" lists (three of them in this example):[4]

```
exception BadTriple

let rec zip3 list_triple =
  match list_triple with
  | ([],[],[]) -> []
  | (hd1::tl1,hd2::tl2,hd3::tl3) -> (hd1,hd2,hd3) :: zip3 (tl1,tl2,tl3)
  | _ -> raise BadTriple

let rec unzip3 lst =
  match lst with
  | [] -> ([],[],[])
  | (a,b,c)::tl -> let (l1,l2,l3) = unzip3 tl in
                   (a::l1,b::l2,c::l3)
```

This example checks that a list of integers is sorted:

```
let rec nondecreasing intlist =
  match intlist with
  | [] -> true
  | _::[] -> true
  | head::(neck::rest) -> (head <= neck && nondecreasing (neck::rest))
```

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

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:

```
type sign = P | N | Z

let multsign (x1,x2) =
  let sign x = if x=0 then Z else if x>0 then P else N in
  match (sign x1,sign x2) with
  | (Z,_) -> Z
  | (_,Z) -> Z
  | (P,P) -> P
  | (N,N) -> P
  | _     -> N (* many say bad style; probably okay here *)
```

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 match expressions. We also have seen the good style of pattern-matching each-of types in let bindings or function arguments and that this is what a "multi-argument function" really is. But is there a way to match against one-of types in variable and function bindings? This seems like a bad idea since we need multiple possibilities. But it turns out OCaml has special syntax for doing this in function definitions with the `function` keyword. Here are two examples, one for our own variant type and one for lists:

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

let rec eval = function
  | Constant i -> i
  | Negate e2 -> - (eval e2)
  | Add(e1,e2) -> (eval e1) + (eval e2)
  | Multiply(e1,e2) -> (eval e1) * (eval e2)

let rec append = function
  | ([],ys) -> ys
  | (x::xs',ys) -> x :: append(xs',ys)
```

As a matter of *taste*, your instructor has never used this style, but it is common among some functional programmers in OCaml and other languages that have support for multiple patterns in function definitions, so you are welcome to use it if you want. As a matter of *semantics*, it is just syntactic sugar for a single function body that is a match expression:

```
let rec eval e =
```

```
  match e with
  | Constant i -> i
  | Negate e2  -> - (eval e2)
  | Add(e1,e2) -> (eval e1) + (eval e2)
  | Multiply(e1,e2) -> (eval e1) * (eval e2)

let rec append e =
  match e with
  | ([],ys) -> ys
  | (x::xs',ys) -> x :: append(xs',ys)
```

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.

As much more questionable or risky style, you can also put patterns that might not match into places where they must match. The type-checker will give you a warning and if the match fails at run-time, an exception is raised. So consider this bad style and use it only where you are *sure* it will not fail (or you want an exception if it does). For example, these functions take an `int option` and returns one more than the number under a `Some`, raising an exception if passed `None`.

```
let risky1 opt =
  let Some x = opt in
  x + 1

let risky2 (Some x) =
  x + 1
```

Exceptions

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

```
let hd xs =
  match xs with
  | [] -> raise (Failure "hd")
  | x::_ -> x
```

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 variant type 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 variant type. So `Failure "hd"`, `MyUndesirableCondition`, and `MyOtherException(3,9)` are all values of type `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, 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.)

```
let rec maxlist (xs,ex) =
   match xs with
   | [] -> raise ex
   | x::[] -> x
   | x::xs' -> Int.max(x,maxlist(xs',ex))
```

Notice that calling `maxlist([3,4,0],Failure "empty list")` 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, Ocaml has try-expressions, which look like `try e1 with 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 try-expression also raises that exception. Similarly, if `e2` raises an exception, then the whole expression also raises an exception.

As with match-expressions, try-expressions 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 OCaml, 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:

```
let rec sum1 xs =
  match xs with
  | [] -> 0
  | i::xs' -> i + sum1 xs'


let sum2 xs =
  let rec f (xs,acc) =
    match xs with
    | [] -> acc
    | i::xs' -> f (xs',i+acc)
  in
  f (xs,0)
```

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 (often 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 OCaml 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. Other times, if we need a particular oder of operations, then using an accumulator may not work.

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:

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

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

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, but not all.

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

```
let rec rev1 xs =
  match xs with
  | [] -> []
  | 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.

```
let rev2 xs =
  let rec aux(xs,acc) =
    match xs with
    | [] -> acc
    | x::xs -> aux (xs, x::acc)
  in
  aux (xs,[])
```

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. Such a precise definition is also important for language implementations since they need to perform tail-call optimization. The definition has one part for each kind of expression; here are several parts:

- If an expression is not in tail position, then none of its subexpressions are in tail position.
- In `let f x = e`, `e` is 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`). (Match-expressions are similar.)
- If `let x = e1 in e2` is in tail position, then `e2` is in tail position (but not `e1`).
- Function-call arguments are not in tail position. (We still have to call the function after evaluating the arguments.)
- For `(e1,e2,...en)`, none of the subexpressions are in tail position. (We still have to create and return the tuple after evaluating the subexpressions).
- ...