Module Systems and ML

Why module systems?

A module system is a mechanism for breaking a program into parts. There are roughly two reasons for module systems to exist:

1. **Separate reasoning:** the ability to break down a system into parts, so that
   - each part's internals can be understood and modified (relatively) independently; and
   - the parts can then be recombined by inclusion into a program.

   One hopes that understanding the recomposition is simpler than understanding all the parts' internals in combination at once.

   The complexity in a system with *N* parts is roughly $O(N^2)$ --- because there are N^2 ways to connect *N* parts. Since modules are smaller than the whole program, and the number of modules is (hopefully) smaller than the number of finer-grained constructs (functions, data types, etc.), you can reduce the number of possible interconnections by using a module system judiciously.

2. **Separate compilation:** the ability to break a system into parts, so that
   - each part can be compiled independently; and
   - the parts can then be recombined by the linker

   This process, adding up the costs of all individual module compilations and the link step, is typically *more* expensive than simply compiling the entire program in one go. However, separate compilation has the advantage that the entire program need not be recompiled when only one module changes. In practical large-scale development, separate compilation is extremely important.

   Separate compilation is a complex topic that is beyond the scope of a first course in programming languages. The one thing we will note here is that a necessary (but not sufficient) condition for separate compilation is **separate typechecking**: it must be possible to typecheck a module in isolation (delare it type-correct), without forcing the compiler to access the details of how other modules are implemented.

Key to Separate Reasoning: Encapsulation and interfaces

Encapsulation (a.k.a. **information hiding**) = A module only reveals what its clients need to know, and hides (or encapsulates, or abstracts) the rest.

- Use of the module (i.e., recomposition of the decomposed program) will be easier --- clients need not think about what they don't need to know.
- A module can change the hidden portions without affecting clients.

The notion of *hiding* some parts of a module implies that some parts are exposed. The exposed part is usually called the **interface** of the module. An interface imposes constraints on how the client can use the module spoken of; how these constraints work is perhaps the key problem in module system design.

Note: some people seem to think encapsulation and information hiding are different. I have never seen anyone define a technical distinction between the two.

A bit of terminology

Language designers usually distinguish between the **core language** and the **module language**. The core language is what we've learned in ML prior to this lecture --- mechanisms for constructing and manipulating computations over ordinary values. The module language consists of mechanisms for arranging and combining core language constructs on a larger scale.

To some extent, features of module systems can be applied freely and independently to various core languages. For example, the notion of bundling core language declarations inside a module for namespace management is applicable in nearly every language. However, more complex module mechanisms typically depend on properties of the core language.

Why a separate module system?

Note our definition of module system: a mechanism for breaking a program into parts. This is pretty vague. Let's suppose that we want a mechanism for

- Putting a bunch of values into a bundle, and
- Hiding some of those things outside the bundle.

Well, we already have a mechanism in the core language for putting a bunch of values into a bundle: records. Let's use a record to bundle up a bunch of functions for manipulating lists (ignore for the moment the fact that ML's syntax doesn't allow in-line definitions of recursive functions inside records; this is not a fundamental barrier, since we could just make up some syntax):

```
val List = {
    map     = fn ...,
    reverse = fn ...,
    ...
};
```

If this value were provided by default in the top-level environment, we could then refer to the map function as `#map(List)`. Syntactically ugly, perhaps, but otherwise fine: it is "the `map` function in the `List` module".

This has at least two major drawbacks:

1. **Record types cannot hide fields.** Recall that a record value's type must include the name and types of all its fields. So, for example, if the programmer of the `List` record wanted to define "private" function in the course of implementing several of these records, then the programmer would not be able to leave that function out of the record type.
2. **Record members can only be values, not types.** You could not, for example, bundle up the list data type in the same record with its functions.

Both of these problems could be solved by extending the capabilities of the record data type, but they would lead to such fundamental changes, and such an increase in complexity, that you would lose the pleasing simplicity of the ML record type.

Similar drawbacks apply to using any other core language construct for modules.

ML modules

ML modules are sequences of declarations inside a `struct` expression; `struct` expressions must be bound to a name using a `structure` declaration:

```
- structure S = struct
    val x = 3
    type t = int * string
    datatype d = D of t
    fun f y = (y, Int.toString x)
end;
structure S :
  sig
    datatype d = D of t
    type t = int * string
    val f : 'a -> 'a * string
    val x : int
  end
```

The declared types, as well as the types of the structure's values, are reflected in the **signature**, which the SML/NJ interpreter echoed back between `sig ... end`. Note that values are erased in the signature; only types remain. A signature is "the type of a structure".

To refer to a structure member, use the syntax *`structExpr.name`*, e.g.:

```
- S.x;
val it = 3 : int
- S.f;
val it = fn : 'a -> 'a * string
- val (a:S.t) = (3, "hi");
val a = (3,"hi") : S.t
- S.D a;
val it = D (3,"hi") : S.d
```

This simplest use of modules allows **namespace management**: in other words, programmers can group related names together inside a module. This is a very weak (although indispensible) form of encapsulation --- it only prevents accidental name clashes (a name is only revealed to clients who ask for it by accessing it through the structure path).

Structures can be nested, with the obvious semantics:

```
- structure Outer = struct
  val x = 5
  structure Inner = struct
    val x = "hi"
  end
end;
structure Outer :
  sig
    structure Inner :
    val x : int
  end

- val outerX = Outer.x;
val outerX = 5 : int
- val innerX = Outer.Inner.x;
val innerX = "hi" : string
```

Signature ascription for abstraction

When a structure is declared, ML automatically generates a **principal signature** for the structure --- the one that reflects all its bindings and types. However, the programmer may wish to hide parts of this signature, in order to abstract away (encapsulate) some details.

This is done using **signature ascription**, which is rather analogous to type ascription, except that there are two kinds of signature ascription: transparent and opaque. Both forms of ascription check that the principal signature matches the ascribed signature. The difference is in what each reveals about types:

- **Transparent** ascription: this allows the definitions of types in the principal signature to "show through", so long as that type is *declared* (not necessarily defined) in the transc.
- **Opaque** ascription: this gives the structure the ascribed signature --- and no more. Therefore, some information in the structure's types may be hidden.

Transparent signature ascription for modules is written with a `:`, like type ascription for values:

```
- structure STransparent:sig
    type t;
    type d;
    val f:int -> t
end = S;
structure STransparent :
  sig
    type t = int * string
    datatype d = ...
    val f : int -> t
  end
```

Note that the signature allows the definitions of `t` and `d` to "show through" the ascription.

Opaque ascription is written with a `:>`

```
- structure SOpaque :> sig
    type t;
    type d;
    val f:int -> t
end = S;
structure SOpaque :
  sig
    type t
    type d
    val f : int -> t
  end
```

Note that `t` and `d` are now fully abstract --- nothing is known about them.

Both transparent and opaque ascription hide any types or values that are not declared in the signature at all. In the example above, both ascribed structures omit the `x` value.

Signatures can be bound to names using the special `signature` declaration:

```
- signature Sig = sig
    type t
    type d
    val f:int -> t
end;
signature Sig =
  sig
    type t
    type d
    val f : int -> t
  end
```

Opening modules and `local`

Sometimes it is inconvenient to refer to module members by the full structure path. To save typing and visual clutter, ML provides an `open` declaration, which dumps a module's members into the scope in which the `open` declaration appears:

```
- open Outer.Inner;
opening Outer.Inner
  val x : string

- structure OpenDemo = struct
    open Outer;
end;
structure OpenDemo :
  sig
    structure Inner :
    val x : int
  end
```

Generally, it is a bad idea to open a module, because it's difficult to keep track of what names will be dumped into the current environment.

If you must open a module, it is best to limit the extent of the resulting names. ML has a `local` construct for this purpose. `local` declarations have the syntax

```
local openDecls in bodyDecls end
```

The `open` statements in *`openDecls`* make the names in the opened structures visible for the declarations in *`bodyDecls`*. Then, *`bodyDecls`* are added to the scope containing the `local` declaration.

```
- structure ToOpen = struct val a = "foo" end;
structure ToOpen : sig val a : string end

- local
    open ToOpen
  in
    val b = a
  end;
val b = "foo" : string

- b;
val it = "foo" : string
```

`local` declarations are rather like a "declaration-level `let`" (Notice that let-expressions have no mechanism for adding declarations to the *surrounding* environment.)

Functors: Parameterized modules (or, functions over modules)

At the expression level, functions abstract over expressions. Likewise, at the module level, we would like some mechanism to abstract over modules --- if we have a common "module pattern" for more than one combinations of types and values, then we would like to write that pattern once, and apply it to several modules.

In ML, a **functor** is a "function over structures". Functor declarations have the form

```
functor name(params):optionalSig = structBody
```

name is the name of the functor, *params* are the the formal functor parameters, which will typically be structures; structure parameters must have ascribed signatures, because parameters' signatures are not inferred at the module level. As with structures, the user may optionally provide a signature for the functor, *optionalSig*. Finally, *structBody* is the body of the functor.

In core language functions, the body is an expression, and the parameter names are available as expressions in the body. In module language, the body is a structure; and the *functor* parameter names are available in the *functor signature and body* as as modules.

Ullman's textbook defines the `BTreeFun` functor shown in abbreviated form below along with the helper signature `TOTALORDER`:

```
signature TOTALORDER = sig
  type element
  val lt : element * element -> bool
end;

functor MakeBST(Lt:TOTALORDER):
  sig
    type 'a btree
    val create : Lt.element btree;
    val lookup : Lt.element * Lt.element btree -> bool
    val insert : Lt.element * Lt.element btree
                   -> Lt.element btree
  end
  =
  struct
    open Lt

    datatype 'a btree =
             Empty
           | Node of 'a * 'a btree * 'a btree;

    val create = Empty

    fun lookup(x, Empty) = false
      | lookup(x, Node(v, left, right)) =
        if Lt.lt(x, v)      then lookup(x, left)
        else if Lt.lt(v, x) then lookup(x, right)
        else true

    fun insert(x, Empty) = Node(x, Empty, Empty)
      | insert(x, n as Node(v, left, right)) =
        if Lt.lt(x, v)    then Node(v, insert(x, left), right)
        else if Lt.lt(v, x) then Node(v, right, insert(x, right))
        else n
  end;
```

Leaving aside signatures for the moment, this works much like you'd expect a function over structures to work:

- The parameter `Lt` is a structure with signature `TOTALORDER`.
- `TOTALORDER` contains a type `element` and a function value `lt`.
- In the functor signature, we refer to the `Lt.element` type in the profile of various functions that will use `Lt.lt`.
- In the functor body, we refer to the `Lt.lt` function value in the bodies of `lookup` and `insert`.

This is analogous to having a function at the expression level which takes a record, and then accesses record members inside the function body. Since structures (unlike records) can contain types, it is also useful for the parameters to be accessible

To instantiate this functor into a structure, you need a structure satisfying signature `TOTALORDER`:

```
- structure StringLt : TOTALORDER =
struct
  type element = string
  val lt:(string * string -> bool) = op <;
end;
structure StringLt : TOTALORDER

- structure StringBST = MakeBST(StringLt);
structure StringBST :
  sig
    datatype 'a btree = ...
    val create : Lt.element btree
    val lookup : Lt.element * Lt.element btree -> bool
    val insert : Lt.element * Lt.element btree -> Lt.element btree
  end
```

Functors vs. records of functions

Previously, we achieved results very similar to the above by defining a datatype that held a record of functions:

```
datatype 'a BTNode = Empty | Node of 'a * 'a BTNode * 'a BTNode;
datatype 'a BTree = Tree of {lt:'a * 'a -> bool, root:'a BTNode}
```

This was simpler than the functor-based approach above. Furthermore, in both approaches, the programmer has the flexibility to write search trees with any data type, and even provide different function(s) that compare values. Why would we ever want to use functors?

This question becomes even more pressing when we consider the complexity of functor semantics. We have only scratched the surface in the example above --- remembering the full semantics of functors exactly is challenging even for Ph.D.'s in programming languages. (In fact, functors are a rather controversial topic in the programming languages community.)

Nevertheless, there are at least two answers to why we'd want to use functors:

- It can get cumbersome to organize large chunks of code using only records of functions (imagine some more complex set of interrelated datatypes and functions).
- More importantly: functor applications are generative --- each application generates a fresh structure, including fresh `datatype`s (if the functor defines any). Therefore, for example, it is not possible to mix up `btree` instances from two different applications of `MakeBST`.

  Consider a `mergeTrees` function in the datatype-based approach; it would have type

  ```
  ('a BTree * 'a BTree) -> 'a BTree
  ```

  Suppose the user constructed two `BTree` instances with the same element type, but different `lt` functions:

```
- type Point = {x:int, y:int};
type Point = {x:int, y:int}
- fun lt_X ({x=x1, y=_}:Point, {x=x2, y=_}:Point) = x1 < x2-
val lt_X = fn : Point * Point -> bool
- fun lt_Y ({x=_, y=y1}:Point, {x=_, y=y2}:Point) = y1 < y2
val lt_Y = fn : Point * Point -> bool

- val xTree = Tree {lt=lt_X, root=Empty}
val xTree = Tree {lt=fn,root=Empty} : Point BTree
- val yTree = Tree {lt=lt_Y, root=Empty}
val yTree = Tree {lt=fn,root=Empty} : Point BTree
```

The trees `xTree` and `yTree` have the same type, because the instantiation of the type variable does not track the identity of the `lt` member. We might accidentally apply `mergeTrees` to two trees with different `lt` functions.

On the other hand, the functor-based approach generates a fresh structure for each kind of `BTree` type.

Suggested exercise: write the `Point` example above in functor-based style. Then, write an expression or function that does not typecheck, and whose failure to typecheck proves that the two kinds of trees are not compatible.

Modules are not first class

Module values and module expressions are *not* first-class. In order to bind a module to a name, you must use the special `structure` binding rather than a `val` binding. You cannot have a "conditional module" expression:

```
structure S = if someComplicatedExpression
              then struct type T = ... end
              else struct type T = ... end;
```

This is not an accident. There are proposals to make modules first-class, but there are subtle issues with typechecking (which are beyond the scope of this course) and there is some disagreement within the languages community as to whether it's even desirable to make modules first-class.