Encapsulation and Information Hiding

• To prevent data from being used in unexpected ways that result in hard-to-discover bugs
• To encourage reuse of code by allowing the definitions supporting a common idea to be packaged with a simple and precisely defined interface

ML module system - Structures, signatures and functors

Structures

Structures are collections of types, datatypes, functions, exceptions, constants, and other elements that want to be encapsulated.

Form:

```
structure <identifier> =
   struct <elements of the structure> end
```

Want to reason about binary search trees, making sure that we can create trees, look up values, insert, delete.

BST implementation

```
fun lower(nil) = nil
|   lower(x::cs) = (Char.toLower c)::lower(cs);

fun lt(x:string, y) = implode(lower(explode(x))) <
       implode(lower(explode(y)));

structure StringBST = struct
  datatype 'label btree = Empty |
     Node of 'label * 'label btree * 'label btree;
  fun lookup(x, Empty) = false
  |   lookup(x, Node(y, left, right)) =
          if lt(x,y) then lookup(x,left)
          else if lt(y,x) then lookup(x,right)
          else (* x = y *) true;

  fun insert(x,Empty)  = Node(x,Empty,Empty)
  |   insert(x, T as Node(y,left,right)) =
          if lt(x,y) then Node(y,insert(x,left),right)
          else if lt(y,x) then Node(y,left,insert(x,right))
          else (* x = y *) T;
          (* do nothing; x was already there *)
  exception EmptyTree;

(* deletemin(T) returns a pair consisting of the least
  element y in tree T and the tree that results if we
  delete y from Y.  Error if T is empty *)
  fun deletemin(Empty) = raise EmptyTree
      (* the critical case.  If the left subtree is empty,
         then the element at current node is min. *)
  |   deletemin(Node(y,Empty,right)) = (y,right)
```


BST implementation (Cont'd)

```
  |   deletemin(Node(w,left,right)) =
        let val (y,L) = deletemin(left)
        in  (y, Node(w,L,right)) end;

  fun delete(x, Empty) = Empty
  |   delete(x, Node(y,left,right)) =
          if lt(x,y) then Node(y,delete(x,left),right)
          else if lt(y,x) then Node(y,left,delete(x,right))
          else (* x = y *)
             case (left,right) of
                 (Empty,r) => r |
                 (l,Empty) => l |
                 (l,r) => let val (z,r1) = deletemin(r)
                          in Node(z,l,r1) end;
end; (* structure *)
```

ML responds with

```
  structure StringBST :
    sig
     datatype 'a btree
       con Empty : 'a btree
       con Node : 'a * 'a btree * 'a btree -> 'a btree
      exception EmptyTree
      val delete : string * string btree -> string btree
      val deletemin : 'a btree -> 'a * 'a btree
      val insert : string * string btree -> string btree
      val lookup : string * string btree -> bool
    end
```

Using Elements of Structures

- By <struct name>.<func name>, e.g.

```
StringBST.lookup();
val lookup = StringBST.lookup;
```

- Or by opening a structure:

```
- open StringBST;
- val t1 = Empty;
val t1 = Empty : 'a btree
- val t2 = insert("foo", t1);
val t2 = Node ("foo",Empty,Empty) : string btree
- val t3 = insert("bar", t2);
val t3 = Node ("foo",Node ("bar",Empty,Empty),
         Empty) : string btree
- lookup("bar",t3);
val it = true : bool
- val t4 = delete("baz",t3);
val t4 = Node ("foo",Node ("bar",Empty,Empty),
         Empty) : string btree
```


Signatures

Signatures are collections of information describing the types and other specifications for some of the elements of a structure

- A signature is a "type" for a structure
- (or) The signature is a declaration, while the structure is a definition.
- Format: `sig` <specifications> `end`
- Structure elements:

`type` followed by an identifier, possibly parameterized by type variables

Ex: `type foo` and `type ('a, 'b) bar`

`eqtype` followed by a (parameterized) identifier

`exception` followed by exception name

`val` followed by an identifier, a colon and a type expression. Type expressions can include functions.

Signatures, Cont'd

- Can give signature a name:

`signature` <identifier> =

`sig` <specifications> `end`
- A signature can specify a more restrictive type for an identifier than is implicit in the structure definition.
- A signature can omit certain identifiers, thus hiding them in any structure defined to have this signature.

Example:
- don't want to be able to access `deletemin` directly in BSTs
- might want to prevent some user from being able to modify trees, allowing just lookup, etc.

Another BST

```
- signature BST_SIG = sig
    type 'a btree;
    exception EmptyTree;
    val delete : string * string btree -> string btree;
    val insert : string * string btree -> string btree;
    val lookup : string * string btree -> bool;
  end;

- structure NewBST: BST_SIG = StringBST;
structure NewBST : BST_SIG
- open NewBST;
open NewBST
exception EmptyTree = EmptyTree
val delete = fn : string * string btree -> string btree
val insert = fn : string * string btree -> string btree
val lookup = fn : string * string btree -> bool
- val t0 = Empty;
std_in:0.0-0.0 Error: unbound variable or
            constructor: Empty
```

Functors

Functors are operations that take as arguments one or more elements such as structures and produce a structure that combines the functor's arguments in some way.

- Idea - want to create a large number of similar structures, e.g. different BST's parameterized by `lt` function.
- With signatures and structures - user could redefine `lt` and get different behavior of BSTs. We want to be able to always use the original definition!
- After proper incorporation, can apply BSTs to any type, after we write appropriate `lt` function.
- Form:

  `functor` <identifier>
  (<structure name>: <signature>)
  = < structure definition>

Steps for Using Functions

1. Define a signature (`TOTALORDER`) that is satisfied only by structures that are suitable as functor inputs.
2. Define a functor (`MakeBST`) that takes a structure (S) with the appropriate signature (`TOTALORDER`) defined in (1) and produces a structure as a result.
3. Define a structure (`String`) that describes an appropriate type. (The signature should be `TOTALORDER` and should include a function `lt`.)
4. Apply the functor defined in (2) (`MakeBST`) to structure defined in (3) (`String`), to produce the desired structure (`StringBST`).

BST Example

```
signature TOTALORDER = sig
  type element;
  val lt : element * element -> bool
end;
functor MakeBST(Lt: TOTALORDER):
  sig
     type 'label btree;
     exception EmptyTree;
     val create : Lt.element btree;
     val lookup : Lt.element * Lt.element btree -> bool;
     val insert : Lt.element * Lt.element btree 0>
            Lt.element btree;
     val deletemin : Lt.element btree 0>
            Lt.element * Lt.element btree;
     val delete : Lt.element * Lt.element btree ->
            Lt.element btree
  end
=
  struct
    open Lt;
    datatype 'label btree = Empty |
        Node of 'label * 'label btree * 'label btree;
    val create = Empty;
    fun lookup(x,Empty) = ...;
    fun insert(x,Empty) = ...;
    exception EmptyTree;
    fun deletemin(Empty) = ...;
    fun delete(x,Empty) = ...;
  end;
```


BST Example (Cont'd)

Defining a Functor Argument:

```
structure String: TOTALORDER =
  struct
    type element = string;
    fun lt(x,y) = let
      fun lower(nil) = nil
      |   lower(c::cs) =
             (Char.toLower c)::lower(cs);
    in implode(lower(explode(x))) <
         implode(lower(explode(y))) end;
end;
```

Applying the functor:

- General form:

  structure <new structure name> =
  <functor name>(<structure argument>)

```
structure StringBST = MakeBST(String);
```

Alternative way

• Can provide argument structure explicitly, instead of by name (i.e., skip definition of `String`):

```
structure StringBST = MakeBST(
  struct
    type element = string;
    fun lt(x,y) = let
        fun lower(nil) = nil
        |   lower(c::cs) =
              (Char.toLower c)::lower(cs);
      in implode(lower(explode(x))) <
           implode(lower(explode(y))) end;
  end;
);
```


Information Hiding

Information Hiding - making all, or certain parts of, a package definition inaccessible to the user, to ensure that the user does not use them incorrectly.
ML has:
• Ability to define signatures that fail to mention some hidden elements (saw before)
• "Opaque" structures that hide certain structure elements
• "Abstract types" that make data constructors invisible
• "Local" definitions within an abstract type or structure to hide certain elements

Want the user of BSTs to not use exception, function `deletemin`, data constructors. Want her to think it is just a dictionary.

Another Signature Example on BSTs

Definition:

```
signature NEW_STRING_BST = sig
  type 'label btree;
  val create : string btree;
  val insert : string * string btree -> string btree;
  val delete : string * string btree -> string btree;
  val lookup : string * string btree -> book
end;

structure NewStringBST : NEW_STRING_BST
                         = MakeBST(String);
```

Usage:

```
- open NewStringBST;
- insert("foo", create);
  val it = Node("foo",Empty,Empty) : string btree
- deletemin(create);
  Error: unbound variable or constructor: deletemin
- val t = Empty;
  Error: unbound variable or constructor: Empty
- val t = Node("foo", Empty, Empty);
  Error: unbound variable or constructor: Node;
- type foo = int btree;
  type foo = int btree
```

However, cannot use `foo` since type constructors are not available.

Abstract datatypes

An *abstract type* is a datatype that hides its data constructors. Form:

`abstype` <datatype definition>
  `with`
    <declarations using the constructors>
  `end`

Example:

```
fun lt(x,y) = ...
val it = fn :  string * string -> bool
abstype sbtree = Empty |
        Node of string * sbtree * sbtree
  with
    val create = Empty;
    fun lookup(x,T) = ...;
    fun insert(x,T) = ...;
    exception EmptyTree;
    fun deletemin(T) = ...;
    fun delete(x,Y) = ...;
   end;
```

Example (Cont'd)

```
type sbtree
val create = _: sbtree
val lookup = fn : string * sbtree -> bool
val insert = fn : string * sbtree -> sbtree
exception EmptyTree
val deletemin = fn : sbtree -> string * sbtree
val delete = fn : string * sbtree -> sbtree
```

Using the abstract datatypes:

```
- insert("foo", create);
val it = - :  string btree
- Node("foo", Empty, Empty);
Error:  unbound variable or constructor:  Node
```


Local Definitions

Can hide some elements in an abstract type or structure definition by using a construct

`local` <definitions> `in` <definitions> `end`

Example:

```
- fun lt(x,y) = ...;
  val it = fn : string * string -> bool;
- abstype sbtree = Empty |
        Node of string * sbtree * sbtree
    with
        val create = Empty;
        fun lookup(x,T) = ...;
        fun insert(x,T) = ...;
        local
          exception EmptyTree;
          fun deletemin(T) = ...;
        in
          fun delete(x,T) = ...;
        end
    end;
type sbtree
val create - -: sbtree
val lookup - fn : string * sbtree -> bool
val insert = fn: string * sbtree -> sbtree
val delete - fn: string * sbtree -> sbtree
```


Summary of Encapsulation Features

- Structures allow grouping related information together
- Signatures allow making some elements inaccessible outside the structures or making the types less general
- Local definitions can also make some elements inaccessible outside the structures
- Abstract datatypes make data constructors inaccessible outside the structures
- Functors allow parameterizing of structures, and signatures again can reduce visibility of certain functor elements

Example - code for assignment 2b.

Functional Programming in ML Review

- Primitive types, records, lists, tuples
- Programming in ML:
  - Recursive functions
  - Pattern matching
- Type definition
  - Recursive datatypes
- String manipulation
- Type checking:
  - Type variables
  - Parametric polymorphism
  - Overloading
  - Eqtypes
- Higher-order programming
- Exception handling
- ML Module system

ML - Review

Covered chapter 8 and 9.1-9.5 of Sethi

- ML is a strongly typed language which does static typechecking.
- An expression is not correct if ML cannot figure out its type.
- Differentiates between records (heterogeneous) and lists (homogeneous).
- Allows parametric polymorphism - ability to define functions to operate on data of any type and still be able to type check them.

ML - Review (cont'd)

- Uses structural equivalence.
- Has type variables.
- Overloading, or ad-hoc polymorphism, conflicts with parametric.
- Allows definition of new types, as enumerated constants or as recursive types.
- Has pattern matching in functions.
- Uses pattern matching to process user-defined types.
- Has good encapsulation facilities

ML vs Scheme - Similarities

- Functional languages
- Interpreted, although compilers are also available
- Can evaluate expressions
- Can provide bindings
- Can define named and unnamed functions.
- Can operate on lists
- Does implicit memory management
- Have some encapsulation features

But Scheme has no typechecking (Scheme fails "dynamically"):

```
(define (sum x) (cond (null?  x) 0)
  (else (+ (car x) (sum (cdr x)))))
(sum '(1 2 "de"))
```

ML users pay a price: user-defined types, type variables, parametric polymorphism, overloading...

food for slide eater