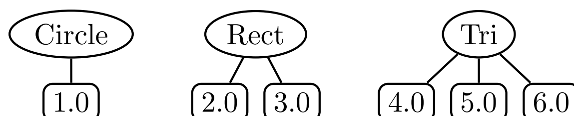
CS251 Programming Languages — **Handout # 15**
Prof. Lyn Turbak — **February 12, 2004**
Wellesley College — ***Revised February 19, 2004***

Data Types and Data Abstraction in OCAML

1 Sum-of-Product Datatypes

1.1 Geometric Figures

Every general-purpose programming language must allow the processing of values with different structure that are nevertheless considered to have the same "type". For example, in the processing of simple geometric figures, we want a notion of a "figure type" that includes circles with a radius, rectangles with a width and height, and triangles with three sides. Abstractly, figure values might be depicted as shown below:



The name in the oval is a *tag* that indicates which kind of figure the value is, and the branches leading down from the oval indicate the *components* of the value. Such types are known as **sum-of-product data types** because they consist of a sum of tagged types, each of which holds on to a product of components.

In OCAML we can declare a new `figure` type that represents these sorts of geometric figures as follows:

```
type figure =
    Circ of float (* radius *)
  | Rect of float * float (* width, height *)
  | Tri of float * float * float (* side1, side2, side3 *)
```

Such a declaration is known as a **data type** declaration. It consists of a series of `|`-separated clauses of the form

constructor-name `of` *component-types*,

where *constructor-name* must be capitalized. The names `Circ`, `Rect`, and `Tri` are the **constructors** of the `figure` type. Each serves as a function-like entity that turns components of the appopriate type into a value of type `figure`. For example, we can make a list of the three figures depicted above:

```
# let figs = [Circ 1.; Rect (2.,3.); Tri(4.,5.,6.)];; (* List of sample figures *)
val figs : figure list = [Circ 1.; Rect (2., 3.); Tri (4., 5., 6.)]
```

It turns out that constructors are *not* functions and cannot be manipulated in a first-class way. For example, we cannot write

```
List.map Circ [7.;8.;9.] (* Does not work, since Circ is not a function *)
```

However, we can always embed a constructor in a function when we need to. For example, the following does work:

```
List.map (fun r -> Circ r) [7.;8.;9.] (* This works *)
```

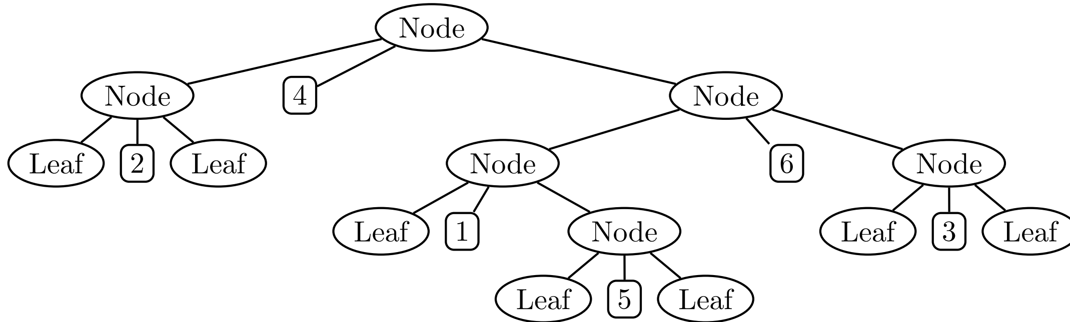
We manipulate a value of the `figure` type by using the OCAML `match` construct to perform a case analysis on the value and name its components. For example, Fig. 1 shows how to calculate figure perimiters and scale figures.

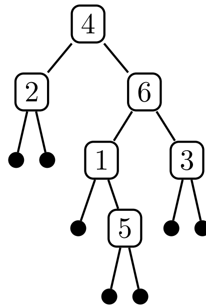
```
# let pi = 3.14159;;
val pi : float = 3.14159

(* Use pattern matching to define functions on sum-of-products datatype values *)
# let perim fig = (* Calculate perimeter of figure *)
    match fig with
      Circ r -> 2.*.pi*.r
    | Rect (w,h) -> 2.*.(w+.h)
    | Tri (s1,s2,s3) -> s1+.s2+.s3;;
val perim : figure -> float = <fun>

# List.map perim figs;;
- : float list = [6.28318; 10.; 15.]

# let scale n fig = (* Scale figure by factor n *)
  match fig with
    Circ r -> Circ (n*.r)
  | Rect (w,h) -> Rect (n*.w, n*.h)
  | Tri (s1,s2,s3) -> Tri (n*.s1, n*.s2, n*.s3);;
val scale : float -> figure -> figure = <fun>

# List.map (scale 3.) figs;;
- : figure list = [Circ 3.; Rect (6., 9.); Tri (12., 15., 18.)]

# List.map (FunUtils.o perim (scale 3.)) figs;;
- : float list = [18.84954; 30.; 45.]
```

Figure 1: Manipulations of `figure` values.

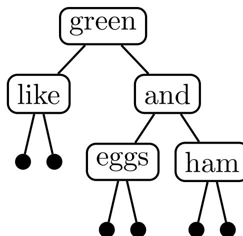
1.2 Binary Trees

```
(* Binary tree datatype abstracted over type of node value *)
type 'a bintree =
    Leaf
  | Node of 'a bintree * 'a * 'a bintree (* left subtree, value, right subtree *)

(* Sample tree of integers *)
# let int_tree =
      Node(Node(Leaf, 2, Leaf),
           4,
           Node(Node(Leaf, 1, Node(Leaf, 5, Leaf)),
                6,
                Node(Leaf, 3, Leaf)));;
val int_tree : int bintree =
  Node (Node (Leaf, 2, Leaf), 4,
   Node (Node (Leaf, 1, Node (Leaf, 5, Leaf)), 6, Node (Leaf, 3, Leaf)))
```






```
(* Sample tree of strings *)
# let string_tree =
      Node(Node(Leaf, "like", Leaf),
           "green",
           Node(Node(Leaf, "eggs", Leaf),
                "and",
                Node(Leaf, "ham", Leaf)));;
val string_tree : string bintree =
  Node (Node (Leaf, "like", Leaf), "green",
   Node (Node (Leaf, "eggs", Leaf), "and", Node (Leaf, "ham", Leaf)))
```




```
# let rec nodes tr = (* Returns number of nodes in tree *)
    match tr with
      Leaf -> 0
    | Node(l,v,r) -> 1 + (nodes l) + (nodes r);;
val nodes : 'a bintree -> int = <fun>

# nodes int_tree;;
- : int = 6

# nodes string_tree;;
- : int = 5
```

```
# let rec height tr = (* Returns height of tree *)
    match tr with
      Leaf -> 0
    | Node(l,v,r) -> 1 + max (height l) (height r);;
val height : 'a bintree -> int = <fun>

# height int_tree;;
- : int = 4

# height string_tree;;
- : int = 3

# let rec sum tr = (* Returns sum of nodes in tree of integers *)
    match tr with
      Leaf -> 0
    | Node(l,v,r) -> v + (sum l) + (sum r);;
val sum : int bintree -> int = <fun>

# sum int_tree;;
- : int = 21


# let rec prelist tr = (* Returns pre-order list of leaves *)
    match tr with
      Leaf -> []
    | Node(l,v,r) -> v :: (prelist l) @ (prelist r);;
val prelist : 'a bintree -> 'a list = <fun>

# prelist int_tree;;
- : int list = [4; 2; 6; 1; 5; 3]

# prelist string_tree;;
- : string list = ["green"; "like"; "and"; "eggs"; "ham"]

# let rec inlist tr = (* Returns in-order list of leaves *)
    match tr with
      Leaf -> []
    | Node(l,v,r) -> (inlist l) @ [v] @ (inlist r);;
val inlist : 'a bintree -> 'a list = <fun>

# inlist int_tree;;
- : int list = [2; 4; 1; 5; 6; 3]

# inlist string_tree;;
- : string list = ["like"; "green"; "eggs"; "and"; "ham"]
```

```
(* Returns post-order list of leaves *)
# let rec postlist tr =
    match tr with
      Leaf -> []
    | Node(l,v,r) -> (postlist l) @ (postlist r) @ [v];;
val postlist : 'a bintree -> 'a list = <fun>

# postlist int_tree;;
- : int list = [2; 5; 1; 3; 6; 4]

# postlist string_tree;;
- : string list = ["like"; "eggs"; "ham"; "and"; "green"]

# let rec map f tr = (* Map a function over every value in a tree *)
    match tr with
      Leaf -> Leaf
    | Node(l,v,r) -> Node(map f l, f v, map f r);;
val map : ('a -> 'b) -> 'a bintree -> 'b bintree = <fun>

# map (( * ) 10) int_tree;;
- : int bintree =
Node (Node (Leaf, 20, Leaf), 40,
 Node (Node (Leaf, 10, Node (Leaf, 50, Leaf)), 60, Node (Leaf, 30, Leaf)))

# map String.uppercase string_tree;;
- : string bintree =
Node (Node (Leaf, "LIKE", Leaf), "GREEN",
 Node (Node (Leaf, "EGGS", Leaf), "AND", Node (Leaf, "HAM", Leaf)))

# map String.length string_tree;;
- : int bintree =
Node (Node (Leaf, 4, Leaf), 5,
 Node (Node (Leaf, 4, Leaf), 3, Node (Leaf, 3, Leaf)))

# map ((flip String.get) 0) string_tree;;
- : char bintree =
Node (Node (Leaf, 'l', Leaf), 'g',
 Node (Node (Leaf, 'e', Leaf), 'a', Node (Leaf, 'h', Leaf)))
```

```
# let rec fold glue lfval tr = (* Divide/conquer/glue on trees *)
    match tr with
      Leaf -> lfval
    | Node(l,v,r) -> glue (fold glue lfval l) v (fold glue lfval r);;
val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a = <fun>

# let sumlist = fold (fun l v r -> l + v + r) 0;; (* Alternative definition *)
val sumlist : int bintree -> int = <fun>
(* can define nodes, height similarly *)

# let prelist tr = fold (fun l v r -> v :: l @ r) [] tr;; (* Alternative definition *)
val prelist : 'a bintree -> 'a list = <fun>
(* can define inlist, postlist similarly *)

# let toString valToString tr =
    fold (fun l v r -> "(" ^ l ^ " " ^ (valToString v) ^ " " ^ r ^ ")")
         "*"
         tr;;
val toString : ('a -> string) -> 'a bintree -> string = <fun>

# toString string_of_int int_tree;;
- : string = "((* 2 *) 4 ((* 1 (* 5 *)) 6 (* 3 *)))"

# toString FunUtils.id string_tree;;
- : string = "((* like *) green ((* eggs *) and (* ham *)))"
```

1.3 Other Simple Data Types

```
type 'a myOption = None | Some of 'a;;
```

```
type 'a myList = Nil | Cons of 'a * 'a myList
```

2 S-Expressions

2.1 Overview

A **symbolic expression** (**s-expression** for short) is a simple notation for representing tree structures using linear text strings containing matched pairs of parentheses. Each leaf of a tree is a **symbolic tokens**, which (to first approximation) is any sequence of characters that does not contain a left parenthesis ('`(`'), a right parenthesis ('`)`'), or a whitespace character (space, tab, newline, etc.).[1] Examples of symbolic tokens include `x`, `this-is-a-token`, `anotherKindOfToken`, `17`, `3.14159`, `4/3*pi*r^2`, `a.b[2]%3`, `'Q'`, and `"a (string) token"`. A node in a tree is represented by a pair of parentheses surrounding zero or more s-expressions that represent the node's subtrees. For example, the s-expression

```
((this is) an ((example) (s-expression tree)))
```

[1]But as we shall see, string and character literals *can* contain parentheses and whitespace characters.

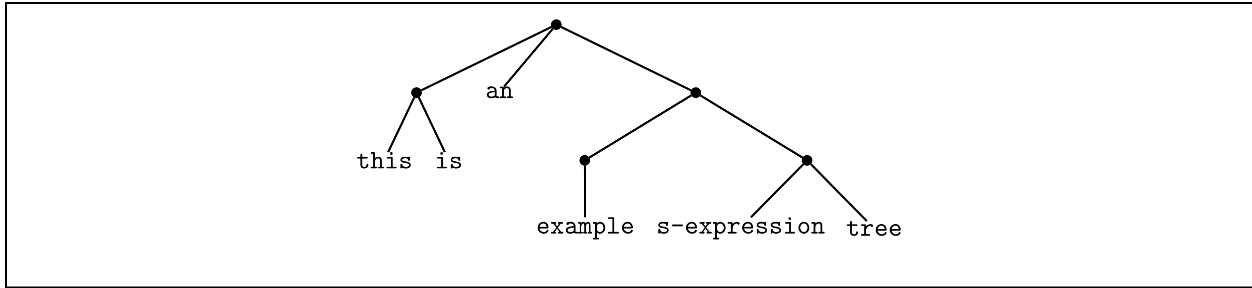


Figure 2: Viewing `((this is) an ((example) (s-expression tree)))` as a tree.

designates the structure depicted in Fig. 2. Whitespace is necessary for separating symbolic tokens that appear next to each other, but can be used liberally to enhance (or obscure!) the readability of the structure. Thus, the above s-expression could also be written as

```
((this is)
 an
 ((example)
  (s-expression
   tree)))
```

or (less readably) as

```
                  (
     (           this
is) an  (    (         example
          ) (
s-expression tree       )
  )
          )
```

without changing the structure of the tree.

S-expressions were pioneered in LISP as a notation for data as well as programs (which we have seen are just particular kinds of tree-shaped data!). We shall see that s-expressions are an exceptionally simple and elegant way of solving the parsing problem.[2] For this reason, the mini-languages we study will (at least initially) have a concrete syntax based on s-expressions.

The fact that LISP dialects (including SCHEME) have a built-in primitive for parsing s-expressions (`read`) and treating them as literals (`quote`) makes them particularly good for manipulating programs (in any language) written with s-expressions. It is not quite as convenient to manipulate s-expression program syntax in other languages, such as OCAML, but we shall see that it is still far easier than solving the parsing problem for more general notations.

2.2 S-Expression Representations of Sum-of-Product Trees

The abstract syntax trees we are trying to model (such as the one in Fig. 3) don't quite have the tree structure for s-expressions depicted above. In particular, ASTs are **sum-of-product trees** in which each node is labeled with a tag indicating the summand represented by the node, while the simplest way of interpreting s-expressions involves label-less nodes.

[2]There are detractors who hate s-expressions and claim that LISP stands for **L**ots of **I**rritating **S**illy **P**arenthesis. Apparently such people lack a critical aesthetic gene that prevents them from appreciating beautiful designs. Strangely, many such people seem to prefer the far more verbose encoding of trees in XML notation discussed later. Go figure!

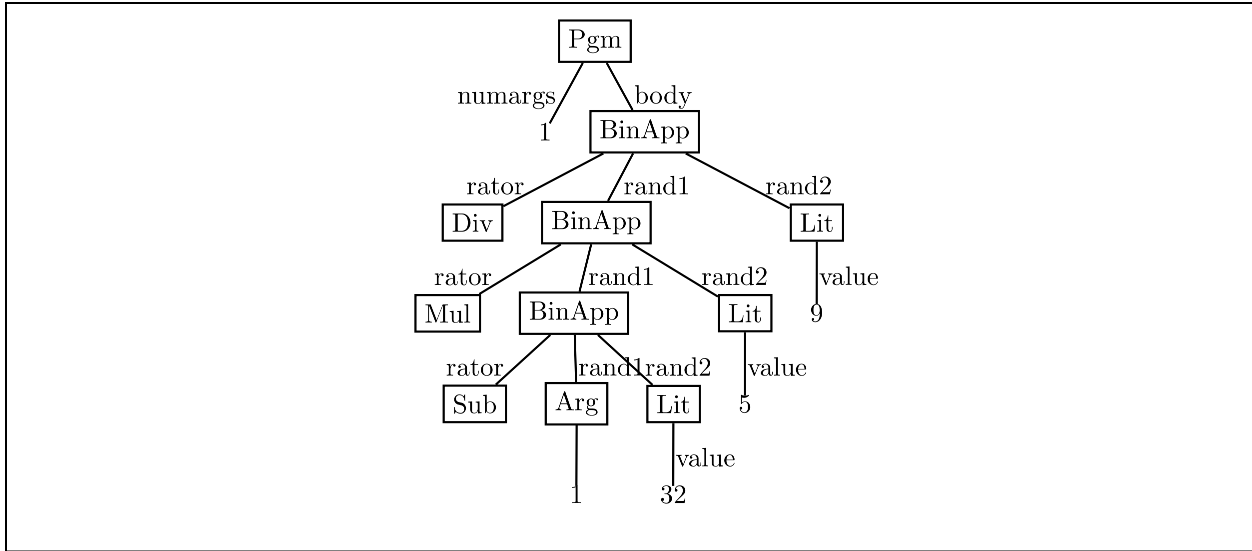


Figure 3: AST for the Fahrenheit-to-Celsius converter.

But this difference is easy to address. To model sum-of-product trees with s-expressions, we adopt the simple **prefix convention** in which the first s-expression in a parenthesized sequence is a token that is the summand node label and the remaining s-expressions are arbitrary s-expressions that represent the product components. For example, using this convention with our conversion program yields the following s-expression:

```
(pgm 1
     (binapp div
             (binapp mul
                     (binapp sub (arg 1) (lit 32))
                     (lit 5))
             (lit 9)))
```

This is still more verbose than we'd like, so we'll use some tricks to make the notation more concise/readable:

- If we assume that numbers stand for themselves, we can avoid the explicit `lit` tag. Thus, we will shorten `(lit 32)` to `32`.
- Since we must be able to distinguish argument references from integer literals, we *cannot* similarly shorten `(arg 1)` to `1`. But we can use a shorter tag name, such as `$`, in which case `(arg 1)` becomes `($ 1)`.
- The only non-leaf node is a binary application, so we can dispense with the `binapp` tag without introducing ambiguity. Using traditional operator symbols (`+`, `-`, `*`, `/`, `%`) in place of names (`add`, `sub`, `mul`, `div`, `rem`) further shortens the notation. For example, `(binapp sub ($ 1) 32)` becomes `(- ($ 1) 32)`.
- To distinguish Intex program from other programs in other mini-languages we will study, we replace `pgm` by `intex`. This is not shorter, but helps to disambiguate programs from different languages.

The result of applying all of the above tricks is

```
(intex 1 (/ (* (- ($ 1) 32) 5) 9),
```

which is significantly shorter that the OCAML notation or the unoptimized s-expression notation. This is the s-expression notation that we will adopt for INTEX. We will make similar abbreviations in other languages. Note that the syntax of LISP dialects is effectively determined by this process – prefix tags are used everywhere except for literals (e.g., numbers, booleans, strings, characters) and for applications (which are written without an explicit `apply` tag, as in `(fact 5)` rather than `(apply fact 5)`).

It is worth noting that there are other common notations for representing sum-of-product trees. The most popular of these are the XML and XML document description languages. In these languages, summand tags appear in begin/end markups and product components are encoded both in the association lists of markups as well as in components nested within the begin/end markups. For instance, Fig. 4 shows how the Fahrenheit-to-Celsius expression might be encoded in XML. The reader is left to ponder why XML, which at one level is a verbose encoding of s-expressions, is a far more popular standard for expressing structured data than s-expressions.

```
<arithop>
  <op name="/"/>
  <rand1>
    <arithop>
      <op name="*"/>
      <rand1>
        <lit num=5/>
      </rand1>
      <rand2>
        <arithop>
          <op name="-"/>
          <rand1>
            <arg index=1/>
          </rand1>
          <rand2>
            <lit num=32/>
          </rand2>
        </arithop>
      </rand2>
    </arithop>
  </rand1>
  <rand2>
    <lit num=9/>
  </rand2>
</arithop>
```

Figure 4: The Fahrenheit-to-Celsius expression in XML notation.

2.3 Representing S-Expressions in OCAML

As with any other kind of tree-shaped data, s-expressions can be represented in OCAML as values of an appropriate datatype. The OCAML datatype representing s-expression trees is presented in Fig. 5.

Recall that the leaves of s-expression trees are symbolic tokens. There are five kinds of symbolic tokens, distinguished by type:

```
type sexp =
        Int of int
      | Flt of float
      | Str of string
      | Chr of char
      | Sym of string
      | Seq of sexp list
```

Figure 5: OCAML s-expression datatype.

1. integer literals (constructed via `Int`);
2. floating point literals (constructed via `Flt`);
3. string literals (constructed via `Str`);
4. character literals (constructed via `Chr`); and
5. symbols (i.e. name tokens, constructed via `Sym`);

The nodes of s-expression trees are represented via the `Seq` constructor, whose `sexp list` argument denotes any number of s-expression subtrees.

For example, the s-expression given by the concrete notation

```
(stuff (17 3.14159) ("foo" 'c' bar))
```

would be expressed in OCAML as:

```
Seq [Sym("stuff");
     Seq [Int(17); Flt(3.14159)];
     Seq [Str("foo"); Chr('c'); Sym("bar")]]
```

As another example, the s-expression notation for the Fahrenheit-to-Celsius INTEX program,

```
(intex 1 (/ (* (- ($ 1) 32) 5) 9),
```

would be expressed in OCAML as:

```
Seq [Sym("intex");
     Int(1);
     Seq [Sym("/");
          Seq [Sym("*");
               Seq [Sym("-");
                    Seq [Sym("$"); Int(1)]
                    Int(32)]
               Int(5)]
          Int(9)]
```

3 Modules

Why modules?

- Program structure: divide big program into smaller parts.
- Data abstraction: separate specification of data abstraction (**signature**) from implementation (**structure**); allow multiple implementations of same signature.

- Name control: e.g.`List.map` vs. `Bintree.map`; export key values but hide internal auxiliary values.
- Allow abstracting one module over another (**functors**).

3.1 Structures

We can collect related declarations into a module using the notation:

```
struct module-declarations end
```

This creates a an entity called a **structure**, which is OCAML's terminology for a module. A structure can be named via the notation:

```
module module-name = structure
```

For example, Fig. 6 shows a structure named `Bintree` that collects together the binary tree declarations studied earlier in Sec. 1.2.

OCAML uses so-called **qualified names** of the form *`module-name.component-name`* ("dot notation") to extract module components from a module via their name. For example, here is an expression that can be written outside the `Bintree` module:

```
# Bintree.Node(Bintree.Leaf, 17, Bintree.map ((+) 1) Bintree.int_tree);;
- : int Bintree.bintree =
Bintree.Node (Bintree.Leaf, 17,
 Bintree.Node (Bintree.Node (Bintree.Leaf, 3, Bintree.Leaf), 5,
  Bintree.Node
   (Bintree.Node (Bintree.Leaf, 2,
     Bintree.Node (Bintree.Leaf, 6, Bintree.Leaf)),
   7, Bintree.Node (Bintree.Leaf, 4, Bintree.Leaf))))
```

Note how OCAML uses qualified names in the type reconstructed for the expression as well as in the printed value of the expression.

Qualified names are important for distinguishing values that have the same component name in two different modules. For example, we can use `List.map` and `Bintree.map` in the same expression:

```
# List.map (( * ) 2) (Bintree.prelist (Bintree.map ((+) 1) Bintree.int_tree));;
- : int list = [10; 6; 14; 4; 12; 8]
```

Using qualified names everywhere can be cumbersome. The OCAML `open` declaration "opens up" a module and permits its components to be used with their unqualified names. The `open` declaration can be used in the top-level interpreter or inside a structure. For example, here is sample top-level use:

```
# open Bintree;;
# Node(Leaf, 17, map ((+) 1) int_tree);;
- : int Bintree.bintree =
Node (Leaf, 17,
 Node (Node (Leaf, 3, Leaf), 5,
  Node (Node (Leaf, 2, Node (Leaf, 6, Leaf)), 7, Node (Leaf, 4, Leaf))))
```

Note that OCAML still tracks the module name in the reconstructed type (`int Bintree.bintree`), but drops it from constructors in the printed representation of the tree value.

As an example of using `open` within a structure, consider:

```
module Bintree = struct

  (* Binary tree datatype abstracted over type of node value *)
  type 'a bintree =
      Leaf
    | Node of 'a bintree * 'a * 'a bintree (* left subtree, value, right subtree *)

  (* Sample tree of integers *)
  let int_tree =
      Node(Node(Leaf, 2, Leaf),
           4,
           Node(Node(Leaf, 1, Node(Leaf, 5, Leaf)),
                6,
                Node(Leaf, 3, Leaf)));;

  (* Sample tree of strings *)
  let string_tree =
      Node(Node(Leaf, "like", Leaf),
           "green",
           Node(Node(Leaf, "eggs", Leaf),
                "and",
                Node(Leaf, "ham", Leaf)));;

  (* Map a function over every value in a tree *)
  let rec map f tr =
    match tr with
      Leaf -> Leaf
    | Node(l,v,r) -> Node(map f l, f v, map f r)

  (* Divide/conquer/glue on trees *)
  let rec fold glue lfval tr =
    match tr with
      Leaf -> lfval
    | Node(l,v,r) -> glue (fold glue lfval l) v (fold glue lfval r)

  let nodes tr = fold (fun l v r -> 1 + l + r) 0 tr

  let height tr = fold (fun l v r -> 1 + (max l r)) 0 tr

  let sum tr = fold (fun l v r -> l + v + r) 0 tr

  let prelist tr = fold (fun l v r -> v :: l @ r) [] tr

  let inlist tr = fold (fun l v r -> l @ [v] @ r) [] tr

  let postlist tr = fold (fun l v r -> l @ r @ [v]) [] tr

  let toString valToString tr =
    fold (fun l v r -> "(" ^ l ^ " " ^ (valToString v) ^ " " ^ r ^ ")") "*" tr

end
```

Figure 6: A `Bintree` module.

```
module Test1 = struct
  open Bintree
  let f x y = List.map (( * ) x) (prelist (map ((+) y) int_tree))
  let g z = map ((^) z) string_tree
end
```

In this case, the `open` declaration permits the use of unqualified names from the `Bintree` module in the remainder of the body of `BintreeTest1`.

It is possible to open multiple modules within a structure declaration. If two modules export the same name, the unqualified name refers to the component from the module opened last. For example:

```
module Test2 = struct
  open Bintree
  open List
  let f x y = map (( * ) x) (prelist (Bintree.map ((+) y) int_tree))
  let g z s = Bintree.map ((^) z) s
end
```

Note how the first `map` is `List.map`, since `List` was opened after `Bintree`. However, the other occurrences of `map` must be explicitly qualified to distinguish them from `List.map`.

The `module` declaration can be used to introduce synonyms for structure names within another structure. In the following module, the `Bintree` and `List` modules are not opened but are given one-letter abbreviations that makes the explicitly qualified names more concise.

```
module Test3 = struct
  module B = Bintree
  module L = List
  let f x y = L.map (( * ) x) (B.prelist (B.map ((+) y) B.int_tree))
  let g z s = B.map ((^) z) s
end
```

The OCAML module system has a sophisticated type analysis that is able to track types through `open` and module renamings. For example, `Test1.g`, `Test2.g`, and `Test3.g` all have the type

```
string -> string Bintree.bintree -> string Bintree.bintree.
```

Indeed, we can use them all together:

```
# Test1.g "a" (Test2.g "b" (Test3.g "c" Bintree.string_tree));;
- : string Bintree.bintree =
Node (Node (Leaf, "abclike", Leaf), "abcgreen",
 Node (Node (Leaf, "abceggs", Leaf), "abcand", Node (Leaf, "abcham", Leaf)))
```

We may also use one `module` declaration within another to define nested structures. An example of this is shown in Fig. 7. A sequence of module qualifications can be used to extract the innermost components:

```
# Nested.Funs.f2 (Nested.Funs.f1 1 Nested.Data.t2);;
- : int = 12
```

An OCAML structure is somewhat like records/structs/objects in other languages. For example, dot notation is used to extract record components in PASCAL, struct components in C, and object components in JAVA. There are two key differences between OCAML structures and traditional record values:

1. OCAML structures can include type components as well as value components. For example,

```
module Nested = struct

  open Bintree

  module Data = struct
    let t1 = Node(Leaf, 1, Leaf)
    let t2 = Node(t1, 2, t1)
  end

  module Funs = struct
    let f1 x = map ((+) x)
    let f2 = fold (fun l v r -> l * v * r) 1
  end

end
```

Figure 7: An example of nested structures.

the `Bintree` module in Fig. 6 include the definition of the `bintree` data type. It turns out that handling modules with type components requires a sophisticated type system.

2. Unlike traditional record values, structures are *second-class* entities in OCAML – they can be manipulated only in limited ways. For instance, structures cannot be named with a `let`, passed as arguments to functions, returned from functions as results, or stored in data structures.[3] This limitation is imposed to simplify the type system.

3.2 Signatures

If the structure in Fig. 6 is stored in the file `Bintree.ml`, then we can load it into the top-level interpreter as follows:

```
# #use "Bintree.ml";;
module Bintree :
  sig
    type 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree
    val int_tree : int bintree
    val string_tree : string bintree
    val map : ('a -> 'b) -> 'a bintree -> 'b bintree
    val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a
    val nodes : 'a bintree -> int
    val height : 'a bintree -> int
    val sum : int bintree -> int
    val prelist : 'a bintree -> 'a list
    val inlist : 'a bintree -> 'a list
    val postlist : 'a bintree -> 'a list
    val toString : ('a -> string) -> 'a bintree -> string
  end
```

A module has a type, which is called its **signature**. A signature consists of a collection of declaration types between keywords `sig` and `end`. Note how the signature of `Bintree` includes the parameterized `bintree` datatype declaration as well as the declarations of all the values in the structure.

[3] OCAML also provides traditional record structures that *are* first class.

It is possible to name signatures and to declare that structures have a signature. For instance, we can modify the file `Bintree.ml` as shown in Fig. 8. The notation

`module type` *signature-name* `=` *signature*

introduces a named signature. We can declare that a structure has a particular signature by writing

`module` *module-name* `:` *signature* `=` *structure*,

where *signature* is either a signature name, or an explicit signature of the form `sig` ... `end`.

```
module type BINTREE = sig
  type 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree
  val map : ('a -> 'b) -> 'a bintree -> 'b bintree
  val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a
  val nodes : 'a bintree -> int
  val height : 'a bintree -> int
  val sum : int bintree -> int
  val prelist : 'a bintree -> 'a list
  val inlist : 'a bintree -> 'a list
  val postlist : 'a bintree -> 'a list
  val toString : ('a -> string) -> 'a bintree -> string
end

module Bintree : BINTREE = struct
  same declarations as above
end
```

Figure 8: Modified `Bintree.ml` file containing both signature and structure.

When we load the modified `Bintree.ml` into the top-level interpreter, we see the following:

```
# #use "Bintree.ml";;
module type BINTREE =
  sig
    type 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree
    val int_tree : int bintree
    val string_tree : string bintree
    val map : ('a -> 'b) -> 'a bintree -> 'b bintree
    val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a
    val nodes : 'a bintree -> int
    val height : 'a bintree -> int
    val sum : int bintree -> int
    val prelist : 'a bintree -> 'a list
    val inlist : 'a bintree -> 'a list
    val postlist : 'a bintree -> 'a list
    val toString : ('a -> string) -> 'a bintree -> string
  end
module Bintree : BINTREE
```

Note how the notation `module Bintree : BINTREE` is used to declare that the `Bintree` structure has the `BINTREE` signature. There is no need to put signatures and structures into the same file; we can store them in separate files if we wish.

Signatures can be used to hide module components. When a module is given an explicit signature, only the names mentioned in the signature are exported from the module; not other names can be extracted from the module. For example, we can defined a restricted version `BT` of the

`Bintree` module as follows:

```
module BT : sig type 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree
                val height : 'a bintree -> int
            end
  = Bintree
```

The `BT` module exports only the `bintree` type, the constructors `BT.Leaf` and `BT.Node`, and the function `BT.height`. Other functions are not exported. For example, we cannot use `BT.fold` even though `fold` is used internally to to define `height`.

3.3 Abstract Data Types

The hiding feature of signatures can be used to hide the implementation of a type. For example, consider the module declaration:

```
module BT2 : sig type 'a bintree
                 val int_tree : int bintree
                 val string_tree : string bintree
                 val map : ('a -> 'b) -> 'a bintree -> 'b bintree
                 val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a end
  = Bintree
```

`BT2` is a version of `Bintree` that exports only the `bintree` type, the functions `map` and `fold`, and the trees `int_tree` and `string_tree`. The declaration `type 'a bintree` says that `BT2` exports a parameterized `bintree` type without giving the definition of this type. Such a type is said to be an **abstract type** because its representation is not known outside the module.

The top-level intepreter will not divulge any details about the representation of the abstract binary trees in `BT2`:

```
# BT2.map ((+) 1) BT2.int_tree;;
- : int BT2.bintree = <abstr>
```

In contrast, it will give details on the concrete binary trees in `Bintree`:

```
# Bintree.map ((+) 1) Bintree.int_tree;;
- : int Bintree.bintree =
Node (Node (Leaf, 3, Leaf), 5,
 Node (Node (Leaf, 2, Node (Leaf, 6, Leaf)), 7, Node (Leaf, 4, Leaf)))
```

Since `BT2` does not export the `Leaf` and `Node` constructors, it is not possible to make arbitrary new trees using the `BT2` module. The only way to create a new tree is to use the `BT2.map` operator on an existing tree (`BT2.int_tree` or `BT2.string_tree`. The types `BT2.intree` and `Bintree.intree` are considered to be different, so we can't try to pass a tree constructed with `Bintree` to a `BT2` function. For example, the expression `BT2.map ((+) 1) Bintree.Leaf` is not well-typed.

The hiding feature provided by OCAML modules is ideal for realizing an **abstract data type (ADT)**, in which a contract serves as an abstraction barrier that separates the client and implementer of a collection of functions that manipulate an abstract value. A classic example of an ADT is a set. From the client's perspective, a set is an abstract collection of values that contains each value at most once and which supports operations like membership testing, insertion, deletion, and the union, intersection, and difference of sets. An implementer can use any concrete data representation and algorithms to implement the set as long as the set operations work as expected. For example, the implementation may involve collections of elements potentially containing duplicate entries as long as the set functions make it appear as though the set contains exactly one occurrence

of each element.

In OCAML an ADT contract is represented as a signature and an ADT implementation is a module satisfying that signature. For example, Fig. 9 shows the signature for a set ADT. Each type declaration in the signature is accompanied by an English description specifying the meaning of the declared operation or value. By not giving a concrete definition of the `set` type, the declaration `type 'a set` guarantees that the ADT is truly abstract. A client can only use the operations in the signature to create and manipulate sets. The type system prevents any attempt by the client to manipulate whatever the underlying concrete representation type of the set might be. For instance, if sets are represented as lists, then any attempt by the client to perform list operations directly on a set will fail.

```
module type SET = sig

  type 'a set
  val empty : 'a set                    (* the empty set *)
  val singleton : 'a -> 'a set          (* a set with one element *)
  val insert : 'a -> 'a set -> 'a set   (* insert elt into given set *)
  val delete : 'a -> 'a set -> 'a set   (* delete elt from given set *)
  val member : 'a -> 'a set -> bool     (* is elt a member of given set? *)
  val union: 'a set -> 'a set -> 'a set (* union of two sets *)
  val intersection: 'a set -> 'a set -> 'a set (* intersection of two sets *)
  val difference: 'a set -> 'a set -> 'a set   (* difference of two sets *)
  val fromList : 'a list -> 'a set      (* create a set from a list *)
  val toList : 'a set -> 'a list        (* list all set elts, sorted low to high *)
  val toSexp : ('a -> Sexp.sexp)
               -> 'a set -> Sexp.sexp   (* return an s-expression rep. of a list *)
  val fromSexp : (Sexp.sexp -> 'a)
                -> Sexp.sexp -> 'a set   (* return an s-expression rep. of a list *)
  val toString : ('a -> string)
                -> 'a set -> string     (* string representation of the set *)

end
```

Figure 9: A signature for a set abstract data type (ADT).

The signature gives great latitude for an implementer to choose a representation for the ADT. In the case of sets, a simple representation for a set is a list of elements without duplicates sorted from low to high. For the ordering criteria, we use the built-in ordering that OCAML provide for any type. A handy collection of functions for manipulating such lists is provided in the `ListSetUtils` module (Fig. 10), whose signature is:

```
module type LIST_SET_UTILS = sig
  val member: 'a -> 'a list -> bool
  val insert: 'a -> 'a list -> 'a list
  val delete: 'a -> 'a list -> 'a list
  val union: 'a list -> 'a list -> 'a list
  val intersection: 'a list -> 'a list -> 'a list
  val difference: 'a list -> 'a list -> 'a list
end
```

```
let fromList xs = xs
```

because the list `xs` might contain elements out of order or contain duplicate elements.

A set implementation using these functions is the `SortedListSet` module presented in Fig. 11.

```
module ListSetUtils : LIST_SET_UTILS = struct

  let rec member x ys =
    match ys with
      [] -> false
    | y::ys' -> (x = y) || ((x > y) && (member x ys'))

  (* Insert an element into a sorted list *)
  let rec insert x ys =
    match ys with
      [] -> [x]
    | y::ys' -> if x < y then x::ys
                else if x = y then ys
                else y::(insert x ys')

  (* Delete an element from a sorted list *)
  let rec delete x ys =
    match ys with
      [] -> []
    | y::ys' -> if x = y then ys'
                else if x < y then ys
                else y::(delete x ys')

  (* Merge two sorted lists, removing duplicates *)
  let rec union xs ys =
    match (xs, ys) with
      ([], _) -> ys
    | (_, []) -> xs
    | (x::xs',y::ys') -> if x = y then x::(union xs' ys')
                         else if x < y then x::(union xs' ys)
                         else y::(union xs ys')

  (* Intersection of two sorted lists *)
  let rec intersection xs ys =
    match (xs, ys) with
      ([], _) -> []
    | (_, []) -> []
    | (x::xs',y::ys') -> if x = y then x::(intersection xs' ys')
                         else if x < y then intersection xs' ys
                         else intersection xs ys'

  (* Difference of two sorted lists *)
  let rec difference xs ys =
    match (xs, ys) with
      ([], _) -> []
    | (_, []) -> xs
    | (x::xs',y::ys') -> if x = y then difference xs' ys
                         else if x < y then x::(difference xs' ys)
                         else difference xs ys'

end
```

Figure 10: Utilities used to process sorted lists.

Of particular interest is the `fromList` function, which uses `insert` to insert all elements of the given list into the resulting set. This preserves the invariant that the set must be a sorted list without duplicates. It would be incorrect to defined `fromList` as

```
module SortedListSet : SET = struct

  module LSU = ListSetUtils (* Abbreviation for list set utilities *)

  type 'a set = 'a list

  let empty = []

  let singleton x = [x]

  let insert x s = LSU.insert x s

  let delete x s = LSU.delete x s

  let member x s = LSU.member x s

  let union s1 s2 = LSU.union s1 s2

  let intersection s1 s2 = LSU.intersection s1 s2

  let difference s1 s2 = LSU.difference s1 s2

  let toList s = s

  let fromList xs = List.fold_right insert xs empty

  let fromSexp eltFromSexp sexp =
    match sexp with
      Sexp.Seq elts -> List.map eltFromSexp elts
    | _ -> raise (Failure "unrecognized s-expression")

  let toSexp eltToSexp xs = Sexp.Seq(List.map eltToSexp xs)

  let toString eltToString s = StringUtils.listToString eltToString s

end
```

Figure 11: An implementation of the set ADT using sorted lists.

Of course, the `SortedListSet` module is only one possible implementation of the set ADT. There are many other possible implementations, particularly variants of **binary search trees (BSTs)** – binary trees of elements in which all elements in the left subtree of each node are strictly less than the element value of that node, and all elements in the right subtree of each node are strictly greater than the element value of that node.

The OCAML type system is sophisticated enough to allow several implementations of the same ADT to be used in the same program. The hard part about this is that it must be a type error for the operations of one implementation to be used on a value created by another implementation. For instance, suppose we have a `BSTSet` module implementing the `SET` signature in addition to the `SortedListSet` module, and we make the following two sets:

```
# let sls = SortedListSet.fromList [1;2;3];;
val sls : int SortedListSet.set = <abstr>

# let bst = BSTSet.fromList [2;3;4];;
val bst : int BSTSet.set = <abstr>
```

Then it should be a type error to use a `SortedListSet` operation on `bst` or to use a `BSTSet` operation on `sls`. And indeed it is:

```
# SortedListSet.insert 1 bst;;
Characters 23-26:
  SortedListSet.insert 1 bst;;
                         ^^^
This expression has type int BSTSet.set but is here used with type
  int SortedListSet.set

# BSTSet.union sls bst;;
Characters 13-16:
  BSTSet.union sls bst;;
               ^^^
This expression has type int SortedListSet.set but is here used with type
  'a BSTSet.set
```

OCAML is able to determine this by keeping track of which module the sets come from. In this case, `sls` has type `int SortedListSet.set`, while `bst` has type `int BSTSet.set`, and these types are considered distinct by the type system.

3.4 Functors

There are many situations where we would like to abstract over the particular structure that is used to implement a given signature. For example, we want to be able to write testing code for a set implementation that gives us confidence that the implementation is implemented corrected. Because we only care about the abstract behavior of sets in our testing code, we would like to be able to use the same testing code with any set implementation, regardless of its concrete representation.

Since structures are second-class entities in OCAML, we cannot use functions to abstract over them. However, OCAML supplies us with a function-like entity called a **functor** that *is* able to abstract over structures. In order to provide type safety guarantees, OCAML makes functors more restrictive than functions – they can only be declared and used in limited ways. Nevertheless, functors are still a powerful way to abstract over the details of particular structures.

As a simple example of a functor, consider the set-testing functor `SimpleSetTest` shown in Fig. 12. `SimpleSetTest` is a functor that takes as its single argument any structure `Set` satisfying the `SET` signature. As its result, it returns a structure with the single declaration for a testing function named `test`. This `test` function uses operations in the the `Set` structure to manipulate sets of the type `int Set.set`. It returns a triple of (1) a set containing the elements 1,2,4,5,6; (2) a list of integer lists showing the results of various set operations; and (3) a list of string lists that shows the results of some other set operations.

We can load `SimpleSetTest` into the top-level interpreter as follows:

```
# #use "../sets/SimpleSetTest.ml";;
module SimpleSetTest :
  functor (Set : SET) ->
    sig val test : unit -> int Set.set * int list list * string list end
```

```
module SimpleSetTest =
  functor (Set: SET) -> struct
    let test () =
      let s1 = Set.fromList [5;2;6;1;4]
      and s2 = Set.fromList [2;8;6;3]
      in ( s1,

           [Set.toList s1;
            Set.toList s2;
            Set.toList (Set.insert 3 s1);
            Set.toList (Set.delete 5 s1);
            Set.toList (Set.union s1 s2);
            Set.toList (Set.intersection s1 s2);
            Set.toList (Set.difference s1 s2)],

           [Set.toString string_of_int s1;
            Sexp.sexpToString(Set.toSexp (fun x -> Sexp.Int x) s1)]

            )
 end
```

Figure 12: A simple set-testing functor.

Note how the first component of the returned triple refers to the `Set` argument given to the functor. A type in which result types depend on argument types is known as a **dependent type**.

We can now give `SimpleSetTest` a spin on different set structures:

```
# module SLST = SimpleSetTest(SortedListSet);;
module SLST :
  sig
    val test : unit -> int SortedListSet.set * int list list * string list
  end

# SLST.test();;
- : int SortedListSet.set * int list list * string list =
(<abstr>,
[[1; 2; 4; 5; 6]; [2; 3; 6; 8]; [1; 2; 3; 4; 5; 6]; [1; 2; 4; 6];
 [1; 2; 3; 4; 5; 6; 8]; [2; 6]; [1; 4; 5]],
["[1,2,4,5,6]"; "(1 2 4 5 6)"])

# module BSTST = SimpleSetTest(BSTSet);;
module BSTST :
  sig val test : unit -> int BSTSet.set * int list list * string list end

# BSTST.test();;
- : int BSTSet.set * int list list * string list =
(<abstr>,
[[1; 2; 4; 5; 6]; [2; 3; 6; 8]; [1; 2; 3; 4; 5; 6]; [1; 2; 4; 6];
 [1; 2; 3; 4; 5; 6; 8]; [2; 6]; [1; 4; 5]],
["((* 1 (* 2 *)) 4 ((* 5 *) 6 *))"; "((1 (2)) 4 ((5) 6))"])
```

By using the printed representation `<abstr>`, OCAML hides the implementation details of the given set structure. However, the `toSexp` and `toString` functions expose the details of which structure is used in this example. This is not a failure of the OCAML module system; it just reflects that

these two operations are defined in an ambiguous way that allows them to return different results for different implementations.