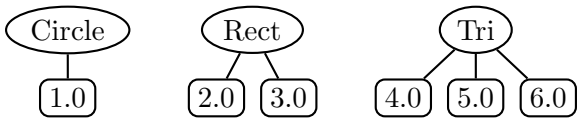
CS251 Programming Languages **Handout # 25**
Prof. Lyn Turbak **February 28, 2007**
Wellesley College

Sum-of-Product Data Types

1 Introduction

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. They are also known as **algebraic data types**.

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 called 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 perimeters and scale figures.

Using data type declarations, we can create user-defined versions of some of the built-in OCAML types that we have been using. For instance, here is a definition of the `option` type:

```
# 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.

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

This is an example of a parameterized data type declaration in which the `'a` can be instantiated to any type. `option` is an example of a `type constructor`.

```
# None;;
- : 'a option = None

# Some 3;;
- : int option = Some 3

# Some true;;
- : bool option = Some true
```

We can even construct our own version of a list type (though it won't support the infix `::` operator or the square-bracket list syntax):

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

Note that lists are a recursive data type. Data types are implicitly recursive without the need for any keyword like `rec`. Here are our lists in action:

```
# let ns = Cons(1, Cons(2, Cons(3, Nil)));;
val ns : int myList = Cons (1, Cons (2, Cons (3, Nil)))

# let bs = Cons('a', Cons('b', Cons('c', Nil)));;
val bs : char myList = Cons ('a', Cons ('b', Cons ('c', Nil)))
```

```
# let ss = Cons("d", Cons("n", Cons("t", Nil)));;
val ss : string myList = Cons ("d", Cons ("n", Cons ("t", Nil)))

# let rec map f xs =
    match xs with
      Nil -> Nil
    | Cons(x,xs') -> Cons(f x, map f xs');;
      val map : ('a -> 'b) -> 'a myList -> 'b myList = <fun>

# map ((+) 1) ns;;
- : int myList = Cons (2, Cons (3, Cons (4, Nil)))

# map ((^) "a") ss;;
- : string myList = Cons ("ad", Cons ("an", Cons ("at", Nil)))
```

Data type declarations may be mutually recursive if they are glued together with `and`. For example, here is the definition of data types and constructors for lists of even length and odd length:

```
# type 'a evenList = ENil | ECons of 'a * ('a oddList)
  and 'a oddList = OCons of 'a * ('a evenList);;
type 'a evenList = ENil | ECons of 'a * 'a oddList

type 'a oddList = OCons of 'a * 'a evenList

# OCons(2, ECons(3, OCons(5, ENil)));;
- : int oddList = OCons (2, ECons (3, OCons (5, ENil)))

# ECons(2, ENil);;
Characters 9-13:
  ECons(2, ENil);;
           ^^^^

This expression has type 'a evenList but is here used with type int oddList
```

The last example shows that the type system cannot be fooled by declaring an odd-length list to have even length.

Sometimes it is helpful for data type declarations to have multiple type parameters. These must be enclosed in parenthesis and separated by commas. For example:

```
# type ('a,'b) swaplist = SNil | SCons of 'a * (('b,'a) swaplist);;
type ('a, 'b) swaplist = SNil | SCons of 'a * ('b, 'a) swaplist

# let alts = SCons(1, SCons(true, SCons(2, SCons(false, SNil))));;
val alts : (int, bool) swaplist =
  SCons (1, SCons (true, SCons (2, SCons (false, SNil))))

# let stail xs =
    match xs with
      SNil -> raise (Failure "attempt to take tail of empty swaplist")
    | SCons(x,xs') -> xs';;
val stail : ('a, 'b) swaplist -> ('b, 'a) swaplist = <fun>

# stail alts;;
- : (bool, int) swaplist = SCons (true, SCons (2, SCons (false, SNil)))

# let srev xs =
    let rec loop olds news =
      match olds with
        SNil -> news
      | SCons(x,xs') -> loop xs' (SCons(x,news))
    in loop xs SNil;;
val srev : ('a, 'a) swaplist -> ('a, 'a) swaplist = <fun>
```

Note that the OCAML type reconstruction process forces both `swaplist` type parameters to be the same. Intuitively, this is because the type of the first element of the reversed list depends on whether the list length is even or odd. When these two types are unified, `srev` works on any length of list.

```
# srev alts;;
Characters 5-9:
  srev alts;;
       ^^^^
This expression has type (int, bool) swaplist but is here used with type
  (int, int) swaplist
```

Invoking `srev` on `alts` fails because it has type `(int, bool) swaplist`, which is not match the form `('a, 'a) swaplist`. However, we can use `srev` on lists that do match this form:

```
# srev (SCons(true, SCons(false, SNil)));;
- : (bool, bool) swaplist = SCons (false, SCons (true, SNil))

# srev (SCons(1, SCons(2, SCons(3, SNil))));;
- : (int, int) swaplist = SCons (3, SCons (2, SCons (1, SNil)))
```

Data type declarations are different from type abbreviations, even though both are are introduced via the `type` keyword. Consider the following:

```
# type 'a doubloon1 = 'a * 'a;; (* Type abbreviation *)
type 'a doubloon1 = 'a * 'a

# type 'a doubloon2 = Doubloon of 'a * 'a;; (* Data type declaration *)
type 'a doubloon2 = Doubloon of 'a * 'a
```

The presence of the capitalized constructor name `Doubloon` (as well as the keyword `of`[1]) is the syntactic marker that indicates that `doubloon2` is a sum-of-products data type rather than a type abbreviation. Recall that constructor names *must* be capitalized in OCAML.

Note that `doubloon2` is an example of a data type with just a single constructor. Such data types can be useful as a simple data abstraction mechanism in cases where it is desirable to distinguish representations that happen to have the same concrete type.[2] For example:

```
# let swap1 ((x,y) : 'a doubloon1) = (y, x);;
val swap1 : 'a doubloon1 -> 'a * 'a = <fun>

# let swap2 d = match d with Doubloon (x,y) -> Doubloon (y,x);;
val swap2 : 'a doubloon2 -> 'a doubloon2 = <fun>

# swap1 (1,2);;
- : int * int = (2, 1)

# swap2 (Doubloon(1,2));;
- : int doubloon2 = Doubloon (2, 1)

# swap1 (Doubloon(1,2));;
Characters 7-20:
  swap1 (Doubloon(1,2));;
         ^^^^^^^^^^^^^
This expression has type 'a doubloon2 but is here used with type
  'b doubloon1 = 'b * 'b
```

[1] The `of` keyword is not required for nullary constructors, such as `None` in the `option` data type.

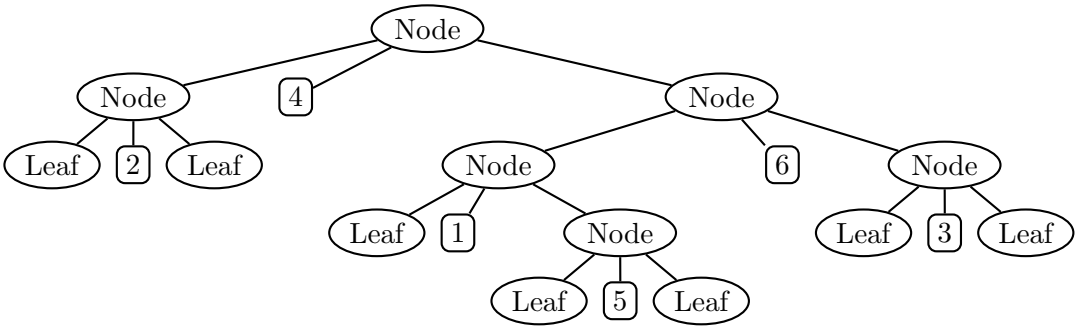
[2] However, this is a very crude form of abstraction, since the concrete representation can be manipulated via pattern matching. For true data abstraction, the module mechanism described in Handout #23 should be used.

```
# swap2 (1,2);;
Characters 7-10:
  swap2 (1,2);;
         ^^^
This expression has type int * int but is here used with type 'a doubloon2
```

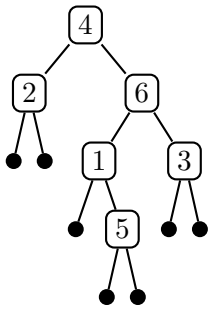
2 Binary Trees

2.1 Background

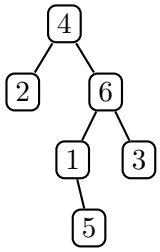
A classic example of a parameterized recursive data type is the **binary tree**. A binary tree is either (1) a leaf or (2) a node with a left subtree, value, and right subtree. Here is a depiction of a sample binary tree:



The depiction is more compact if we put each value in the node position and use ● for each leaf:



The notation is even more compact if we do not draw the leaves:



2.2 Binary Trees in OCaml

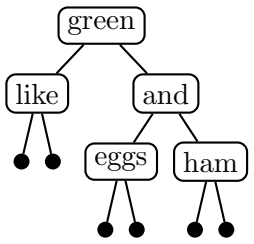
We can express the type of a binary tree in OCaml as follows:

```
(* 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 *)
```

Here is how we create our sample binary tree of integers in OCaml:

```
# 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)))
```

Of course, we can create a binary tree of strings instead, such as:




```
# 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)))
```

Now we'll define some simple functions that manipulate trees in Figs. 2–4. In each case we will use the `match` construct to test whether a given tree is a leaf or a node and to extract the components of the node.

```
# let rec nodes tr = (* Returns number of nodes in tree *)



val nodes : 'a bintree -> int = <fun>

# nodes int_tree;;
- : int = 6

# nodes string_tree;;
- : int = 5

# let rec height tr = (* Returns height of tree *)



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 *)



val sum : int bintree -> int = <fun>

# sum int_tree;;
- : int = 21
```

Figure 2: Binary tree functions, part 1.

```
# let rec prelist tr = (* Returns pre-order list of leaves *)



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 *)



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 =



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"]
```

Figure 3: Binary tree functions, part 2.

```
# let rec map f tr = (* Map a function over every value in a tree *)



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 *)



val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a = <fun>

# let sum = fold (fun l v r -> l + v + r) 0;; (* Alternative definition *)
val sum : 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 =



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 *)))"
```

Figure 4: Binary tree functions, part 3.

2.3 Binary Search Trees

A common use of binary trees in practice is to implement **binary search trees** (BSTs). The **BST condition** holds at a non-leaf binary tree node if (1) all elements in the left subtree of the node are $\leq$ the node value and (2) all elements in the right subtree of the node are $\geq$ the node value. A tree is a BST if the BST condition holds at all non-leaf nodes in the tree.

For example, below are some of the many possible BSTs containing the numbers 1 through 7. Note that a BST is not required to be balanced in any way.

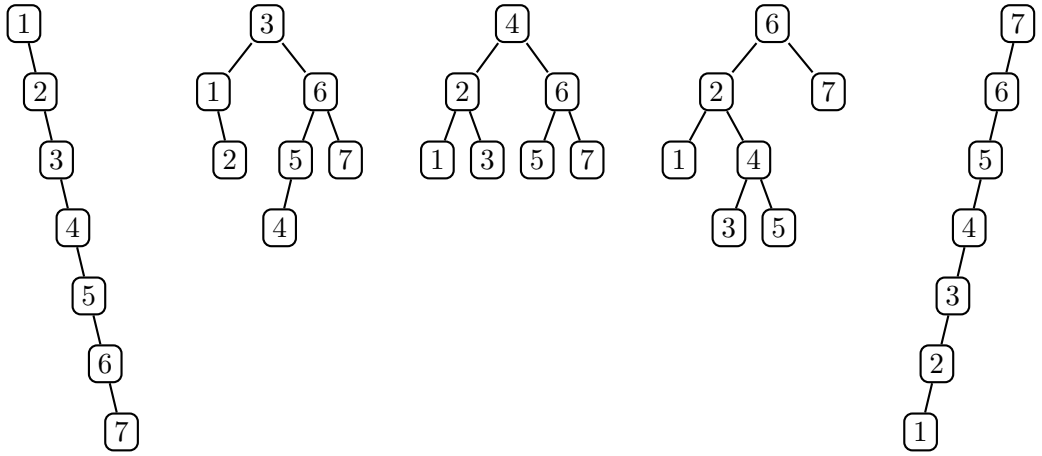


Fig. 5 presents a `BSTSet` module in which sets are represented as binary search trees. (This is the `BSTSet` module mentioned in the discussion of sets in Handout #23.) The combination of sum-of-product data types, pattern matching on these types, and higher-order functions makes this implementation remarkably concise. It is worthwhile for you to compare this implementation to a corresponding implementation in other languages, such as JAVA or C.

3 S-Expressions

3.1 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 an **atom**, 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.).[3] Examples of atoms include `x`, `this-is-an-atom`, `anotherKindOfAtom`, `17`, `3.14159`, `4/3*pi*r^2`, `a.b[2]%3`, `'Q'`, and `"a (string) atom"`. A node in an s-expression tree is represented by a pair of parentheses surrounding zero or s-expressions that represent the node's subtrees. For example, the s-expression

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

designates the structure depicted in Fig. 6. Whitespace is necessary for separating atoms 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

[3] As we shall see, string and character literals *can* contain parentheses and whitespace characters.

```
module BSTSet : SET = struct

  open Bintree (* exports bintree datatype and functions from previous section *)

  type 'a set = 'a bintree

  let empty = Leaf

  let singleton x = Node(Leaf, x, Leaf)

  let rec insert x t =
    match t with
      Leaf -> singleton x
    | Node(l,v,r) -> if x = v then t
                     else if x < v then Node(insert x l, v, r)
                     else Node(l, v, insert x r)

  let rec member x s =
    match s with
      Leaf -> false
    | Node(l,v,r) -> if x = v then true
                     else if x < v then member x l
                     else member x r

  (* Assume called on non-empty tree *)
  let rec deleteMax t =
    match t with
      Leaf -> raise (Failure "shouldn't happen")
    | Node(l,v,Leaf) -> (v, l)
    | Node(l,v,r) -> let (max, r') = deleteMax r in
                       (max, Node(l,v,r'))

  let rec delete x s =
    match s with
      Leaf -> Leaf
    | Node(l,v,Leaf) when v = x -> l
    | Node(Leaf,v,r) when v = x -> r
    | Node(l,v,r) -> if x = v then
                       let (pred, l') = deleteMax l in Node(l', pred, r)
                     else if x < v then Node(delete x l, v, r)
                     else Node(l, v, delete x r)

  let rec toList s = inlist s

  let rec union s1 s2 = ListUtils.foldr insert s2 (postlist s1)
  (* In union and below, postlist helps to preserve balance more than
     inlist or prelist in a foldr.  For a foldl, prelist would be best. *)

  let rec intersection s1 s2 =
    ListUtils.foldr (fun x s -> if member x s1 then insert x s
                                else s)
                    empty
                    (postlist s2)

  let rec difference s1 s2 = ListUtils.foldr delete s1 (postlist s2)

  let rec fromList xs = ListUtils.foldr insert empty xs

  let rec toString eltToString s = StringUtils.listToString eltToString (toList s)

end
```

Figure 5: An implementation of the SET signature using binary search trees.

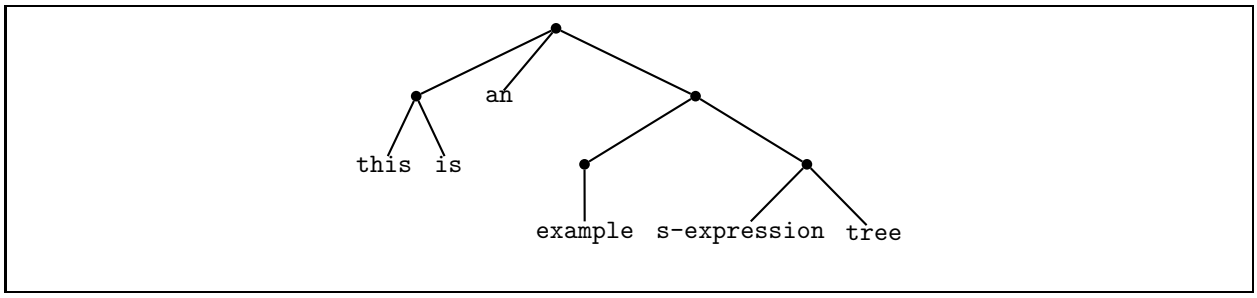


Figure 6: Viewing `((this is) an ((example) (s-expression tree)))` as a tree.

```
((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**: translating string-based representations of data structures and programs into the tree structures they denote.[4] For this reason, all the mini-languages we study later in this course 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.

3.2 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. 7.

There are five kinds of atoms, distinguished by type; these are the leaves of s-expression trees:

1. integer literals, constructed via `Int`;
2. floating point literals, constructed via `Flt`;

[4]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 in the course. Go figure!

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

Figure 7: OCAML s-expression datatype.

3. string literals, constructed via `Str`;
4. character literals, constructed via `Chr`; and
5. symbols, which are name tokens (as distinguished from quoted strings), 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")]]
```

The `Sexp` module in `~/cs251/utils/Sexp.ml` contains several handy utilities for manipulating `sexp` trees. In particular, it contains functions for **parsing** s-expression strings into `sexp` trees and for **unparsing** `sexp` trees into s-expression trees. The signature `SEXP` for the `Sexp` module is presented in Fig. 8. We will not study the implementation of the `Sexp` functions, but will use them as black boxes.

Here are some sample invocations of functions from the `Sexp` module:

```
# let s = Sexp.stringToSexp "(stuff (17 3.14159) (\"foo\" 'c' bar))";;
val s : Sexp.sexp =
  Sexp.Seq
   [Sexp.Sym "stuff"; Sexp.Seq [Sexp.Int 17; Sexp.Flt 3.14159];
    Sexp.Seq [Sexp.Str "foo"; Sexp.Chr 'c'; Sexp.Sym "bar"]]

# Sexp.sexpToString s;;
- : string = "(stuff (17 3.14159) (\"foo\" 'c' bar))"

# let ss = Sexp.stringToSexps "stuff (17 3.14159) (\"foo\" 'c' bar)";;
val ss : Sexp.sexp list =
  [Sexp.Sym "stuff"; Sexp.Seq [Sexp.Int 17; Sexp.Flt 3.14159];
   Sexp.Seq [Sexp.Str "foo"; Sexp.Chr 'c'; Sexp.Sym "bar"]]

# Sexp.sexpsToString ss;;
- : string = "stuff\n\n(17 3.14159)\n\n(\"foo\" 'c' bar)"
```

```
module type SEXP = sig

  (* The sexp type is exposed for the world to see *)
  type sexp =
      Int of int
    | Flt of float
    | Str of string
    | Chr of char
    | Sym of string
    | Seq of sexp list

  exception IllFormedSexp of string
  (* This exception is used for all errors in s-expression manipulation *)

  val stringToSexp : string -> sexp
  (* (stringToSexp <str>) returns the sexp tree represented by the s-expression
     <str>. Raise an IllFormedSexp exception if <str> is not a valid
     s-expression string. *)

  val stringToSexps : string -> sexp list
  (* (stringToSexps <str>) returns the list of sexp trees represented by <str>, which
     is a string containing a sequence of s-expressions. Raise an IllFormedSexp
     exception if <str> not a valid representation of a sequence of s-expressions. *)

  val fileToSexp : string -> sexp
  (* (fileToSexp <filename>) returns the sexp tree represented by the s-expression
     contents of the file named by <filename>. Raises an IllFormedSexp exception
     if the file contents is not a valid s-expression. *)

  val fileToSexps : string -> sexp list
  (* (fileToSexps <filename>) returns the list of sexp trees represented by the
     contents of the file named by <filename>. Raises an IllFormedSexp exception if
     the file contents is not a valid representation of a sequence of s-expressions. *)

  val sexpToString : sexp -> string
  (* (sexpToString <sexp>) returns an s-expression string representing <sexp> *)

  val sexpToString' : int -> sexp -> string
  (* (sexpToString' <width> <sexp>) returns an s-expression string representing
     <sexp> in which an attempt is made for each line of the result to be
     <= <width> characters wide. *)

  val sexpsToString : sexp list -> string
  (* (sexpsToString <sexps>) returns string representations of the sexp trees
     in <sexps> separated by two newlines. *)

  val sexpToFile : sexp -> string -> unit
  (* (sexpsToFile <sexp> <filename>) writes a string representation of <sexp>
     to the file name <filename>. *)

  val readSexp : unit -> sexp
  (* Reads lines from standard input until a complete s-expression has been
     found, and returns the sexp tree for this s-expresion. *)

end
```

Figure 8: The SEXP signature.

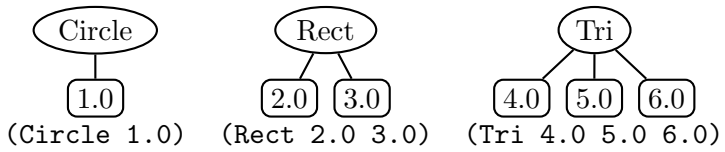
```
# Sexp.readSexp();;
(a b
   (c d e)
   (f (g h))
   i)
- : Sexp.sexp =
Sexp.Seq
 [Sexp.Sym "a"; Sexp.Sym "b";
  Sexp.Seq [Sexp.Sym "c"; Sexp.Sym "d"; Sexp.Sym "e"];
  Sexp.Seq [Sexp.Sym "f"; Sexp.Seq [Sexp.Sym "g"; Sexp.Sym "h"]];
  Sexp.Sym "i"]
```

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

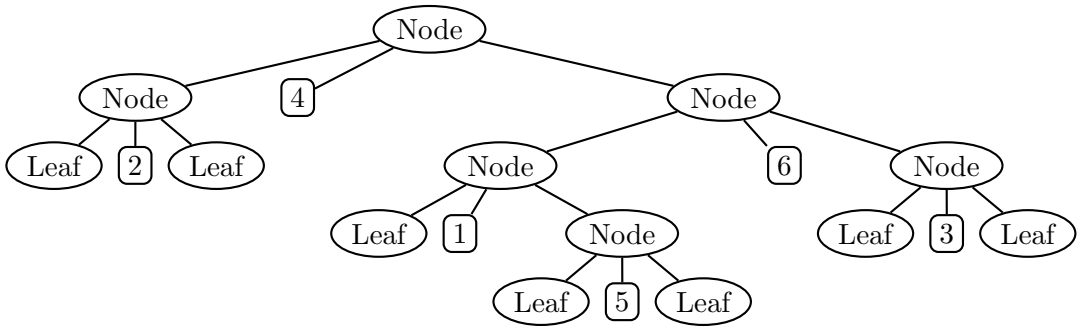
We will mainly use s-expressions for concisely representing the trees implied by sum-of-product data type constructor invocations. We will represent a tree node with tag *tag* and subtrees $t_1 \ldots t_n$ by an s-expression of the form:

(*tag* <s-expression for t_1> ... <s-expression for t_n>)

For instance, consider the following figure trees



And here is a binary tree example:



Using the above conventions, this would be represented via the s-expression:

```
(Node (Node (Leaf) 2 (Leaf))
      4
     (Node (Node (Leaf) 1 (Node (Leaf) 5 (Leaf)))
           6
           (Node (Leaf) 3 (Leaf))))  ; Let's call this the ''verbose'' notation
```

This is not a very compact representation! We can often develop more compact representations for particular data types. For instance, here are other s-expressions representing the binary tree above:

```
((* 2 *) 4 ((* 1 (* 5 *)) 6 (* 3 *))) ; The ''compact'' notation

((2) 4 ((1 (5)) 6 (3)))  ; The ''dense'' notation
```

Fig. 9 shows functions that convert between binary trees and these s-expression notations.

```
let rec toVerboseSexp eltToSexp tr =




let rec toCompactSexp eltToSexp tr =




let rec toDenseSexp eltToSexp s =




let rec fromVerboseSexp eltFromSexp sexp =




let rec fromCompactSexp eltFromSexp sexp =




let rec fromDenseSexp eltFromSexp sexp =
```

Figure 9: Functions for converting between binary trees and s-expressions.