CS109A Notes for Lecture 2/21/96

Type Shorthands

```
type ILL = int list list;
```
type ILL = int list list

```
fun ilhead(x::xs : ILL) = x;
```
val ilhead = fn : ILL → int list

- Type parameters can go after `type`.

```
type ('eltype) ELL =
    'eltype list list;
```
type 'a ELL = 'a list list

```
fun ilhead(x::xs : int ELL) = x;
```
val ilhead = fn : int ELL → int list

- Whole thing is no big deal, just a shorthand.
- But the following story, "datatypes," is a major deal.

Datatypes

We may define a new *datatype* T by specifying one or more *data constructors* for T.

- The values for T are prefix expressions that use the data constructors as operators and use operands of the appropriate type(s).
  - Operands may be values for T, or values of other types, depending on how T is defined.
- The declaration of a datatype consists of
  1. The keyword `datatype`.
  2. A parenthesized list of type parameters, as for type declarations.
  3. An = sign.
  4. A list of one or more *constructor expressions* separated by bars.
- A constructor expression consists of:

1. A constructor name, usually an indentifier beginning with a capital.

2. The keyword `of`.

3. A type expression, possibly involving the type parameters.

□ (2) and (3) are optional, but normal.

Example: The simplest examples look like enumerated types, e.g.

```
datatype buildingMaterials =
    Straw | Wood | Brick;
```
datatype buildingMaterials
con Straw
con Wood
con Brick

- Its values are nothing more than the 3 data constructors, e.g., `Straw`.

Example: Datatypes can simulate C's union types, but the values are each wrapped in an appropriate data constructor, to tell what kind it is.

```
datatype rori =
    Int of int |
    Real of real;
```
datatype rori
con Int : int → rori
con Real : real → rori

- Values of datatype `rori` include $Int(23)$, $Real(23.0)$, and $Real(2.34)$.

- Note the ML description of data constructors makes them look as if they were functions. That makes sense, since a data constructor does take values as "arguments" and produces a new value.

- Data constructors can appear naturally in patterns of functions.

```
fun getReal(Int(i)) = real(i)
|   getReal(Real(r)) = r;
```
val getReal = fn : rori → real

An Expression Type

Here is a datatype that defines expressions involving sets and the operators $\cup$ and $\cap$.

```
datatype 'elt Set =
    Union of 'elt Set * 'elt Set |
    Inter of 'elt Set * 'elt Set |
    Op of 'elt list;
datatype 'a Set
con Inter : 'a Set * 'a Set → 'a Set
con Op : 'a list → 'a Set
con Union : 'a Set * 'a Set → 'a Set
```

Values of the datatype `Set` may be thought of as expressions.

Basis: Set represented by the data constructor `Op` (operand) and a list of the elements of the set.

- Elements are of some type `'elt`, e.g., integers.

Induction: The data constructors `Union` and `Inter` take two set expressions as arguments to create the obvious expressions.

Example: The value of datatype `Set`:

```
val set1 = Union(Op([1,2,3]),
    Inter(Op([2,3,4]), Op([4,5,6])));
val set1 = Union (Op [1,2,3],Inter (Op #,Op #)) : int Set
```

represents the set-expression

$$\{1,2,3\} \cup (\{2,3,4\} \cap \{4,5,6\})$$

Here is a function that tests whether an element x is a member of the set denoted by some set expression.

```
fun member(x,Op(nil)) = false
|   member(x,Op(y::ys)) =
        if x=y then true
        else member(x,Op(ys))
|   member(x,Union(s,t)) =
        member(x,s) orelse member(x,t)
|   member(x,Inter(s,t)) =
        member(x,s) andalso member(x,t);
val member = fn : ''a * ''a Set → bool

member(4,set1);
val it = true : bool
```