**The Datatype Mechanism**

The `datatype` mechanism specifies new data types using value constructors.

For example,

```sml
datatype color = red|blue|green;
datatype color = blue | green | red
```

Pattern matching works too using the type's constructors:

```sml
fun translate red = "rot"
  | translate blue = "blau"
  | translate green = "gruen";
val translate = fn : color -> string
fun jumble red = blue
  | jumble blue = green
  | jumble green = red;
val jumble = fn : color -> color
translate (jumble green);
val it = "rot" : string
```

**SML Examples**

Source code for most of the SML examples presented here may be found in `~cs538-1/public/sml/class.sml`

**Parameterized Constructors**

The constructors used to define data types may be parameterized:

```sml
datatype money =
  none
  | coin of int
  | bill of int
  | iou of real * string;
datatype money =
  bill of int | coin of int
  | iou of real * string | none
```

Now expressions like `coin(25)` or `bill(5)` or `iou(10.25,"Lisa")` represent valid values of type `money`.

We can also define values and functions of type `money`:

```sml
val dime = coin(10);
val dime = coin 10 : money
val deadbeat = iou(25.00,"Homer Simpson");
val deadbeat = iou (25.0,"Homer Simpson") : money
fun amount(none) = 0.0
  | amount(coin(cents)) = real(cents)/100.0
  | amount(bill(dollars)) = real(dollars)
  | amount(iou(amt,_)) = 0.5*amt;
val amount = fn : money -> real
```
Polymorphic Datatypes

A user-defined data type may be polymorphic. An excellent example is

datatype 'a option = none | some of 'a;

datatype 'a option = none | some of 'a

val zilch = none;
val mucho = some(10e10);

val zilch = none : 'a option
val mucho = some 100000000000.0 : real option

type studentInfo = (name:string, ssNumber:int option);

type studentInfo = {name:string, ssNumber:int option}

Datatypes may be Recursive

Recursive datatypes allow linked structures without explicit pointers.

datatype binTree = null |
leaf |
ode of binTree * binTree;

datatype binTree = leaf |
ode of binTree * binTree |
ull

fun size(null) = 0 |
size(leaf) = 1 |
size(node(t1,t2)) = size(t1)+size(t2) + 1

val size = fn : binTree -> int

Recursive Datatypes may be Polymorphic

datatype 'a binTree = null |
leaf of 'a |
ode of 'a binTree * 'a binTree

datatype 'a binTree = leaf of 'a |
ode of 'a binTree * 'a binTree |
ull

fun frontier(null) = [] |ronter(leaf(v)) = [v] |ronter(node(t1,t2)) = frontier(t1) @ frontier(t2)

val frontier = fn : 'a binTree -> 'a list
We can model n-ary trees by using lists of subtrees:

```ml
datatype 'a Tree =
  null
| leaf of 'a
| node of 'a Tree list;
```

```ml
datatype 'a Tree = leaf of 'a | node of 'a Tree list | null
```

```ml
fun frontier(null) = []
| frontier(leaf(v)) = [v]
| frontier(node(h::t)) = frontier(h) @ frontier(node(t))
| frontier(node([])) = []
```

```ml
val frontier = fn : 'a Tree -> 'a list
```

---

**Abstract Data Types**

ML also provides abstract data types in which the implementation of the type is hidden from users. The general form is

```
abstype name = implementation
with
  val and fun definitions
end;
```

Users may access the name of the abstract type and the val and fun definitions that follow the with, but the implementation may be used only with the body of the abstype definition.

---

**Example**

```ml
abstype 'a stack =
  stk of 'a list
with
  val Null = stk([])
  fun empty(stk([])) = true
  | empty(stk(_::_)) = false
  fun top(stk(h:::_)) = h
  fun pop(stk(_::_)) = stk(t)
  fun push(v, stk(L)) = stk(v::L)
end
```

```ml
type 'a stack
val Null = - : 'a stack
val empty = fn : 'a stack -> bool
val top = fn : 'a stack -> 'a
val pop = fn : 'a stack -> 'a
val push = fn : 'a * 'a stack -> 'a stack
```
abstype 'a stack = 
  stk of 'a list 
with 
  local 
    fun size(stk(L))=length(L); 
in 
  val Null = stk([\]) 
  fun empty(s) = 
    (size(s) = 0) 
  fun top(stk(h::\_)) = h 
  fun pop(stk(_::t)) = stk(t) 
  fun push(v,stk(L)) = 
    stk(v::L) 
end 
end 

Why are abstract data types useful?
Because they hide an implementation of a type from a user, allowing implementation changes without any impact on user programs.
Consider a simple implementation of queues:
abstype 'a queue = 
  q of 'a list 
with 
  val Null = q(\[]) 
  fun front(q(h::\_)) = h 
  fun rm(q(_::t)) = q(t) 
  fun enter(v,q(L)) = 
    q(rev(v::rev(L))) 
end 

A more efficient (but less obvious) implementation of a queue is to store it as two lists. One list represents the “front” of the queue. It is from this list that we extract the front value, and from which we remove elements.
The other list represents the “back” of the queue (in reversed order). We add elements to the rear of the queue by adding elements to the front of the list. From time to time, when the front list becomes null, we “promote” the rear list into the front list (by reversing it). Now access to both the front and the back of the queue is fast and direct. The new implementation is:

val rm = 
  fn : 'a queue -> 'a queue 
val enter = 
  fn : 'a * 'a queue -> 'a queue 

This implementation of queues is valid, but somewhat inefficient. In particular to enter a new value onto the rear end of a queue, we do the following:
fun enter(v,q(L)) = 
  q(rev(v::rev(L)))

We reverse the list that implements the queue, add the new value to the head of the reversed queue then reverse the list a second time.
From the user’s point of view, the two implementations are identical (they export exactly the same set of values and functions). Hence the new implementation can replace the old implementation without any impact at all to the user (except, of course, performance!).

We see “match failure” since our definition of top is incomplete! In ML we can raise an exception if an illegal or unexpected operation occurs. Asking for the top of an empty stack ought to raise an exception since the requested value does not exist.

ML contains a number of predefined exceptions, including

\textbf{Match Empty Div Overflow}

(exception names usually begin with a capital letter).

Predefined exception are raised by illegal values or operations. If they are not caught, the run-time prints an error message.

Exception Handling

Our definitions of stacks and queues are incomplete. Reconsider our definition of stack:

abstype 'a stack =  
  stk of 'a list  
with  
  val Null = stk([])  
  fun empty(stk([])) = true  
  | empty(stk(_::_)) = false  
  fun top(stk(h:::_)) = h  
  fun pop(stk(_::_)) = stk(t)  
  fun push(v,stk(L)) =  
    stk(v::L)
end

What happens if we evaluate\top(Null);
fun f(1) = 2;
val f = fn : int -> int
f(2);
uncaught exception nonexhaustive
match failure
hd [];
uncaught exception Empty
1000000*1000000;
uncaught exception overflow
(1 div 0);
uncaught exception divide by zero
1.0/0.0;
val it = inf : real
(inf is the IEEE floating-point
standard “infinity” value)

User Defined Exceptions
New exceptions may be defined as
exception name;
or
exception name of type;
For example
exception IsZero;
exception IsZero
exception NegValue of real;
exception NegValue of real

Exceptions May be Raised
The raise statement raises
(throws) an exception:
raise exceptionName;
Or
raise exceptionName(expr);
For example
fun divide(a,0) = raise IsZero
| divide(a,b) = a div b;
val divide = 
fn : int * int -> int
divide(10,3);
val it = 3 : int
divide(10,0);
uncaught exception IsZero

val sqrt = Real.Math.sqrt;
val sqrt = fn : real -> real
fun sqroot(x) = 
if x < 0.0
then raise NegValue(x)
else sqrt(x);
val sqroot = fn : real -> real
sqroot(2.0);
val it = 1.41421356237 : real
sqroot(-2.0);
uncaught exception NegValue
**Exception Handlers**

You may catch an exception by defining a *handler* for it:

```plaintext
(expr) handle exception1 => val1
    || exception2 => val2
    || ... ;
```

For example,

```plaintext
(sqroot ~100.0)
    handle NegValue(v) =>
        (sqrt (~v));
val it = 10.0 : real
```