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
type 'a queue
val Null = - : 'a queue
val front = fn : 'a queue -> 'a
```



```
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.

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:


```
abstype 'a queue =
  q of 'a list * 'a list
with
  val Null = q([],[])
  fun front(q(h::_,_)) = h
   |  front(q([],L)) =
        front(q(rev(L),[]))
  fun rm(q(_::t,L)) = q(t,L)
   |  rm(q([],L)) =
        rm(q(rev(L),[]))
  fun enter(v,q(L1,L2)) =
     q(L1,v::L2)
end
type 'a queue
val Null = - : 'a queue
val front = fn :
  'a queue -> 'a
val rm = fn :
 'a queue -> 'a queue
val enter = fn :
 'a * 'a queue -> 'a queue
```

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!).

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(_::t)) = stk(t)
  fun push(v,stk(L)) =
   stk(v::L)
end
```

What happens if we evaluate

```
top(Null);
```


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

`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.


```
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:

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

For example,

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

Stacks Revisited

We can add an exception, **EmptyStk**, to our earlier stack type to handle **top** or **pop** operations on an empty stack:

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



```
type 'a stack
val Null = - : 'a stack
exception EmptyStk
val empty = fn : 'a stack -> bool
val top = fn : 'a stack -> 'a
val pop = fn :
 'a stack -> 'a stack
val push = fn : 'a * 'a stack ->
'a stack

pop(Null);
uncaught exception EmptyStk
top(Null) handle EmptyStk => 0;
val it = 0 : int
```


User-Defined Operators

SML allows users to define symbolic operators composed of non-alphanumeric characters. This means operator-like symbols can be created and used. Care must be taken to avoid predefined operators (like +, -, ^, @, etc.).

If we wish, we can redo our stack definition using symbols rather than identifiers. We might use the following symbols:

```
top    |=
pop    <==
push   ==>
null   <@>
empty  <?>
```


We can have expressions like

```
<?> <@>;
val it = true : bool
|= (==> (1,<@>));
val it = 1 : int
```

Binary functions, like ==> (push) are much more readable if they are infix. That is, we'd like to be able to write

```
1 ==> 2+3 ==> <@>
```

which pushes **2+3**, then **1** onto an empty stack.

To make a function (either identifier or symbolic) infix rather than prefix we use the definition

```
infix level name
```

or

```
infixr level name
```

`level` is an integer representing the “precedence” level of the infix operator. `0` is the lowest precedence level; higher precedence operators are applied before lower precedence operators (in the absence of explicit parentheses).

`infix` defines a left-associative operator (groups from left to right). `infixr` defines a right-associative operator (groups from right to left).

Thus

```
fun cat(L1,L2) = L1 @ L2;
infix 5 cat
```

makes `cat` a left associative infix operator at the same

precedence level as `@`. We can now write

```
[1,2] cat [3,4,5] cat [6,7];
val it = [1,2,3,4,5,6,7] : int list
```

The standard predefined operators have the following precedence levels:

Level	Operator
3	`o`
4	`=` `<>` `<` `>` `<=` `>=`
5	`::` `@`
6	`+` `-` `^`
7	`*` `/` `div` `mod`

If we define `==>` (push) as

```
infixr  2  ==>
```

then

```
1 ==> 2+3 ==> <@>
```

will work as expected, evaluating expressions like `2+3` before doing any pushes, with pushes done right to left.


```
abstype 'a stack =
  stk of 'a list
with
   val <@> = stk([])
   exception emptyStk
   fun <?>(stk([])) = true
    |   <?>(stk(_::_)) = false

   fun |=(stk(h::_)) = h
    |  |=(stk([])) =
           raise emptyStk

   fun <==(stk(_::t)) = stk(t)
   |  <==(stk([])) =
          raise emptyStk

   fun ==>(v,stk(L)) =
            stk(v::L)
   infixr 2 ==>
end
```

```
type 'a stack
val <@> = - : 'a stack
exception emptyStk
val <?> = fn : 'a stack -> bool
val |= = fn : 'a stack -> 'a
val <== = fn :
  'a stack -> 'a stack
val ==> = fn : 'a * 'a stack ->
'a stack
infixr 2 ==>
```

Now we can write

```
val myStack =
 1 ==> 2+3 ==> <@>;
val myStack = - : int stack
|= myStack;
val it = 1 : int
|= (<== myStack);
val it = 5 : int
```


Using Infix Operators as Values

Sometimes we simply want to use an infix operator as a symbol whose value is a function.

For example, given

```
fun dupl f v = f(v,v);
val dupl =
 fn : ('a * 'a -> 'b) -> 'a -> 'b
```

we might try the call

```
dupl ^ "abc";
```

This fails because SML tries to parse `dupl` and `"abc"` as the operands of `^`.

To pass an operator as an ordinary function value, we prefix it with `op` which tells the

SML compiler that the following symbol is an infix operator.

Thus

```
dupl op ^ "abc";
val it = "abcabc" : string
```

works fine.

The Case Expression

ML contains a `case` expression patterned on switch and case statements found in other languages.

As in function definitions, patterns are used to choose among a variety of values.

The general form of the `case` is

```
case expr of
 pattern1 => expr1|
 patternn => expr2|
   ...
 patternn => exprn;
```

If no pattern matches, a `Match` exception is thrown.

It is common to use _ (the wildcard) as the last pattern in a **case**.

Examples include

```
case c of
 red   => "rot" |
 blue  => "blau" |
 green => "gruen";

case pair of
 (1,_) => "win" |
 (2,_) => "place" |
 (3,_) => "show" |
 (_,_) => "loser";

case intOption of
 none => 0 |
 some(v)  => v;
```


Imperative Features of ML

ML provides references to heap locations that may be updated. This is essentially the same as access to heap objects via references (Java) or pointers (C and C++).

The expression

ref val

creates a reference to a heap location initialized to val. For example,

```
ref 0;
val it = ref 0 : int ref
```

The prefix operator ! fetches the value contained in a heap location (just as * dereferences a pointer in C or C++).

Thus

```
! (ref 0);
val it = 0 : int
```

The expression

ref := val

updates the heap location referenced by **ref** to contain **val**. The unit value, **()**, is returned.

Hence

```
val x = ref 0;
val x = ref 0 : int ref
!x;
val it = 0 : int
x:=1;
val it = () : unit
!x;
val it = 1 : int
```