Example

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

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 stack
val push = fn :
  'a * 'a stack -> 'a stack

Local value and function definitions, not to be exported to users of the type can be created using the local definition mechanism described earlier:

local
  val and fun definitions
in
  exported definitions
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

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:

\[
\text{abstype } 'a \text{ stack } = \\
\text{stk of } 'a \text{ list}
\]

\[
\begin{aligned}
\text{val Null } &= \text{stk([]} \\
\text{fun empty} &\text{(stk([])) } = \text{true} \\
\text{ | empty(stk(\_::\_)) } &= \text{false} \\
\text{fun top(stk(h::\_)) } &= \text{h} \\
\text{fun pop(stk(\_::t)) } &= \text{stk(t)} \\
\text{fun push}(v, \text{stk(L)}) &= \text{stk(v::L)} \\
\end{aligned}
\]

What happens if we evaluate \(\text{top(Null)}\);

We get a “match failure” because our definition of \(\text{top}\) is incomplete!

In ML we can \textit{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 \text{Match Empty Div Overflow} (exception names by convention begin with a capital letter).

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

User Defined Exceptions

New exceptions may be defined as

- exception name;
- exception name of type;

For example

\[
\begin{aligned}
\text{exception IsZero} &; \\
\text{exception IsZero} &; \\
\text{exception NegValue of real}; \\
\text{exception NegValue of real} \\
\end{aligned}
\]
**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,

```
(val sqrt ~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 choose the following symbols:

\[ \text{top} \quad |\text{=}\quad \text{pop} \quad <=\quad \text{push} \quad => \quad \text{null} \quad <@> \quad \text{empty} \quad <?> \]

Now we can have expressions like
\[ <?> <@>; \]
\[ \text{val it = true : bool} \]
\[ | = (==> (1,<@>)); \]
\[ \text{val it = 1 : int} \]

Binary functions, like \[ ==>((\text{push}) \]
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
\[ \text{infix level name} \]
or
\[ \text{infixr level name} \]

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
\[ \text{fun cat}(L1,L2) = L1 @ L2; \]
\[ \text{infix 5 cat} \]
makes \text{cat} a left associative infix operator at the same precedence level as @. We can now write
\[ [1,2] \text{ cat } [3,4,5] \text{ cat } [6,7]; \]
\[ \text{val it = [1,2,3,4,5,6,7] : int list} \]
The standard predefined operators have the following precedence levels:

<table>
<thead>
<tr>
<th>Level</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>o</td>
</tr>
<tr>
<td>4</td>
<td>=  &lt;&gt;  &lt;  &gt;  &lt;=  &gt;=</td>
</tr>
<tr>
<td>5</td>
<td>::  @</td>
</tr>
<tr>
<td>6</td>
<td>+  -  ^</td>
</tr>
<tr>
<td>7</td>
<td>*  /  div  mod</td>
</tr>
</tbody>
</table>

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.

```sml
abstype 'a stack =
  stk of 'a list

with
  | <@> = 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
```

```sml

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);
```

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.

```sml
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
```
Thus

dupl op ^ "abc";
val it = "abcabc" : string
works fine.