**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 choose the following symbols:

top  |=
pop  <==
push  ==> 
null  @@
empty  <?>
Now 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:

<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.
abstract '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 ==>
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

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

we might try the call

```plaintext
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
  pattern_1 => expr_1 |
  pattern_n => expr_2 |
  . . .
  pattern_n => expr_n;
```

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

\[
\text{ref val}
\]

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

\[
\text{ref 0};
\]

\[
\text{val it = ref 0 : int ref}
\]

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

Thus

\[
! (\text{ref 0});
\]

\[
\text{val it = 0 : int}
\]
The expression

\[ \text{ref} := \text{val} \]

updates the heap location referenced by \text{ref} to contain \text{val}. The unit value, \text{()}, is returned.

Hence

\begin{verbatim}
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
\end{verbatim}
Sequential Composition

Expressions or statements are sequenced using “;”. Hence

val a = (1+2; 3+4);
val a = 7 : int
(x:=1; !x);
val it = 1 : int

Iteration

while expr1 do expr2
implements iteration (and returns unit); Thus

(while false do 10);
val it = () : unit
while !x > 0 do x:= !x-1;
val it = () : unit
!x;
val it = 0 : int
Simple I/O

The function

\[
\text{print;}
\]

\[
\text{val it = fn : string -> unit}
\]

prints a string onto standard output. For example,

\[
\text{print("Hello World\n");}
\]

Hello World

The conversion routines

\[
\text{Real.toString;}
\]

\[
\text{val it = fn : real -> string}
\]

\[
\text{Int.toString;}
\]

\[
\text{val it = fn : int -> string}
\]

\[
\text{Bool.toString;}
\]

\[
\text{val it = fn : bool -> string}
\]

convert a value (real, int or bool) into a string. Unlike Java, the call must be explicit.
For example,

```plaintext
print (Int.toString (123));
123
```

Also available are

```plaintext
Real.fromString;
val it = fn : string -> real option
```

```plaintext
Int.fromString;
val it = fn : string -> int option
```

```plaintext
Bool.fromString;
val it = fn : string -> bool option
```

which convert from a string to a real or int or bool if possible. (That’s why the option type is used).
For example,

case (Int.fromString("123")) of
    SOME(i) => i | NONE => 0;

val it = 123 : int

case (Int.fromString("One two three")) of
    SOME(i) => i | NONE => 0;

val it = 0 : int
Text I/O

The structure TextIO contains a wide variety of I/O types, values and functions. You load these by entering:

open TextIO;

Among the values loaded are

- **type instream**
  This is the type that represents input text files.

- **type outstream**
  This is the type that represents output text files.

- **type vector = string**
  Makes vector a synonym for string.

- **type elem = char**
  Makes elem a synonym for char.
• val stdIn : instream
  val stdOut : outstream
  val stdErr : outstream
  Predefined input and output streams.

• val openIn :
  string -> instream
val openOut :
  string -> outstream
Open an input or output stream. For example,
val out =
  openOut("/tmp/test1");
val out = - : outstream

• val input :
  instream -> vector
Read a line of input into a string (vector is defined as equivalent to string). For example (user input is in red):
val s = input(stdIn);
Hello!
val s = "Hello!\n" : vector

• val inputN :
  instream * int -> vector
Read the next N input characters into a string. For example,
val t = inputN(stdIn,3);
abcde
val t = "abc" : vector

• val inputAll :
  instream -> vector
Read the rest of the input file into a string (with newlines separating lines). For example,
val u = inputAll(stdIn);
Four score and
seven years ago . . .
val u = "Four score and\nseven years ago . . .\n" : vector
• val endOfStream : instream -> bool
  Are we at the end of this input stream?

• val output : outstream * vector -> unit
  Output a string on the specified output stream. For example,
  output (stdOut, "That’s all folks!\n");
  That’s all folks!
**String Operations**

ML provides a wide variety of string manipulation routines. Included are:

- The string concatenation operator, `^`
  
  "abc" ^ "def" = "abcdef"

- The standard 6 relational operators:
  
  `< > <= >= = <>`

- The string size operator:
  
  ```
  val size : string -> int
  size ("abcd");
  val it = 4 : int
  ```

- The string subscripting operator (indexing from 0):
  
  ```
  val sub =
    fn : string * int -> char
  sub("abcde",2);
  val it = #"c" : char
  ```
• The substring function
  
  val substring : string * int * int -> string
  This function is called as
  substring(string, start, len)
  start is the starting position, counting from 0.
  len is the length of the desired substring. For example,
  substring("abcdefgij",3,4)
  val it = "defg" : string

• Concatenation of a list of strings into a single string:
  concat :
    string list -> string
  For example,
  concat ["What’s"," up","?" ];
  val it = "What’s up?" : string
• Convert a character into a string:
   \[ str : \text{char} \rightarrow \text{string} \]
   For example,
   \[ str(#"x"); \]
   \[ \text{val it = } "x" : \text{string} \]

• “Explode” a string into a list of characters:
   \[ \text{explode} : \text{string} \rightarrow \text{char list} \]
   For example,
   \[ \text{explode("abcde");} \]
   \[ \text{val it = } ["a","b","c","d","e"] : \text{char list} \]

• “Implode” a list of characters into a string.
   \[ \text{implode} : \text{char list} \rightarrow \text{string} \]
   For example,
   \[ \text{implode ["a","b","c","d","e"];} \]
   \[ \text{val it = } "abcde" : \text{string} \]