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.
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

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

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 |
  pattern2 => 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
```
Sequential Composition

Expressions or statements are sequenced using “;”. Hence

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

Iteration

```plaintext
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

```plaintext
print;
val it = fn : string -> unit
```

prints a string onto standard output. For example,

```plaintext
print("Hello World\n");
Hello World
```

The conversion routines

```plaintext
Real.toString;
val it = fn : real -> string
Int.toString;
val it = fn : int -> string
Bool.toString;
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
Int.fromString;
val it = fn : string -> int option
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,

```plaintext
case (Int.fromString("123")) of
  SOME(i) => i | NONE => 0;
val it = 123 : int
```

```plaintext
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**
  Predefined input stream.

- **val stdOut : outstream**
  Predefined output stream.

- **val stdErr : outstream**
  Predefined error stream.

- **val openIn : string -> instream**
  Open an input stream.

- **val openOut : string -> outstream**
  Open an output stream.

- **val input : instream -> vector**
  Read a line of input into a string.

- **val inputN : instream * int -> vector**
  Read the next N input characters into a string.

- **val inputAll : instream -> vector**
  Read the rest of the input file into a string.

- **val endOfStream : instream -> bool**
  Are we at the end of this input stream?

- **val output : outstream * vector -> unit**
  Output a string.


```plaintext
val s = input(stdIn);
Hello!
val s = "Hello!\n": vector

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

val inputAll : instream -> vector
Read the rest of the input file into a string.
val u = inputAll(stdIn);
Four score and seven years ago ...
val u = "Four score and\nseven years ago ...
": vector

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.
For example,
val u = inputAll(stdIn);
Four score and seven years ago ...
val u = "Four score and\nseven years ago ...
": vector

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.
For example,
val u = inputAll(stdIn);
Four score and seven years ago ...
val u = "Four score and\nseven years ago ...
": vector
```