Standard ML of New Jersey

—

Base Environment

(Version 0.93)

February 15, 1993

Contents

NAME

Intro — Introduction to the base environment

DESCRIPTION

This chapter describes the *base environment* that is built in to the **sml** executable, also known as the *initial*, *pervasive*, or *standard* environment. None of the structures, types, values, *etc.* contained in this environment need to be loaded from external source files.

Some structures, such as **General**, **List**, and **Integer** are opened in the base environment, so that it is not necessary to prefix their components by a structure name. Thus, **map** and **List.map** both refer to the same function in the base environment. The latter might be used to refer to list mapping if the identifier **map** were rebound locally. In this manual, the **SYNOPSIS** of such a structure will mention that it is pre-opened.

Other structures, such as **Array** and **Vector**, are not pre-opened. To use functions and types from these structures, one must either **open** the structure, or qualify references to components with the name of the structure.

Infix operators

Infix declarations cannot be exported from structures or specified in signatures. The following fixities are predefined in the base environment:

```
  infix 0 before
  infix 3 o
  infix 4 = <> > < >= <=
  infixr 5 :: @
  infix 6 + - ^
  infix 7 * / div mod quot rem
```

When infix declarations occur in the **SIGNATURE** sections of this manual it is just a reminder that the indicated symbol is infix in the base environment.

Overloaded operators

Standard ML does not allow user-defined overloading. The following operaters are overloaded in the base environment:

```
makestring : ('a -> string) as Bool.makestring and Real.makestring
                            and Integer.makestring
print : ('a -> unit)   as Bool.print and Integer.print
                       and Real.print and String.print
~ :   ('a -> 'a)        as Integer.~   and Real.~
+ :   ('a * 'a -> 'a)   as Integer.+   and Real.+
- :   ('a * 'a -> 'a)   as Integer.-   and Real.-
* :   ('a * 'a -> 'a)   as Integer.*   and Real.*
< :   ('a * 'a -> bool) as Integer.<   and Real.<  and String.<
> :   ('a * 'a -> bool) as Integer.>   and Real.>  and String.>
<= :  ('a * 'a -> bool) as Integer.<=  and Real.<= and String.<=
>= :  ('a * 'a -> bool) as Integer.>=  and Real.>= and String.>=
abs : ('a -> 'a)        as Integer.abs and Real.abs
```

In the scope of a redeclaration of such an operator, the operator is not overloaded. Thus

```
let val (op +) = fn (s,t) => s^t in "Hello, " + "World" end
```

is legal, but

```
let val (op +) = fn (s,t) => s^t in 3+4 end
```

is not. Hence these operators will cease to be overloaded if they are rebound at top level

NAME

Array — updateable constant-time indexable sequences

SYNOPSIS

signature ARRAY
structure Array : ARRAY

SIGNATURE

```
eqtype 'a array
exception Size
exception Subscript
val array: int * '_a -> '_a array
val arrayoflist: '_a list -> '_a array
val tabulate: int * (int -> '_a) -> '_a array
val sub: 'a array * int -> 'a
val update: 'a array * int * 'a -> unit
val length: 'a array -> int
```

DESCRIPTION

The `Array` structure provides one-dimensional, zero-based, updateable arrays.

array (n, v)
create an n-element, zero-based array with each element initialized to v. Raises `Size` if $n < 0$ or if $n \geq 2^{25}$.

arrayoflist l
create an array whose elements are initialized to the elements of l.

tabulate (n, f)
create an n element array whose ith element is initialized to $f(i)$.

sub (a, i)
extract (subscript) the ith element of array a. Raises `Subscript` if $i < 0$ or $i \geq \text{length}(a)$. It is conventional, if `sub` is declared infix, to use precedence 9. However, `sub` is not infix in the base environment.

update (a, i, v)
replace the ith element of a by the value v. Raises `Subscript` if $i < 0$ or $i \geq \text{length}(a)$.

length a
the number of elements in the array a.

Type α `array` is an equality type even if α is not. Thus, the `eqtype` specification in the signature ARRAY does not quite capture the equality semantics of arrays.

All zero-length arrays are equal to each other. Nonzero-length arrays a and b, created by different calls to `array`, are always unequal, even if their elements are equal.

SEE ALSO

Array2(LIB), ByteArray(BASE), RealArray(BASE), Vector(BASE)

CAVEATS

The signature ARRAY should not be used to constrain the Array structure because this will prevent some operations from being coded inline (this is a bug).

NAME

Bits — bitwise logical operations on integers

SYNOPSIS

signature BITS
structure Bits : BITS

SIGNATURE

```
type int
val orb : int * int -> int
val andb : int * int -> int
val xorb : int * int -> int
val lshift : int * int -> int
val rshift : int * int -> int
val notb : int * int -> int
```

DESCRIPTION

The `Bits` structure provides bitwise logical operations on ordinary integers. In version 0.93 of SML/NJ, these are 31-bit two's complement binary numbers.

int
: the integer type, identical to Integer.int.

orb (i, j)
: the bitwise inclusive-or (union) of i and j.

andb (i, j)
: the bitwise and (intersection) of i and j.

xorb (i, j)
: the bitwise exclusive-or of i and j.

not i
: the bitwise complement of i.

lshift (i, j)
: i shifted left by j bits. Zero bits are shifted from the right.

rshift (i, j)
: i shifted right by j bits. If i is negative, the bits shifted from the left are unspecified.

NAME

Bool — Boolean type and operations upon it

SYNOPSIS

signature BOOL
structure Bool : BOOL
Opened in the initial environment of SML/NJ.

SIGNATURE

```
datatype bool = true | false
datatype 'a option = NONE | SOME of 'a
val not: bool -> bool
val print: bool -> unit
val makestring: bool -> string
```

DESCRIPTION

bool
the Boolean type.

α option
a generally useful data type.

not x
the logical negation of x.

print x
write `true or false` on the standard output.

makestring x
the string `true` or `false`.

CAVEATS

The `option` datatype doesn't belong here, and will eventually move somewhere else.

NAME

ByteArray — compact arrays of bytes

SYNOPSIS

signature BYTEARRAY
structure ByteArray : BYTEARRAY

SIGNATURE

```
eqtype bytearray
exception Subscript
exception Range
exception Size
val array : int * int -> bytearray
val sub : bytearray * int -> int
val update : bytearray * int * int -> unit
val length : bytearray -> int
val extract : bytearray * int * int -> string
val fold : ((int * 'b) -> 'b) -> bytearray -> 'b -> 'b
val revfold : ((int * 'b) -> 'b) -> bytearray -> 'b -> 'b
val app : (int -> 'a) -> bytearray -> unit
val revapp : (int -> 'b) -> bytearray -> unit
```

DESCRIPTION

Byte arrays are just like arrays of integers (with zero-based indexing, of course), with the restriction that the values of the component integers must be between 0 and 255. They are represented much more compactly than ordinary arrays: each element occupies one byte of storage space.

Subscript identical to String.Ord but distinct from Array.Subscript.

array (n, v)
: create an n-element, zero-based byte array array with each element initialized to v. Raises `Size` if $n < 0$ or if $n \geq 2^{25}$. Raises `Range` if $v < 0$ or $v \geq 256$.

sub (a, i)
: extract (subscript) the ith element of array a. Raises `Subscript` if $i < 0$ or $i \geq \text{length}(a)$.

update (a, i, v)
: replace the ith element of a by the value v. Raises `Subscript` if $i < 0$ or $i \geq \text{length}(a)$.

length a
: the number of elements in the array a.

extract (a, i, n)
: a string containing characters $a_i, a_{i+1}, \ldots a_{i+n-1}$. Raises `Subscript` if $i < 0$ or $i + n > \text{length}(a)$ or $n < 0$.

fold $f\ a\ z$
: $f(a_0, f(a_1, \ldots f(a_{|a|-1}, z) \ldots))$.

revfold $f\ a\ z$
: $f(\ldots f(f(z, a_0), a_1), \ldots, a_{|a|-1})$.

app $f\ a$
: $(f(a_0); f(a_1); \ldots f(a_{|a|-1}); ())$.

revapp $f\ a$
: $(f(a_{|a|-1}); f(a_{|a|-2}); \ldots f(a_0); ())$.

Two byte arrays are equal if and only if they are both of length 0 or they were created by the same call to `ByteArray.array`.

SEE ALSO

Array(BASE)

CAVEATS

The signature BYTEARRAY should not be used to constrain the ByteArray structure because this will prevent some operations from being coded inline (this is a bug).

NAME

callcc, throw — call with current continuation

SYNOPSIS

signature GENERAL
structure General : GENERAL
Opened in the initial environment of SML/NJ.

SIGNATURE

```
. . .
type 'a cont
val callcc : ('1a cont -> '1a) -> '1a
val throw : 'a cont -> 'a -> 'b
```

DESCRIPTION

First class continuations are created with `callcc` and invoked with `throw`.

α **cont**

the type of continuations accepting arguments of type α.

callcc *f*

Apply f to the "current continuation". If f invokes this continuation with argument x, it is as if (`callcc` f) had returned x as a result.

throw *k x*

Invoke continuation k with argument x.

The continuation of an expression is an abstraction of what the system will do with the value of the expression. For example in the expression:

```
if a orelse b then foo() else goo()
```

the continuation of the expression "a orelse b" can be described in words as "if the value is true then compute `foo()` otherwise compute `goo()`" and then continue in the context of the `if` expression. Usually the continuation of an expression is implicit, however, the primitive `callcc` allows the programmer to capture and use these continuations.

The primitive `callcc` takes a function as an argument and applies it to the current continuation. The continuation of an expression of type α has type α cont and is a first-class object. To capture the continuation described above one would write:

```
if callcc(fn k => a orelse b) then foo() else goo()
```

Here the continuation of the `callcc` application is captured by being bound to k, but it is not used. Because the continuation is not used the result is simply the result of the expression "`a orelse b`." To use the continuation a value must be supplied, and the computation continues as if that value where the result of the `callcc` application. This is called throwing the continuation a value; it is performed by applying `throw` to the continuation and the value.

```
if callcc(fn k => (throw k false) orelse b)
   then foo() else goo()
```

Here, when the continuation k is thrown the value false, "`orelse b`" is simply ignored, the `callcc` application returns false, and `goo`() is then evaluated.

The type returned by a `throw` expression is unconstrained like that of a raise-expression and for the same reason: neither of these expressions ever return.

The continuation captured by `callcc` also contains the exception context of the expression.

REFERENCES

Robert Harper, Bruce M. Duba, David B. MacQueen, "Typing first-class continuations in ML," *Journal of Functional Programming*, (to appear) 1993.

John H. Reppy, "First-class synchronous operations in Standard ML," TR 89-1068, Department of Computer Science, Cornell University, December 1989. *(An example of the use of* **callcc** *in ML.)*

NAME

exportML, exportFn — create executable ML files

SYNOPSIS

signature IO
structure IO : IO
Opened in the initial environment of SML/NJ.

SIGNATURE

```
. . .
val exportML : string -> bool
val exportFn : string * (string list * string list -> unit)
                -> unit
```

DESCRIPTION

exportML s

export the current executing image into file s. Return $false$. When s is executed, exportML returns $true$ and execution continues exactly where it left off, except that all files have been closed (and `std_in`, `std_out`, `std_err` have been re-opened in the new process). Access to command-line arguments and Unix environment variables when running the saved image may be accomplished by System.argv and System.environ.

exportFn (s, f)

Create an executable file s that contains the function f and as little else as possible. When s is executed, then $f(argv,\ environ)$ is executed, where $argv$ is a list of operating-system-provided arguments (starting with the name of the executable itself), and $environ$ is the operating-system-provided environment (a list of strings).

The compiler is not accessible in the exported file; thus, `use` (and related functions) will fail if called by f. This makes the exported file several megabytes smaller, *provided that exportFn is run from within a "noshare" version of* **sml**. In many installations this will be named **sml-noshare**; see the installation guide (release-notes) or contact your system administrator if you don't have an **sml-noshare**. An alternative is to export from a "*pervshare*" version of **sml** that only incorporates the code for the base (aka pervasive) environment in the text segment.

EXAMPLE

```
if exportML("saved")
then print "this is the saved image\n"
else print "this is the original process\n"
```

NAME

General — general-purpose predeclared identifiers

SYNOPSIS

signature GENERAL
structure General : GENERAL
Opened in the initial environment of SML/NJ.

SIGNATURE

```
exception Bind
exception Match
exception Interrupt (* never raised; see text *)
exception Fail of string

infix 3 o
val o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)

infix before
val before : ('a * 'b) -> 'a

type 'a cont
val callcc : ('1a cont -> '1a) -> '1a
val throw : 'a cont -> 'a -> 'b

datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a

type exn
type unit
infix 4 = <>
val = : ''a * ''a -> bool
val <> : ''a * ''a -> bool
```

DESCRIPTION

The `General` structure provides general purpose standard identifiers. The infix directives are in the base environment but not the structure `General` or the signature `GENERAL`.

Bind

raised for nonexhaustive **val** bindings whose pattern fails to match

Match

raised for nonexhaustive **case**, **fn**, and **fun** matches that fail to match their argument. (Exception handlers—that is, matches after the keyword `handle`—simply re-raise their argument if the match fails.)

Interrupt

never raised; it is provided for compatibility with *The Definition of Standard ML*. Pressing control-C raises a signal, not an exception, in SML/NJ; see section SIGNAL(SYS).

Fail

The top-level loop recognizes this exception and prints the message string.

o

the infix function-composition operator.

before

This infix operator evaluates both of its arguments and returns the first one; useful in sequencing side effects. (“after” might have been a better name.)

α cont
the type of continuations accepting α. See section callcc(BASE).

callcc
call with current continuation; see section callcc(BASE).

throw
throw to a continuation; see section callcc(BASE).

frag
a datatype used for the *quote/antiquote* mechanism. See section Quote(BASE).

exn
the type of all exception values.

unit
a type containing exactly one value, which is denoted `()`.

=
the polymorphic equality function.

<>
the polymorphic inequality function.

Many of these functions are described in *The Definition of Standard ML* or in any textbook on Standard ML.

SEE ALSO
Quote(BASE), callcc(BASE)

NAME

Integer — fixed-precision integers and operations upon them

SYNOPSIS

signature INTEGER
structure Integer : INTEGER
Opened in the initial environment of SML/NJ.

SIGNATURE

```
 infix 7 * div mod quot rem
 infix 6 + -
 infix 4 > < >= <=
exception Sum and Diff and Prod and Neg and Quot and Abs
exception Div and Mod
exception Overflow
type int
val ~ : int -> int
val * : int * int -> int
val div : int * int -> int
val mod : int * int -> int
val quot : int * int -> int
val rem : int * int -> int
val + : int * int -> int
val - : int * int -> int
val >  : int * int -> bool
val >= : int * int -> bool
val <  : int * int -> bool
val <= : int * int -> bool
val min : int * int -> int
val max : int * int -> int
val abs : int -> int
val print : int -> unit
val makestring : int -> string
```

DESCRIPTION

The `Integer` structure provides fixed precision integers. In current implementations these are 31-bit integers, ranging from -2^{30} to $2^{30}-1$.

int

the integer type.

Overflow

raised by various arithmetic operators (`+`, `-`, `*`, `div`, `~`, etc.) if the result is too large to be representable.

Div

raised upon division by zero.

Mod

raised upon mod by zero. This exception is simply a renaming of the Div exception.

Sum, Diff, Prod, Neg, Quot, Abs

synonyms for Overflow, for semi-compatibility with *The Definition of Standard ML*.

`~` i

the negation of i.

i `*` j
: the product of i and j.

i `+` j
: the sum of i and j.

i `-` j
: the difference of i and j.

i `div` j
: the quotient of i and j, *rounded toward negative infinity.*

i `quot` j
: the quotient of i and j, *rounded toward zero.*

i `mod` j
: a quantity k, such that $0 \leq k < j$ or $j < k \leq 0$, and $j(i \text{ div } j) + k = i$.

i `rem` j
: a quantity k, such that $0 \leq k < |j|$ and $j(i \text{ quot } j) + k = i$.

i `>` j
: i greater than j.

i `>=` j
: i greater than or equal to j.

i `<` j
: i less than j.

i `<=` j
: i less than or equal to j.

min (i,j)
: the minimum of i and j.

max (i,j)
: the maximum of i and j.

abs i
: the absolute value of i.

print i
: print the decimal representation of i on the standard output. Overloaded; see INTRO(BASE).

makestring i
: the ASCII decimal representation of i. Overloaded; see INTRO(BASE).

CAVEATS

The signature INTEGER should not be used to constrain the Integer structure because this will prevent some operations from being coded inline (this is a bug).

NAME

IO — Input/Output structure

SYNOPSIS

signature IO
structure IO : IO
Opened in the initial environment of SML/NJ.

SIGNATURE

```
type instream
type outstream
exception Io of string
val std_in : instream
val std_out : outstream
val std_err : outstream
val open_in : string -> instream
val open_out : string -> outstream
val open_append : string -> outstream
val open_string : string -> instream
val close_in : instream -> unit
val close_out : outstream -> unit
val output : outstream * string -> unit
val outputc : outstream -> string -> unit
val input : instream * int -> string
val inputc : instream -> int -> string
val input_line : instream -> string
val lookahead : instream -> string
val end_of_stream : instream -> bool
val can_input : instream -> int
val flush_out : outstream -> unit
val is_term_in : instream -> bool
val is_term_out : outstream -> bool
val set_term_in : instream * bool -> unit
val set_term_out : outstream * bool -> unit
val execute : string * string list -> instream * outstream
val execute_in_env : string * string list * string list
                         -> instream * outstream
val exportML : string -> bool
val exportFn : string * (string list * string list -> unit) -> unit
```

DESCRIPTION

The `IO` structure provides buffered input and output streams, and operations upon them.

instream
the type of input streams

outstream
the type of output streams

Io *s*
the Io exception with message *s*. *s* is a string with format "*command* `"`*filename*`":` *syscall* `failed,` *message*" where *command* is the ML IO function that failed, *filename* is the name under which the instream or outstream was opened, *syscall* is the low-level system call that failed, and *message* is the operating system's explanation of the problem.

std_in
the standard input stream

std_out
the standard output stream

std_err
the standard error (output) stream

open_in *s*
opens the file named *s* for reading and returns an instream.

open_out *s*
opens the file named *s* for writing and returns an outstream.

open_append *s*
opens the file named *s* for appending (writing at the end) and returns an outstream.

open_string *s*
make an instream whose contents are the string *s*.

close_in *f*
close the instream *f*.

close_out *f*
close the outstream *f*, flushing the output buffer.

output (f, s)
write characters *s* to outstream *f*. The characters may not immediately appear to the operating system because of internal buffering. Writing a newline character to an interactive outstream flushes the buffer, however.

outputc *f s*
equivalent to output(f, s).

input (f, n)
read *n* characters from an input stream *f*. If fewer than *n* characters remain before end of file, return them. Otherwise block until either *n* characters are available or end of file has been reached. Raises `Io` if *f* has been closed.

inputc *f n*
Not equivalent to input(f, n)! If end of file has been reached, return the empty string. Otherwise block until *at least one* character is available. Then return a string with at least one, and no more than *n* characters read from the stream; *not necessarily all the available characters.* **Note:** `inputc` is both more efficient, and has more useful semantics, than `input`. Raises `Io` if *f* has been closed.

input_line *f*
return all characters up through and including either the first newline or the end of stream, whichever comes first. It blocks until either a newline or the end of stream is reached.

lookahead *f*
yield the next character from *f* without removing it from the readable input; or the empty string if at end of file. Blocks if no character available but not at end of file.

end_of_stream *f*
true if at end of stream *f*.

can_input *f*
the number of characters available to input from *f* without blocking.

flush_out f
ensure that all characters that have been output to f appear to the operating system.

is_term_in f
true iff f is an interactive stream, usually. This makes no difference to the buffered I/O system, but is provided as a service to clients.

is_term_out f
true iff f is an interactive stream, usually. Interactive streams are flushed at each newline.

set_term_in f
make f appear to be an interactive stream.

set_term_out f
make f appear to be an interactive stream.

execute (c, l)
return the pair (f, g); execute a Unix command c with arguments l its output piped to instream f and its input piped from outstream g. The argument list l should not include the command name.

execute_in_env (c, l, e)
like `execute` but with an environment argument e.

exportML s
see section exportML(BASE).

exportFn (s, f)
see section exportML(BASE).

use s
Compile and execute the ML declarations and expressions in the file named s. For best results, use `use` or `use_stream` only at top level, or at top level within a `use`d file.

use_stream f
Compile and execute the ML declarations and expressions in the already-opened instream f.

CAVEATS

The argument of the `Io` exception should be structured. But this would lose compatibility with the *Definition*.

SEE ALSO

SysIO(SYS), Format(LIB), ListFormat(LIB), Makestring(LIB), StringCvt(LIB)

NAME

List — operations on lists

SYNOPSIS

signature LIST
structure List : LIST
Opened in the initial environment of SML/NJ.

SIGNATURE

```
infixr 5 :: @
datatype 'a list = :: of ('a * 'a list) | nil
exception Hd
exception Tl
exception Nth
exception NthTail
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val null : 'a list -> bool
val length : 'a list -> int
val @ : 'a list * 'a list -> 'a list
val rev : 'a list -> 'a list
val map :  ('a -> 'b) -> 'a list -> 'b list
val fold : (('a * 'b) -> 'b) -> 'a list -> 'b -> 'b
val revfold : (('a * 'b) -> 'b) -> 'a list -> 'b -> 'b
val app : ('a -> 'b) -> 'a list -> unit
val revapp : ('a -> 'b) -> 'a list -> unit
val nth : 'a list * int -> 'a
val nthtail : 'a list * int -> 'a list
val exists : ('a -> bool) -> 'a list -> bool
```

DESCRIPTION

The `List` structure provides the `list` datatype and operations upon it.

a :: *b*
: The list with head a and tail b.

nil
: the empty list.

`[`$a_1, \ldots, a_n$`]` an n-element list expression or list pattern, for $n \geq 0$.

Hd
: raised by `hd(nil)`.

Tl
: raised by `tl(nil)`.

Nth
: raised by `nth`(l, k), where $k \geq \text{length}(l)$.

NthTail
: raised by `nth`(l, k), where $k > \text{length}(l)$.

hd *l*
: the head (first element) of l.

tl l
: the tail (all elements after the first) of l.

null l
: true if $l =$ `nil`.

length l
: number of elements in l.

a `@` b
: the concatenation of lists a and b. Pronounced "append." Copies list a.

rev l
: the reverse of list l.

map f l
: the list of results obtained by applying f to each element of l.

fold f l z
: Fold a binary operator f over list l, with "identity" z, associating from right to left. In APL, this is called "reduce:" $f(l_0, f(l_1, \ldots f(l_{|l|-1}, z) \ldots))$.

revfold f l z
: Like fold, but associates from left to right; note that the list elements are used as left-hand arguments to f, not right-hand: $f(\ldots f(f(l_0, z), l_1), \ldots, l_{|l|-1})$.

app f l
: Apply f to each element of l, from beginning to end, discarding the results:
$(f(l_0); f(l_1); \ldots f(l_{|l|-1}); ())$.

revapp f l
: Apply f to each element of l, from end to beginning, discarding the results:
$(f(l_{|l|-1}); f(l_{n-2}); \ldots f(l_0); ())$.

nth(l,n)
: the nth element of l, counting from 0.

nthtail(l,n)
: the result of dropping the first n elements of l: $\text{nthtail}(l, 0) = l$.

exists p l
: true iff $p(l_i)$ is true for some element l_i of l.

CAVEATS

These functions are ill-chosen, and their arguments are badly arranged in some cases.

SEE ALSO

ListFormat(LIB), ListMergeSort(LIB), ListUtil(LIB)

NAME

Quote/Antiquote — object language embedding

SYNOPSIS

signature GENERAL
structure General : GENERAL
Opened in the initial environment of SML/NJ.

System.Control.quotation := true;

SIGNATURE

```
  . . .
datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a
```

DESCRIPTION

Standard ML is often used to implement another language L, *e.g.*, the syntax of the HOL logic in hol90 or the syntax of CCS for the Concurrency Workbench. Typically, one defines the abstract syntax of L by a `datatype` declaration. Then useful functions over the datatype can be defined (such as finding the free variables of a formula when L is a logic). Soon afterwards, one concludes that concrete syntax is easier for humans to read than abstract syntax, and so writes a parser and prettyprinter for L.

In the situation just outlined, Standard ML is called the *metalanguage*, and L is called the *object language*, or OL.[1] The purpose of a quotation/antiquotation mechanism is to allow one to embed expressions in the object language's concrete syntax inside of ML programs, and to mix the object language expressions with ML expressions.

Quotation and Antiquotation The quote/antiquote mechanism is enabled by setting

```
System.Control.quotation : bool ref
```

to true. Then the backquote character ceases to be legal in symbolic identifiers, and takes on a special meaning.

A *quotation* is a special form of literal expression that represents the concrete syntax of an OL phrase. The backquote character (`) is used to delimit quotations.

For a running example, suppose our OL is a simple propositional logic with propositions represented as values of abstract type `prop`. We might wish to write propositional expressions such as `A/\B/\C`.

The most common approximation to quotation is strings. This is not pleasant at times, especially when dealing with backslashes and newlines. Still, strings are bearable. Strings are not adequate, however, for the following idea.

The ML-OL relationship invites a notion of *antiquotation*: the temporary abandonment of parsing so that an ML value can be spliced into the middle of a quotation. Operations like this have cropped up under various names in various places: antiquote is due to Milner; Quine had a version called *quasi-quotation* in his 1940 book; Carnap used a notation much like it. It also closely resembles the Lisp *backquote* facility.

Using backquote, we write

[1] Edinburgh/INRIA/Cambridge ML, the precursor to Standard ML, was originally a programming metalanguage for a particular object language, the LCF logic.

```
- val f = `A /\ B \/ C`;
val f : 'a frag = [QUOTE "A /\\ B /\\ C"] : 'a frag list
```

More commonly, we invoke an OL parser to parse, enforce precedence, etc. By naming the parser something concise, such as `%`, we can use the syntax

```
- val % = my_proposition_parser;
val % : prop frag -> prop
- val p = %`A /\ B \/ C`;
val p = -: prop
```

An antiquote is written as a caret (`^`) followed by either an SML identifier or a parenthesized SML expression. Antiquotation can be used to conveniently express *contexts*, which are often used as a descriptive tool for syntax. A context could be defined as a function taking a `prop` and directly placing it at a location in a quotation.

```
- fun foo a = %`^a ==> A`;
val foo : prop -> prop
```

In this case, `foo p` would denote the same proposition as

```
%`(A /\ B \/ C) ==> A`
```

Antiquotations can have nested quotations (which may contain antiquotes of their own, etc.):

```
- let val K x y = x
      val I x = x
  in
  %`A /\ ^(K (%`B`) (I (%`C`)))
      \/ C`
  end;
```

gives the same `prop` as that denoted by `p`. We note in passing that the power of the OL parser is completely up to its author: for example, in the framework offered here, one could write an OL "parser" for Scheme that parses program plus arguments, evaluates the program on the arguments, and finally prints the returned value.

Implementation of OL parsers A concrete syntax quotation is mapped by the SML compiler into a `frag list`. Intuitively, a `frag` is a contiguous part of a quotation: `` `A /\ B` `` maps to `[ QUOTE "A /\\ B" ]` while `` `^x /\ ^y` `` maps to

```
[QUOTE "",ANTIQUOTE x, QUOTE "/\\", ANTIQUOTE y, QUOTE ""]
```

In this approach, the value of a quotation has type `ol frag list` where `ol` is the type of object language expressions; the type of the OL parser is `ol frag list -> ol`.

The OL parser (in our example, `%`) must handle these lists and insert the antiquoted ML values in the right places.

CAVEATS

Uses whatever `QUOTE` and `ANTIQUOTE` constructors happen to be in scope. This bug may be fixed some day.

Often one wants to parse stratified languages, such as first order logic, or typed lambda calculus, which requires a trick. Also, there is a bit of trickery when one wants to deal with ML-Yacc and ML-Lex, especially when functorizing the parser.

SEE ALSO

PrettyPrint(SYS)

NAME

Real — floating-point numbers and operations thereupon

SYNOPSIS

signature REAL
structure Real : REAL
Opened in the initial environment of SML/NJ.

SIGNATURE

```
infix 7 * /
infix 6 + -
infix 4 > < >= <=
type real
exception Overflow and Sum and Diff and Prod and Floor and Exp
exception Sqrt and Ln
exception Div
val ~ : real -> real
val + : (real * real) -> real
val - : (real * real) -> real
val * : (real * real) -> real
val / : (real * real) -> real
val > : (real * real) -> bool
val < : (real * real) -> bool
val >= : (real * real) -> bool
val <= : (real * real) -> bool
val abs : real ->  real
val real : int -> real
val floor : real -> int
val truncate : real -> int
val ceiling : real -> int
val sqrt : real -> real
val sin : real -> real
val cos : real -> real
val arctan : real -> real
val exp : real -> real
val ln : real -> real
val print : real -> unit
val makestring : real -> string
```

DESCRIPTION

The `Real` structure provides IEEE (on most machines) double-precision floating point numbers.

real
: the floating-point type.

Overflow
: raised when the result of an operation has too large an exponent to be representable. Identical to the Integer.Overflow exception.

Sum, Diff, Prod, Floor, Exp
: synonyms for Overflow, for (almost) compatibility with *The Definition of Standard ML.*

Sqrt
: raised by the `sqrt` function.

Ln
: raised by the `ln` function.

`~` x
: the negation of x.

x `+` y
: the sum of x and y.

x `-` y
: the difference of x and y.

x `*` y
: the product of x and y.

x `/` y
: the quotient of x and y.

x `>` y
: x greater than y.

x `>=` y
: x greater than or equal to y.

x `<` y
: x less than y.

x `<=` y
: x less than or equal to y.

abs x
: the absolute value of x.

real i
: the conversion of i into floating point.

floor x
: the highest integer not greater than x.

ceiling x
: the lowest integer not less than x.

truncate x
: x rounded towards zero.

sqrt x
: the square root of x; raises Sqrt if $x < 0$.

ln x
: the natural logarithm of x; raises Ln if $x \leq 0$.

print x
: print a decimal representation of x on the standard output. In future versions we intend to ensure that the printed representation is sufficient to exactly recover the value of x, but this not true of version 0.93.

makestring x
: an ASCII decimal representation of x.

CAVEATS

The signature REAL should not be used to constrain the Real structure because this will prevent some operations from being coded inline (this is a bug).

NAME

RealArray — compact arrays of real numbers

SYNOPSIS

structure RealArray

SIGNATURE

```
eqtype realarray
exception RealSubscript
exception Size
val length: realarray -> int
val array: int * real -> realarray
val sub: realarray * int -> real
val update: realarray * int * real -> unit
```

DESCRIPTION

RealArrays are just like Arrays of Real, except that they are represented more compactly. In current implementations of the compiler, access to a `realarray` is not faster than access to a `real array`; in other words, space is saved but not time. In future implementations time may also be saved.

SEE ALSO

Array(BASE), Real(BASE)

NAME

Ref — operations on references

SYNOPSIS

signature REF
structure Ref : REF
Opened in the initial environment of SML/NJ.

SIGNATURE

```
infix 3 :=
(* special  datatype 'a ref = ref of 'a *)
con ref : '1a -> '1a ref
val !   : 'a ref -> 'a
val :=  : 'a ref * 'a -> unit
val inc : int ref -> unit
val dec : int ref -> unit
```

DESCRIPTION

The `Ref` structure provides reference cells and operations upon them. Warning: the data constructor `ref` is "special." Matching this structure to a signature will yield a `ref` constructor that does not create updateable reference cells. Thus the signature REF does not contain the datatype `ref`.

α **ref**

the type of references to values of type α.

ref x

a reference cell initialized to contain the value x. Note that the ref constructor has a weak polymorphic type when used in expressions.

`!` r

the contents of reference r.

r `:=` y

update r to contain y.

inc r

increment the integer-ref r.

dec r

decrement the integer-ref r.

SEE ALSO

Discussion of weak polymorphism in the GUIDE chapter.

NAME

String — operations on character strings

SYNOPSIS

signature STRING
structure String : STRING
Opened in the initial environment of SML/NJ.

SIGNATURE

```
type string
exception Substring
val size : string -> int
val length : string -> int
val substring : string * int * int -> string
val explode : string -> string list
val implode : string list -> string
val <= : string * string -> bool
val <  : string * string -> bool
val >= : string * string -> bool
val >  : string * string -> bool
val ^  : string * string -> string
exception Chr
val chr : int -> string
exception Ord
val ord : string -> int
val ordof : string * int -> int
val print : string -> unit
```

DESCRIPTION

The `String` structure contains the character string type and operations upon it.

string
the character string type. Each string is a sequence of zero or more characters; each character has a code ranging from 0 to 255.

size s
the number of characters in s.

length s
same as size(s).

Substring
raised by substring

substring(s,i,n**)**
yields an n-character substring of string s starting after the first i characters. substring$(s, 0, 1)$ is the first character in s; substring$(s, 0, \text{size } s) = s$. Raises `Substring` if $n < 0$, $i < 0$, or $i + n > \text{size}(s)$.

explode s
a list of single-character strings which, when concatenated, would yield s.

implode l
the concatenation of all the elements of l. The elements of l need not be single-character strings.

s `<=` *t*
: Lexicographic less-than-or-equal.

s `<` *t*
: Lexicographic less-than.

s `>=` *t*
: Lexicographic greater-or-equal.

s `<` *t*
: Lexicographic greater-than.

s `^` *t*
: The concatenation of s and t. Note that implode is more efficient for the concatenation of more than two strings.

Chr
: raised by chr.

chr *i*
: A single-character string containing the character whose code is i. Raises `Chr` if $i < 0$ or $i > 255$.

Ord
: raised by ord and ordof.

ord *s*
: The code of the first character in s. Raises `Ord` if s is the empty string.

ordof (s, i)
: The code of the ith character of s, counting from 0. ordof$(s, 0)$ is equivalent to ord(s). Raises `Ord` if $i < 0$ or $i \geq$ size(s).

print *s*
: write s to the standard output.

CAVEATS

The signature STRING should not be used to constrain the String structure because this will prevent some operations from being coded inline (this is a bug).

SEE ALSO

CharSet(LIB), Ctype(LIB), Format(LIB), HashString(LIB), Makestring(LIB), Name(LIB), StringCvt(LIB), StringUtil(LIB)

NAME

Vector — immutable, constant-time indexable sequences

SYNOPSIS

signature VECTOR
structure Vector : VECTOR

#[$exp_1, exp_2, \ldots, exp_n$]
#[$pat_1, pat_2, \ldots, pat_n$]

SIGNATURE

```
eqtype 'a vector
exception Size
exception Subscript
val vector: 'a list -> 'a vector
val tabulate: int * (int -> 'a) -> 'a vector
val sub: 'a vector * int -> 'a
val length: 'a vector -> int
```

DESCRIPTION

The `Vector` structure provides one-dimensional immutable indexable arrays.

vector l
: a zero-based, indexable vector whose elements are those of l, with the same length and in the same order.

tabulate (n, f)
: a vector whose ith element is $f(i)$. Raises `Size` if $n < 0$ or $n \geq 2^{25}$.

sub (v, i)
: the ith element of v. Raises Subscript if $i < 0$ or $i \geq \text{length}(v)$. If `sub` is declared `infix`, it is conventional to use precedence 9.

length v
: the number of elements in v.

Vectors a and b are equal if and only if $\text{length}(a) = \text{length}(b)$ and $a_i = b_i$ for $0 \leq i \leq \text{length}(a)$.

In Standard ML of New Jersey, the expression `#[`$exp_0, exp_1, \ldots, exp_{n-1}$`]` where $n \geq 0$ creates a vector of length n whose elements are the values of the corresponding subexpressions. Vectors may be pattern-matched by vector-patterns of the form `#[`$pat_0, pat_1, \ldots, pat_{n-1}$`]`.

Vector expressions and vector patterns are more compact and efficient than lists, and are comparable in cost to records.

SEE ALSO

Array(BASE)

CAVEATS

The signature VECTOR should not be used to constrain the Vector structure because this will prevent some operations from being coded inline (this is a bug).