# Standard ML of New Jersey

—

## System Modules

(Version 0.93)

February 15, 1993

# Contents

## NAME

Intro — introduction to system-dependent features

## DESCRIPTION

The structure `System` (see **SYSTEM**) contains substructures and values for system-dependent features of Standard ML of New Jersey.

## SEE ALSO

System(SYS)

## NAME

Ast — unelaborated ML syntax trees

## SYNOPSIS

signature AST
structure System.Ast : AST

## SIGNATURE

```
type fixity
type symbol      same as System.Symbol.symbol
val infixleft : int -> fixity
val infixright : int -> fixity
type filePos     positions in files
type path        symbolic path, a list of symbols
EXPRESSIONS
datatype exp
    = VarExp of path                                          variable
    | FnExp of rule list                                      abstraction
    | AppExp of {function:exp,argument:exp}                   application
    | CaseExp of {expr:exp,rules:rule list}                   case expression
    | LetExp of {dec:dec,expr:exp}                            let expression
    | SeqExp of exp list                                      sequence of expressions
    | IntExp of int                                           integer
    | RealExp of string                                       floating point literal,
                                                              coded by its string
    | StringExp of string                                     string
    | RecordExp of (symbol * exp) list                        record
    | TupleExp of exp list                                    tuple (derived form)
    | SelectorExp of symbol                                   selector of a record field
    | ConstraintExp of {expr:exp,constraint:ty}               type constraint
    | HandleExp of {expr:exp, rules:rule list}                exception handler
    | RaiseExp of exp                                         raise an exception
    | IfExp of {test:exp, thenCase:exp, elseCase:exp}
                                                              if expression (derived form)
    | AndalsoExp of exp * exp                                 andalso (derived form)
    | OrelseExp of exp * exp                                  orelse (derived form)
    | VectorExp of exp list                                   vector
    | WhileExp of {test:exp,expr:exp}                         while (derived form)
    | MarkExp of exp * filePos * filePos                      mark an expression
RULE for case functions and exception handler
and rule = Rule of {pat:pat,exp:exp}
PATTERN
and pat = WildPat                                             empty pattern
        | VarPat of path                                      variable pattern
        | IntPat of int                                       integer
        | RealPat of string                                   floating literal
        | StringPat of string                                 string
        | RecordPat of {def:(symbol * pat) list,
                        flexibility:bool}                     record
        | TuplePat of pat list                                tuple
        | AppPat of {constr:path,argument:pat}                application
        | ConstraintPat of {pattern:pat,constraint:ty}
                                                              constraint
        | LayeredPat of {varPat:pat,expPat:pat}               as expressions
        | VectorPat of pat list                               vector pattern
        | MarkPat of pat * filePos * filePos                  mark a pattern
```

*STRUCTURE EXPRESSION*

```
and strexp = VarStr of path                                    variable structure
           | StructStr of dec                                  defined structure
           | AppStr of path * (strexp * bool) list             application
           | LetStr of dec * strexp                            let in structure
           | MarkStr of strexp * filePos * filePos             mark
```

*FUNCTOR EXPRESSION*

```
and fctexp = VarFct of path * fsigexp option                   functor variable
           | FctFct of {                                       definition of a functor
               params   : (symbol option * sigexp) list,
               body         : strexp,
               constraint : sigexp option}
           | LetFct of dec * fctexp
           | AppFct of path * (strexp * bool) list
                     * fsigexp option                          application
           | MarkFct of fctexp * filePos * filePos             mark
```

*SIGNATURE EXPRESSION*

```
and sigexp = VarSig of symbol                                  signature variable
           | SigSig of spec list                               defined signature
           | MarkSig of sigexp * filePos * filePos             mark
```

*FUNCTOR SIGNATURE EXPRESSION*

```
and fsigexp = VarFsig of symbol                                funsig variable
            | FsigFsig of {param: (symbol option * sigexp) list,
                           def:sigexp}                         defined funsig
            | MarkFsig of fsigexp * filePos * filePos
                                                               mark a funsig
```

*SPECIFICATION FOR SIGNATURE DEFINITIONS*

```
and spec =
    StrSpec of (symbol * sigexp) list                          structure
  | TycSpec of ((symbol * tyvar list) list * bool)             type
  | FctSpec of (symbol * fsigexp) list                         functor
  | ValSpec of (symbol * ty) list                              value
  | DataSpec of db list                                        datatype
  | ExceSpec of (symbol * ty option) list                      exception
  | FixSpec of  {fixity: fixity, ops: symbol list}             fixity
  | ShareSpec of path list                                     structure sharing
  | ShatycSpec of path list                                    type sharing
  | LocalSpec of spec list * spec list                         local specif
  | IncludeSpec of symbol                                      include specif
  | OpenSpec of path list                                      open structures
  | MarkSpec of spec * filePos * filePos                       mark a spec
```

*DECLARATIONS (let and structure)*

```
and dec       =
    ValDec of vb list                                              values
  | ValrecDec of rvb list                                recursive values
  | FunDec of fb list                                    recurs functions
  | TypeDec of tb list                                           type dec
  | DatatypeDec of {datatycs: db list, withtycs: tb list}
                                                             datatype dec
  | AbstypeDec of {abstycs: db list, withtycs: tb list, body: dec}
                                                            abstract type
  | ExceptionDec of eb list                                     exception
  | StrDec of strb list                                         structure
  | AbsDec of strb list                                   abstract struct
  | FctDec of fctb list                                           functor
  | SigDec of sigb list                                         signature
  | FsigDec of fsigb list                                          funsig
  | LocalDec of dec * dec                                       local dec
  | SeqDec of dec list                                    sequence of dec
  | OpenDec of path list                                  open structures
  | OvldDec of symbol * ty * exp list               overloading (internal)
  | FixDec of {fixity: fixity, ops: symbol list}                   fixity
  | ImportDec of string list                              import (unused)
  | MarkDec of dec * filePos * filePos                         mark a dec
```

*VALUE BINDINGS*

```
and vb = Vb of {pat:pat, exp:exp}
       | MarkVb of vb * filePos * filePos
```

*RECURSIVE VALUE BINDINGS*

```
and rvb = Rvb of {var:symbol, exp:exp, resultty: ty option}
        | MarkRvb of rvb * filePos * filePos
```

*RECURSIVE FUNCTIONS BINDINGS*

```
and fb = Fb of {var:symbol, clauses:clause list}
       | MarkFb of fb * filePos * filePos
```

*CLAUSE: a definition for a single pattern in a function binding*

```
and clause = Clause of {pats: pat list,
                        resultty: ty option, exp:exp}
```

*TYPE BINDING*

```
and tb = Tb of {tyc : symbol, def : ty, tyvars : tyvar list}
       | MarkTb of tb * filePos * filePos
```

*DATATYPE BINDING*

```
and db = Db of {tyc : symbol, tyvars : tyvar list,
                def : (symbol * ty option) list}
       | MarkDb of db * filePos * filePos
```

*EXCEPTION BINDING*

```
and eb = EbGen of {exn: symbol, etype: ty option}
                                 Exception definition
       | EbDef of {exn: symbol, edef: path}
                                 defined by equality
       | MarkEb of eb * filePos * filePos
```

*STRUCTURE BINDING*

```
and strb = Strb of {name: symbol,def: strexp,constraint: sigexp option}
         | MarkStrb of strb * filePos * filePos
```

*FUNCTOR BINDING*

```
and fctb = Fctb of {name: symbol,def: fctexp}
         | MarkFctb of fctb * filePos * filePos
```

*SIGNATURE BINDING*
```
and sigb = Sigb of {name: symbol,def: sigexp}
         | MarkSigb of sigb * filePos * filePos
```
*FUNSIG BINDING*
```
and fsigb = Fsigb of {name: symbol,def: fsigexp}
          | MarkFsigb of fsigb * filePos * filePos
```
*TYPE VARIABLE*
```
and tyvar = Tyv of symbol
          | MarkTyv of tyvar * filePos * filePos
```
*TYPES*
```
and ty
    = VarTy of tyvar                          type variable
    | ConTy of symbol list * ty list          type constructor
    | RecordTy of (symbol * ty) list          record
    | TupleTy of ty list                      tuple
    | MarkTy of ty * filePos * filePos        mark type
```

## DESCRIPTION

System.Ast defines a set of datatypes that represent unelaborated ML syntax trees (i.e. the result of parsing before static analysis, or elaboration, has been performed). They are produced by System.Compile.parse and can be compiled by System.Compile.compileAst. Another set of abstract syntax types are use to represent programs after elaboration, but these are not externalized. See src/absyn.

## SEE ALSO

Compile(SYS)

## NAME

System.Control.CG — code generator/optimizer control flags

## SYNOPSIS

signature CGCONTROL
structure System.Control.CG : CGCONTROL

## SIGNATURE

```
structure M68 : sig val trapv : bool ref end
val tailrecur : bool ref
val recordopt : bool ref
val tail : bool ref
val allocprof : bool ref
val closureprint : bool ref
val closureStrategy : int ref
val lambdaopt : bool ref
val cpsopt : bool ref
val rounds : int ref
val path : bool ref
val betacontract : bool ref
val eta : bool ref
val selectopt : bool ref
val dropargs : bool ref
val deadvars : bool ref
val flattenargs : bool ref
val switchopt : bool ref
val handlerfold : bool ref
val branchfold : bool ref
val arithopt : bool ref
val betaexpand : bool ref
val unroll : bool ref
val knownfiddle : bool ref
val invariant: bool ref
val targeting: int ref
val lambdaprop: bool ref
val newconreps : bool ref
val unroll_recur : bool ref
val hoistup : bool ref
val hoistdown : bool ref
val maxregs : int ref
val recordcopy : bool ref
val tagopt : bool ref
val recordpath : bool ref
val machdep : bool ref
val misc1 : bool ref
val misc2 : bool ref
val misc3 : int ref
val misc4 : int ref
val hoist : bool ref
val argrep : bool ref
val reduce : bool ref
val bodysize : int ref
val reducemore : int ref
val alphac : bool ref
val comment : bool ref
```

```
val knownGen : int ref
val knownClGen : int ref
val escapeGen : int ref
val calleeGen : int ref
val spillGen : int ref
val foldconst : bool ref
val etasplit : bool ref
val printLambda : bool ref
val printit : bool ref
val printsize : bool ref
val scheduling : bool ref
val cse : bool ref
val optafterclosure : bool ref
val calleesaves : int ref
val extraflatten : bool ref
val uncurry : bool ref
val ifidiom : bool ref
val comparefold : bool ref
val csehoist : bool ref
val rangeopt : bool ref
val floatargs : int ref
val floatvars : int ref
val floatreg_params : bool ref
val icount : bool ref
val representations : bool ref
```

## DESCRIPTION

There is little point in fiddling with these flags to improve the performance of the optimizer. Each flag either has little effect or is already set to an optimum value.

**M68.trapv** generate arithmetic traps on the MC680x0.

**tailrecur** obsolete.

**recordopt** constant-folding of record expressions.

**tail** obsolete.

**allocprof** generate allocation-profiling code.

**closureprint** print information about generated closures.

**closureStrategy** choose flat vs. linked closures.

**lambdaopt** perform reduction of lambda-language before CPS-conversion.

**cpsopt** perform optimization of CPS representation.

**rounds** how many alternating rounds of expansion/contraction.

**path** ?

**betacontract** perform $\beta$-contraction of functions called only once.

**eta** perform $\eta$ reduction on CPS.

**selectopt** constant-folding of SELECT expressions.

**dropargs** remove unused arguments to known functions.

**deadvars** eliminate dead variables.

**flattenargs** flatten tupled arguments of known functions.

**switchopt** constant-folding of SWITCH expressions.

**handlerfold** constant-folding of exception handlers.

**branchfold** optimization of conditional branches whose clauses are $\alpha$-equivalent.

**arithopt** constant-folding of arithmetic operators.

**betaexpand** $\beta$-reduction of functions called more than once.

**unroll** loop unrolling.

**knownfiddle** scope localization of free variables.

**invariant** hoist loop-invariant computations.

**targeting** register targeting of results of subexpressions.

**lambdaprop** ?

**newconreps** use efficient data-constructor representations.

**unroll_recur** unrolling of non-tail-recursive functions.

**hoistup** enlarge scope of variables to merge closures.

**hoistdown** contract scope of variables to merge closures.

**maxregs** limit the number of registers used on the target machine.

**recordcopy** ?

**tagopt** ?

**recordpath** perform select-path optimization on record components.

**machdep** do machine-dependent optimizations (e.g. load-delay scheduling).

**misc1, misc2, misc3, misc4** miscellaneous control flags.

**hoist** hoistup or hoistdown.

**argrep** use trace-based argument register selection.

**reduce** perform the contract phase (which encompasses many of the above optimizations).

**bodysize** control the optimism of the inline expander.

**reducemore** control the propensity of the optimizer to give up when] few optimizations are found.

**alphac** perform alpha conversion when necessary.

**comment** generate comments in the assembly code.

**knownGen** count of the number of known functions without closures ever generated.

**knownClGen** count of the number of known functions with closures ever generated.

**escapeGen** the number of escaping functions ever generated.

**calleeGen** the number of callee-save closures ever generated.

**spillGen** the number of spills ever generated.

**foldconst** do constant folding

**etasplit** local/global calling sequences for escaping functions.

**printLambda** show the lambda code of each compilation.

**printit** show the CPS code after each phase.

**printsize** show the size of CPS after each phase.

**scheduling** instruction scheduling.

**cse** common subexpression elimination.

**optafterclosure** perform CPS optimization after the closure phase.

**calleesaves** number of callee-save registers.

**extraflatten** flatten tupled arguments even when it might slow down some calls.

**uncurry** do the uncurrying optimization.

**ifidiom** optimize the special SWITCH statements generated by if expressions.

**comparefold** constant-fold integer comparisons.

**csehoist** hoist common subexpressions.

**rangeopt** do range analysis.

**floatargs** number of floating-point argument registers.

**floatvars** number of floating-point temporary registers.

**floatreg_params** ?

**icount** enable instruction counting.

**representations** do Leroy-style representation analysis

## REFERENCES

Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

## NAME

CInterface — low-level operating system interface functions

## SYNOPSIS

signature CINTERFACE
structure System.Unsafe.CInterface : CINTERFACE

## SIGNATURE

```
exception CFunNotFound of string
exception SysError of (int * string)
exception SystemCall of string

type time

val c_function  : string -> ('a -> 'b)
val c_string    : string -> string
val wrap_sysfn  : string -> ('a -> 'b) -> 'a -> 'b

val argv        : unit -> string list
val environ     : unit -> string list
val gethostname : unit -> string
val exec        : (string * string list * string list) -> (int * int)
val system      : string -> int
val export      : int -> bool
val blas        : (int * 'a) -> int
val salb        : string -> 'a
val gettime     : unit -> {usr : time, sys : time, gc : time}
val setitimer   : (int * time * time) -> unit
val gc          : int -> unit
val syscall     : (int * string list) -> int

val exit        : int -> 'a
val getpid      : unit -> int
val getuid      : unit -> int
val getgid      : unit -> int
val chdir       : string -> unit
val gethostid   : unit -> string
val gettimeofday: unit -> time
```

## DESCRIPTION

**c_function**
get a function from the runtime system. Raises `CFunNotFound` with the name of the missing function.

**c_string**
convert a string to null-terminated (C language) format.

**wrap_sysfn**
get, and wrap exception-handlers around, a C function.

**argv ()**
get the command line arguments.

**environ ()**
get the Unix "environment" as a list of strings.

**gethostname ()**
what it says.

**exec** $(c, a, e)$
Pipe, fork and execute Unix command $c$ with arguments $a$ and environment $e$; return input and output file descriptors.

**system** $s$
Execute shell command $s$.

**export** $fd$
like exportML but to a file descriptor.

**blas** $(fd, x)$
blast-write $x$ to a file descriptor.

**salb** $s$
Convert a character string to a data structure. Essentially the inverse of **blas**.

**gettime ()**
Get the cumulative execution time of the process, divided into non-gc user time, system time, and garbage collection time.

**setitimer** $(i, t_1, t_2)$
Set an interval timer (see the Unix manual).

**gc** $n$
Invoke the garbage collector. For a minor collection, $n = 0$; for a major collection, $n > 0$. Future releases may have more than two levels of collection.

**syscall (**$n$,$l$**)**
Perform system call #$n$ with arguments $l$. The arguments may, in fact, be integer values, byte arrays, etc., as well as strings; use **cast**. Use of this function is not portable.

**exit** $c$
Terminate the **sml** process with error code $c$.

**getpid(), getuid(), getgid(), gethostid()**
Unix system calls.

**chdir** $s$, **gettimeofday()**
Unix system calls.

## NAME

System.Unsafe.CleanUp — cleanup functions to be executed on exit

## SYNOPSIS

signature CLEANUP
structure System.Unsafe.CleanUp : CLEANUP

## SIGNATURE

```
datatype clean_mode
  = CleanForExportML | CleanForExportFn
  | CleanForQuit | CleanForInit
val addCleaner : (string * (clean_mode -> unit)) -> bool
val removeCleaner : string -> unit
val cleanup : clean_mode -> unit
val shutdown : unit -> 'a
```

## DESCRIPTION

Allows the installation of cleanup functions that will be executed when the system is about to exit for some reason. Normally used by the i/o stream system to keep track of which files need buffers flushed, etc.

## NAME

Code — interface to machine code objects

## SYNOPSIS

signature CODE
structure System.Code : CODE

## SIGNATURE

```
structure IO : sig type instream end
type code
val mkCode : string -> code
val inputCode : (IO.instream * int) -> code
val sizeOf : code -> int
val apply : code -> ('a -> 'b)
```

## DESCRIPTION

System.Code provides an interface for creating and operating on machine code objects, which are essentially a special form of string. Code objects can also be generated by the functions System.Compile.compile and System.Compile.compileAst, but in these cases the code is bundled inside values of type codeUnit. One reason for creating a special type for code is to support special memory management for code.

**structure IO**
Imported version of IO providing the type instream.

**type code**
The type of code values. Structurally the same as strings.

**mkCode** *s*
Coerces a string to a code value.

**inputCode (***instream*, *n***)**
Read the first n bytes from the instream, producing a code value.

**sizeOf** *code*
Returns the size in bytes of the code.

**apply** *code*
Coerces code to an executable function. The highly polymorphic type of the result reflects the fact that code of different "types" is produced by compiler functions (e.g. the batch compiler and the interactive top level), so the arguments expected when the code is executed differ. This will change after 0.93.

## SEE ALSO

IO(BASE), Compile(SYS)

## NAME

Compile — interface to compiler

## SYNOPSIS

signature COMPILE
structure System.Compile : COMPILE

## SIGNATURE

```
structure PP : sig type ppconsumer end
structure IO : sig type instream type outstream end
structure Ast : sig type dec end
type source
type staticUnit
type codeUnit
type compUnit (* = staticUnit * codeUnit *)
exception Compile of string
val makeSource  : string * int * IO.instream * bool * PP.ppconsumer
                  -> source
val closeSource : source -> unit
val changeLvars : staticUnit -> staticUnit
val elaborate   : source * staticEnv -> staticUnit
val parse       : source * staticEnv -> Ast.dec * staticEnv
val compile     : source * staticEnv -> compUnit
val compileAst  : Ast.dec * staticEnv * source option -> compUnit
val execute     : compUnit * environment -> environment
val eval_stream : IO.instream * environment -> environment
val use         : string -> unit
val use_stream  : IO.instream -> unit
```

## DESCRIPTION

A unit of separate compilation is, in terms of source code, a sequence of signature, structure, and module declarations. The resulting environment containing only signature, structure, and functor bindings can also be thought of as the compiled form of the compilation unit. The declarations need not be closed: they can contain free references not only to one another, but to pervasives and to modules defined in "earlier" compilation units.

System.Compile: COMPILE is the external interface providing access to functions defined in functors CompileUnit (src/build/compile.sml) and Interact (src/build/interact.sml).

functions.

### structure PP

Imported for type PP.ppconsumer.
See section **PrettyPrint**(SYS).

### structure IO

Imported for type instream. See section **IO**(BASE).

### structure Ast

Imported for type dec. See section **Ast**(SYS).

### type source

External version of ErrorMsg.inputSource. This wraps an input stream (instream) in a datastructure that maintains location information for error reporting. It may also contain an output stream for gathering indexing information.

**type staticUnit**
The static information for a unit of compilation. It contains a static environment whose elements are module bindings, plus a list of the lvars of these elements.

**type codeUnit**
The code part of a compilation unit. It contains the code itself, plus a persistent specification of imported components, i.e. structures and functors mentioned freely in the declarations of its elements.

**type compUnit**
An abbreviation for `(staticUnit * codeUnit)`.

**exception Compile of string**
Exception raised to signal compilation errors. The string argument identifies the error and other relevant information.

**makeSource** (*s*, *n*, *str*, *interact*, *consumer*)
Probably the first argument to makeSource should be a "string option". Perhaps the bool argument indicating whether the instream is interactive could be eliminated, if this attribute can be determined from the stream itself.

**closeSource** *source*
Close the input steam and indexing stream, if any.

**changeLvars** *unit*
"Alpha-convert" a static unit by replacing its bound lvars with newly created ones. This is used to prevent clash of lvar bindings when importing a previously compiled staticUnit.

**elaborate** (*source*, *senv*)
Parse and analyze input from the source using the staticEnv and producing a staticUnit as the result. The result contains only the new bindings, i.e. the delta environment. Terminated by eof in the source or an error causing the Compile exception.

**parse** (*source*, *senv*)
Parse the source relative to the static environment *senv*, yielding an `Ast.dec` and a resultant incremental environment containing the new infix bindings introduced by the declaration parsed.

**compile** (*dec*, *senv*)
Elaborate and generate code for the declaration syntax tree *dec*. The result is a staticUnit and corresponding codeUnit.

**compileAst** (*dec*, *senv*, *source_opt*)
Parse, elaborate, and generate code for the declarations input from the source. The result is a staticUnit and corresponding codeUnit. Terminated by eof in the source or an error causing the Compile exception.

**execute** (*unit*, *env*)
Execute the code from the codeUnit against the given environment. The result is bound in a dynamic environment that is combined with with static environment of the staticUnit to form the result environment. The imports from the codeUnit are looked up in the static part of the environment arg and their stamps are validated.

**use** *file_name*
The string *file_name* is interpreted as a file name. The file is opened and used to create

a source that is compiled against the top-level environment. The resulting environment is concatenated onto the top level environment.

**use_stream** *instream*
Similar to use, but starting with an instream instead of a file name.

## SEE ALSO

Symbol(SYS), Env(SYS), Ast(SYS)

## CAVEATS

This interface will change substantially after version 0.93, though no functionality will be lost.

## NAME

Control — ways of controlling compiler modes

## SYNOPSIS

signature CONTROL
structure System.Control : CONTROL

## SIGNATURE

```
structure Runtime : RUNTIMECONTROL
structure MC : MCCONTROL
structure CG : CGCONTROL
structure Print : PRINTCONTROL
structure Debug : DEBUG
val allocProfReset : unit -> unit
val allocProfPrint : unit -> unit
val prLambda       : (unit -> unit) ref
val debugging            : bool ref
val primaryPrompt  : string ref
val secondaryPrompt: string ref
val internals      : bool ref
val weakUnderscore : bool ref
val interp         : bool ref
val debugLook      : bool ref
val debugCollect   : bool ref
val debugBind      : bool ref
val saveLambda     : bool ref
val saveLvarNames  : bool ref
val timings        : bool ref
val reopen         : bool ref
val markabsyn      : bool ref
val indexing       : bool ref
val instSigs       : bool ref
val quotation      : bool ref
```

## DESCRIPTION

System.Control contains many knobs for controlling the specifics of compiler operation. The substructures are documented in their own sections of this chapter.

**allocProfReset ()**
reset the allocation profiler.

**allocProfPrint ()**
show statistics of allocation profiling.

**primaryPrompt**
Printed between compilation units at top level; the default is "`- `"

**secondaryPrompt**
Printed between compilation units at top level; the default is "`= `"

**internals**
show types and values with compiler internal mumbo-jumbo.

**weakUnderscore**
show weak type variables with underscore syntax for (almost) compatibility with *The Definition of Standard ML.*

**interp**
: Interpret programs instead of generating native code. Makes compilation go an order of magnitude faster, makes execution two orders of magnitude slower.

**debugLook, debugCollect, debugBind**
: turn on debug print statements for various parts of the compiler front end.

**saveLvarNames**
: keep track of source-language variable names associated with internal compiler temporaries.

**timings**
: display the execution time of each phase of the compiler.

**markabsyn**
: keep track of source-code line numbers in abstract syntax.

**indexing**
: create index files for GNU-Emacs tags program.

**quotation**
: enable the quote/antiquote mechanism.

## NAME

Directory — operations on Unix directory files

## SYNOPSIS

signature DIRECTORY
structure System.Directory : DIRECTORY

## SIGNATURE

```
val isDir : string -> bool
exception NotDirectory
val listDir : string -> string list
val cd : string -> unit
val getWD : unit -> string
```

## DESCRIPTION

**isDir** *s*
true if *s* names a directory.

**listDir** *s*
a list of files in directory *s*; raises `NotDirectory` if *s* is not a directory.

**cd** *s*
make *s* current working directory.

**getWD ()**
get the name of the current working directory.

## SEE ALSO

UnixPath(LIB)

## NAME

Env — interface to compiler environments

## SYNOPSIS

signature ENVIRONMENT
structure System.Env : ENVIRONMENT

## SIGNATURE

```
type environment
type staticEnv
exception Unbound
val emptyEnv        : unit -> environment
val concatEnv       : environment * environment -> environment
val layerEnv        : environment * environment -> environment
val staticPart      : environment -> staticEnv
val layerStatic     : staticEnv * staticEnv -> staticEnv
val filterEnv       : environment * symbol list -> environment
val filterStaticEnv : staticEnv * symbol list -> staticEnv
val catalogEnv      : staticEnv -> symbol list
val describe        : staticEnv -> symbol -> unit
val pervasiveEnvRef : environment ref
val topLevelEnvRef  : environment ref
```

## DESCRIPTION

System.Env is an external version of the internal Env structure used by the compiler.

**type environment**
External version of Environment.environment.

**type staticEnv**
External version of Environment.staticEnv.

**exception Unbound**
Raised when attempting to look up a symbol not bound in the environment in question.

**emptyEnv ()**
The empty environment (Environment.emptyEnv).

**concatEnv (*e1*, *e1*)**
Concatenate two environments, with the first argument overlaying the second argument. The concatenation eliminates bindings in the second argument's dynamic environment that become hidden as the result of the concatenation.

**layerEnv (*env1*, *env1*)**
Concatenate two environments, with the first argument overlaying the second argument, but don't bother to eliminate hidden dynamic bindings. This is used for temporary combinations of environments.

**staticPart *env***
Extract the static part (statenv and invenv) of an environment.

**layerStatic (*senv1*, *senv1*)**
Layer the two components of the staticEnvs.

**filterEnv (*env*, *symbols*)**
The result is the subenvironment of the first argument formed by selecting only the bindings

of the symbols in the second environment. Any symbols in the symbol list that are not bound in the first environment are ignored. This function allows us to cut down an environment to a desired minimum.

**catalogEnv** *senv*
Produces a sorted list of all symbols bound in the staticEnv *senv*.

**describe** *senv symbol*
Prints a description of the binding of the symbol in the staticEnv *senv*.

**pervasiveEnvRef**
A reference containing the pervasive environment. This is used as the base environment for most compilations. If this reference is updated, the change will not affect evaluation until the top-level evaluation loop is restarted, which will happen on an interrupt.

**topLevelEnvRef**
A reference containing the current top-level environment. This is changed by each interactive evaluation, and implicitly by functions like "use". Changes take effect on the next top-level evaluation.

## SEE ALSO

Symbol(SYS), Compile(SYS)

## CAVEATS

This interface will change after version 0.93, though no functionality will be lost.

## NAME

MC — control of the pattern-match compiler

## SYNOPSIS

signature MCCONTROL
structure System.Control.MC : MCCONTROL

## SIGNATURE

```
val printArgs : bool ref
val printRet : bool ref
val bindContainsVar : bool ref
val bindExhaustive : bool ref
val matchExhaustive : bool ref
val matchRedundant : bool ref
val expandResult : bool ref
```

## DESCRIPTION

These flags control the operation of the analyzer that converts ML pattern matches to decision trees during the compilation process.

**printArgs**
whether to (for diagnostic purposes) print the clauses of each match analyzed.

**printRet**
whether to (for diagnostic purposes) print the results of each pattern-match compilation

**bindContainsVar**
whether to issue warning messages of the form, "binding contains no variable."

**bindExhaustive**
whether to issue warning messages of the form, "binding not exhaustive."

**matchExhaustive**
whether to issue warning messages of the form, "match not exhaustive."

**matchRedundant**
whether to issue warning messages of the form, "redundant pattern in match."

**expandResult**
whether to create duplicate copies of right-hand-sides in some cases (default is false). Has no effect on semantics, but can cause exponential code blowup if true.

## NAME

PolyCont — unsafe call-with-current-continuation

## SYNOPSIS

signature POLY_CONT
structure System.Unsafe.PolyCont : POLY_CONT

## SIGNATURE

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
type 'a control_cont
val capture : ('a control_cont -> 'a) -> 'a
val escape : 'a control_cont -> 'a -> 'b
```

## DESCRIPTION

`PolyCont.callcc` is a more loosely typed creator of first-class continuations. With the looser typing it is possible (if you are clever enough) to poke holes in the type system. However, for some applications the weak typing is too restrictive.

`capture` and `escape` are just like `callcc` and `throw` except that when a continuation is invoked it gets the exception handler of the invoker. This is a bug, not a feature, but it can be more efficient for applications that don't care about the exception handler: `capture` can be tail-recursive since it doesn't have to save an exception handler. Certain space leaks can thus be avoided.

## SEE ALSO

Callcc(BASE)

## NAME

PrettyPrint — prettyprinting values

## SYNOPSIS

signature PRETTYPRINT
structure System.PrettyPrint : PRETTYPRINT

## SIGNATURE

```
type ppstream
type ppconsumer = consumer: string -> unit,
                  flush: unit -> unit,
                  linewidth: int
datatype break_style = CONSISTENT | INCONSISTENT
val mk_ppstream    : ppconsumer -> ppstream
val dest_ppstream  : ppstream -> ppconsumer
val add_break      : ppstream -> int * int -> unit
val add_newline    : ppstream -> unit
val add_string     : ppstream -> string -> unit
val begin_block    : ppstream -> break_style -> int -> unit
val end_block      : ppstream -> unit
val clear_ppstream : ppstream -> unit
val flush_ppstream : ppstream -> unit
val with_pp        : ppconsumer -> (ppstream -> unit) -> unit
val install_pp     : string list -> (ppstream -> 'a -> unit)
                       -> unit
val pp_to_string  : int -> (ppstream -> 'a -> unit) -> 'a -> string
```

## DESCRIPTION

System.PrettyPrint implements a facility for defining and installing user-defined prettyprinters over (monomorphic) user-defined types for use by the top-level loop of the SML compiler. The underlying algorithm is that of Oppen (TOPLAS, 1980). There are more expressive prettyprinting languages around, notably that of PPML, but the Oppen interface has the benefit of being efficiently implementable. Thus it is a good option for modest prettyprinting tasks that need to be quick, such as the printing of SML values. These same prettyprinting facilities are used by the compiler for such tasks as printing values in the interactive top level and printing expressions in error messages.

The high-level view is that the user will define a prettyprinter for a datatype and install it in a prettyprinter table. When it comes time for the compiler to print a value, it looks first in the prettyprinter table, to see if a prettyprinter is installed for that type. If so, it calls the prettyprinter on the value, otherwise, it calls the default printing routine.

The Oppen algorithm provides a *block* abstraction: a block establishes a level of indentation. Since blocks can be nested and offset from one another, levels of indentation can be achieved. A block can be broken up by one or more *breaks*, which mark possible places to add carriage returns. There are two styles of block: CONSISTENT and INCONSISTENT. If a consistent block does not fit completely onto the current line, a carriage return will be added after each component of the block. If an INCONSISTENT block does not fit completely onto the current line, a carriage return is added after the last item that does fit on the line; this style of block conserves on vertical space.

### type ppstream

This is an abstract type of prettyprint streams. A ppstream encapsulates the state of an Oppen prettyprinter.

### type ppconsumer

This type describes a record that provides the ultimate consumer of the characters produced by the prettyprint operations. A ppconsumer is supplied to `mk_ppstream` and `with_pp` and is used by those operations to build ppstreams.

**type break_style**
This defines the behavior of breaks within a block. If the style is `CONSISTENT`, then if any break occurs, all breaks occur. If the style is `INCONSISTENT`, then breaks will only occur when the current line is full.

**mk_ppstream** *consumer*
A new ppstream is constructed, with character output directed to the consumer.

**dest_ppstream** *pps*
Returns the consumer record used by the ppstream.

**install_pp** *path ppfn* The first argument is a path designating a type constructor in the current interactive top-level environment. This type constructor must be a nullary constructor (i.e it must be a type constant like "int" rather than a constructor with parameters like "list"), and it must be generative (i.e. it must be a datatype or abstype constructor). The second argument is a prettyprinting function for that type. Note that the connection between the type constructor designated by the path and the printing function is not enforced by type checking, so this operation is unsafe. Also note that install_pp should only be called at top_level; if you try to install a prettyprinter for a datatype defined within a local context (like a structure) from within that same context, it won't work.

The function ppfn is installed in a prettyprinter table maintained by the interactive system, and it will be used when printing values of the designated type.

It is planned that this facility will be extended to allow nonnullary type constructors in the future.

**begin_block** *pps bs*
This begins a new level of indentation, at the current offset from the left margin. The break_style argument determines how the block is to be broken. The third argument determines the new offset if the block gets broken.

**end_block** *pps* Prettyprinting for pps reverts to the previous level of indentation.

**add_break** *pps* (*size*, *offset*)
Notify the ppstream pps that a carriage return is possible. The argument to this function is a pair; the first member is the size of the break, the second member is an offset. This "break" offset is used for finer control over indentation than that offered by the block offset. The current block style and the size of the block determine what action is to be taken:

- block_style = CONSISTENT and the entire block fits on the remainder of the current line =¿ output size spaces.
- block_style = CONSISTENT and the block does not fit on the rest of the line =¿ add a carriage return after each component in the block and add the block offset to the current indentation level. Each component will be further indented by offset spaces.
- block_style = INCONSISTENT and the next component of the current block fits on the rest of the line =¿ output size spaces.
- block_style = INCONSISTENT and the next component doesn't fit on the rest of the line =¿ output a carriage return and indent to the current indentation level plus the block offset plus offset extra spaces.

Size is taken into account when determining if there is enough room to print the next component in the block.

**add_string** *pps s*
Outputs the given string to pps.

**add_newline** *pps*
Forces the output of a line break on pps.

**lear_ppstream** *pps*
Clears the state of pps.

**flush_ppstream** *pps*
Flushes currently accumulated text to pps's consumer and calls the flush operation of the consumer.

**with_pp** *consumer printfn*
Constructs a new ppstream from consumer and applies the printfn to it.

**pp_to_string** *linewidth printit x*
Constructs a new ppstream whose consumer accumulates output in a string. Applies the function printit to this ppstream and the value x, and then returns the string s containing the prettyprinting output.

## EXAMPLE

```
open System.PrettyPrint;
fun out_ppstream outstrm n =
    mk_ppstream{linewidth = n,
                flush = fn () => flush_out outstrm,
                consumer = outputc outstrm}
fun with_ppstream ppstrm =
    {add_string=add_string ppstrm,
     add_break=add_break ppstrm,
     begin_block=begin_block ppstrm,
     end_block=fn () => end_block ppstrm,
     flush_ppstream=fn () => flush_ppstream ppstrm};
datatype Num = Zero | Suc of Num;
fun pp_Num ppstrm n =
    let fun ppn Zero = add_string ppstrm "Z"
          | ppn (Suc n) = ( ppn n; add_string ppstrm "'")
     in begin_block ppstrm CONSISTENT 0;
        ppn n;
        end_block ppstrm
    end;
val _ = install_pp ["Num"] pp_Num;
datatype Nexp = Atom of Num
             | Add of (Nexp*Nexp);
fun pp_Nexp ppstrm ne =
    let fun pr (Atom  a) = pp_Num ppstrm a
          | pr (Add(a1,a2)) =
             (pr a1;
              add_string ppstrm " +";
              add_break ppstrm (1,0);
              pr a2)
     in begin_block ppstrm CONSISTENT 0;
        pr ne;
        end_block ppstrm
    end;
val _ = install_pp ["Nexp"] pp_Nexp;
```

```
datatype exp = A of Nexp
             | ITE of (bool*exp*exp);
fun pp_exp ppstrm e =
    let val {add_string,add_break,begin_block,
              end_block,flush_ppstream} =
                with_ppstream ppstrm
        val pp_Nexp = pp_Nexp ppstrm
        fun pr (A a) = pp_Nexp a
          | pr (ITE(b,e1,e2)) =
               (begin_block CONSISTENT 0;
                add_string"if ";
                add_string (if b then "true" else "false");
                add_break(1,2);
                add_string "THEN ";
                pr e1;
                add_break(1,2);
                add_string "ELSE ";
                pr e2;
                add_break(1,0);
                add_string "END";
                end_block ())
     in begin_block CONSISTENT 0;
        pr e;
        end_block ()
    end;
val _ = install_pp ["exp"] pp_exp;
(* some example values to try *)
val N = Add(Atom(Suc(Suc(Suc(Suc(Suc(Suc(Suc(Suc(Suc(Zero))))))))),
            Atom(Suc(Suc(Suc(Suc(Suc(Suc(Suc(Suc(Suc(Zero))))))))));
val a = ITE(true,A(N),A(N));
val b = ITE(false,a,A(N));
val c = ITE(true,b,b);
```

## FILES

| | |
|---|---|
| $SMLSRC/boot/system.sig | contains the `PRETTYPRINT` signature. |
| $SMLSRC/util/pp.sml | contains the internal `PrettyPrint` structure implementation. |

## CAVEATS

Be sure that all prettyprinting is done between calls of `begin_block` and `end_block`, and that calls to these functions are balanced (dynamically, not just statically!). This can be tricky to check.

Prettyprint functions that are going to be installed by install_pp should not call flush_ppstream. The system will do the flushing for you.

In general, expect to spend some time experimenting to get the printing to work the way you want it to.

## NAME

Print — control of top-level interactive value printing

## SYNOPSIS

signature PRINTCONTROL
structure System.Print : PRINTCONTROL

## SIGNATURE

```
type outstream
val printDepth : int ref
val printLength : int ref
val stringDepth : int ref
val printLoop : bool ref
val signatures : int ref
val out : outstream ref
val linewidth : int ref
val say : string -> unit
val flush: unit -> unit
```

## DESCRIPTION

**printDepth**
the depth of nesting of recursive data structure at which ellipsis begins.

**printLength**
the length of lists at which ellipsis begins.

**stringDepth**
the length of strings at which ellipsis begins.

**printLoop**
whether to treat loops (involving ref cells) specially when printing.

**signatures**
whether to print signature bodies at their declarations.

**out**
The stream to which all compiler messages and top-level value printing should go. Initialized to `std_out`. To suppress all printout by the compiler, open `/dev/null` for output and assign the resulting outstream to `System.Print.out`:

```
System.Print.out := open_out "/dev/null"
```

**linewidth**
The number of characters per line for prettyprinting to `out`.

**say** *s*
write *s* to `!System.Print.out`.

**flush ()**
`flush_out(!out)`.

The function `print` in the initial environment is overloaded, and can be used to print integers, reals, strings, or booleans. It's output is directed to the output stream `std_out`, which represents the Unix standard output, and this cannot be changed. So output by applications that use the `print` function cannot easily be controlled. The function `System.Print.say` can be used as an alternative — it's output is via `System.Print.out`.

## NAME

Runtime — control flags for the runtime system

## SYNOPSIS

signature RUNTIMECONTROL
structure System.Control.Runtime : RUNTIMECONTROL

## SIGNATURE

```
val collected : int ref
val collectedfrom : int ref
val gcmessages : int ref
val majorcollections : int ref
val minorcollections : int ref
val ratio : int ref
val softmax : int ref
val lastratio : int ref
```

## DESCRIPTION

**collected**
cumulative kilobytes of live objects ever found by the garbage collector.

**collectedfrom**
cumulative kilobytes of live+dead objects ever examined by the garbage collector.

**gcmessages**
Which level of garbage collection messages to print:

**0** None.

**1** Only heap resizings.

**2** Major collections and heap resizings.

**3** Minor collections and all of the above.

$> 3$ Levels greater than 3 are legal; in current versions they have the same effect as level 3.

**majorcollections**
how many major collections have ever occurred.

**minorcollections**
how many minor collections have ever occurred.

**ratio**
The target ratio of heap size to live data, as long as memory usage is less than the soft max.

**softmax**
What level of physical memory usage (in bytes) the system should aim for. If $m_h$ is the hard max (the "datasize" operating system resource limit), $m_s$ is the soft max, $r$ is the heap ratio, and $d$ is the amount of live data, then the heap size will be approximately

$$\min(m_h, \max(3d, \min(rd, m_s))).$$

**lastratio**
The ratio of heap size to live data actually achieved at the last major garbage collection.

## NAME

Signals — interface to Unix signal system

## SYNOPSIS

signature SIGNALS
structure System.Signals : SIGNALS

## SIGNATURE

```
datatype signal
   = SIGHUP | SIGINT | SIGQUIT | SIGALRM | SIGVTALRM
   | SIGTERM | SIGURG | SIGCHLD | SIGIO | SIGWINCH
   | SIGUSR1 | SIGUSR2 | SIGPROF | SIGTSTP | SIGCONT
   | SIGGC
val setHandler: (signal *
                 ((int * unit cont) -> unit cont) option)
                -> unit
val inqHandler: signal ->
                 ((int * unit cont) -> unit cont) option
val maskSignals : bool -> unit
val masked : unit -> bool
val pause : unit -> unit
```

## DESCRIPTION

**setHandler** ($s$, SOME $h$)
Install handler $h$ for signal $s$. When the signal arrives, the handler $h(n, c)$ is invoked. Argument $n$ is the number of occurrences of the signal that have arrived by the time the system has had an opportunity to invoke the handler. Argument $c$ is the continuation function of the interrupted program. When $h$ returns a continuation, this re-enables the signal, which is blocked during execution of the handler.

If $h$ simply returns $c$ as a result, this resumes the interrupted process. Or, $h$ can return some other continuation, which will abandon the current thread of control. In this case, $h$ may want to save $c$ in a data structure for later resumption.

**setHandler** ($s$, NONE)
Remove the handler (if any) for signal $s$.

**inqHandler** $s$
get the current handler (if any) for signal $s$.

**SIGHUP ... SIGCONT**
Unix signal names

**SIGGC**
a special signal raised after each garbage colleciton.

**maskSignals true**
block the invocation of all signal handlers.

**maskSignals false**
resume the delivery of signals, including those that arrived while signals were masked.

**masked ()**
are signals blocked now?

**pause ()**
sleep until the next signal arrives.

## CAVEATS

SIGCONT is not yet supported.

## REFERENCES

John H. Reppy, "Asynchronous Signals in Standard ML", TR 90-1144, Cornell University, Dept. of Computer Science, 1990.

## NAME

Symbol — compiler symbols

## SYNOPSIS

signature SYMBOL
structure System.Symbol : SYMBOL

## SIGNATURE

```
type symbol
datatype namespace =
   VALspace | TYCspace | SIGspace | STRspace | FCTspace | FIXspace |
   LABspace | TYVspace | FSIGspace%

val valSymbol : string -> symbol
val tycSymbol : string -> symbol
val tyvSymbol : string -> symbol
val sigSymbol : string -> symbol
val strSymbol : string -> symbol
val fsigSymbol : string -> symbol
val fctSymbol : string -> symbol
val fixSymbol : string -> symbol
val labSymbol : string -> symbol
val name      : symbol -> string
val makestring: symbol -> string
val kind      : symbol -> string
val nameSpace : symbol -> namespace
val makeSymbol: namespace * string -> symbol
```

## DESCRIPTION

System.Symbol is an external interface to the internal Symbol structure used by the compiler.

**type symbol**

External version of the internal type *Symbol.symbol.* Symbols are needed when working with abstract syntax trees (System.Ast) and environments (System.Env). Though symbols belong to one type, `symbol`, they come in nine different flavors, corresponding to the different name spaces of Standard ML: variables and constructors, type constructors, type variables, signatures, structures, functor signatures, functors, fixities, and labels.

**type namespace**

External version of the internal type *Symbol.namespace.* The constructors of this datatype correspond to the nine namespaces that symbols are partitioned into. Each symbol has a unique namespace.

**valSymbol** *s*

Creates a value symbol in name space `VALspace`.

**tycSymbol** *s*

Creates a type constructor symbol in name space `TYCspace`.

**tyvSymbol** *s*

Creates a type variable symbol in name space `TYVspace`. The string s should start with an apostrophy, but this is not checked.

**sigSymbol** *s*

Creates a signature symbol in name space `SIGspace`.

**strSymbol** *s*
: Creates a structure symbol in name space `STRspace`.

**fsigSymbol** *s*
: Creates a functor signature symbol in name space `FSIGspace`.

**fctSymbol** *s*
: Creates a functor symbol in name space `FCTspace`.

**fixSymbol** *s*
: Creates a fixity symbol in name space `FIXspace`. Fixity symbols are bound to fixity information. An infix directive binds a fixity symbol; a later value declaration for the same identifier creates a separate binding of a value symbol with the same name.

**labSymbol** *s*
: Creates a label symbol in name space `LABspace`.

**name** *sym*
: Returns the string from which the symbol was constructed.

**makestring** *sym*
: Returns a print representation of the symbol, consisting of the name with an annotation indicating the name space. E.g. value symbol "foo" is mapped to "VAL$foo".

**kind** *sym*
: Returns a string naming the name space of the symbol. The names of the namespaces are "variable or constructor", "type constructor", "signature", "structure", "functor", "fixity", "label", "type variable", and "functor signature".

**nameSpace** *sym*
: Returns the namespace of the symbol *sym*.

**makeSymbol (***ns*, *s***)**
: Creates a new symbol with namespace *ns* and name *s*.

## SEE ALSO

Env(SYS), Compile(SYS), Ast(SYS)

## CAVEATS

This interface will change in minor ways after version 0.93. There are some missing functions and types, such as symbol equality and symbol mappings.

## NAME

SysIO — system calls for unbuffered input/output

## SYNOPSIS

signature SYSIO
structure System.Unsafe.SysIO : SYSIO

## SIGNATURE

```
type bytearray
type time

type fd  = int
eqtype fileid
datatype fname  = DESC of fd | PATH of string
datatype mode   = O_READ | O_WRITE | O_APPEND
datatype whence = L_SET | L_INCR | L_XTND
datatype access = A_READ | A_WRITE | A_EXEC
datatype file_type     = F_REGULAR | F_DIR | F_SYMLINK
                       | F_SOCK | F_CHR | F_BLK
val dtablesize   : int
val openf        : (string * mode) -> fd
val close        : fd -> unit
val unlink       : string -> unit
val pipe         : unit -> (fd * fd)
val connect_unix : string -> fd
val connect_inet : (string * string) -> fd
val link         : (string * string) -> unit
val symlink      : (string * string) -> unit
val mkdir        : (string * int) -> unit
val dup          : fd -> fd
val read         : (fd * bytearray * int) -> int
val readi        : (fd * bytearray * int * int) -> int
val write        : (fd * bytearray * int) -> unit
val writei       : (fd * bytearray * int * int) -> unit
val writev       : (fd * (bytearray * int) list) -> unit
val send_obd     : (fd * bytearray * int) -> unit
val getdirent    : fd -> string list
val readlink     : string -> string
val truncate     : (fname * int) -> unit
val lseek        : (fd * int * whence) -> int
val getmod       : fname -> int
val chmod        : (fname * int) -> unit
val umask        : int -> int
val access       : (string * access list) -> bool
val isatty       : fd -> bool
val fionread     : fd -> int
val getfid       : fname -> fileid
val ftype        : fname -> file_type
val getownid     : fname -> (int * int)
val fsize        : fname -> int
val atime        : fname -> time
val ctime        : fname -> time
val mtime        : fname -> time
val select   : (fd list * fd list * fd list * time option)
                -> (fd list * fd list * fd list)
```

## DESCRIPTION

These functions allow the manipulation of Unix file descriptors and unbuffered input/output operations upon them. Most of these operations correspond directly to Unix system calls. See a Unix manual, chapter 2, for detailed semantics.

**fd**
Unix file descriptor (a small integer).

**fileid**
A representation of the device and inode numbers; a unique descriptor of a file (no matter how reopened or linked).

**fname**
Some system calls accept either an unopened file (represented by its name) or a file descriptor; this union type is useful for arguments to those calls.

**mode**
What you intend to do with a file: read, write, or append (write at the end).

**whence**
Useful as an argument to **lseek**.

**access**
File protection modes.

**file_type**
Types of Unix file.

**dtablesize**
The number of file descriptors that can be simultaneously open.

**openf** $(s, m)$
Open an unbuffered file named $s$ for reading ($m$ = `O_READ`), writing ($m$ = `O_WRITE`), or appending ($m$ = `O_APPEND`).

**close** *fd*
Close a file. The only effect this has is to clear a slot in the file descriptor table maintained by the operating system (if this table becomes full, it is impossible to open new files).

**unlink** $s$
Unlink (remove) the file named $s$ from its directory.

**pipe ()**
Create a pipe, and return both ends as file descriptors $(fd_0, fd_1)$. Data written to $fd_1$ may be read from $fd_0$.

**connect_unix** $s$
Open a Unix-domain socket with file pathname $s$.

**connect_inet** $(h, p)$
Open an internet-domain socket connection to host $h$ (specified as an ASCII string of decimal numbers separated by dots, e.g. `"128.112.128.1"`) on port $p$ (specified as an ASCII decimal string, e.g. `"6000"`).

**link** $(s, l)$
Create a link: Make $l$ a new alternate name for file $s$.

**symlink** $(s, l)$
Create a symbolic link.

**mkdir** $(s, m)$
Make a directory named $s$ with protection (access) mode $m$.

**dup** *fd*
Duplicate a file descriptor *fd*.

**read** $(fd, a, n)$
Read no more than $n$ bytes into byte-array $a$ from file *fd*. Returns the number of bytes actually read.

**readi** $(fd, a, n, i)$
Just like **read**, but starts filling $a$ at position $i$ instead of position 0.

**write** $(fd, a, n)$
Write no more than $n$ bytes from byte-array $a$ to file *fd*. Returns the number of bytes actually written.

**writei** $(fd, a, n, i)$
Like **write**, but starts at position $i$ of $a$ instead of position 0.

**writev** $(fd, l)$
Given a list $l$ of buffers, each of which is a byte-array and the number of characters to write from it, write the contents of the buffers (in the given order) to file *fd*.

**send_obd** $(fd, a, n)$
Send $n$ bytes from byte-array $a$ as out-of-band data on socket *fd*.

**getdirent** *fd*
Get a list of files in directory *fd*.

**readlink** $s$
If $s$ names a symbolic link, get the name of the file it points to.

**truncate** $(n, i)$
Truncate file $n$ to $i$ bytes.

**lseek** $(fd, i, w)$
Position file *fd* for reading or writing from $i$ bytes beyond the (beginning / current position / end) of the file, if $w$ is (`L_SET` / `L_INCR` / `L_XTND`). The offset $i$ may be negative.

**getmod** $n$
Get the mode (access protection) of file $n$.

**chmod** $(n, m)$
Set the mode of file $n$ to $m$.

**umask** $m$
Set the user mask to $m$ (this affects the mode of newly-created files). Returns the previous value of the mask.

**access** $(s, l)$
Returns true iff the file $s$ exists and is accessible (by the current user) with permission to do all the things specified by $l$. If $l$ is empty, this is an existence test.

**isatty** *fd*
True if *fd* is associated with a terminal device.

**fionread** *fd*
Returns the number of bytes ready to read without blocking from *fd*.

**getfid** $n$
Get a unique descriptor for file $n$ (see type `fileid`, above).

**ftype** $n$
Find the type (ordinary file, directory, i/o device, etc.) of $n$.

**getownid** $n$
?

**fsize** $n$
The number of bytes in file $n$.

**atime** $n$
The last access time of $n$.

**ctime** $n$
The time $n$'s status last changed.

**mtime** $n$
The time of last modification of $n$.

**select** $(l_r, l_w, l_e, t)$
Block until either: one of $l_r$ is ready for reading, one of $l_w$ is ready for writing, or one of $l_e$ has an exceptional condition, or $t =$ SOME $t'$ and more than $t'$ seconds have elapsed. Returns the sublists of file descriptors $(l'_r, l'_w, l'_e)$ on which the respective i/o operations are possible without blocking.

## SEE ALSO

The Unix (TM) manual.

## CAVEATS

**dtablesize** isn't necessarily constant in Unix (especially after **exportML**), but is constant here.

## NAME

System — system-dependent features of SML/NJ

## SYNOPSIS

signature SYSTEM
structure System : SYSTEM
*(present but not pre-opened in the* **sml** *executable)*

## SIGNATURE

```
structure Hooks : HOOKS
structure Symbol : SYMBOL
structure Env : ENVIRONMENT
structure Ast : AST
structure Code : CODE
structure Compile: COMPILE
structure PrettyPrint : PRETTYPRINT
structure Control : CONTROL
structure Tags : TAGS
structure Timer : TIMER
structure Stats : STATS
structure Unsafe : UNSAFE
structure Signals : SIGNALS
structure Directory : DIRECTORY

val exn_name     : exn -> string
val version      : string
val architecture : string ref
val runtimeStamp : string ref
val interactive  : bool ref
val system       : string -> int
val argv         : unit -> string list
val environ      : unit -> string list
```

## DESCRIPTION

Each of the substructures is described in its own section of this chapter. The `val` components are:

**exn_name** *e*

get the name from an exception. Note the following behavior, however:

```
exception E1
exception J = E1
```

`exn_name(E1)` is `"E1"` and `exn_name(J)` is also `"E1"`.

**version**

The version of SML/NJ currently running, e.g.

```
Standard ML of New Jersey, Version 0.93, February 1, 1993
```

**architecture**

the instruction set architure of the executing machine, e.g. `.mipseb` for the big-endian MIPS. Other architectures are `.sparc`, `.mipsel`, `.vax`, `.m68`, `.hppa`, etc.

**runtimeStamp**

an identifier for the current version of the runtime system. May change each time the runtime system is recompiled.

**interactive**
whether the standard input is (nominally) from an interactive source such as a terminal.

**system** *s*
execute the shell *s*, yielding a return code.

**argv ()**
get the command-line arguments

**environ ()**
get the Unix "environment"

## NAME

Timer — execution timing

## SYNOPSIS

signature TIMER
structure System.Timer : TIMER

## SIGNATURE

```
datatype time = TIME of {sec : int, usec : int}
type timer
val start_timer     : unit -> timer
val check_timer     : timer -> time
val check_timer_sys : timer -> time
val check_timer_gc  : timer -> time
val makestring      : time -> string
val add_time        : time * time -> time
val sub_time        : time * time -> time
val earlier         : time * time -> bool
```

## DESCRIPTION

**time**
A time value, divided into seconds (`sec`) and microseconds (`usec`).

**timer**
An abstract time value: the time at which a timer was started.

**start_timer ()**
Start a timer. Any number of timers may be simultaneously active.

**check_timer** $t$
The amount of non-gc user execution time since $t$ was started.

**check_timer_sys** $t$
The amount of operating system execution time since $t$ was started.

**check_timer_gc** $t$
The amount of garbage collection time since $t$ was started.

**makestring** $v$
Convert time $v$ into a decimal string (e.g. `123.103200` for 123 seconds and 103200 microseconds).

**add_time, sub_time**
Arithmetic on time values.

**earlier** $(t_1, t_2)$
True if $t_1$ is less than $t_2$.

## EXAMPLE

```
fun timeit f x =
  let open System.Timer
      val timeofday = System.Unsafe.CInterface.gettimeofday
      val t = start_timer()
      val rt = timeofday()
      val z = f x
      val rt' = sub_time(timeofday(),rt)
      val t' = check_timer t
      val ts = check_timer_sys t
      val tg = check_timer_gc t
   in implode[" ",makestring t'," ",makestring ts," ", makestring tg," ",
              makestring rt'," "]
  end
```

The result of `timeit f x` is that $f$ is applied to $x$, the result is discarded, and a string containing the non-gc user time, the system time, the garbage collection time, and the wallclock time is returned.

## NAME

Unsafe — unsafe features of SML/NJ

## SYNOPSIS

signature UNSAFE
structure System.Unsafe : UNSAFE

## SIGNATURE

```
type object
type instream and outstream
structure Assembly : ASSEMBLY
structure CInterface : CINTERFACE
structure SysIO : SYSIO
structure CleanUp : CLEANUP
structure Weak : WEAK
structure Susp : SUSP
structure PolyCont : POLY_CONT
val boxed : 'a -> bool
val ordof : 'a * int -> int
val slength : 'a -> int
val objLength : 'a -> int
val getObjTag : 'a -> int
val special : (int * 'a) -> 'b
val setSpecial : ('a * int) -> unit
val getSpecial : 'a -> int
val store : string * int * int -> unit
val bstore : Assembly.A.bytearray * int * int -> unit
val subscript : 'a array * int -> 'a
val update : 'a array * int * 'a -> unit
val subscriptv : 'a vector * int -> 'a
val subscriptf : Assembly.A.realarray * int -> real
val updatef : Assembly.A.realarray * int * real -> unit
val getvar : unit -> 'a
val setvar : 'a -> unit
val gethdlr : unit -> 'a
val sethdlr : 'a -> unit
val boot : 'a -> ('b -> 'c)
val cast : 'a -> 'b
val blast_write : outstream * 'a -> int
val blast_read : instream * int -> 'a
val create_s : int -> string
val create_b : int -> Assembly.A.bytearray
val store_s : string * int * int -> unit
val lookup_r : (int -> object) ref
val lookup : int -> object
val toplevelcont : unit cont ref
val pstruct : core: object, initial: object, math: object ref
exception Boxity
val tuple : object -> object vector
val string : object -> string
val real : object -> real
val int : object -> int
datatype datalist = DATANIL
                  | DATACONS of (string * string * datalist)
val datalist : datalist
val profiling : bool ref
```

## DESCRIPTION

Many of the components of `System.Unsafe` can cause Standard ML of New Jersey to dump core, or to use a more technical term, to "go wrong." Furthermore, `System.Unsafe` may be rearranged at the whim of the implementors, and **nothing in here is guaranteed from one release to the next.**

Furthermore, this module is hodgepodge of unsafe features that are of genuine interest to ordinary (but sophisticated) users, and of those that are of use only to implementors of the system.

Each of the substructures is descibed in its own section of this chapter. The value components are:

**boxed** $x$
: true if $x$ is represented as a pointer.

**ordof** $(s, i)$
: just like String.ordof, but without range checking.

**slength** $s$
: just like String.size but may dump core on single-character strings.

**objLength** $x$
: Gives the length of a vector, array, or record; but undefined or may dump core on zero-element or two-element records.

**getObjTag** $x$
: Get the low-order tag bits of a boxed value.

**special** $(i, x)$
: Make a "special" object with tag $i$. Used for weak pointers, lazy thunks, etc.

**setSpecial** $(x, i)$
: Change the tag of special object $x$ to $i$.

**store** $(s, i, c)$
: Change the $i$th character of string $s$ to the character whose code is $c$. Bombs on single-character strings, or on string literals embedded in programs.

**bstore** $(a, i, c)$
: Like ByteArray.update, but without range checking.

**subscript** $(a, i)$
: Like Array.sub, but without range checking.

**update** $(a, i, v)$
: Like Array.update, but without range checking.

**subscriptv** $(v, i)$
: Like Vector.sub, but without range checking.

**subscriptf** $(a, i)$
: Like RealArray.sub, but without range checking.

**updatef** $(a, i, x)$
: Like RealArray.update, but without range checking.

**getvar ()**
: Get the value of the global register variable.

**setvar** *x*
: Set the value of the global register variable.

**gethdlr ()**
: Get the exception-handler continuation.

**sethdlr** *c*
: Set the exception-handler continuation.

**boot** *s*
: Interpret string *s* (which contains machine instructions) as a function.

**cast** *x*
: Treat *x* as a value of a different type.

**blast_write** $(f, x)$
: Write *x* to outstream *f* as a structured value. Returns the number of characters written.

**blast_read** $(f, n)$
: Read *n* characters from instream *f*, and interpret them as a data structure.

**create_s** *n*
: Make an uninitialized string of length *n*.

**create_b** *n*
: Make an uninitialized byte array of length *n*.

## CAVEATS

Indeed.

## NAME

Weak — weak pointers

## SYNOPSIS

signature WEAK
structure System.Unsafe.Weak : WEAK

## SIGNATURE

```
type 'a weak
val weak : 'a -> 'a weak
val strong : 'a weak -> 'a option
```

## DESCRIPTION

A weak pointer to object $x$ is one that, by itself, is not sufficient to keep $x$ alive. If there are no conventional (strong) references to $x$, it will be reclaimed by the collector, and the weak pointer will be zapped.

$\alpha$ **weak**
: the type of weak pointers to things of type $\alpha$.

**weak** $x$ A weak pointer to $x$.

**strong** $p$ returns NONE if $x$ no longer exists. Returns SOME $x$ if $x$ still exists, or if $x$ no longer exists but the garbage collector hasn't gotten to it yet,

Of course, this is semantically ill-defined, especially because the semantics of ML talks about values, not objects.

## CAVEATS

May not be supported forever.