BNF Syntax of ML and SML97 Overview

By K.W. Regan—based on syntax diagrams in J.D. Ullman, *Elements of ML Programming, ML'97 ed.*, but expanded and regrouped with some fixes. ALL-CAPS are used for nonterminals, and all-lowercase for literal keywords. Literal `( ) [ ] { }` are quoted to distinguish them from BNF syntax.

```
PGM ::= EXP; | {DEC | MODULE | ;}    (Semicolon == "compile me now")
DEC ::= val [TVARS] PAT = EXP {and PAT = EXP}
  | val rec [TVARS] ID = fn MATCH {and ID = fn MATCH}   (SML-NJ; Std
  | fun [TVARS] FBIND {and FBIND}     ML'97 has "val [TVARS] rec..")
  | type TBINDS
  | datatype DTBIND {and DTBIND} [withtype TBINDS]
  | datatype ID = datatype LID
  | abstype DTBIND {and DTBIND} [withtype TBINDS] with {DEC[;]} end
  | exception ID [of TYPE | = LID] {and ID [of TYPE | = LID]}
  | local {DEC[;]} in {DEC[;]} end
  | open LID {LID}           dump public structure items into scope
  | infix[r] [INT] ID {ID}   make [right] assoc. infix fn, prec. INT
  | nonfix ID {ID}           "nonfix +;" makes you write "+(x,y)".

TVARS  ::=  TVAR | "("TVAR{,TVAR}")"
TBINDS ::=  [TVARS] ID = TYPE {and [TVARS] ID = TYPE}
DTBIND ::=  [TVARS] ID = ID [of TYPE] {"|" ID [of TYPE]}
FBIND  ::=  ID APAT {APAT} = EXP {"|" ID APAT {APAT} = EXP}
```

Optional TVARS create local scope so type annotations avoid clashing with same-named type variables outside. In FBIND, each occurrence of ID must be the same name each time. Unlike MATCH, FBIND allows atomic patterns separated by spaces. From TYPE to PAT, rule order indicates precedence:

```
TYPE ::= TVAR | RTYPE | BTYPE | LID         (L)ID must name a type
  | TYPE list | TYPE ref | TYPE array | TYPE option | TYPE vector
  | TYPE LID  |  "("TYPE{,TYPE}")" LID      ID is a datatype/abstype
  | TYPE * TYPE                             Builds tuple types
  | TYPE -> TYPE                            Builds function types
  | "("TYPE")"                              Note: (T*T)*T != T*T*T !
RTYPE ::= "{"LABEL: TYPE{, LABEL: TYPE}"}"  Field names-part of type
BTYPE ::= int | real | char | string | unit | exn | word | substring
  | bool | order | TextIO.instream | TextIO.outstream (and BinIO."")
```

Expressions. Any ID, EXP, or PAT below may be followed by a colon : and a legal type; this binds looser than `before` but tighter than `andalso`. Some lines are redundant—for suggestion and clarity.

```
EXP ::= CONST | LID | PREFIXFN | PREFCON    #LABEL passable as a fn
  | #LABEL                          #2 (a,b) = b, #c {c=3, e=4} = 3
  | "{"LABEL=EXP {, LABEL=EXP}"}"   record.  Note {2=a, 1=b} = (b,a)
  | "("EXP{, EXP}")"                tuple.  Note "(EXP)" is legal
  | "("EXP{; EXP}")"                sequence-of-"statements"
  | "["EXP{, EXP}"]"                list of known length
  | "#["EXP{,EXP}"]"                SML-NJ vector: unbreakable list
  | let {DEC[;]} in EXP{;EXP} end   Makes a scope. Last EXP has no ;

  | EXP EXP           Includes PREFIXFN EXP.  Just space, no comma!
  | ! EXP             Dereference--EXP must eval to a ref variable
  | ref EXP           Here EXP must have a non-polymorphic type
  | EXP INFIXFN EXP   See top-level infix ops below, precedence 4-7
  | EXP o EXP         Function composition: literal o. Precedence 3
  | EXP := EXP        First EXP must eval to ref var.  Precedence 3
  | EXP before EXP    Value is lhs; do rhs for side-effects. Prec.0
  | EXP andalso EXP   Short-circuit Booleans are "special", not fns
  | EXP orelse EXP
  | EXP handle MATCH              Catch exn.  EXP may need (...)
  | raise EXP                     Throw an exception.
  | if EXP then EXP else EXP      Just like "EXP ? EXP : EXP" in C
  | while EXP do EXP              EXPs must use !, :=, or similar
  | case EXP of MATCH             Often needs (...) around it
  | fn MATCH                      Best use: anonymous functions
```

Matches and Patterns. Wildcard _ means "ignore"; no (other) ID may appear twice in a PAT.

```
MATCH::=  PAT => EXP {"|" PAT => EXP}
PAT  ::=  APAT                      atomic pattern
  | PREFCON PAT                     pattern with a constructor
  | PAT INFIXCON PAT                includes list pattern x::rest
  | ID as PAT                       binds ID to a match for PAT
APAT ::=  CONST                     includes MONOCONs and MONOEXNs
  | ID | _                          ID is *written to*, not read!
  | "(" PAT{, PAT} ")"              tuple pattern
  | "(" APAT{"|" APAT} ")"          "OR-pattern", in SML-NJ only?
  | "[]" | "[" PAT{,PAT} "]"        list pattern of fixed size
  | "#[]" | "#[" APAT{,APAT} "]"    vector pattern, SML-NJ addition
  | "{" FPAT{, FPAT}[, ...] "}"     record pattern, "..." wildcard
FPAT ::= LABEL=PAT | ID [as PAT]    ID names field & gets its value
```

The Modules System. Factories, not classes. No inheritance or late binding. `S :> SIG` opaques types in S, and `functor Foo(S : BAR) = …` is like Java `class Foo<S extends Bar> {...}`

```
MODULE  ::=  STRDEC | SIGDEC | FUNCTDEC
STRDEC  ::=  DEC  |  local {STRDEC[;]} in {STRDEC[;]} end
  | structure ID [:[>] SIG] = STRUCT {and ID [:[>] SIG] = STRUCT}
STRUCT  ::=  struct {STRDEC[;]} end          (can nest structures!)
  | ID "(" [STRUCT | {STRDEC[;]}] ")"        (functor application)
  | LID  |  let {STRDEC[;]} in STRUCT end    (ID names a structure)

SIGDEC::= signature ID = SIG [QUAL] {and ID = SIG [QUAL]}
SIG   ::= sig {SPEC[;]} end | ID             (ID names a signature)
QUAL  ::= where type [TVARS] ID = TYPE {and type [TVARS] ID = TYPE}
SPEC  ::= val ID : TYPE {and ID : TYPE}
  | [eq]type [TVARS] ID [= TYPE] {and [TVARS] ID [= TYPE]}
  | datatype DTBIND {and DTBIND}
  | datatype ID = datatype LID
  | exception ID [of TYPE] {and ID [of TYPE]}
  | structure ID : SIG {and ID : SIG}
  | sharing [type] LID = LID {= LID} {and [type] LID = LID {= LID}}
  | include ID {ID}                    (inlcudes other signatures)

FUNCTDEC  ::=  functor FUNCTOR {and FUNCTOR}
FUNCTOR   ::=  ID "(" [ID : SIG | SPECS] ")" [:[>] SIG] = STRUCT
```

Identifiers and Literals. Note tilde ~ for unary minus, not -. Chars cannot be written as 'A'.

```
LETTER  ::=  a | b | ... | z | A | B | ... | Z |  /implem.-defined/
SYMBOL  ::=  + - / * < > = ! @ # $ % ^ & ‘ ~ \ ? : "|"
DIGIT   ::=  0 | 1 | ... | 9
HEXDIGIT::=  DIGIT | a | b | c | d | e | f
CHAR    ::=  #"<char>"  e.g. #"A" #" " #"\t" #"\000" #"\255" #"\^a"
STRING  ::=  "anything in quotes"         (use \" for " in strings)
INT     ::=  [~]DIGIT{DIGIT}                      (leading 0s are OK)
REAL    ::=  INT.DIGIT{DIGIT} | INT[.DIGIT{DIGIT}]"E"INT
WORD    ::=  0wDIGIT{DIGIT}  |  0wxHEXDIGIT{HEXDIGIT}

LID     ::=  {ID.}ID     (long ID, prefixed by (sub)structure names)
ID      ::=  ALPHID | SYMBID
ALPHID  ::=  LETTER{LETTER | ’ | _}          (minus reserved words)
SYMBID  ::=  SYMBOL{SYMBOL}                  (minus reserved words)
LABEL   ::=  ALPH_ID | SYMB_ID | DIGIT{DIGIT}
TVAR    ::=  ’ALPH_ID | ’’ALPH_ID            (’’ for equality type)
CONST   ::=  INT | REAL| CHAR | STRING | WORD | () | MONOCON|MONOEXN
INFIXFN ::=  *  |  /  |  div  |  mod                  (precedence 7)
          |  +  |  -  |  ^                          (precedence 6)
          |  @                 (list cons :: also has precedence 5)
          |  =  |  <>  |  <  |  >  |  <=  |  >=      (precedence 4)
          |  :=  |  o                               (precedence 3)
          |  ID                          (defaults to precedence 0)
PREFIXFN::=  op INFIXFN  |  !  |  ~  |  not  |  abs  |  LID
PREFCON ::=  "SOME"  |  Fail : string->exn  |  LID
INFIXCON::=  ::  |  ID  (infix ID in structure is prefix outside it)
MONOCON ::=  true|false|nil| "NONE" | "LESS"|"EQUAL"|"GREATER" | LID
MONOEXN ::=  Bind | Chr | Div | Domain | Empty | Match | Option
          |  Overflow | Size | Span | Subscript | LID
```

Other Top-Level Standard Basis Functions. Each is given with its type signature.

```
floor,ceil,trunc,round : real -> int;        real : int -> real;
concat : string list -> string;               chr : int -> char;
explode : string -> char list;                ord : char -> int;
implode : char list -> string;                str : char -> string;
substring : string*int*int -> string;        size : string -> int;
exnName,exnMessage : exn -> string;         print : string -> unit;
use : string -> unit   (compile a file)
hd : ’a list -> ’a;                          tl : ’a list -> ’a list;
null : ’a list -> bool;                     rev : ’a list -> ’a list;
map : (’a -> ’b) -> ’a list -> ’b list;     length : ’a list -> int;
app : (’a -> unit) -> ’a list -> unit;      vector:’a list->’a vector;
foldl,foldr : (’a * ’b -> ’b) -> ’b -> ’a list -> ’b list;
isSome : ’a option -> bool;                 valOf : ’a option -> ’a;
getOpt : ’a option * ’a -> ’a;              ignore : ’a -> unit;
```

Prefer `if (null L)` … to `if (L = nil)` … since the latter forces `L` to have an equality type. Use `map` to apply a function to each element in a list, and note `foldr op @ y [x1,x2,...,xn] = x1 @ (x2 @ (... @ (xn @ y)...))`, and `foldl f y L = foldr f y (rev L)`. Note that `op` binds super-tight. Strings index from 0, so `substring("abcdef",2,3)` = `"cde"`.

File I/O. `let val IN=TextIO.openIn("dir/foo.txt"); val OUT=TextIO.stdOut; val w=TextIO.inputLine(IN) in TextIO.output (OUT,"Hi"); TextIO.output (valOf w) end;` shows sample usage. Structure `BinIO` has similar for `Word8.word`.

```
TextIO.stdIn: TextIO.instream TextIO.stdOut,stdErr: TextIO.outstream
TextIO.openIn : string -> instream    open file, can give full path
TextIO.openOut: string -> outstream   flushes file if already exists
TextIO.openAppend: ""  -> outstream   preserves contents and appends
TextIO.input1:instream -> char option gives SOME c or NONE if at EOF
TextIO.inputN:instream*int -> string  read N chars or all chs to EOF
TextIO.input: instream -> string      reads next "batch" of chars
TextIO.inputAll:instream->string      reads stream to EOF
TextIO.lookahead:instream->char option    does not consume next char
TextIO.inputLine:instream->string option  =NONE at EOF, not in BinIO
TextIO.endOfStream : instream -> bool     blocks if stream stalled
TextIO.output : outstream*string->unit    write a whole string
TextIO.output1: outstream*char -> unit
TextIO.closeIn : instream -> unit         often automatic on exit
TextIO.closeOut: outstream -> unit
TextIO.flushOut: outstream -> unit        does not close the stream
IO.Io : exn                               NOTE: input1 consumes EOF
```

Standard Basis Structures and Contents. Here `val arr = Array.array(i,v)` equals Java `T[] arr` = $\{v,\ldots,v\}$ with i-many occurrences of the same value v (of type T), and `Array.sub(arr,i)` is clunky syntax for `arr[i]`. `Array.update(arr,i,x)` executes `arr[i] = x` by reference, while in SML-NJ, `Vector.update(vec,i,x)` returns a copy of vector `vec` with `x` in field `i`. `Array` and `Vector` also have variations of `foldl` and `foldr`. `Array2` gives 2D arrays. Our SML-NJ nonstandardly allows pattern-matching on (fixed-length!) vectors, but omits `extract` (for taking slices of vectors) from the SML library.

```
Int.toString : int -> string; Real, Bool, Word8, IntInf have similar
Int.rem,quot,max,min: int*int -> int;          Int.sign: int -> int;
Int.minInt,maxInt: int;    See also structures LargeInt,IntInf,Word8
Word.toInt,toIntX: word -> int;           Word.fromInt: int -> word;
Word.andb,orb,xorb,<<,>>: word*word -> word;        (all are prefix)
Real.sign: real -> int;           Real. max,min,: real*real -> real;
Real.Math.sin,cos,atan,sqrt,exp,ln: real->real; pow: real*real->real
Char.isAlpha,isDigit,isSpace,isPrint: char-> bool;
Char.toLower,toUpper: char -> char;
Char.contains: string -> char -> bool         (WideChar has similar)
String.maxLen: int;               String.sub: string * int -> char;
String.tokens: (char->bool) -> string -> string list;  (for parsing)
List.take,drop: ’a list*int -> ’a list; List.nth: ’a list*int -> ’a;
List.filter: (’a -> bool) -> ’a list -> ’a list;
Array.array: int * ’a -> ’a array;    Array.length: ’a array -> int;
Array.sub: ’a array * int -> ’a;
Array.update: ’a array * int * ’a -> unit;
Array.fromList: ’a list -> ’a array;
Array.vector: ’a array -> ’a vector;
Vector.concat: ’a vector list -> ’a vector
Time.toReal: Time.time -> real;    Time.fromReal: real -> Time.time;
Timer.startRealTimer: unit -> real_timer;    Timer.startCPUTimer: ""
Timer.checkRealTimer: real_timer->Time.time; Timer.checkCPUTimer: ""
```