2 Syntax of the Core

2.1 Reserved Words

The following are the *reserved words* used in the Core. They may not (except `=`) be used as identifiers.

```
abstype   and   andalso   as   case   datatype   do   else
end   exception   fn   fun   handle   if   in   infix
infixr   let   local   nonfix   of   op   open   orelse
raise   rec   then   type   val   with   withtype   while
(  )   [  ]   {  }   ,   :   ;   ...   _   |   =   =>   ->   #
```

2.2 Special constants

An *integer constant (in decimal notation)* is an optional negation symbol (`~`) followed by a non-empty sequence of decimal digits `0`,..,`9`. An *integer constant (in hexadecimal notation)* is an optional negation symbol followed by `0x` followed by a non-empty sequence of hexadecimal digits `0`,..,`9` and `a`,..,`f`. (`A`,..,`F` may be used as alternatives for `a`,..,`f`.)

A *word constant (in decimal notation)* is `0w` followed by a non-empty sequence of decimal digits. A *word constant (in hexadecimal notation)* is `0wx` followed by a non-empty sequence of hexadecimal digits. A *real constant* is an integer constant in decimal notation, possibly followed by a point (`.`) and one or more decimal digits, possibly followed by an exponent symbol (`E` or `e`) and an integer constant in decimal notation; at least one of the optional parts must occur, hence no integer constant is a real constant. Examples: `0.7  3.32E5  3E~7` . Non-examples: `23  .3  4.E5  1E2.0` .

We assume an underlying alphabet of N characters ($N \geq 256$), numbered 0 to $N-1$, which agrees with the ASCII character set on the characters numbered 0 to 127. The interval $[0, N-1]$ is called the *ordinal range* of the alphabet. A *string constant* is a sequence, between quotes (`"`), of zero or more printable characters (i.e., numbered 33–126), spaces or escape sequences. Each escape sequence starts with the escape character `\` , and stands for a character sequence. The escape sequences are:

`\a`	A single character interpreted by the system as alert (ASCII 7)
`\b`	Backspace (ASCII 8)
`\t`	Horizontal tab (ASCII 9)
`\n`	Linefeed, also known as newline (ASCII 10)
`\v`	Vertical tab (ASCII 11)
`\f`	Form feed (ASCII 12)
`\r`	Carriage return (ASCII 13)
`\^`*c*	The control character *c*, where *c* may be any character with number 64–95. The number of `\^`*c* is 64 less than the number of *c*.
`\`*ddd*	The single character with number *ddd* (3 decimal digits denoting an integer in the ordinal range of the alphabet).

\u*xxxx*	The single character with number *xxxx* (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
\"	"
\\	\
\$f\cdots f$\	This sequence is ignored, where $f\cdots f$ stands for a sequence of one or more formatting characters.

The *formatting characters* are a subset of the non-printable characters including at least space, tab, newline, formfeed. The last form allows long strings to be written on more than one line, by writing \ at the end of one line and at the start of the next.

A *character constant* is a sequence of the form #*s*, where *s* is a string constant denoting a string of size one character.

Libraries may provide multiple numeric types and multiple string types. To each string type corresponds an alphabet with ordinal range $[0, N-1]$ for some $N \geq 256$; each alphabet must agree with the ASCII character set on the characters numbered 0 to 127. When multiple alphabets are supported, all characters of a given string constant are interpreted over the same alphabet. For each special constant, overloading resolution is used for determining the type of the constant (see Appendix E).

We denote by SCon the class of *special constants*, i.e., the integer, real, word, character and string constants; we shall use *scon* to range over SCon.

2.3 Comments

A *comment* is any character sequence within comment brackets (* *) in which comment brackets are properly nested. No space is allowed between the two characters which make up a comment bracket (* or *). An unmatched (* should be detected by the compiler.

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 1. We use *vid*, *tyvar* to range over VId, TyVar etc. For each class X marked "long" there is a class longX of *long identifiers*; if *x* ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

longx	::=	*x*	identifier
		$strid_1.\cdots.strid_n.x$	qualified identifier ($n \geq 1$)

VId	(value identifiers)	long
TyVar	(type variables)	
TyCon	(type constructors)	long
Lab	(record labels)	
StrId	(structure identifiers)	long

Figure 1: Identifiers

The qualified identifiers constitute a link between the Core and the Modules. Throughout this document, the term "identifier", occurring without an adjective, refers to non-qualified identifiers only.

An identifier is either *alphanumeric*: any sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime, or *symbolic*: any non-empty sequence of the following *symbols*

```
!  %  &  $  #  +  -  /  :  <  =  >  ?  @  \  ~  `  ^  |  *
```

In either case, however, reserved words are excluded. This means that for example `#` and `|` are not identifiers, but `##` and `|=|` are identifiers. The only exception to this rule is that the symbol `=`, which is a reserved word, is also allowed as an identifier to stand for the equality predicate. The identifier `=` may not be re-bound; this precludes any syntactic ambiguity.

A type variable *tyvar* may be any alphanumeric identifier starting with a prime; the subclass EtyVar of TyVar, the *equality* type variables, consists of those which start with two or more primes. The classes VId, TyCon and Lab are represented by identifiers not starting with a prime. However, `*` is excluded from TyCon, to avoid confusion with the derived form of tuple type (see Figure 23). The class Lab is extended to include the *numeric* labels `1 2 3` ⋯, i.e. any numeral not starting with `0`. The identifier class StrId is represented by alphanumeric identifiers not starting with a prime.

TyVar is therefore disjoint from the other four classes. Otherwise, the syntax class of an occurrence of identifier *id* in a Core phrase (ignoring derived forms, Section 2.7) is determined thus:

1. Immediately before "." – i.e. in a long identifier – or in an `open` declaration, *id* is a structure identifier. The following rules assume that all occurrences of structure identifiers have been removed.

2. At the start of a component in a record type, record pattern or record expression, *id* is a record label.

3. Elsewhere in types *id* is a type constructor.

4. Elsewhere, *id* is a value identifier.

By means of the above rules a compiler can determine the class to which each identifier occurrence belongs; for the remainder of this document we shall therefore assume that the classes are all disjoint.

2.5 Lexical analysis

Each item of lexical analysis is either a reserved word, a numeric label, a special constant or a long identifier. Comments and formatting characters separate items (except within string constants; see Section 2.2) and are otherwise ignored. At each stage the longest next item is taken.

2.6 Infixed operators

An identifier may be given *infix status* by the `infix` or `infixr` directive, which may occur as a declaration; this status only pertains to its use as a *vid* within the scope (see below) of the directive, and in these uses it is called an *infixed operator*. (Note that qualified identifiers never have infix status.) If *vid* has infix status, then "exp_1 *vid* exp_2" (resp. "pat_1 *vid* pat_2") may occur – in parentheses if necessary – wherever the application "*vid*`{1=`exp_1`,2=`exp_2`}`" or its derived form "*vid*`(`exp_1`,`exp_2`)`" (resp "*vid*`(`pat_1`,`pat_2`)`") would otherwise occur. On the other hand, an occurrence of any long identifier (qualified or not) prefixed by `op` is treated as non-infixed. The only required use of `op` is in prefixing a non-infixed occurrence of an identifier *vid* which has infix status; elsewhere `op`, where permitted, has no effect. Infix status is cancelled by the `nonfix` directive. We refer to the three directives collectively as *fixity directives*.

The form of the fixity directives is as follows ($n \geq 1$):

$$\texttt{infix}\ \langle d \rangle\ vid_1 \cdots vid_n$$

$$\texttt{infixr}\ \langle d \rangle\ vid_1 \cdots vid_n$$

$$\texttt{nonfix}\ vid_1 \cdots vid_n$$

where $\langle d \rangle$ is an optional decimal digit d indicating binding precedence. A higher value of d indicates tighter binding; the default is `0`. `infix` and `infixr` dictate left and right associativity respectively. In an expression of the form $exp_1\ vid_1\ exp_2\ vid_2\ exp_3$, where vid_1 and vid_2 are infixed operators with the same precedence, either both must associate to the left or both must associate to the right. For example, suppose that `<<` and `>>` have equal precedence, but associate to the left and right respectively; then

```
x << y << z   parses as   (x << y) << z
x >> y >> z   parses as   x >> (y >> z)
x << y >> z   is illegal
x >> y << z   is illegal
```

The precedence of infixed operators relative to other expression and pattern constructions is given in Appendix B.

The *scope* of a fixity directive *dir* is the ensuing program text, except that if *dir* occurs in a declaration *dec* in either of the phrases

$$\texttt{let}\ dec\ \texttt{in} \cdots \texttt{end}$$

$$\texttt{local}\ dec\ \texttt{in} \cdots \texttt{end}$$

then the scope of *dir* does not extend beyond the phrase. Further scope limitations are imposed for Modules (see Section 3.3).

These directives and `op` are omitted from the semantic rules, since they affect only parsing.

AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
Mrule	match rules
Dec	declarations
ValBind	value bindings
TypBind	type bindings
DatBind	datatype bindings
ConBind	constructor bindings
ExBind	exception bindings
AtPat	atomic patterns
PatRow	pattern rows
Pat	patterns
Ty	type expressions
TyRow	type-expression rows

Figure 2: Core Phrase Classes

2.7 Derived Forms

There are many standard syntactic forms in ML whose meaning can be expressed in terms of a smaller number of syntactic forms, called the *bare* language. These derived forms, and their equivalent forms in the bare language, are given in Appendix A.

2.8 Grammar

The phrase classes for the Core are shown in Figure 2. We use the variable *atexp* to range over AtExp, etc. The grammatical rules for the Core are shown in Figures 3 and 4.

The following conventions are adopted in presenting the grammatical rules, and in their interpretation:

- The brackets $\langle\ \rangle$ enclose optional phrases.
- For any syntax class X (over which x ranges) we define the syntax class Xseq (over which *xseq* ranges) as follows:

$$\begin{array}{lcll} xseq & ::= & x & \text{(singleton sequence)} \\ & & & \text{(empty sequence)} \\ & & (x_1, \cdots, x_n) & \text{(sequence, } n \geq 1\text{)} \end{array}$$

(Note that the "$\cdots$" used here, meaning syntactic iteration, must not be confused with "..." which is a reserved word of the language.)

- Alternative forms for each phrase class are in order of decreasing precedence; this resolves ambiguity in parsing, as explained in Appendix B.
- L (resp. R) means left (resp. right) association.
- The syntax of types binds more tightly than that of expressions.
- Each iterated construct (e.g. *match*, ···) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. "`fn` *match*", if this occurs within a larger match.

atpat	::=		wildcard
		scon	special constant
		⟨`op`⟩*longvid*	value identifier
		{ ⟨*patrow*⟩ }	record
		(*pat*)	
patrow	::=	...	wildcard
		lab = *pat* ⟨ , *patrow*⟩	pattern row
pat	::=	*atpat*	atomic
		⟨`op`⟩*longvid atpat*	constructed pattern
		pat_1 *vid* pat_2	infixed value construction
		pat : *ty*	typed
		⟨`op`⟩*vid*⟨: *ty*⟩ `as` *pat*	layered
ty	::=	*tyvar*	type variable
		{ ⟨*tyrow*⟩ }	record type expression
		tyseq longtycon	type construction
		ty `->` *ty′*	function type expression (R)
		(*ty*)	
tyrow	::=	*lab* : *ty* ⟨ , *tyrow*⟩	type-expression row

Figure 3: Grammar: Patterns and Type expressions

2.9 Syntactic Restrictions

- No expression row, pattern row or type-expression row may bind the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this applies also to value identifiers within a *datbind*.
- No *tyvarseq* may contain the same *tyvar* twice.
- For each value binding *pat* = *exp* within `rec`, *exp* must be of the form `fn` *match*. The derived form of function-value binding given in Appendix A, page 57, necessarily obeys this restriction.

- No *datbind*, *valbind* or *exbind* may bind `true`, `false`, `nil`, `::` or `ref`. No *datbind* or *exbind* may bind `it`.
- No real constant may occur in a pattern.
- In a value declaration `val` *tyvarseq valbind*, if *valbind* contains another value declaration `val` *tyvarseq′ valbind′* then *tyvarseq* and *tyvarseq′* must be disjoint. In other words, no type variable may be scoped by two value declarations of which one occurs inside the other. This restriction applies after *tyvarseq* and *tyvarseq′* have been extended to include implicitly scoped type variables, as explained in Section 4.6.

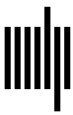
atexp	::=	*scon*	special constant
		⟨op⟩*longvid*	value identifier
		{ ⟨*exprow*⟩ }	record
		`let` *dec* `in` *exp* `end`	local declaration
		(*exp*)	
exprow	::=	*lab* = *exp* ⟨ , *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp_1 *vid* exp_2	infixed application
		exp : *ty*	typed (L)
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ | *match*⟩	
mrule	::=	*pat* => *exp*	
dec	::=	`val` *tyvarseq valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaration
		`datatype` *tycon* = `datatype` *longtycon*	datatype replication
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration ($n \geq 1$)
			empty declaration
		dec_1 ⟨;⟩ dec_2	sequential declaration
		`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (L) directive
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (R) directive
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive
valbind	::=	*pat* = *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
typbind	::=	*tyvarseq tycon* = *ty* ⟨`and` *typbind*⟩	
datbind	::=	*tyvarseq tycon* = *conbind* ⟨`and` *datbind*⟩	
conbind	::=	⟨op⟩*vid* ⟨`of` *ty*⟩ ⟨ | *conbind*⟩	
exbind	::=	⟨op⟩*vid* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		⟨op⟩*vid* = ⟨op⟩*longvid* ⟨`and` *exbind*⟩	

Figure 4: Grammar: Expressions, Matches, Declarations and Bindings

The Definition of Standard ML

By: Robin Milner, Robert Harper, David MacQueen, Mads Tofte

Citation:
The Definition of Standard ML
By: Robin Milner, Robert Harper, David MacQueen, Mads Tofte
DOI: 10.7551/mitpress/2319.001.0001
ISBN (electronic): 9780262287005
Publisher: The MIT Press
Published: 1997



The MIT Press

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

The definition of standard ML: revised / Robin Milner ... et al.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-63181-4 (alk. paper)
1. ML (Computer program language) I. Milner, R. (Robin), 1934-
QA76.73.M6D44 1997
005.13'3—dc21

97-59
CIP

10 9 8 7 6 5 4 3 2