B Appendix: Full Grammar

The full grammar of programs is exactly as given at the start of Section 8.

The full grammar of Modules consists of the grammar of Figures 5–8 in Section 3, together with the derived forms of Figures 18 and 19 in Appendix A.

The remainder of this Appendix is devoted to the full grammar of the Core. Roughly, it consists of the grammar of Section 2 augmented by the derived forms of Appendix A. But there is a further difference: two additional subclasses of the phrase class Exp are introduced, namely AppExp (application expressions) and InfExp (infix expressions). The inclusion relation among the four classes is as follows:

$$\mathrm{AtExp} \subset \mathrm{AppExp} \subset \mathrm{InfExp} \subset \mathrm{Exp}$$

The effect is that certain phrases, such as "`2 + while` $\cdots$ `do` $\cdots$ ", are now disallowed.

The grammatical rules are displayed in Figures 20, 21, 22 and 23. The grammatical conventions are exactly as in Section 2, namely:

- 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:

$xseq$	::=	x	(singleton sequence)
			(empty sequence)
		$(x_1,\cdots,x_n)$	(sequence, $n \geq 1$)

  (Note that the "$\cdots$" used here, a meta-symbol indicating syntactic repetition, 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 precedence resolves ambiguity in parsing in the following way. Suppose that a phrase class — we take exp as an example — has two alternative forms F_1 and F_2, such that F_1 ends with an exp and F_2 starts with an exp. A specific case is

  F_1: **if** exp_1 **then** exp_2 **else** exp_3
  F_2: exp **handle** $match$

  It will be enough to see how ambiguity is resolved in this specific case.

  Suppose that the lexical sequence

  $\cdots\ \cdots$ **if** $\cdots$ **then** $\cdots$ **else** exp **handle** $\cdots\ \cdots$

  is to be parsed, where exp stands for a lexical sequence which is already determined as a subphrase (if necessary by applying the precedence rule). Then the higher precedence of F_2 (in this case) dictates that *exp* associates to the right, i.e. that the correct parse takes the form

`··· ··· if ··· then ··· else (`*exp* `handle ···) ···`

not the form

`··· (··· if ··· then ··· else` *exp*`) handle ··· ···`

Note particularly that the use of precedence does not decrease the class of admissible phrases; it merely rejects alternative ways of parsing certain phrases. In particular, the purpose is not to prevent a phrase, which is an instance of a form with higher precedence, having a constituent which is an instance of a form with lower precedence. Thus for example

`if ··· then while ··· do ··· else while ··· do ···`

is quite admissible, and will be parsed as

`if ··· then (while ··· do ···) else (while ··· do ···)`

- 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.

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvid*	value identifier
		`{` ⟨*exprow*⟩ `}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		`(`exp_1 `,` ⋯ `,` exp_n`)`	n-tuple, $n \geq 2$
		`[`exp_1 `,` ⋯ `,` exp_n`]`	list, $n \geq 0$
		`(`exp_1 `;` ⋯ `;` exp_n`)`	sequence, $n \geq 2$
		`let` *dec* `in` exp_1 `;` ⋯ `;` exp_n `end`	local declaration, $n \geq 1$
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
appexp	::=	*atexp*	
		appexp atexp	application expression
infexp	::=	*appexp*	
		$infexp_1$ *vid* $infexp_2$	infix expression
exp	::=	*infexp*	
		exp `:` *ty*	typed (L)
		exp_1 `andalso` exp_2	conjunction
		exp_1 `orelse` exp_2	disjunction
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`if` exp_1 `then` exp_2 `else` exp_3	conditional
		`while` exp_1 `do` exp_2	iteration
		`case` *exp* `of` *match*	case analysis
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	

Figure 20: Grammar: Expressions and Matches

dec	::=	`val` *tyvarseq valbind*	value declaration
		`fun` *tyvarseq fvalbind*	function declaration
		`type` *typbind*	type declaration
		`datatype` *datbind* ⟨`withtype` *typbind*⟩	datatype declaration
		`datatype` *tycon* `=` `datatype` *longtycon*	datatype replication
		`abstype` *datbind* ⟨`withtype` *typbind*⟩ `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, $n \geq 1$
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (R) directive, $n \geq 1$
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive, $n \geq 1$
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
fvalbind	::=	⟨`op`⟩*vid* $atpat_{11} \cdots atpat_{1n}$⟨`:` *ty*⟩`=`$exp_1$	$m, n \geq 1$
		`|`⟨`op`⟩*vid* $atpat_{21} \cdots atpat_{2n}$⟨`:` *ty*⟩`=`$exp_2$	See also note below
		`|` $\cdots$ $\cdots$	
		`|`⟨`op`⟩*vid* $atpat_{m1} \cdots atpat_{mn}$⟨`:` *ty*⟩`=`$exp_m$ ⟨`and` *fvalbind*⟩	
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*⟩	

Note: In the *fvalbind* form, if *vid* has infix status then either `op` must be present, or *vid* must be infixed. Thus, at the start of any clause, "`op` *vid* `(`*atpat*`,` *atpat′*`)` ⋯" may be written "`(`*atpat vid atpat′*`)` ⋯"; the parentheses may also be dropped if "`:` *ty*" or "`=`" follows immediately.

Figure 21: Grammar: Declarations and Bindings

atpat	::=	_	wildcard
		scon	special constant
		⟨op⟩*longvid*	value identifier
		{ ⟨*patrow*⟩ }	record
		()	0-tuple
		(pat_1 , ⋯ , pat_n)	n-tuple, $n \geq 2$
		[pat_1 , ⋯ , pat_n]	list, $n \geq 0$
		(*pat*)	
patrow	::=	...	wildcard
		lab = *pat* ⟨ , *patrow*⟩	pattern row
		vid⟨: *ty*⟩ ⟨as *pat*⟩ ⟨, *patrow*⟩	label as variable
pat	::=	*atpat*	atomic
		⟨op⟩*longvid atpat*	constructed value
		pat_1 *vid* pat_2	constructed value (infix)
		pat : *ty*	typed
		⟨op⟩*vid*⟨: *ty*⟩ as *pat*	layered

Figure 22: Grammar: Patterns

ty	::=	*tyvar*	type variable
		{ ⟨*tyrow*⟩ }	record type expression
		tyseq longtycon	type construction
		ty_1 * ⋯ * ty_n	tuple type, $n \geq 2$
		ty -> *ty′*	function type expression (R)
		(*ty*)	
tyrow	::=	*lab* : *ty* ⟨ , *tyrow*⟩	type-expression row

Figure 23: Grammar: Type expressions

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

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