Parsing in the SML Kit

Nick Rothwell

September 10, 1992

1 Introduction

This document describes the parsing scheme of the SML Kit. We introduce the grammar datatype, or abstract syntax, and the grammar specification itself, explaining the conventions used. There are various mechanisms employed to resolve the ambiguities and parsing difficulties of SML, and we describe these together with the mechanisms which deal with infix operators, derived forms, and so on.

We use the **ML-Lex**[1] and **ML-Yacc**[2] packages. Both have been modified to produce portable code running under the SML Library[3], rather than SML/NJ-specific code, so that the resulting parser can be compiled under other SML systems. In fact, these versions of **ML-Lex** and **ML-Yacc** are themselves portable, so that all Kit development work can take place using a different SML compiler.

The generated lexing and parsing functors are supported by a number of utility functors for dealing with infixes, derived-form elimination, and so on. There are also utility functors for lexing, providing positional information for error messages, and so on. All these components are gathered together in the functor `Parse` (Section 6).

Section 2 describes the abstract syntax and its parameterisation. Section 3 covers lexing issues, and Section 4 overviews the general structure of the parser. We cover the grammar specification in Section 5, and the functor structure and linkage in Section 6. The constituent files of the parsing system are described in Section 7.

At this stage we should probably mention the main thing that the parsing phase does *not* do. The Definition states a number of syntactic restrictions (Section 2.9), mainly concerned with repeated identifiers of various kinds. In fact, some of these restrictions are *not* syntactic since they are impossible to enforce at parse time, and require elaboration. Specifically, repeated identifiers in value bindings and patterns cannot be detected at parse time since it is legal to repeat value and exception constructors, and these cannot be determined before elaboration. We have chosen to enforce *none* of the rules about repeated identifiers, for the sake of simplicity; it is cleaner to deal with these errors during elaboration. The one remaining syntactic restriction involving `val rec` is truly syntactic and can be dealt with in the grammar.

2 The Abstract Syntax

The abstract syntax in the Kit is split into two levels: the abstract syntax of the Core and that of the Modules. This is the same division as found in the Definition, and with the same provisos: qualified identifiers and `open` declarations are part of the Core syntax.

The Core abstract syntax provides the language phrases up to the level of declarations (*dec*), and the Modules abstract syntax provides the phrases up to top-level declarations (*topdec*). There is no type representing the phrase class *program*; instead, the top-level loop of the interpreter is responsible for parsing and evaluating a sequence of top-level declarations.

The Core and Modules abstract syntax are each a set of mutually recursive datatypes, parameterised on a number of more primitive types representing constants, identifiers of various kinds, and in general the leaves of the syntax. In addition, the Modules abstract syntax is parameterised on the datatype of declarations, *dec*.

An important aspect of any SML implementation is the classification of identifiers. In SML there is no lexical distinction between variables, data constructors and exception constructors; they are all conform to the same lexical convention. In addition, it is not always possible for the parser to infer the class of an identifier from its syntactic context. The parser therefore generates an abstract syntax tree with many of the identifiers unresolved.

The later stages of the interpreter require that all identifiers be resolved. One way of doing this is to decorate the abstract syntax with an indication of the class of each ambiguous identifier. However, there is another method which requires no decoration and is also type-secure, in that the later stages of the interpreter can only ever be passed abstract syntax which has had the identifiers resolved. This is done by parameterising the abstract syntax on the type of identifiers. The abstract syntax functor is used twice: once with generic identifiers, and once with resolved identifiers (represented as a datatype). The result is two distinct abstract syntax types, one of which can have its identifier type shared with the resolved identifier datatype.

The abstract syntax tree produced by the parser is unresolved, and contains the following type identities:

$$\texttt{TopdecGrammar.id} = \texttt{DecGrammar.id} = \texttt{Basics.Ident.id}$$
$$\texttt{DecGrammar.longid} = \texttt{Basics.Ident.longid}$$

The abstract syntax produced by the elaboration is resolved, and contains the identities

$$\texttt{TopdecGrammar.id} = \texttt{DecGrammar.id} = \texttt{Basics.Var.var}$$
$$\texttt{DecGrammar.longid} = \texttt{ResIdent.longid}$$

where

```
datatype longid = LONGVAR of longvar
                | LONGCON of longcon
                | LONGEXCON of longexcon
```

Corresponding to the two levels of abstract syntax in the Kit, there are two signatures and two functors. `DEC_GRAMMAR` and `DecGrammar` provide the abstract syntax for the Core, and `TOPDEC_GRAMMAR` and `TopdecGrammar` the abstract syntax for Modules. The signatures contain the full datatype specifications (since the interfaces to the abstract syntax modules are essentially the datatypes themselves). `DecGrammar` is parameterised on the types `id` and `longid` as described above, on a type `GrammarInfo` which provides the *info* nodes on the syntax tree, and on structures which provide the leaves of the syntax tree (`LAB` for record labels, `SCON` for special constants, `CON` for data constructors, `TYVAR` for type variables, `TYCON` for type constructors, `EXCON` for exception constructors, `STRID` for structure identifiers).[1] `TopdecGrammar` is parameterised on a similar set of types (`strid`, `longstrid`, `funid`, `sigid`, `id`, `tyvar`, `tycon`, `longtycon`, `con`, `excon`, `GrammarInfo`) as well as two syntax types from the Core language (`dec` and `ty`).

The functors are applied in two places in the linking stage. In the first (functor `TopdecParsing`), the unresolved abstract syntax is built along with the parser and the pretty-printers for the syntax. In the second (`Elaboration`) the abstract syntax is built in terms of the resolved identifiers, together with the elaborator and pretty-printers for the resolved syntax.

3 Lexing

The lexer is built using **ML-Lex**. The resulting functor is parameterised on a token specification produced by **ML-Yacc**, and a couple of utility functors (`LexBasics` and `LexUtils`) described below. The lexical specification is fairly straightforward. The lexer carries an abstract argument (of type `LexUtils.LexArgument`) which is used to accumulate string constants and to keep track of comment depth. Comments and strings are consumed in separate lexing states (`C` and `S`).

The lexing specification does not contain keywords, since these would add complexity to the specification and would result in large lexing tables. Instead, the lexer generates generic identifiers which are translated into keywords in module `LexUtils`.

[1] `DecGrammar` does not actually need to take all these as structure arguments; many could just be types.

There is some complexity in the handling of the source character stream, for two reasons. Firstly, the lexing specification needs to provide a type `pos` required by **ML-Lex** and **ML-Yacc** so that the generated lexer and parser fit together, with the positional information of the tokens finding its way into the abstract syntax. Secondly, these `pos` objects have to be converted back into values suitable for printing, including the input file name, line and column number, and the actual source text. This is all done using the abstract stream package provided by `LexBasics`.

Firstly, we consider the type of positions, `pos`. A position is either `DUMMY` (a marker used for end-of-file), or is a suspension containing a function which returns the current file name, line, column and a function mapping line numbers to lines of source text. The suspension is for efficiency; the overhead of position calculation is incurred when a position is printed, rather than for each token.

`SourceReader` is the type of input streams. A source reader is composed of a lexing function to produce the source text and a function mapping absolute character number to position. Source readers are built using `lexFromFile` (to read source text from a file) or `lexFromString` (to read from a string).

The generated lexer has access to the absolute character position of each token as it is recognised. From these, a position object (of type `pos`) can be built for the start and end of each token. This is the linkage required between the **ML-Lex**-generated lexer and **ML-Yacc**-generated parser, and results in abstract syntax whose tokens contain positional information. This information is propagated into the info fields of the interior nodes of the abstract syntax in the actions of the grammar (Section 5). Since a `pos` object can be transformed into its file name, line number, column number, and line-printing function, the original source position and surrounding lines of text can be reconstituted and printed (using `printPosition` and `displaySource`) from any part of the abstract syntax.

The purpose of the module `LexBasics` is to provide source readers and the `pos` type. `LexUtils` provides operations on the lexer argument (type `LexArgument`, aliased to type `arg` in order to interface to **ML-Lex** and **ML-Yacc**), and general functions to build tokens. The lexer argument contains the current source reader, together with a record of the current comment level and accumulated characters for string literals.

Source readers are important in the implementation of the `use` function for reading and executing SML source from files. Each invocation of `use` causes a new source reader to be created for the used source file. If an error is detected in the file, then the source reader can be discarded. The implementation and treatment of `use` is in fact quite complex, and described in detail in a separate document [4].

4 Parsing

Syntactically, SML is a nightmare. Parts of the syntax require indefinite lookahead, so SML is certainly not LALR/1. The meaning of the symbols "`=`" and "`*`" changes according to context, and the requirement of user-definable infixes (which can include "`=`" and "`*`") makes the situation even more complex. The grammar described in the Definition is ambiguous in places (specifically, in the treatment of "`local`"), and the description of some of the derived forms is inconsistent with descriptions of the syntax elsewhere in the Definition, as described below.

Some of these ambiguities are dealt with the the grammar itself, by using new nonterminals to impose precedence in the usual way. We detail these cases in our description of the grammar in Section 5. The other ambiguities are addressed in the following sub-sections.

Because of the above problems (and others), it is impossible to parse SML directly without using a specially constructed parser which can carry a complex left state encompassing the infix environment and so on. Because we choose to use **ML-Yacc** for parsing, we are forced to adopt an LALR/1 grammar which accepts a clean superset of SML, and then remove illegal constructions either in the rule actions or in a separate post-pass.

In passing, we should point out a couple of inconsistencies in the description of SML's syntax in the Definition. The first is minor, and involves the documentation of derived forms of the Core, as given in Appendix A, figures 15 and 16. We consider the derived forms "`()`", "`#`*lab*" and `[`exp_1`,` ... exp_n`]` to be atomic expressions, even if their equivalent forms are not. (Such equivalent forms are considered parenthesised.) This agrees with the full grammar in figure 19.

The second ambiguity involves `val rec` declarations. The Definition states that the right hand side of a `val rec` must be of the form "`fn` *match*", possibly constrained by one or more type expressions (Section 2.9, page 9). This is inconsistent: it is not possible to constrain an entire `fn` expression without bracketting it. We have chosen to allow bare `fn` expressions and constrained `fn` expressions which are contained within a single pair of brackets; this seems to be the minimal syntax which includes the cases described and which also makes sense.

4.1 Infixes and the Post-Pass

Because the grammar presented to **ML-Yacc** is fixed, it cannot deal with SML's facility to change the infix status or precedence of arbitrary identifiers, especially since such changes are subject to scoping rules. The situation is complicated by the context-dependent meaning of the symbols "`*`" and "`=`", both of which can have their infix status changed. Our only choice is to resolve infixes after they have been parsed. Rather than maintain an infix environment during the parsing process, and tackle infixes in the rule actions, it is easiest to resolve infixes after the top-level parse has completed.

The superset language accepted by the main parser contains the rules[2]

$$Pat ::= AtPat^{+}$$
$$Exp ::= AtExp^{+}$$

so that a pattern can comprise a sequence of one or more atomic patterns, and an expression can be one or more atomic expressions. The post-pass either resolves these phrases (`UNRES_INFIXexp`, `UNRES_INFIXpat`), or reports a syntax error if the actual input is not legal SML. The post-pass removes all `UNRES_INFIXexp` and `UNRES_INFIXpat` nodes, replacing them with "proper" expressions containing the operators.

4.2 Lookahead ambiguities

The ambiguities are as follows. Firstly, consider the following two (legal) phrases (which are both patterns):

$$id \ \texttt{:} \ ty \ \texttt{as} \ pat$$
$$atpat \ \texttt{:} \ ty$$

If the parser encounters an atomic pattern followed by "`:`", it is not possible to ascertain whether the phrase is legal without parsing the "`:`" and the following type expression. If the type is followed by "`as`" then the original atomic pattern must be an *id* (optionally preceeded by "`op`"), otherwise it may be any atomic pattern.

This problem is dealt with in a similar way to patterns containing infixes. The parser accepts a superset of SML syntax, by allowing the form

$$pat \ (\texttt{as} \ pat)^{?}$$

This accepts

$$pat \ \texttt{:} \ ty$$

in isolation, and will also pass

$$pat \ \texttt{as} \ pat$$

through to be transformed into

$$id \ \texttt{:} \ ty \ \texttt{as} \ pat$$

[2] We are using a richer metagrammar than that in the Definition for conciseness in some of the later examples. Superscripts $*$, $+$ and $?$ denote zero or more, one or more, and optional phrases, respectively.

or rejected. Unlike infix analysis, this can be done during the parsing stage, in the action for the appropriate *pat* rule.

The second ambiguity involves "`fun`" declarations. Consider the phrase

`fun` $(pat_1\ infix_1\ pat_2)\ infix_2\ pat_3$ `= ...`

The phrase $(pat_1\ infix_1\ pat_2)$ has two interpretations. If it is followed by an $infix_2$, then it is the first argument of the definition of a function called $infix_2$. If not, the function being defined is $infix_1$.

`Fun` declarations are dealt with using a grammar superset which is resolved (or rejected) in a post-pass, similar to the technique used for infix patterns and expressions. One reason for this is that `fun` declarations need to be analysed in the context of an infix environment anyway. The Kit grammar accepts function bindings of the form

`fun` $atpat^{+}\ (:\ ty)^{?}$ `=` exp

which result in `UNRES_FUNdec` nodes in the abstract syntax. These are replaced by the equivalent `val`-bindings in the post-pass.

4.3 Differences Between the SML Syntax Definition and the Abstract Syntax

To summarise, the main differences between the SML syntax described in the Definition (figure 19), and the abstract syntax of the Kit, are:

- Identifiers are often just of type `id` or `longid` rather than `var`, `longcon`, etc. This is because the parser cannot in general resolve identifier status. Where status can be resolved (for example, in `datatype` declarations), it is done so.

- The Kit syntax retains the "`op`" keyword. This is needed to resolve infixes in patterns, expressions and `fun`-bindings, and is retained to aid printing (rather than being folded into the *info* nodes, which is another possibility but would add complexity). The Definition considers "`op`" to be purely syntactic, and it does not occur in any of the semantic rules.

- The abstract syntax datatype contains declaration constructors `INFIX`, `INFIXR` and `NONFIX`. These are needed for the infix-resolving post-pass. Again, the Definition considers these declarations to be purely syntactic.

- The abstract syntax datatype contains the expression constructor `UNRES_INFIXexp`, the pattern constructor `UNRES_INFIXpat`, and the declaration constructor `UNRES_FUNdec`. These are generated by the parser but eliminated by the parsing post-pass, so that the abstract syntax passed to the elaborator and the later stages of the Kit will never contain `UNRES` syntax.

5 Grammar

We now describe the grammar of SML used by the Kit, as opposed to the abstract syntax datatype. The two do not correspond directly; for example, the grammar accepts the derived forms, although there are no constructs for them in the abstract syntax. The main purpose of this section is to present the conventions adopted in the grammar, and to describe the less obvious features of it.

5.1 Tokens

There are some peculiarities in the tokens (terminal symbols) adopted by the grammar. Firstly, qualified identifiers are recognised as individual `QUAL_ID` tokens. (In fact, this is not peculiar at all, and is what the Definition suggests, but other ML compilers, specifically SML/NJ, lex qualified identifiers as a series of tokens, which is illegal.) Secondly, the grammar distinguishes between single digits, positive integers of two or more digits, and negative (signed) integers. This is because there are contexts in which single digits are required (`infix` and `infixr` declarations), and in which positive integers are required (record labels).

The treatment of identifiers is rather complicated. The syntax of SML requires unqualified identifiers in some contexts (`infix` declarations and record labels, for example) while allowing qualified identifiers elsewhere. The symbol "`*`" is allowed as an identifier except in type expressions, where it serves a specific purpose. In other words, "`*`" is disallowed as a type constructor; in particular, qualified type constructors differ from other qualified identifiers in that they cannot end in "`.*`". The symbol "`=`" is a reserved word which is required to serve as an identifier, except in contexts where that would be ambiguous. We allow qualified use of "`=`" in all contexts. In addition, we syntactically allow redefinition of "`=`", since our interpreter builds the equality predicate in terms of a predefined pervasive function. This facility is disabled for user programs in the semantic analysis stage.

To get the correct treatment of identifiers, we classify them into a number of distinct nonterminals, as follows:

`Ident`: Any valid unqualified ML identifier, including "`*`" but excluding "`=`".

`OpIdent`: An identifier, as above, optionally preceeded by "`op`".

`EqIdent`: An identifier, as above, or "`=`".

`TypeIdent`: An identifier, as above, but omitting "`*`".

`LongIdent`: An identifier or long (qualified) identifier. This includes qualified identifiers such as "`S.*`" and "`*.S`" (which are legal), as well as qualified identifiers ending in "`=`" (since we may wish to provide the equality predicate within a structure, even though it is not legal for programs to rebind it).

`LongTypeIdent`: A `LongIdent` excluding "`*`" and qualified identifiers ending with "`.*`".

`LongOpIdent`: A `LongIdent` optionally preceeded by "`op`".

`LongOpEqIdent`: A `LongOpIdent` or "`=`" or "`op =`".

`Label`: A record label, which is either an identifier (including "`*`" but excluding "`=`"), or a positive integer of indefinite size and without leading zeros.

These classes of identifier are nonterminals, built from a set of terminal symbols which includes unqualified identifiers (`ID`) and qualified identifiers (`FULL_QUAL_ID`, `TYPE_QUAL_ID`), and which treats both "`*`" and "`=`" as reserved words. (According to the Definition, "`*`" is not reserved, although its use and meaning vary according to context.)

Because of the large number of identifier classes in the grammar, we choose not to distinguish between the different *semantic* classes of identifier (*var*, *con*, *excon*), even in contexts where identifier status is known at parse time. This is to avoid having an even larger set of identifier nonterminals. Identifiers encountered in resolvable contexts are converted into the correct semantic objects in the rule actions, so that the abstract syntax has identifiers resolved in all the cases where it is possible to do so.

5.2 Naming Conventions

There are some general naming conventions in the grammar. Terminals are in upper case, nonterminals in mixed case. Note the trailing underscores in the nonterminals for expressions (`Exp_`) and matches (`Match_`) to differentiate the identifiers from the pervasive exception constructors.

The Definition has a number of conventions when describing the syntax of SML. Optional phrases are enclosed in braces ($<\ >$), there is a convention for sequences of phrases, and alternative forms are listed in order of decreasing precedence. We discuss the issue of precedence later.

Optional phrases in the grammatical rules are reflected directly in the abstract syntax of the Kit and in the **ML-Yacc** specification. Corresponding to each optional phrase of some class t the abstract syntax contains a field of type t `Option`, and the grammar contains an explicit "option" nonterminal, named according to the optional phrase it contains and the terminal symbol (if any) that is used to introduce it. Sequenced phrases (with syntax class Xseq for some X) are implemented with specific nonterminals to deal with the sequencing. More specifically, the conventions are as follows:

- Nonterminals ending in "`_opt`" are optional phrases. Many (such as `BarConDesc_opt` and `AndDatDesc_opt`) are introduced by keywords (`BAR` and `AND` in these cases); some (such as `SigDec_opt`) are not.

- Sequences of items are represented by nonterminals ending in "`_seq0`" (zero or more items), "`_seq1`" (one or more items), "`_seq2`" (two or more) and so on. Some (such as `ExpComma_seq0` and `LongIdentEq_seq2`) are named to reflect the tokens which separate the items; others (such as `AtExp_seq1`) have no separator.

- `TyVarSeq` is a special case, being a sequenced construction as described in the Definition. The Definition also uses the construction *tyseq*; we do not, since it would introduce an ambiguity into the grammar (indefinite lookahead would be required to disambiguate type constructions from parenthesised type expressions).

5.3 Ambiguities resolved in the grammar

The grammar has to resolve the ambiguities present in the rules of the syntactic description contained in the Definition. Many of these ambiguities are matters of precedence, covered by the Definition's conventions in the presentation of the rules. There is, however, one remaining ambiguity: a phrase of the form

`local` *dec* `in` *dec* `end`

can be interpreted either at the Modules level (`local` over two *strdec*'s which happen to be Core declarations) or the Core level (a single *strdec* which happens to be a Core declaration containing `local`). In addition, treatment of semicolons is problematic since we wish to parse

dec ; *dec*

as two separate phrases at top-level but as a single phrase when embedded in an enclosing syntactic construct.

To solve all these problems, the grammar is layered in a fairly obvious manner. Nonterminals with names of the form `XXX_sans_YYY` represent ambiguity-resolving intermediate phrases. Top-level declarations using `local` are regarded as modules constructs, in order to avoid indefinite lookahead when resolving phrases such as

`local` *dec* `in structure` *strbind* `end`

The various nonterminals occur as follows:

`OneDec_or_SEMICOLON`: A single core-level declaration or a semicolon;

`NonEmptyDec`: A non-empty core-level declaration, possibly containing semicolons;

`OneDec`: A single declaration, no semicolons;

`OneDec_sans_LOCAL`: Single Core-level declaration phrase, excluding the `local` form;

`StrDec_sans_SEMICOLON`: A structure declaration with no semicolons at the outermost level;

`SigDec_sans_SEMICOLON`: As above, for signature declarations;

`FunDec_sans_SEMICOLON`: As above, for functor declarations;

`Ty_sans_STAR`: Type expression without "`*`" at the outermost level.

The various `_sans_LOCAL` declaration forms eliminate the ambiguity of `local` by consigning it to the Modules level or pushing it down into an unambiguous context. Semicolons are treated similarly so that the parser will deal with them properly in nested constructs but not shift past them at top-level. `Ty_sans_STAR` is a disambiguating phrase class for the "`*`" operator in type expressions.

Semicolon separators (in phrase classes *dec*, *strdec* and *spec*) are a particularly nasty case, since they are optional and they separate optional (i.e. possibly empty) phrases. We deal with this problem by giving "`;`" the same status as the phrase class it serves to separate. The "`compose`" functions from `GrammarUtils` are there to swallow empty abstract syntax phrases which can result from this treatment. One side-effect of this treatment is that it is not possible to reconstitute semicolons when reconstituting source text from the abstract syntax (since that would require an abstract syntax class for them at least).

6 Linking

The various modules of the parsing system, including the **ML-Lex**-generated lexer, the **ML-Yacc**-generated parser, and the support modules, are linked into a single functor, `Parse`, with the signature shown in figure 1. The interface has been kept as simple as possible: parsing is done from a source of input text held in a string or a file, and in the context of the current top-level infix basis. The result of the parse, if it succeeds, is the abstract syntax for a top-level declaration, plus any top-level infix basis established by this declaration.

The reason for the intermediate type `State` is to handle parsing errors elegantly. In order to deal with error recovery properly at the (interactive) top level, source readers operate on a line-by-line basis so that the remainder of an input line can be discarded if an error is detected. An interactive source reader is created by `Parse.sourceFromStdIn`. If a phrase parses and executes successfully, the remainder of the line must be retained in case it contains additional phrases. Hence, `Parse.begin` returns a `State` which can be used repeatedly by `Parse.parse` to read phrases. If the current input line needs to be discarded, the state is discarded and a new source reader created using `sourceFromStdIn` again.

```
signature PARSE =
  sig
    type InfixBasis
    type topdec
    type SourceReader

    val nameOf: SourceReader -> string

    val sourceFromStdIn: unit -> SourceReader
    val sourceFromFile: string -> SourceReader
    val sourceFromString: string -> SourceReader

    type Report

    type State
    datatype Result = SUCCESS of InfixBasis * topdec * State
                    | ERROR of Report
                    | LEGAL_EOF

    val begin: SourceReader -> State
    val parse: InfixBasis * State -> Result
  end
```

Figure 1: The signature PARSE

We use portable versions of the **ML-Lex** and **ML-Yacc** packages which compile under Standard ML with the Edinburgh Library (the packages bundled with SML/NJ are nonportable), and our versions generate portable code (the SML/NJ ones do not). In addition, there are various support modules for **ML-Yacc**, including a generic parsing engine. We use a portable version of these with the "abstractions" removed and outermost structures replaced by functors.

The interfaces and linkage of **ML-Lex** and **ML-Yacc** are messy, and we are not going to document their operation here. Our `Parse` functor links the generated files in the obvious way. `Parse` contains functor applications for the other support modules, and the top-level success and failure handling routines.

7 Files

The abstract syntax modules are in the `Common/` subdirectory because they are required by the entire Kit. The other parsing-related files are in the `Parsing/` subdirectory. The relevant files are as follows:

`Common/DEC_GRAMMAR.sml, Common/DecGrammar.sml`: The signature and functor for the abstract syntax datatype of the Core.

`Common/TOPDEC_GRAMMAR.sml, Common/TopdecGrammar.sml`: The signature and functor for the abstract syntax datatype of the Modules.

`Parsing/Topdec.lex`: The lexical specification (**ML-Lex** source file) for SML.

`Parsing/Topdec.grm`: The grammar (**ML-Yacc** source file) for SML, as described in Section 5.

`Parsing/LEX_BASICS.sml, Parsing/LexBasics.sml`: The module which provides the machinery for source readers (Section 3) and source position annotation and printing.

`Parsing/LEX_UTILS.sml, Parsing/LexUtils.sml`: General lexing utilities: accumulation of literal strings, construction of integer and real constants, and comment nesting. `LexUtils` provides the lexer argument (type `LexArgument`) and operations on it for string and comment accumulation.

`Parsing/Topdec.lex.sml`: The **ML-Lex**-generated lexer.

`Parsing/Topdec.grm.sig`: Signature for the **ML-Yacc** output; contains the token (terminal symbol) specifications.

`Parsing/Topdec.grm.sml`: The **ML-Yacc**-generated parser.

`Parsing/GRAMMAR_UTILS.sml, Parsing/GrammarUtils.sml`: Utility module for the parser. Its main purpose is to expand the derived forms; the expansion functions are called from the actions in the grammar rules. It also brings together the functions for turning strings into the various kinds of identifiers (`con`, `excon`, `lab` and so on), does special constant construction, and provides translation from positional information (`LexBasics.pos`) to *info*.

`Parsing/PARSE.sml, Parsing/Parse.sml`: The top-level of the parser. `PARSE` provides the interface to the rest of the Kit. `Parse` performs all the functor applications and links all the subsidiary modules, and provides the top-level exception handling and error reporting for the parser.

`Parsing/MyBase.sml`: Library files for the generated lexer and parser. A portable, functorised, version of the modules provided with **ML-Lex** and **ML-Yacc**.

`Parsing/INFIXING.sml, Parsing/Infixing.sml`: Infix resolution. Resolves the `infix` declarations for a top-level declaration, and expands `fun` bindings into their equivalent (underived) form.

`Parsing/INFIX_STACK.sml, Parsing/InfixStack.sml`: Internal implementation module for `Infixing`; provides generic analysis and resolution of infix constructions for both patterns and expressions.

`Parsing/HOOKS.sml`: Calls to `use` for the generated lexer and parser, since the generated files cannot carry `Make` tags.

References

[1] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Online documentation distributed with SML/NJ, 1989.

[2] Andrew W. Appel and David R. Tarditi. ML-Yacc, version 2.1. documentation for release version. Online documentation distributed with SML/NJ, March 1991.

[3] Dave Berry. The Edinburgh SML Library. Technical Report ECS–LFCS–91–148, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[4] Nick Rothwell. Miscellaneous design issues in the ML Kit. Technical Report ECS–LFCS–92–237, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.