ML-Yacc User's Manual
Version 2.4

David R. Tarditi[1]
Andrew W. Appel[2]

[1]Microsoft Research

[2]Department of Computer Science
Princeton University
Princeton, NJ 08544

April 24, 2000

New in this version: Improved error correction directive `%change` that allows multi-token insertions, deletions, substitutions. Explanation of how to build a parser (Section 5) and the Calc example (Section 7) revised for SML/NJ Version 110 and the use of CM.

Contents

1 Introduction

1.1 General

ML-Yacc is a parser generator for Standard ML modeled after the Yacc parser generator. It generates parsers for LALR languages, like Yacc, and has a similar syntax. The generated parsers use a different algorithm for recovering from syntax errors than parsers generated by Yacc. The algorithm is a partial implementation of an algorithm described in [1]. A parser tries to recover from a syntax error by making a single token insertion, deletion, or substitution near the point in the input stream at which the error was detected. The parsers delay the evaluation of semantic actions until parses are completed successfully. This makes it possible for parsers to recover from syntax errors that occur before the point of error detection, but it does prevent the parsers from affecting lexers in any significant way. The parsers can insert tokens with values and substitute tokens with values for other tokens. All symbols carry left and right position values which are available to semantic actions and are used in syntactic error messages.

ML-Yacc uses context-free grammars to specify the syntax of languages to be parsed. See [2] for definitions and information on context-free grammars and LR parsing. We briefly review some terminology here. A context-free grammar is defined by a set of terminals T, a set of nonterminals NT, a set of productions P, and a start nonterminal S. Terminals are interchangeably referred to as tokens. The terminal and nonterminal sets are assumed to be disjoint. The set of symbols is the union of the nonterminal and terminal sets. We use lower case Greek letters to denote a string of symbols. We use upper case Roman letters near the beginning of the alphabet to denote nonterminals. Each production gives a derivation of a string of symbols from a nonterminal, which we will write as $A \rightarrow \alpha$. We define a relation between strings of symbols α and β, written $\alpha \vdash \beta$ and read as α derives β, if and only if $\alpha = \delta A \gamma$, $\beta = \delta \phi \gamma$ and there exists some production $A \rightarrow \phi$. We write the transitive closure of this relation as $\vdash_*$. We say that a string of terminals α is a valid sentence of the language, *i.e.* it is derivable, if the start symbol $S \vdash_* \alpha$. The sequence of derivations is often visualized as a parse tree.

ML-Yacc uses an attribute grammar scheme with synthesized attributes. Each symbol in the grammar may have a value (i.e. attribute) associated with it. Each production has a semantic action associated with it. A production with a semantic action is called a rule. Parsers perform bottom-up, left-to-right evaluations of parse trees using semantic actions to compute values as they do so. Given a production $P = A \rightarrow \alpha$, the corresponding semantic action is used to compute a value for A from the values of the symbols in α. If A has no value, the semantic action is still evaluated but the value is ignored. Each parse returns the value associated with the start symbol S of the grammar. A parse returns a nullary value if the start symbol does not carry a value.

parsing structure	$\longrightarrow$	values for a particular parser
values for a particular parser	$\longrightarrow$	lexical analyzer
parsing structure, values for a particular parser, lexical analyzer	$\longrightarrow$	particular parser

Figure 1: Module Dependencies

The synthesized attribute scheme can be adapted easily to inherited attributes. An inherited attribute is a value which propagates from a nonterminal to the symbols produced by the nonterminal according to some rule. Since functions are values in ML, the semantic actions for the derived symbols can return functions which takes the inherited value as an argument.

1.2 Modules

ML-Yacc uses the ML modules facility to specify the interface between a parser that it generates and a lexical analyzer that must be supplied by you. It also uses the ML modules facility to factor out a set of modules that are common to every generated parser. These common modules include a parsing structure, which contains an error-correcting LR parser[1], an LR table structure, and a structure which defines the representation of terminals. ML-Yacc produces a functor for a particular parser parameterized by the LR table structure and the representation of terminals. This functor contains values specific to the parser, such as the LR table for the parser[2], the semantic actions for the parser, and a structure containing the terminals for the parser. ML-Yacc produces a signature for the structure produced by applying this functor and another signature for the structure containing the terminals for the parser. You must supply a functor for the lexing module parameterized this structure.

Figure 1 is a dependency diagram of the modules that summarizes this information. A module at the head of an arrow is dependent on the module at the tail.

1.3 Error Recovery

The error recovery algorithm is able to accurately recover from many single token syntax errors. It tries to make a single token correction at the token in the input stream at which the syntax error was detected and any of the 15 tokens[3] before that token. The algorithm checks corrections before the point of

[1] A plain LR parser is also available.

[2] The LR table is a value. The LR table structure defines an abstract LR table type.

[3] An arbitrary number chosen because numbers above this do not seem to improve error correction much.

error detection because a syntax error is often not detected until several tokens beyond the token which caused the error.[4]

The algorithm works by trying corrections at each of the 16 tokens up to and including the token at which the error was detected. At each token in the input stream, it will try deleting the token, substituting other tokens for the token, or inserting some other token before the token.

The algorithm uses a parse check to evaluate corrections. A parse check is a check of how far a correction allows a parser to parse without encountering a syntax error. You pass an upper bound on how many tokens beyond the error point a parser may read while doing a parse check as an argument to the parser. This allows you to control the amount of lookahead that a parser reads for different kinds of systems. For an interactive system, you should set the lookahead to zero. Otherwise, a parser may hang waiting for input in the case of a syntax error. If the lookahead is zero, no syntax errors will be corrected. For a batch system, you should set the lookahead to 15.

The algorithm selects the set of corrections which allows the parse to proceed the farthest and parse through at least the error token. It then removes those corrections involving keywords which do not meet a longer minimum parse check. If there is more than one correction possible after this, it uses a simple heuristic priority scheme to order the corrections, and then arbitrarily chooses one of the corrections with the highest priority. You have some control over the priority scheme by being able to name a set of preferred insertions and a set of preferred substitutions. The priorities for corrections, ordered from highest to lowest priority, are preferred insertions, preferred substitutions, insertions, deletions, and substitutions.

The error recovery algorithm is guaranteed to terminate since it always selects fixes which parse through the error token.

The error-correcting LR parser implements the algorithm by keeping a queue of its state stacks before shifting tokens and using a lazy stream for the lexer. This makes it possible to restart the parse from before an error point and try various corrections. The error-correcting LR parser does not defer semantic actions. Instead, ML-Yacc creates semantic actions which are free of side-effects and always terminate. ML-Yacc uses higher-order functions to defer the evaluation of all user semantic actions until the parse is successfully completed without constructing an explicit parse tree. You may declare whether your semantic actions are free of side-effects and always terminate, in which case ML-Yacc does not need to defer the evaluation of your semantic actions.

1.4 Precedence

ML-Yacc uses the same precedence scheme as Yacc for resolving shift/reduce conflicts. Each terminal may be assigned a precedence and associativity. Each

[4] An LR parser detects a syntax error as soon as possible, but this does not necessarily mean that the token at which the error was detected caused the error.

rule is then assigned the precedence of its rightmost terminal. If a shift/reduce conflict occurs, the conflict is resolved silently if the terminal and the rule in the conflict have precedences. If the terminal has the higher precedence, the shift is chosen. If the rule has the higher precedence, the reduction is chosen. If both the terminal and the rule have the same precedence, then the associativity of the terminal is used to resolve the conflict. If the terminal is left associative, the reduction is chosen. If the terminal is right associative, the shift is chosen. Terminals may be declared to be nonassociative, also, in which case an error message is produced if the associativity is need to resolve the parsing conflict.

If a terminal or a rule in a shift/reduce conflict does not have a precedence, then an error message is produced and the shift is chosen.

In reduce/reduce conflicts, an error message is always produced and the first rule listed in the specification is chosen for reduction.

1.5 Notation

Text surrounded by brackets denotes meta-notation. If you see something like {parser name}, you should substitute the actual name of your parser for the meta-notation. Text in a bold-face typewriter font (**`like this`**) denotes text in a specification or ML code.

2 ML-Yacc specifications

An ML-Yacc specification consists of three parts, each of which is separated from the others by a **`%%`** delimiter. The general format is:

```
{user declarations}
%%
{ML-Yacc declarations}
%%
{rules}
```

You can define values available in the semantic actions of the rules in the user declarations section. It is recommended that you keep the size of this section as small as possible and place large blocks of code in other modules.

The ML-Yacc declarations section is used to make a set of required declarations and a set of optional declarations. You must declare the nonterminals and terminals and the types of the values associated with them there. You must also name the parser and declare the type of position values. You should specify the set of terminals which can follow the start symbol and the set of non-shiftable terminals. You may optionally declare precedences for terminals, make declarations that will improve error-recovery, and suppress the generation of default reductions in the parser. You may declare whether the parser generator should create a verbose description of the parser in a ".desc" file. This is useful for finding the causes of shift/reduce errors and other parsing conflicts.

You may also declare whether the semantic actions are free of significant side-effects and always terminate. Normally, ML-Yacc delays the evaluation of semantic actions until the completion of a successful parse. This ensures that there will be no semantic actions to "undo" if a syntactic error-correction invalidates some semantic actions. If, however, the semantic actions are free of significant side-effects and always terminate, the results of semantic actions that are invalidated by a syntactic error-correction can always be safely ignored.

Parsers run faster and need less memory when it is not necessary to delay the evaluation of semantic actions. You are encouraged to write semantic actions that are free of side-effects and always terminate and to declare this information to ML-Yacc.

A semantic action is free of significant side-effects if it can be reexecuted a reasonably small number of times without affecting the result of a parse. (The reexecution occurs when the error-correcting parser is testing possible corrections to fix a syntax error, and the number of times reexecution occurs is roughly bounded, for each syntax error, by the number of terminals times the amount of lookahead permitted for the error-correcting parser).

The rules section contains the context-free grammar productions and their associated semantic actions.

2.1 Lexical Definitions

Comments have the same lexical definition as they do in Standard ML and can be placed anywhere in a specification.

All characters up to the first occurrence of a delimiting **%%** outside of a comment are placed in the user declarations section. After that, the following words and symbols are reserved:

```
of for = { } , * -> : | ( )
```

The following classes of ML symbols are used:

identifiers: nonsymbolic ML identifiers, which consist of an alphabetic character followed by one or more alphabetic characters, numeric characters, primes "'", or underscores "_".

type variables: nonsymbolic ML identifier starting with a prime "'"

integers: one or more decimal digits.

qualified identifiers: an identifer followed by a period.

The following classes of non-ML symbols are used:

% identifiers: a percent sign followed by one or more lowercase alphabet letters. The valid % identifiers are:

```
%arg %eop %header %keyword %left %name
%nodefault %nonassoc %nonterm %noshift %pos
%prec %prefer %pure %right %start %subst
%term %value %verbose
```

code: This class is meant to hold ML code. The ML code is not parsed for syntax errors. It consists of a left parenthesis followed by all characters up to a balancing right parenthesis. Parentheses in ML comments and ML strings are excluded from the count of balancing parentheses.

2.2 Grammar

This is the grammar for specifications:

spec ::= user-declarations `%%` cmd-list `%%` rule-list

ML-type ::= nonpolymorphic ML types (see the Standard ML manual)

symbol ::= identifier

symbol-list ::= symbol-list symbol
    | ϵ

symbol-type-list ::= symbol-type-list `|` symbol `of` ML-type
    | symbol-type list `|` symbol
    | symbol `of` ML-type
    | symbol

subst-list ::= subst-list `|` symbol `for` symbol
    | ϵ

cmd ::= `%arg` (Any-ML-pattern) `:` ML-type
    | `%eop` symbol-list
    | `%header` code
    | `%keyword` symbol-list
    | `%left` symbol-list
    | `%name` identifier
    | `%nodefault`
    | `%nonassoc` symbol-list
    | `%nonterm` symbol-type list
    | `%noshift` symbol-list
    | `%pos` ML-type
    | `%prefer` symbol-list
    | `%pure`

	|	**`%right`** symbol-list
	|	**`%start`** symbol
	|	**`%subst`** subst-list
	|	**`%term`** symbol-type-list
	|	**`%value`** symbol code
	|	**`%verbose`**
cmd-list	::=	cmd-list cmd
	|	cmd
rule-prec	::=	**`%prec`** symbol
	|	ϵ
clause-list	::=	symbol-list rule-prec code
	|	clause-list `|` symbol-list rule-prec code
rule	::=	symbol `:` clause-list
rule-list	::=	rule-list rule
	|	rule

2.3 Required ML-Yacc Declarations

`%name` You must specify the name of the parser with **`%name`** {name}.

`%nonterm` and `%term` You must define the terminal and nonterminal sets using the **`%term`** and **`%nonterm`** declarations, respectively. These declarations are like an ML datatype definition. The type of the value that a symbol may carry is defined at the same time that the symbol is defined. Each declarations consists of the keyword (**`%term`** or **`%nonterm`**) followed by a list of symbol entries separated by a bar ("`|`"). Each symbol entry is a symbol name followed by an optional "of <ML-type>". The types cannot be polymorphic. Those symbol entries without a type carry no value. Nonterminal and terminal names must be disjoint and no name may be declared more than once in either declaration.

The symbol names and types are used to construct a datatype union for the values on the semantic stack in the LR parser and to name the values associated with subcomponents of a rule. The names and types of terminals are also used to construct a signature for a structure that may be passed to the lexer functor.

Because the types and names are used in these manners, do not use ML keywords as symbol names. The programs produced by ML-Yacc will not compile if ML keywords are used as symbol names. Make sure that the types specified in the **`%term`** declaration are fully qualified types or are available in the background environment when the signatures produced by ML-Yacc are loaded. Do not use any locally defined types from the user declarations section of the specification.

These requirements on the types in the `%term` declaration are not a burden. They force the types to be defined in another module, which is a good idea since these types will be used in the lexer module.

`%pos` You must declare the type of position values using the `%pos` declaration. The syntax is `%pos` <ML-type>. This type MUST be the same type as that which is actually found in the lexer. It cannot be polymorphic.

2.4 Optional ML-Yacc Declarations

`%arg` You may want each invocation of the entire parser to be parameterized by a particular argument, such as the file-name of the input being parsed in an invocation of the parser. The `%arg` declaration allows you to specify such an argument. (This is often cleaner than using "global" reference variables.) The declaration

> `%arg` (Any-ML-pattern) : <ML-type>

specifies the argument to the parser, as well as its type. For example:

```
%arg (filename) :  string
```

If `%arg` is not specified, it defaults to `() :  unit`.

`%eop` and `%noshift` You should specify the set of terminals that may follow the start symbol, also called end-of-parse symbols, using the `%eop` declaration. The `%eop` keyword should be followed by the list of terminals. This is useful, for example, in an interactive system where you want to force the evaluation of a statement before an end-of-file (remember, a parser delays the execution of semantic actions until a parse is successful).

ML-Yacc has no concept of an end-of-file. You must define an end-of-file terminal (EOF, perhaps) in the `%term` declaration. You must declare terminals which cannot be shifted, such as end-of-file, in the `%noshift` declaration. The `%noshift` keyword should be followed by the list of non-shiftable terminals. An error message will be printed if a non-shiftable terminal is found on the right hand side of any rule, but ML-Yacc will not prevent you from using such grammars.

It is important to emphasize that *non-shiftable terminals must be declared.* The error-correcting parser may attempt to read past such terminals while evaluating a correction to a syntax error otherwise. This may confuse the lexer.

`%header` You may define code to head the functor {parser name}LrValsFun here. This may be useful for adding additonal parameter structures to the functor. The functor must be parameterized by the Token structure, so the declaration should always have the form:

```
%header (functor {parser name}LrValsFun(
                              structure Token : TOKEN
                                     ...)
        )
```

`%left`,`%right`,`%nonassoc` You should list the precedence declarations in order of increasing (tighter-binding) precedence. Each precedence declaration consists of % keyword specifying associativity followed by a list of terminals. The keywords are `%left`, `%right`, and `%nonassoc`, standing for their respective associativities.

`%nodefault` The `%nodefault` declaration suppresses the generation of default reductions. If only one production can be reduced in a given state in an LR table, it may be made the default action for the state. An incorrect reduction will be caught later when the parser attempts to shift the lookahead terminal which caused the reduction. ML-Yacc usually produces programs and verbose files with default reductions. This saves a great deal of space in representing the LR tables,but sometimes it is useful for debugging and advanced uses of the parser to suppress the generation of default reductions.

`%pure` Include the `%pure` declaration if the semantic actions are free of significant side-effects and always terminate.

`%start` You may define the start symbol using the `%start` declaration. Otherwise the nonterminal for the first rule will be used as the start nonterminal. The keyword `%start` should be followed by the name of the starting nonterminal. This nonterminal should not be used on the right hand side of any rules, to avoid conflicts between reducing to the start symbol and shifting a terminal. ML-Yacc will not prevent you from using such grammars, but it will print a warning message.

`%verbose` Include the `%verbose` declaration to produce a verbose description of the LALR parser. The name of this file is the name of the specification file with a ".desc" appended to it.

This file has the following format:

1. A summary of errors found while generating the LALR tables.
2. A detailed description of all errors.
3. A description of the states of the parser. Each state is preceded by a list of conflicts in the state.

2.5 Declarations for improving error-recovery

These optional declarations improve error-recovery:

`%keyword` Specify all keywords in a grammar here. The **`%keyword`** should be followed by a list of terminal names. Fixes involving keywords are generally dangerous; they are prone to substantially altering the syntactic meaning of the program. They are subject to a more rigorous parse check than other fixes.

`%prefer` List terminals to prefer for insertion after the **`%prefer`**. Corrections which insert a terminal on this list will be chosen over other corrections, all other things being equal.

`%subst` This declaration should be followed by a list of clauses of the form {terminal} **`for`** {terminal}, where items on the list are separated using a `|`. Substitution corrections on this list will be chosen over all other corrections except preferred insertion corrections (listed above), all other things being equal.

`%change` This is a generalization of **`%prefer`** and **`%subst`**. It takes a the following syntax:

$tokens_{1a}$ `->` $tokens_{1b}$ `|` $tokens_{2a}$ `->` $tokens_{2b}$ *etc.*

where each *tokens* is a (possibly empty) seqence of tokens. The idea is that any instance of $tokens_{1a}$ can be "corrected" to $tokens_{1b}$, and so on. For example, to suggest that a good error correction to try is `IN ID END` (which is useful for the ML parser), write,

```
%change   ->  IN ID END
```

`%value` The error-correction algorithm may also insert terminals with values. You must supply a value for such a terminal. The keyword should be followed by a terminal and a piece of code (enclosed in parentheses) that when evaluated supplies the value. There must be a separate **`%value`** declaration for each terminal with a value that you wish may be inserted or substituted in an error correction. The code for the value is not evaluated until the parse is successful.

Do not specify a **`%value`** for terminals without values. This will result in a type error in the program produced by ML-Yacc.

2.6 Rules

All rules are declared in the final section, after the last **`%%`** delimiter. A rule consists of a left hand side nonterminal, followed by a colon, followed by a list of right hand side clauses.

The right hand side clauses should be separated by bars ("|"). Each clause consists of a list of nonterminal and terminal symbols, followed by an optional `%prec` declaration, and then followed by the code to be evaluated when the rule is reduced.

The optional `%prec` consists of the keyword `%prec` followed by a terminal whose precedence should be used as the precedence of the rule.

The values of those symbols on the right hand side which have values are available inside the code. Positions for all the symbols are also available. Each value has the general form {symbol name}{n+1}, where {n} is the number of occurrences of the symbol to the left of the symbol. If the symbol occurs only once in the rule, {symbol name} may also be used. The positions are given by {symbol name}{n+1}left and {symbol name}{n+1}right. where {n} is defined as before. The position for a null rhs of a production is assumed to be the leftmost position of the lookahead terminal which is causing the reduction. This position value is available in `defaultPos`.

The value to which the code evaluates is used as the value of the nonterminal. The type of the value and the nonterminal must match. The value is ignored if the nonterminal has no value, but is still evaluated for side-effects.

3 Producing files with ML-Yacc

ML-Yacc may be used from the interactive system or built as a stand-alone program which may be run from the Unix command line. See the file **README** in the mlyacc directory for directions on installing ML-Yacc. We recommend thaat ML-Yacc be installed as a stand-alone program.

If you are using the stand-alone version of ML-Yacc, invoke the program "sml-yacc" with the name of the specifcation file. If you are using ML-Yacc in the interactive system, load the file "smlyacc.sml". The end result is a structure ParseGen, with one value parseGen in it. Apply parseGen to a string containing the name of the specification file.

Two files will be created, one named by attaching ".sig" to the name of the specification, the other named by attaching ".sml" to the name of the specification.

4 The lexical analyzer

Let the name for the parser given in the `%name` declaration be denoted by {n} and the specification file name be denoted by {spec name} The parser generator creates a functor named {n}LrValsFun for the values needed for a particular parser. This functor is placed in {spec name}.sml. This functor contains a structure Tokens which allows you to construct terminals from the appropriate values. The structure has a function for each terminal that takes a tuple consisting of the value for the terminal (if there is any), a leftmost position for the terminal, and a rightmost position for the terminal and constructs the terminal from these values.

A signature for the structure Tokens is created and placed in the ".sig" file created by ML-Yacc. This signature is {n}_TOKENS, where {n} is the name given in the parser specification. A signature {n}_LRVALS is created for the structure produced by applying {n}LrValsFun.

Use the signature {n}_TOKENS to create a functor for the lexical analyzer which takes the structure Tokens as an argument. The signature {n}_TOKENS will not change unless the `%term` declaration in a specification is altered by adding terminals or changing the types of terminals. You do not need to recompile the lexical analyzer functor each time the specification for the parser is changed if the signature {n}_TOKENS does not change.

If you are using ML-Lex to create the lexical analyzer, you can turn the lexer structure into a functor using the `%header` declaration. `%header` allows the user to define the header for a structure body.

If the name of the parser in the specification were Calc, you would add this declaration to the specification for the lexical analyzer:

```
%header (functor CalcLexFun(structure Tokens : Calc_TOKENS))
```

You must define the following in the user definitions section:

```
type pos
```

This is the type of position values for terminals. This type must be the same as the one declared in the specification for the grammar. Note, however, that this type is not available in the Tokens structure that parameterizes the lexer functor.

You must include the following code in the user definitions section of the ML-Lex specification:

```
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult  = (svalue,pos) token
```

These types are used to give lexers signatures.

You may use a lexer constructed using ML-Lex with the `%arg` declaration, but you must follow special instructions for tying the parser and lexer together.

5 Creating the parser

Let the name of the grammar specification file be denoted by {grammar} and the name of the lexer specification file be denoted by {lexer} (e.g. in our calculator example these would stand for calc.grm and calc.lex, respectively). Let the parser name in the specification be represented by {n} (e.g. Calc in our calculator example).

To construct a parser, do the following:

1. In the appropriate CM description file (e.g. for your main program or one of its subgroups or libraries), include the lines:

   ```
   ml-yacc-lib.cm
   {lexer}
   {grammar}
   ```

   This will cause ML-Yacc to be run on {grammar}, producing source files {grammar}.sig and {grammar}.sml, and ML-Lex to be run on {lexer}, producing a source file {lexer}.sml. Then these files will be compiled after loading the necessary signatures and modules from the ML-Yacc library as specified by `ml-yacc-lib.cm`.

2. Apply functors to create the parser:

   ```
   structure {n}LrVals =
     {n}LrValsFun(structure Token = LrParser.Token)
   structure {n}Lex =
     {n}LexFun(structure Tokens = {n}LrVals.Tokens)
   structure {n}Parser=
     Join(structure ParserData = {n}LrVals.ParserData
          structure Lex={n}Lex
          structure LrParser=LrParser)
   ```

   If the lexer was created using the `%arg` declaration in ML-Lex, the definition of {n}Parser must be changed to use another functor called JoinWithArg:

   ```
   structure {n}Parser=
     JoinWithArg
       (structure ParserData={n}LrVals.ParserData
        structure Lex={n}Lex
        structure LrParser=LrParser)
   ```

The following outline summarizes this process:

```
(* available at top level *)

TOKEN
LR_TABLE
STREAM
LR_PARSER
PARSER_DATA
```

```
structure LrParser : LR_PARSER

(* printed out in .sig file created by parser generator: *)

signature {n}_TOKENS =
sig
  structure Token : TOKEN
  type svalue
  val PLUS : 'pos * 'pos ->
             (svalue,'pos) Token.token
  val INTLIT : int * 'pos * 'pos ->
               (svalue,'pos) Token.token
  ...
end

signature {n}_LRVALS =
sig
  structure Tokens : {n}_TOKENS
  structure ParserData : PARSER_DATA
  sharing ParserData.Token = Tokens.Token
  sharing type ParserData.svalue = Tokens.svalue
end

(* printed out by lexer generator: *)

functor {n}LexFun(structure Tokens : {n}_TOKENS)=
struct
  ...
end

(* printed out in .sml file created by parser generator: *)

functor {n}LrValsFun(structure Token : TOKENS) =
struct

  structure ParserData =
  struct
    structure Token = Token

    (* code in header section of specification *)

    structure Header = ...
    type svalue = ...
    type result = ...
```

```
    type pos = ...
    structure Actions = ...
    structure EC = ...
    val table = ...
  end

  structure Tokens : {n}_TOKENS =
  struct
    structure Token = ParserData.Token
    type svalue = ...
    fun PLUS(p1,p2) = ...
    fun INTLIT(i,p1,p2) = ...
  end

end

(* to be done by the user: *)

structure {n}LrVals =
  {n}LrValsFun(structure Token = LrParser.Token)

structure {n}Lex =
  {n}LexFun(structure Tokens = {n}LrVals.Tokens)

structure {n}Parser =
  Join(structure Lex = {n}Lex
       structure ParserData = {n}ParserData
       structure LrParser = LrParser)
```

6 Using the parser

6.1 Parser Structure Signatures

The final structure created will have the signature PARSER:

```
signature PARSER =
sig
  structure Token : TOKEN
  structure Stream : STREAM
  exception ParseError

  type pos    (* pos is the type of line numbers *)
  type result (* value returned by the parser *)
  type arg    (* type of the user-supplied argument  *)
```

```
  type svalue (* the types of semantic values *)

  val makeLexer : (int -> string) ->
    (svalue,pos) Token.token Stream.stream
  val parse :
      int * ((svalue,pos) Token.token Stream.stream) *
      (string * pos * pos -> unit) * arg ->
result * (svalue,pos) Token.token Stream.stream
  val sameToken :
      (svalue,pos) Token.token * (svalue,pos) Token.token ->
bool
end
```

or the signature ARG_PARSER if you used `%arg` to create the lexer. This signature differs from ARG_PARSER in that it which has an additional type `lexarg` and a different type for `makeLexer`:

```
type lexarg
val makeLexer : (int -> string)  -> lexarg ->
                    (svalue,pos) token stream
```

The signature STREAM (providing lazy streams) is:

```
signature STREAM =
sig
  type 'a stream
  val streamify : (unit -> 'a) -> 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val get : 'a stream -> 'a * 'a stream
end
```

6.2 Using the parser structure

The parser structure converts the lexing function produced by ML-Lex into a function which creates a lazy stream of tokens. The function `makeLexer` takes the same values as the corresponding `makeLexer` created by ML-Lex, but returns a stream of tokens instead of a function which yields tokens.

The function parse takes the token stream and some other arguments that are described below and parses the token stream. It returns a pair composed of the value associated with the start symbol and the rest of the token stream. The rest of the token stream includes the end-of-parse symbol which caused the reduction of some rule to the start symbol. The function parse raises the exception ParseError if a syntax error occurs which it cannot fix.

The lazy stream is implemented by the `Stream` structure. The function `streamify` converts a conventional implementation of a stream into a lazy stream. In a conventional implementation of a stream, a stream consists of a position in a list of values. Fetching a value from a stream returns the value associated with the position and updates the position to the next element in the list of values. The fetch is a side-effecting operation. In a lazy stream, a fetch returns a value and a new stream, without a side-effect which updates the position value. This means that a stream can be repeatedly re-evaluated without affecting the values that it returns. If f is the function that is passed to `streamify`, f is called only as many times as necessary to construct the portion of the list of values that is actually used.

Parse also takes an integer giving the maximum amount of lookahead permitted for the error-correcting parse, a function to print error messages, and a value of type arg. The maximum amount of lookahead for interactive systems should be zero. In this case, no attempt is made to correct any syntax errors. For non-interactive systems, try 15. The function to print error messages takes a tuple of values consisting of the left and right positions of the terminal which caused the error and an error message. If the `%arg` declaration is not used, the value of type arg should be a value of type unit.

The function sameToken can be used to see if two tokens denote the same terminal, irregardless of any values that the tokens carry. It is useful if you have multiple end-of-parse symbols and must check which end-of-parse symbol has been left on the front of the token stream.

The types have the following meanings. The type `arg` is the type of the additional argument to the parser, which is specified by the `%arg` declaration in the ML-Yacc specification. The type `lexarg` is the optional argument to lexers, and is specified by the `%arg` declaration in an ML-Lex specifcation. The type `pos` is the type of line numbers, and is specified by the `%pos` declaration in an ML-Yacc specification and defined in the user declarations section of the ML-Lex specification. The type `result` is the type associated with the start symbol in the ML-Yacc specification.

7 Examples

See the directory examples for examples of parsers constructed using ML-Yacc. Here is a small sample parser and lexer for an interactive calculator, from the directory examples/calc, along with code for creating a parsing function. The calculator reads one or more expressions from the standard input, evaluates the expressions, and prints their values. Expressions should be separated by semicolons, and may also be ended by using an end-of-file. This shows how to construct an interactive parser which reads a top-level declaration and processes the declaration before reading the next top-level declaration.

7.1 Sample Grammar

```
(* Sample interactive calculator for ML-Yacc *)

fun lookup "bogus" = 10000
  | lookup s = 0

%%

%eop EOF SEMI

(* %pos declares the type of positions for terminals.
   Each symbol has an associated left and right position. *)

%pos int

%left SUB PLUS
%left TIMES DIV
%right CARAT

%term ID of string | NUM of int | PLUS | TIMES | PRINT |
      SEMI | EOF | CARAT | DIV | SUB
%nonterm EXP of int | START of int option

%name Calc

%subst PRINT for ID
%prefer PLUS TIMES DIV SUB
%keyword PRINT SEMI

%noshift EOF
%value ID ("bogus")
%nodefault
%verbose
%%

(* the parser returns the value associated with the expression *)

  START : PRINT EXP (print EXP;
                     print "\n";
                     flush_out std_out; SOME EXP)
        | EXP (SOME EXP)
        | (NONE)
  EXP : NUM             (NUM)
```

```
      | ID              (lookup ID)
      | EXP PLUS EXP    (EXP1+EXP2)
      | EXP TIMES EXP   (EXP1*EXP2)
      | EXP DIV EXP     (EXP1 div EXP2)
      | EXP SUB EXP     (EXP1-EXP2)
      | EXP CARAT EXP   (let fun e (m,0) = 1
                               | e (m,l) = m*e(m,l-1)
                         in e (EXP1,EXP2)
                         end)
```

7.2 Sample Lexer

```
structure Tokens = Tokens

type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult= (svalue,pos) token

val pos = ref 0
val eof = fn () => Tokens.EOF(!pos,!pos)
val error = fn (e,l : int,_) =>
              output(std_out,"line " ^ (makestring l) ^
                                 ": " ^ e ^ "\n")
%%
%header (functor CalcLexFun(structure Tokens: Calc_TOKENS));
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n       => (pos := (!pos) + 1; lex());
{ws}+    => (lex());
{digit}+ => (Tokens.NUM
                (revfold (fn (a,r) => ord(a)-ord("0")+10*r)
                         (explode yytext) 0,
                  !pos,!pos));
"+"      => (Tokens.PLUS(!pos,!pos));
"*"      => (Tokens.TIMES(!pos,!pos));
";"      => (Tokens.SEMI(!pos,!pos));
{alpha}+ => (if yytext="print"
                 then Tokens.PRINT(!pos,!pos)
                 else Tokens.ID(yytext,!pos,!pos)
            );
"-"      => (Tokens.SUB(!pos,!pos));
```

```
"^"      => (Tokens.CARAT(!pos,!pos));
"/"      => (Tokens.DIV(!pos,!pos));
"."      => (error ("ignoring bad character "^yytext,!pos,!pos);
             lex());
```

7.3 Top-level code

You must follow the instructions in Section 5 to create the parser and lexer functors and load them. After you have done this, you must then apply the functors to produce the `CalcParser` structure. The code for doing this is shown below.

```
structure CalcLrVals =
  CalcLrValsFun(structure Token = LrParser.Token)

structure CalcLex =
  CalcLexFun(structure Tokens = CalcLrVals.Tokens);

structure CalcParser =
  Join(structure LrParser = LrParser
       structure ParserData = CalcLrVals.ParserData
       structure Lex = CalcLex)
```

Now we need a function which given a lexer invokes the parser. The function `invoke` does this.

```
fun invoke lexstream =
    let fun print_error (s,i:int,_) =
    TextIO.output(TextIO.stdOut,
  "Error, line " ^ (Int.toString i) ^ ", " ^ s ^ "\n")
     in CalcParser.parse(0,lexstream,print_error,())
    end
```

Finally, we need a function which can read one or more expressions from the standard input. The function `parse`, shown below, does this. It runs the calculator on the standard input and terminates when an end-of-file is encountered.

```
fun parse () =
    let val lexer = CalcParser.makeLexer
                      (fn _ => TextIO.inputLine TextIO.stdIn)
val dummyEOF = CalcLrVals.Tokens.EOF(0,0)
val dummySEMI = CalcLrVals.Tokens.SEMI(0,0)
fun loop lexer =
    let val (result,lexer) = invoke lexer
val (nextToken,lexer) = CalcParser.Stream.get lexer
```

```
      in case result
   of SOME r =>
       TextIO.output(TextIO.stdOut,
      "result = " ^ (Int.toString r) ^ "\n")
    | NONE => ();
        if CalcParser.sameToken(nextToken,dummyEOF) then ()
else loop lexer
    end
     in loop lexer
    end
```

8 Signatures

This section contains signatures used by ML-Yacc for structures in the file base.sml, functors and structures that it generates, and for the signatures of lexer structures supplied by you.

8.1 Parsing structure signatures

```
(* STREAM: signature for a lazy stream.*)

signature STREAM =
sig
  type 'a stream
  val streamify : (unit -> 'a) -> 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val get : 'a stream -> 'a * 'a stream
end

(* LR_TABLE: signature for an LR Table.*)

signature LR_TABLE =
sig
  datatype ('a,'b) pairlist
    = EMPTY
    | PAIR of 'a * 'b * ('a,'b) pairlist
  datatype state = STATE of int
  datatype term = T of int
  datatype nonterm = NT of int
  datatype action = SHIFT of state
                  | REDUCE of int
                  | ACCEPT
                  | ERROR
```

```
  type table

  val numStates : table -> int
  val numRules : table -> int
  val describeActions : table -> state ->
                        (term,action) pairlist * action
  val describeGoto : table -> state ->
                     (nonterm,state) pairlist
  val action : table -> state * term -> action
  val goto : table -> state * nonterm -> state
  val initialState : table -> state
  exception Goto of state * nonterm

  val mkLrTable :
      {actions : ((term,action) pairlist * action) array,
       gotos : (nonterm,state) pairlist array,
       numStates : int, numRules : int,
       initialState : state} -> table
end

(* TOKEN: signature for the internal structure of a token.*)

signature TOKEN =
sig
  structure LrTable : LR_TABLE
  datatype ('a,'b) token = TOKEN of LrTable.term *
                                    ('a * 'b * 'b)
  val sameToken : ('a,'b) token * ('a,'b) token -> bool
end

(* LR_PARSER: signature for a polymorphic LR parser *)

signature LR_PARSER =
sig
  structure Stream: STREAM
  structure LrTable : LR_TABLE
  structure Token : TOKEN

  sharing LrTable = Token.LrTable

  exception ParseError

  val parse:
       {table : LrTable.table,
```

```
        lexer : ('b,'c) Token.token Stream.stream,
        arg: 'arg,
        saction : int *
                 'c *
                 (LrTable.state * ('b * 'c * 'c)) list *
                 'arg ->
                  LrTable.nonterm *
                  ('b * 'c * 'c) *
                  ((LrTable.state *('b * 'c * 'c)) list),
        void : 'b,
        ec: {is_keyword : LrTable.term -> bool,
             noShift : LrTable.term -> bool,
             preferred_subst:LrTable.term -> LrTable.term list,
             preferred_insert : LrTable.term -> bool,
             errtermvalue : LrTable.term -> 'b,
             showTerminal : LrTable.term -> string,
             terms: LrTable.term list,
             error : string * 'c * 'c -> unit
            },
        lookahead : int (* max amount of lookahead used in
                         * error correction *)
       } -> 'b * (('b,'c) Token.token Stream.stream)
end
```

8.2 Lexers

Lexers for use with ML-Yacc's output must match one of these signatures.

```
signature LEXER =
sig
  structure UserDeclarations :
    sig
      type ('a,'b) token
      type pos
      type svalue
    end
  val makeLexer : (int -> string) -> unit ->
       (UserDeclarations.svalue, UserDeclarations.pos)
       UserDeclarations.token
end

(* ARG_LEXER: the %arg option of ML-Lex allows users to
   produce lexers which also take an argument before
   yielding a function from unit to a token.
```

```
*)

signature ARG_LEXER =
sig
  structure UserDeclarations :
    sig
      type ('a,'b) token
      type pos
      type svalue
      type arg
    end
  val makeLexer :
      (int -> string) ->
      UserDeclarations.arg ->
      unit ->
       (UserDeclarations.svalue, UserDeclarations.pos)
       UserDeclarations.token
end
```

8.3 Signatures for the functor produced by ML-Yacc

The following signature is used in signatures generated by ML-Yacc:

```
(* PARSER_DATA: the signature of ParserData structures in
   {n}LrValsFun functor produced by ML-Yacc. All such
   structures match this signature. *)

signature PARSER_DATA =
sig
  type pos       (* the type of line numbers *)
  type svalue    (* the type of semantic values *)
  type arg       (* the type of the user-supplied *)
 (* argument to the parser *)
  type result

  structure LrTable : LR_TABLE
  structure Token : TOKEN
  sharing Token.LrTable = LrTable

  structure Actions :
    sig
      val actions : int * pos *
       (LrTable.state * (svalue * pos * pos)) list * arg ->
       LrTable.nonterm * (svalue * pos * pos) *
```

```
      ((LrTable.state *(svalue * pos * pos)) list)
       val void : svalue
       val extract : svalue -> result
     end

   (* structure EC contains information used to improve
      error recovery in an error-correcting parser *)

   structure EC :
     sig
       val is_keyword : LrTable.term -> bool
       val noShift : LrTable.term -> bool
       val preferred_subst: LrTable.term -> LrTable.term list
       val preferred_insert : LrTable.term -> bool
       val errtermvalue : LrTable.term -> svalue
       val showTerminal : LrTable.term -> string
       val terms: LrTable.term list
     end

   (* table is the LR table for the parser *)

   val table : LrTable.table
 end
```

ML-Yacc generates these two signatures:

```
(* printed out in .sig file created by parser generator: *)

signature {n}_TOKENS =
sig
  type ('a,'b) token
  type svalue
  ...
end

signature {n}_LRVALS =
sig
  structure Tokens : {n}_TOKENS
  structure ParserData : PARSER_DATA
  sharing type ParserData.Token.token = Tokens.token
  sharing type ParserData.svalue = Tokens.svalue
end
```

8.4 User parser signatures

Parsers created by applying the Join functor will match this signature:

```
signature PARSER =
sig
  structure Token : TOKEN
  structure Stream : STREAM
  exception ParseError

  type pos    (* pos is the type of line numbers *)
  type result (* value returned by the parser *)
  type arg    (* type of the user-supplied argument  *)
  type svalue (* the types of semantic values *)

  val makeLexer : (int -> string) ->
   (svalue,pos) Token.token Stream.stream

  val parse :
      int * ((svalue,pos) Token.token Stream.stream) *
      (string * pos * pos -> unit) * arg ->
  result * (svalue,pos) Token.token Stream.stream
  val sameToken :
    (svalue,pos) Token.token * (svalue,pos) Token.token ->
bool
end
```

Parsers created by applying the JoinWithArg functor will match this signature:

```
signature ARG_PARSER =
sig
  structure Token : TOKEN
  structure Stream : STREAM
  exception ParseError

  type arg
  type lexarg
  type pos
  type result
  type svalue

  val makeLexer : (int -> string) -> lexarg ->
     (svalue,pos) Token.token Stream.stream
  val parse : int *
      ((svalue,pos) Token.token Stream.stream) *
```

```
        (string * pos * pos -> unit) *
        arg ->
         result * (svalue,pos) Token.token Stream.stream
    val sameToken :
        (svalue,pos) Token.token * (svalue,pos) Token.token ->
     bool
  end
```

9 Sharing constraints

Let the name of the parser be denoted by {n}. If you have not created a lexer which takes an argument, and you have followed the directions given earlier for creating the parser, you will have the following structures with the following signatures:

```
(* always present *)

signature TOKEN
signature LR_TABLE
signature STREAM
signature LR_PARSER
signature PARSER_DATA
structure LrParser : LR_PARSER

(* signatures generated by ML-Yacc *)

signature {n}_TOKENS
signature {n}_LRVALS

(* structures created by you *)

structure {n}LrVals : {n}_LRVALS
structure Lex : LEXER
structure {n}Parser : PARSER
```

The following sharing constraints will exist:

```
sharing {n}Parser.Token = LrParser.Token =
          {n}LrVals.ParserData.Token
sharing {n}Parser.Stream = LrParser.Stream

sharing type {n}Parser.arg = {n}LrVals.ParserData.arg
sharing type {n}Parser.result = {n}LrVals.ParserData.result
sharing type {n}Parser.pos = {n}LrVals.ParserData.pos =
```

```
                Lex.UserDeclarations.pos
sharing type {n}Parser.svalue = {n}LrVals.ParserData.svalue =
        {n}LrVals.Tokens.svalue = Lex.UserDeclarations.svalue
sharing type {n}Parser.Token.token =
           {n}LrVals.ParserData.Token.token =
           LrParser.Token.token =
           Lex.UserDeclarations.token

sharing {n}LrVals.LrTable = LrParser.LrTable
```

If you used a lexer which takes an argument, then you will have:

```
structure ARG_LEXER
structure {n}Parser : PARSER

(* additional sharing constraint *)

sharing type {n}Parser.lexarg = Lex.UserDeclarations.arg
```

10 Hints

10.1 Multiple start symbols

To have multiple start symbols, define a dummy token for each start symbol. Then define a start symbol which derives the multiple start symbols with dummy tokens placed in front of them. When you start the parser you must place a dummy token on the front of the lexer stream to select a start symbol from which to begin parsing.

Assuming that you have followed the naming conventions used before, create the lexer using the makeLexer function in the {n}Parser structure. Then, place the dummy token on the front of the lexer:

```
val dummyLexer =
    {n}Parser.Stream.cons
        ({n}LrVals.Tokens.{dummy token name}
                 ({dummy lineno},{dummy lineno}),
        lexer)
```

You have to pass a Tokens structure to the lexer. This Tokens structure contains functions which construct tokens from values and line numbers. So to create your dummy token just apply the appropriate token constructor function from this Tokens structure to a value (if there is one) and the line numbers. This is exactly what you do in the lexer to construct tokens.

Then you must place the dummy token on the front of your lex stream. The structure {n}Parser contains a structure Stream which implements lazy streams. So you just cons the dummy token on to stream returned by makeLexer.

10.2 Functorizing things further

You may wish to functorize things even further. Two possibilities are turning the lexer and parser structures into closed functors, that is, functors which do not refer to types or values defined outside their body or outside their parameter structures (except for pervasive types and values), and creating a functor which encapsulates the code necessary to invoke the parser.

Use the `%header` declarations in ML-Lex and ML-Yacc to create closed functors. See section 2.4 of this manual and section 4 of the manual for ML-Lex for complete descriptions of these declarations. If you do this, you should also parameterize these structures by the types of line numbers. The type will be an abstract type, so you will also need to define all the valid operations on the type. The signature INTERFACE, defined below, shows one possible signature for a structure defining the line number type and associated operations.

If you wish to encapsulate the code necessary to invoke the parser, your functor generally will have form:

```
functor Encapsulate(
     structure Parser : PARSER
     structure Interface : INTERFACE
         sharing type Parser.arg = Interface.arg
         sharing type Parser.pos = Interface.pos
         sharing type Parser.result = ...
     structure Tokens : {parser name}_TOKENS
         sharing type Tokens.token = Parser.Token.token
         sharing type Tokens.svalue = Parser.svalue) =
  struct
        ...
  end
```

The signature INTERFACE, defined below, is a possible signature for a structure defining the types of line numbers and arguments (types pos and arg, respectively) along with operations for them. You need this structure because these types will be abstract types inside the body of your functor.

```
signature INTERFACE =
sig
   type pos
   val line : pos ref
   val reset : unit -> unit
   val next : unit -> unit
```

```
    val error : string * pos * pos -> unit

    type arg
    val nothing : arg
end
```

The directory example/fol contains a sample parser in which the code for tying together the lexer and parser has been encapsulated in a functor.

11 Acknowledgements

Nick Rothwell wrote an SLR table generator in 1988 which inspired the initial work on an ML parser generator. Bruce Duba and David MacQueen made useful suggestions about the design of the error-correcting parser. Thanks go to all the users at Carnegie Mellon who beta-tested this version. Their comments and questions led to the creation of this manual and helped improve it.

12 Bugs

There is a slight difference in syntax between ML-Lex and ML-Yacc. In ML-Lex, semantic actions must be followed by a semicolon but in ML-Yacc semantic actions cannot be followed by a semicolon. The syntax should be the same. ML-Lex also produces structures with two different signatures, but it should produce structures with just one signature. This would simplify some things.

References

[1] "A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery", M. Burke and G. Fisher, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1987, pp. 164-167.

[2] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.