CM-Lex and CM-Yacc User's Manual
(Version 2.1)

Karl Crary
Carnegie Mellon University

December 20, 2017

1 CM-Lex

CM-Lex is a lexer generator for Standard ML and Haskell. It converts a specification file into an SML or Haskell source-code file that implements a lexer functor. The programmer may then instantiate that functor with appropriate code for lexing actions to obtain a lexer.

1.1 The specification file

A CM-Lex specification consists of a series of directives:

- `sml` *or* `haskell` *or* `ocaml`

  Specifies the language in which the generated lexer should be implemented. This is a required directive, and it must appear first.

- `name` ⟨*identifier*⟩

  Specifies the name of the generated functor for SML/Ocaml lexers, and the name of the generated module for Haskell lexers. This is a required directive.

  In Haskell and OCaml specifications, the name must be a valid module identifier, so it must begin with a capital letter. In Haskell, a hierarchical name is permitted. In SML specifications the name is not required to be capitalized, but it is standard practice in SML code to capitalize functor names nonetheless.

- `alphabet` ⟨*alphabet size*⟩

  Specifies the size of the input alphabet, which must be a positive integer. The generated lexer will not recognize any symbols whose ordinal is greater or equal to the given value. This is a required directive and it must appear before any `set`, `regexp`, or `function` directives.

- `set` ⟨*name*⟩ `=` ⟨*charset*⟩

  Declares a character set ⟨*name*⟩ with the definition ⟨*charset*⟩.

- `regexp` ⟨*name*⟩ `=` ⟨*regexp*⟩

  Declares a regular expression ⟨*name*⟩ with the definition ⟨*regexp*⟩.

- `function` ⟨*function name*⟩ `:` ⟨*type name*⟩ `=` ⟨*clauses*⟩

  Specifies a lexing function ⟨*function name*⟩ that has type ⟨*type name*⟩ and is defined by ⟨*clauses*⟩, where each clause has the form:

  ⟨*regexp*⟩ `=>` ⟨*action name*⟩

- `monadic`

  Haskell specifications only: Specifies that CM-Lex should generate a monadic lexer.

Comments are written "`/*` *comment text* `*/`" and may appear anywhere in the specification. Comments may be nested.

1.1.1 Character sets and regular expressions

For the purposes of CM-Lex, symbols are identified with nonnegative integers. Typically the number is the ASCII code of a character, but establishing the correspondence is left to the programmer.

⟨*symbol*⟩ ::=

⟨*number*⟩	the indicated number
`'` ⟨*printable character*⟩	the ASCII code of the indicated character

Character sets and regular expressions are specified using the Scheme Shell's SRE notation:

⟨*charset*⟩ ::=

⟨*identifier*⟩	the indicated character set
⟨*symbol*⟩	the singleton set containing ⟨*symbol*⟩
`empty`	the empty set
`any`	the universal set (all symbols up to $N-1$, where N is the alphabet size)
`(or` ⟨*charset*⟩ `...)` `(|` ⟨*charset*⟩ `...)`	set union
`(and` ⟨*charset*⟩ `...)` `(&` ⟨*charset*⟩ `...)`	set intersection
`(-` ⟨*charset*⟩ `...)`	set difference: the first set minus all the following sets
`(~` ⟨*charset*⟩ `...)`	set complement: equivalent to (- any (or ⟨*charset*⟩ ...))
`(range` ⟨*range-spec*⟩ `...)` `(/` ⟨*range-spec*⟩ `...)`	union of the specified ranges

⟨*range-spec*⟩ ::=

⟨*symbol1*⟩ ⟨*symbol2*⟩	the range of symbols from ⟨*symbol1*⟩ to ⟨*symbol2*⟩ (inclusive)

⟨*regexp*⟩ ::=

⟨*identifier*⟩	the indicated character set or regular expression
⟨*symbol*⟩	match the indicated symbol
`any`	match any character

`epsilon`	match the empty string
`empty`	match nothing
`eos`	match end-of-stream
`"` ⟨*string*⟩ `"`	match the indicated string: that is, the sequence of symbols given by the ASCII codes of the characters of ⟨*string*⟩ (the string can contain any printable character other than a quotation mark)
`(seq` ⟨*regexp*⟩ `...)` `(:` ⟨*regexp*⟩ `...)`	concatenation
`(or` ⟨*regexp*⟩ `...)` `(|` ⟨*regexp*⟩ `...)`	choice
`(?` ⟨*regexp*⟩`)`	zero or one matches
`(*` ⟨*regexp*⟩`)`	zero or more matches
`(+` ⟨*regexp*⟩`)`	one or more matches
`(=` ⟨*number*⟩ ⟨*regexp*⟩`)`	exactly ⟨*number*⟩ matches
`(>=` ⟨*number*⟩ ⟨*regexp*⟩`)`	at least ⟨*number*⟩ matches
`(**` ⟨*number1*⟩ ⟨*number2*⟩ ⟨*regexp*⟩`)`	between ⟨*number1*⟩ and ⟨*number2*⟩ matches (inclusive)

1.1.2 Function specifications

A lexing function is specified by the directive:

```
function ⟨function name⟩ : ⟨type name⟩ =
   ⟨regexp⟩ => ⟨action⟩
   ...
```

This generates a function named ⟨*function name*⟩ with return type ⟨*type name*⟩. That function finds the longest prefix of its input stream that matches any of the given regular expressions and activates the corresponding action. If the longest possible match matches two different regular expressions, CM-Lex prefers the one that appears earlier.

If a function is inexhaustive (*i.e.,* if it is possible for an input stream to satisfy none of the regular expressions), CM-Lex adds a default action that raises an exception. The programmer can add an explicit default action by adding a clause `epsilon => my_default` to the end of the clause list. This ensures that the function is exhaustive, because it always can at least match the empty prefix.

An action name can be used more than once, either in the same function specification, or even in different function specifications, provided that the functions have the same return type.

1.2 Invocation

CM-Lex is invoked from the command line with the specification file's name as its argument. The desired output file name can be specified with the `-o` flag. Otherwise, the output file name is derived from from the input file name by appending the appropriate suffix.

```
cmlex foo.cmlex               # generates foo.cmlex.sml or foo.cmlex.hs
                              #   or foo.cmlex.ml
cmlex foo.cmlex -o bar.sml    # generates bar.sml
cmlex foo.cmlex -o bar.hs     # generates bar.hs
cmlex foo.cmlex -o bar.ml     # generates bar.ml
```

1.3 Streams (Standard ML)

The generated lexer takes its input in the form of a lazy stream, which is a possibly-infinite list in which each element is computed the first time it is needed, and then memoized for future uses.

A stream type is included in CM-Lib, a library packaged with CM-Lex and CM-Yacc. The module `Stream` implements the signature `STREAM`, an abbreviated version of which appears below:

```
signature STREAM =
   sig
      type 'a stream
      datatype 'a front = Nil | Cons of 'a * 'a stream

      val front : 'a stream -> 'a front
      val lazy : (unit -> 'a front) -> 'a stream
   end
```

The `front` function forces the computation of the next element of the stream, if any. It returns a front, which exposes whether the stream is empty, and, if not, gives its next element and the remainder of the stream. The `lazy` function builds a stream from a thunk returning the stream's front. A thunk is used so that the front need not be computed until the stream is forced.

The `STREAM` signature also includes a variety of other functions for building and manipulating streams. For example, `fromString` and `fromTextInstream` build character streams that read input from a string and from an SML IO stream, respectively:

```
val fromString : string -> char stream
val fromTextInstream : TextIO.instream -> char stream
```

A lexer need not use streams, it may use any type that implements the type class `STREAMABLE`, which specifies types that are compatible with the stream forcing operation, `front`:

```
signature STREAMABLE =
   sig
      type 'a t

      datatype 'a front = Nil | Cons of 'a * 'a t
      val front : 'a t -> 'a front
   end
```

The canonical implementation of `STREAMABLE` is `StreamStreamable`, which implements `t` as simply `Stream.stream`. However, the programmer may also use `ListStreamable`, which implements `t` as `list`, or any other way he or she chooses.

1.4 The generated functor (Standard ML)

CM-Lex generates a functor from the given specification file. For example, consider the specification:

```
sml
name LexerFun
alphabet 128

function foo : t =
  (* 'a) => astar

function bar : u =
  (* 'b) => bstar
```

CM-Lex generates the functor:

```
functor LexerFun
   (structure Streamable : STREAMABLE
    structure Arg :
       sig
          type symbol
          val ord : symbol -> int

          type t
          type u

          type self = { bar : symbol Streamable.t -> u,
                        foo : symbol Streamable.t -> t }
          type info = { match : symbol list,
                        len : int,
                        start : symbol Streamable.t,
                        follow : symbol Streamable.t,
                        self : self }

          val astar : info -> t
          val bstar : info -> u
       end)
   :>
   sig
      val bar : Arg.symbol Streamable.t -> Arg.u
      val foo : Arg.symbol Streamable.t -> Arg.t
   end
= ... implementation ...
```

The functor takes in two arguments. The first structure argument is an implementation of of the `STREAMABLE`

type class for the lexer to use. This is typically `StreamStreamable` (so `Streamable.t` = `Stream.stream`), but need not be.

The second structure argument, `Arg`, implements the lexer's actions. First, the programmer defines the type of the symbols that the lexer processes. This is typically `char`, but need not be. Whatever symbol type is chosen, `ord` gives the coercion from symbols to the numbers that CM-Lex uses to identify them.

Next, the programmer implements the types used for the functions' result types. The specification uses two type names, `t` and `u`, so the functor demands `t` and `u` as type arguments.

Next are two type definitions we will return to in a moment, and finally the programmer implements the lexing actions. The specification uses two action names, `astar` and `bstar`, so the functor demands `astar` and `bstar` as value arguments.

Since `astar` was used as an action for `foo`, which was declared to return type `t`, the `astar` action must return type `t`. Similarly, `bstar` must return type `u`.

Once given the two structure arguments, the generated functor returns a structure containing the two lexing functions, `foo` and `bar`. Each one takes in an `Arg.symbol Streamable.t` (typically a `char stream`), and returns the appropriate type.

1.4.1 The match info

When a lexing action is invoked by the lexer, it is passed a record containing various matching information:

- `match`: the actual symbols that were matched.
- `len`: the length of match.
- `follow`: the remaining stream after the matched symbols.
- `start`: the stream on which the lexer was called (*i.e.,* beginning with the matched symbols). This is useful for re-lexing some or all of the stream, as in Lex's `yyless` directive.
- `self`: a record containing the lexing functions being defined, so that actions can reinvoke the lexer.

Note that the lexing functor is *stateless*. Nowhere does it maintain the current input stream. Rather, it is the job of the programmer, through the lexing actions, to manage the input stream.

Typically, an action calls the lexer (through the `self` argument) on the remaining stream (*i.e.,* the `follow` argument). However, the programmer may manipulate the stream as he or she desires. For example, one may push additional symbols onto the stream (as in Lex's `yymore` directive), simply by doing so before calling the lexer.

1.5 Streams (OCaml)

We do not use the stream type in the OCaml standard library, which is imperative. Instead we employ a functional stream type similar to the streams we employ for Standard ML (Section 1.3). The module `FStream` implements streams; its interface is abbreviated below:

```
type 'a stream
type 'a front = Nil | Cons of 'a * 'a stream
type 'a t = 'a stream

val front : 'a stream -> 'a front
val eager : 'a front -> 'a stream
val laz : 'a front Lazy.t -> 'a stream
```

For lazy stream construction, we use OCaml's `Lazy` module, instead of a thunk. One can construct a lazy expression using the `lazy` keyword. Thus, a lazy `Cons` can be written `laz (lazy (Cons (...)))`.

As in Standard ML, a lexer need not use a stream, it may use any type that implements the type class `Streamable.S`:

```
module type S =
   sig
      type 'a t

      type 'a front = Nil | Cons of 'a * 'a t
      val front : 'a t -> 'a front
   end
```

The canonical implementation of `Streamable.S` is `Streamable.StreamStreamable`, which implements `t` as simply `FStream.stream`. However, the programmer may also use `Streamable.ListStreamable`, which implements `t` as `list`, or any other way he or she chooses.

1.6 The generated functor (OCaml)

Consider the specification:

```
ocaml
name LexerFun
alphabet 128

function foo : t =
  (* 'a) => astar

function bar : u =
  (* 'b) => bstar
```

CM-Lex generates the functor:

```
module LexerFun
   (Strm : Streamable.S)
   (Arg :
       sig
          type symbol
          val ord : symbol -> int

          type t
          type u

          type self = { bar : symbol Strm.t -> u;
                        foo : symbol Strm.t -> t }
          type info = (symbol list, symbol Strm.t, self) LexInfo.t

          val astar : info -> t
          val bstar : info -> u
       end)
   :
   sig
      val bar : Arg.symbol Strm.t -> Arg.u
      val foo : Arg.symbol Strm.t -> Arg.t
   end
= ... implementation ...
```

This works in a fashion identical to Standard ML lexers (Section 1.4). Each action is passed a `LexInfo.t`, which is defined as follows:

```
type ('a, 'b, 'c) t =
   { matc : 'a;
     len : int;
     start : 'b;
     follow : 'b;
     self : 'c }
```

In particular, `(symbol list, symbol Strm.t, self) LexInfo.t` (as it is used in the functor) works out to:

```
   { matc : symbol list;
     len : int;
     start : symbol Strm.t;
     follow : symbol Strm.t;
     self : self }
```

The information in the `LexInfo.t` is identical to that in Section 1.4.1. The first field is named `matc` because `match` is a reserved word in OCaml.

1.7 Streams (Haskell)

As in Standard ML, Haskell lexers use a type class `Streamable` that specifies types compatible with the stream forcing operation `front`:

```
data Front a b =
   Nil
 | Cons a b

class Monad m => Streamable s m where
   front :: s a -> m (Front a (s a))
```

A type constructor `s` is streamable (with an associated monad `m`) if it provides a `front` operation that takes an `s a` and then — in the monad `m` — returns a stream front with head `a` and tail `s a`. This differs from the `STREAMABLE` class in Standard ML, which makes no explicit mention of a monad since SML permits side-effects.

One important instance is lists. Lists are purely functional streamables, and thus belong to `Streamable` with any monad (but particularly with `Identity`).

For effectful streams, the canonical instance is the `Stream` type:

```
newtype Stream m a =
   Stream (m (Front a (Stream m a)))
```

For example, streams that result from reading input from a file will typically have the type `Stream Char IO`. Forcing such a stream will result in an IO effect and return a `Front` containing a character and the rest of the stream. The effect occurs because it might be necessary to read the file to obtain the next character.

A `Stream` can be formed from a `Front` in two ways:

```
eager :: Monad m => Front a (Stream m a) -> Stream m a
lazy  :: MonadMemo m => m (Front a (Stream m a)) -> m (Stream m a)
```

When a front is held outright, `eager` coerces it to a stream. However, when one has a front under a monad (the typical case for monadic lexers), one uses `lazy`, which memoizes the monadic front before coercing it to a stream. The memoization means that if the stream is forced more than once, the monadic operation is not repeated; instead the prior result is recalled. This requires that the monad belong to the `MonadMemo` class, which includes `Identity`, `IO`, and `ST`, as well as every monad belonging to `MonadIO`.

The `Stream` module contains a variety of other functions for building and manipulating streams. For example, fromList and fromHandle build character streams that read input from a list (or string) and from an IO handle, respectively:

```
fromList :: Monad m => [a] -> Stream m a
fromHandle :: Handle -> IO (Stream IO Char)
```

1.8 The generated functor (Haskell)

Instantiating the generated code for Haskell is conceptually similar to the process for Standard ML. However, Haskell, unlike Standard ML, does not support functors. Instead, the tool generates an ersatz functor using polymorphic functions. For example, again consider the specification from Section 1.4 (now as a Haskell specification):

```
haskell
name LexerFun
alphabet 128

function foo : t =
  (* 'a) => astar

function bar : u =
  (* 'b) => bstar
```

CM-Lex generates a module exporting:[1]

```
data Arg stream symbol t u =
   Arg { ord :: symbol -> Int,

         {- type arguments -}
         t :: Proxy t,
         u :: Proxy u,

         {- action arguments -}
         astar :: LexInfo stream symbol -> t,
         bstar :: LexInfo stream symbol -> u }

foo :: Streamable stream Identity
       => Arg stream symbol t u -> stream symbol -> t
bar :: Streamable stream Identity
       => Arg stream symbol t u -> stream symbol -> u
```

To use the functor, one constructs an inhabitant of the `Arg` type. As in the Standard ML version, an argument contains an `ord` function that converts symbols to integers, implementations of the type arguments (`t` and `u`) in this example, and implementations of the actions (`astar` and `bstar` in this example). One then calls the functions (`foo` and `bar`) with the argument to obtain the lexing functions.

The type arguments are given by proxy fields using the `Data.Proxy` module in the Haskell standard library. The terms put into the proxy fields are not used; their purpose is solely to specify the types. They are filled in with `Proxy :: Proxy T` for the desired `T`.

Each action is passed a `LexInfo`, which is defined as follows:

[1] For readability, we use unqualified names here.

```
data LexInfo stream symbol =
   LexInfo
   { match :: [symbol],
     len :: Int,
     start :: stream symbol,
     follow :: stream symbol }
```

The information in the `LexInfo` is similar to that from Section 1.4.1. However, note that `LexInfo` does not include a `self` field. In Haskell a `self` field is not necessary: since anything can call anything else, no special mechanism is necessary for an action to reinvoke the lexer.

1.8.1 Monadic lexers

If the `monadic` directive is added to the specification, CM-Lex generates a monadic lexer. In a monadic lexer, the module contains:

```
data Arg stream monad symbol t u =
   Arg { ord :: symbol -> Int,

         {- type arguments -}
         monad :: Proxy monad,
         t :: Proxy t,
         u :: Proxy u,

         {- action arguments -}
         astar :: LexInfo stream symbol -> monad t,
         bstar :: LexInfo stream symbol -> monad u }

foo :: Streamable stream monad
       => Arg stream monad symbol t u -> stream symbol -> monad t
bar :: Streamable stream monad
       => Arg stream monad symbol t u -> stream symbol -> monad u
```

Observe that the types of the actions `astar` and `bstar`, and the lexing functions `foo` and `bar`, now end in `monad t` and `monad u`, intead of `t` and `u`.

A worked example of how to use this appears in Section 5.

1.9 The automaton listing

In the generated file, embedded as a comment immediately after the functor's interface specification, is a listing of each lexing function's state machine. For each lexing function, the listing gives the initial state and the total number of states, and for each state, it gives the state that results from each possible input symbol. Symbols that are not listed, implicitly transition to the error state.

2 CM-Yacc

CM-Yacc is an LALR(1) parser generator for Standard ML and Haskell. It converts a specification file into an SML or Haskell source-code file that implements a parser functor. The programmer may then instantiate that functor with appropriate code for lexing actions to obtain a parser.

2.1 The specification file

A CM-Yacc specification consists of a series of directives:

- `sml` *or* `haskell` *or* `ocaml`

  Specifies the language in which the generated parser should be implemented. This is a required directive, and it must appear first.

- `name` ⟨*identifier*⟩

  Specifies the name of the generated functor for SML/Ocaml parser, and the name of the generated module for Haskell parsers. This is a required directive.

  In Haskell and OCaml specifications, the name must be a valid module identifier, so it must begin with a capital letter. In Haskell, a hierarchical name is permitted. In SML specifications the name is not required to be capitalized, but it is standard practice in SML code to capitalize functor names nonetheless.

- `terminal` ⟨*name*⟩ [`of` ⟨*type name*⟩] [⟨*precedence*⟩]

  Declares a terminal. If the "`of` ⟨*type name*⟩" form is used, then the terminal carries a value of type ⟨*type name*⟩. Otherwise it carries no value. If the ⟨*precedence*⟩ form is used, the terminal is marked with a precedence, which is used to resolve shift-reduce conflicts.

- `nonterminal` ⟨*name*⟩ `:` ⟨*type name*⟩ `=` ⟨*production*⟩ `...` ⟨*production*⟩

  Declares a nonterminal carrying a value of type ⟨*type name*⟩. The nonterminal expands using the productions in ⟨*production list*⟩.

- `start` ⟨*name*⟩

  Indicates that the nonterminal ⟨*name*⟩ is the start symbol. This is a required directive.

- `follower` ⟨*name*⟩

  Indicates that the terminal ⟨*name*⟩ is permitted to follow the start symbol. (In other words, it is permitted to be the first symbol after a complete parse.) If no followers are given, only the end of the stream is permitted to follow the start symbol.

- `data` ⟨*identifier*⟩

  Haskell specifications only: Specifies the name that the tool should give to the datatype of terminals. This is a required directive. (In Standard ML, the datatype of terminals is always named `terminal`.)

- `monadic`

  Haskell specifications only: Specifies that CM-Lex should generate a monadic parser.

Comments are written "`/*` *comment text* `*/`" and may appear anywhere in the specification. Comments may be nested.

2.1.1 Productions

Each production has the form:

⟨*production*⟩ ::=
  ⟨*right-hand-side*⟩ `=>` ⟨*action*⟩ [⟨*precedence*⟩]

⟨*right-hand-side*⟩ ::=
  ⟨*constituent*⟩ `...` ⟨*constituent*⟩

⟨*constituent*⟩ ::=
  ⟨*terminal-or-nonterminal*⟩
  ⟨*label*⟩ `:` ⟨*terminal-or-nonterminal*⟩
  `(` ⟨*constituent*⟩ `)`

⟨*label*⟩ ::=
  ⟨*positive integer*⟩
  ⟨*identifier*⟩

Each production gives a right-hand-side (made up of terminals and nonterminals) into which the nonterminal being defined can expand. The right-hand-side may be empty. When a production is used, the action is called to convert the data carried by the right-hand-side's constituents into the data carried by the nonterminal.

The labels on constituents determine which constituents' data are passed to the action. If a constituent omits the label, it passes no data to the action. If no constituents pass data to the action, the domain type of the action function will be `unit` (`()` in Haskell). Otherwise, the labels determine the action's domain type.

A label is either a positive integer or an identifier. In a single production, all the labels must be of the same sort (*i.e.,* all numbers or all identifiers). If the labels are identifiers, the constituents' data are passed in a record, and the labels are used as the names of the record's fields. If the labels are numbers, all the numbers from 1 to n must be used, and the action's domain is an n-tuple. The numbers indicate the order in which the constituents' data appear in the tuple. *Except:* If only the label 1 is used, the datum is passed bare, not in a 1-tuple.

For example, with the directive:

```
nonterminal Production : production =
   2:Constituents ARROW 1:Ident 3:Precedence => single_production
```

the action `single_production` has type `ident * constitutents * precedence -> production` (assuming that `Ident`, `Constituents`, and `Precedence` are declared to carry types `ident`, `constitutents`, and `precedence`, respectively).

2.1.2 Precedence

If precedences are used, the tool consults them to resolve shift-reduce conflicts. They have the form:

⟨*precedence*⟩ ::=

`precl` ⟨*number*⟩	left associative with precedence ⟨*number*⟩
`precr` ⟨*number*⟩	right associative with precedence ⟨*number*⟩
`noprec`	no precedence

Precedences range from 0 to 100, with higher numbers indicating higher precedence (*i.e.,* binding more tightly). For a terminal, a `noprec` precedence is equivalent to omitting the precedence. For a production, if no precedence is given, the production's precedence is inferred from the precedence of its last terminal. A production has no precedence if it is given precedence `noprec`, if it contains no terminals, or if its last terminal does not have a precedence.

2.2 Invocation

CM-Yacc is invoked from the command line with the specification file's name as its argument. The desired output file name can be specified with the `-o` flag. Otherwise, the output file name is derived from from the input file name by appending the appropriate suffix.

```
cmyacc foo.cmyacc               # generates foo.cmyacc.sml or foo.cmyacc.hs
                                #   or foo.cmyacc.ml
cmyacc foo.cmyacc -o bar.sml    # generates bar.sml
cmyacc foo.cmyacc -o bar.hs     # generates bar.hs
cmyacc foo.cmyacc -o bar.ml     # generates bar.ml
```

2.3 The generated functor (Standard ML)

CM-Yacc generates a functor from the given specification file. For example, consider the specification:

```
sml
name ParserFun

terminal NUMBER of t
terminal PLUS
terminal TIMES

nonterminal Factor : t =
  1:NUMBER => number_factor
  1:NUMBER TIMES 2:Factor => times_factor

nonterminal Term : t =
  1:Factor => factor_term
  1:Factor PLUS 2:Term => plus_term

start Term
```

CM-Yacc generates the functor:

```
functor ParserFun
   (structure Streamable : STREAMABLE
    structure Arg :
       sig
          type t

          val plus_term : t * t -> t
          val factor_term : t -> t
          val times_factor : t * t -> t
          val number_factor : t -> t

          datatype terminal =
             NUMBER of t
           | PLUS
           | TIMES

          val error : terminal Streamable.t -> exn
       end)
   :>
   sig
      val parse : Arg.terminal Streamable.t -> Arg.t * Arg.terminal Streamable.t
   end
= ... implementation ...
```

As with CM-Lex, the functor takes two arguments, a `STREAMABLE` and an argument that implements the parser's actions. The `Arg` structure also must supply the datatype that implements terminals, and it must supply an error function. The error function is called when the parser detects a syntax error. It is passed the stream of terminals beginning with the terminal that generated the error, and it should return an `exn`, which the parser then raises.

The result of the functor is a function `parse`. It takes in a stream of terminals, and it returns a pair: the data carried by the start symbol, and the stream of terminals that follows the complete parse. (If no followers are specified, the stream that follows the parse is always empty.)

2.4 The generated functor (OCaml)

Consider the specification:

```
ocaml
name ParserFun

terminal NUMBER of t
terminal PLUS
terminal TIMES

nonterminal Factor : t =
  1:NUMBER => number_factor
  1:NUMBER TIMES 2:Factor => times_factor

nonterminal Term : t =
  1:Factor => factor_term
  1:Factor PLUS 2:Term => plus_term

start Term
```

CM-Yacc generates the functor:

```
module ParserFun
   (Strm : Streamable.S)
   (Arg :
       sig
          type t

          val plus_term : t -> t -> t
          val factor_term : t -> t
          val times_factor : t -> t -> t
          val number_factor : t -> t

          type terminal =
             NUMBER of t
           | PLUS
           | TIMES

          val error : terminal Strm.t -> exn
       end)
   :
   sig
      val parse : Arg.terminal Strm.t -> Arg.t * Arg.terminal Strm.t
   end
= ... implementation ...
```

It works in a fashion identical to Standard ML parsers (Section 2.3).

2.5 The generated functor (Haskell)

The generated code is an ersatz functor, similar to the one generated by CM-Lex (Section 1.8). For example, consider the specification:

```
haskell
name ParserFun

terminal NUMBER of t
terminal PLUS
terminal TIMES

nonterminal Factor : t =
  1:NUMBER => number_factor
  1:NUMBER TIMES 2:Factor => times_factor

nonterminal Term : t =
  1:Factor => factor_term
  1:Factor PLUS 2:Term => plus_term

start Term

data Terminal
```

CM-Yacc generates a module exporting:[2]

```
data Terminal t =
   NUMBER t
 | PLUS
 | TIMES

data Arg stream t =
   Arg { error :: stream (Terminal t) -> SomeException,

         {- type arguments -}
         t :: Proxy t,

         {- action arguments -}
         plus_term :: t -> t -> t,
         factor_term :: t -> t,
         times_factor :: t -> t -> t,
         number_factor :: t -> t }

parse :: Streamable stream Identity
         => Arg stream t -> stream (Terminal t) -> (t, stream (Terminal t))
```

If the `monadic` directive is added to the specification, CM-Yacc instead generates:

[2]For readability, we use unqualified names here.

```
data Terminal t =
   NUMBER t
 | PLUS
 | TIMES

data Arg stream monad t =
   Arg { error :: stream (Terminal t) -> monad SomeException,

         {- type arguments -}
         monad :: Proxy monad,
         t :: Proxy t,

         {- action arguments -}
         plus_term :: t -> t -> t,
         factor_term :: t -> t,
         times_factor :: t -> t -> t,
         number_factor :: t -> t }

parse :: Streamable stream monad
         => Arg stream monad t
               -> stream (Terminal t) -> monad (t, stream (Terminal t))
```

A worked example of how to use this appears in Section 5.

2.6 The automaton listing

In the generated file, embedded as a comment immediately after the functor's interface specification, is a listing of the parsing function's state machine. Its form should be familiar to users of Yacc's automaton listings.

The bulk of the listing is a list of the automaton's states. For example:

```
State 4:

0 : Factor -> . NUMBER  / 1
1 : Factor -> . NUMBER TIMES Factor  / 1
2 : Term -> . Factor  / 0
3 : Term -> . Factor PLUS Term  / 0
3 : Term -> Factor PLUS . Term  / 0

NUMBER => shift 2
Factor => goto 1
Term => goto 6
```

The state begins with a list of LR(1) items, each of which gives one scenario in which the parser could be in the state. For example, the item

```
3 : Term -> Factor PLUS . Term  / 0
```

means that the parser could be in the process of recognizing `Factor PLUS Term` which it would then reduce to `Term` using the 3rd production (counting from zero). The dot is a cursor, indicating that so far it has seen `Factor PLUS`. The "`/ 0`" gives the production's lookahead set: the set of terminals that can follow this production.

The lookahead sets are given after the state listings, such as:

```
lookahead 0 = $
lookahead 1 = $ PLUS
```

The `$` is a special terminal representing the end of the stream. In lookahead set 0, only the end of the stream can follow the production. In lookahead set 1, the `PLUS` terminal can also follow it.

The LR(1) items are given to help the user understand the parser the tool has constructed. Following the LR(1) is the state's action table. For each terminal, the action table indicates shift, reduce, or error. (Error transitions are omitted from the listing.) In a `shift` n transition, the parser consumes the terminal and transitions to state n. In a `reduce` n transition, the parser leaves the terminal on the stream, reduces using production n, and retrieves an old state from its stack.

For example, in the state

```
State 2:

0 : Factor -> NUMBER .  / 1
1 : Factor -> NUMBER . TIMES Factor  / 1

$ => reduce 0
PLUS => reduce 0
TIMES => shift 5
```

the parser will reduce using production 0 if it sees either end-of-stream or `PLUS`. If it sees `TIMES` it will shift it and transition to a state that includes the item:

```
1 : Factor -> NUMBER TIMES . Factor  / 1
```

Following the action table is the state's goto table. When a state is re-entered after a reduce, the goto table tells which state the parser should enter, depending on which nonterminal was just reduced.

2.7 Shift-reduce conflicts

When a grammar is ambiguous, an action table may list multiple possible actions. For example, consider:

```
name ParserFun

terminal NUMBER of t
terminal PLUS
terminal TIMES precl 0

nonterminal Term : t =
  1:NUMBER => number_term
  1:Term PLUS 2:Term => plus_term
  1:Term TIMES 2:Term => times_factor

start Term
```

The grammar is ambiguous, and it declares that `TIMES` is left associative, but it gives no precedence for `PLUS`. Its automaton contains the state:

```
State 5:

1 : Term -> Term . PLUS Term  / 1
2 : Term -> Term . TIMES Term  / 1
2 : Term -> Term TIMES Term .  / 1

$ => reduce 2
PLUS => shift 4, reduce 2  CONFLICT
TIMES => reduce 2, shift 3  PRECEDENCE
```

In this state, only the end-of-stream action is unambiguous. When `PLUS` is seen, the parser could either shift, or reduce using production 2. The actual generated functor uses the first action listed. When `TIMES` is seen, the parser could either shift or reduce, but the left associativity of TIMES indicates that the parser should resolve the conflict in favor of the reduce.

Thus, the `CONFLICT` notation marks an unresolved ambiguity, while the `PRECEDENCE` notation marks an ambiguity resolved by precedence.

Whenever there is an unresolved shift-reduce conflict, the tool prefers the shift. When there is a reduce-reduce conflict, the tool chooses an arbitrary production. Reduce-reduce conflicts usually indicate a serious problem with the grammar, so once one has been detected, the tool no longer uses precedence to resolve shift-reduce conflicts.

3 A Standard ML example

As a more realistic example, we implement a calculator that processes an input stream and returns its value. For simplicity, the calculator stops at the first illegal character (which might be the end of the stream). The lexer specification is:

```
sml
name CalcLexFun
alphabet 128

set digit = (range '0 '9)
set whitechar = (or 32 9 10)  /* space, tab, lf */

function lex : t =
  (+ digit) => number
  '+ => plus
  '* => times
  '( => lparen
  ') => rparen
  (+ whitechar) => whitespace

  /* Stop at the first illegal character */
  epsilon => eof
```

which generates the functor:

```
functor CalcLexFun
   (structure Streamable : STREAMABLE
    structure Arg :
       sig
          type symbol
          val ord : symbol -> int

          type t

          type self = { lex : symbol Streamable.t -> t }
          type info = { match : symbol list,
                        len : int,
                        start : symbol Streamable.t,
                        follow : symbol Streamable.t,
                        self : self }

          val eof : info -> t
          val lparen : info -> t
          val number : info -> t
          val plus : info -> t
          val rparen : info -> t
          val times : info -> t
          val whitespace : info -> t
       end)
   :>
   sig
      val lex : Arg.symbol Streamable.t -> Arg.t
   end
= ... implementation ...
```

The parser specification is:

```
sml
name CalcParseFun

terminal NUMBER of t
terminal PLUS
terminal TIMES
terminal LPAREN
terminal RPAREN

nonterminal Atom : t =
  1:NUMBER => number_atom
  LPAREN 1:Term RPAREN => paren_atom

nonterminal Factor : t =
  1:Atom => atom_factor
  1:Atom TIMES 2:Factor => times_factor

nonterminal Term : t =
  1:Factor => factor_term
  1:Factor PLUS 2:Term => plus_term

start Term
```

which generates the functor:

```
    functor CalcParseFun
       (structure Streamable : STREAMABLE
        structure Arg :
           sig
              type t

              val plus_term : t * t -> t
              val factor_term : t -> t
              val times_factor : t * t -> t
              val atom_factor : t -> t
              val paren_atom : t -> t
              val number_atom : t -> t

              datatype terminal =
                 NUMBER of t
               | PLUS
               | TIMES
               | LPAREN
               | RPAREN

              val error : terminal Streamable.t -> exn
           end)
       :>
       sig
          val parse : Arg.terminal Streamable.t -> Arg.t * Arg.terminal Streamable.t
       end
    = ... implementation ...
```

We then assemble the calculator as follows:

```
structure Calculator
  :>
  sig
    val calc : char Stream.stream -> int
  end
  =
  struct
     open Stream

     datatype terminal =
        NUMBER of int
      | PLUS
      | TIMES
      | LPAREN
      | RPAREN

     structure Lexer =
        CalcLexFun
        (structure Streamable = StreamStreamable
         structure Arg =
            struct
```

```
            type symbol = char
            val ord = Char.ord

            type t = terminal front

            type self = { lex : char stream -> t }
            type info = { match : char list,
                          follow : char stream,
                          self : self,
                          len : int,
                          start : char stream }

            fun number ({ match, follow, self, ... }:info) =
              Cons (NUMBER (Option.valOf (Int.fromString (String.implode match))),
                    lazy (fn () => #lex self follow))

            fun simple terminal ({ follow, self, ... }:info) =
              Cons (terminal, lazy (fn () => #lex self follow))

            val plus = simple PLUS
            val times = simple TIMES
            val lparen = simple LPAREN
            val rparen = simple RPAREN

            fun whitespace ({ follow, self, ...}:info) =
              #lex self follow

            fun eof _ = Nil
          end)

    structure Parser =
       CalcParseFun
       (structure Streamable = StreamStreamable
        structure Arg =
           struct
             type t = int

             fun id x = x

             val number_atom = id
             val paren_atom = id
             val atom_factor = id
             fun times_factor (x, y) = x * y
             val factor_term = id
             fun plus_term (x, y) = x + y

             datatype terminal = datatype terminal

             fun error _ = Fail "syntax error"
           end)

    fun calc strm = #1 (Parser.parse (lazy (fn () => Lexer.lex strm)))
end
```

4 An OCaml example

To build the calculator example in OCaml is similar to Standard ML. The lexer specification is:

```
ocaml
name Fun
alphabet 128

set digit = (range '0 '9)
set whitechar = (or 32 9 10)  /* space, tab, lf */

function lex : t =
  (+ digit) => number
  '+ => plus
  '* => times
  '( => lparen
  ') => rparen
  (+ whitechar) => whitespace

  /* Stop at the first illegal character */
  epsilon => eof
```

which generates the functor:

```
module Fun
   (Strm : Streamable.S)
   (Arg :
       sig
          type symbol
          val ord : symbol -> int

          type t

          type self = { lex : symbol Strm.t -> t }
          type info = (symbol list, symbol Strm.t, self) LexInfo.t

          val eof : info -> t
          val lparen : info -> t
          val number : info -> t
          val plus : info -> t
          val rparen : info -> t
          val times : info -> t
          val whitespace : info -> t
       end)
   :
   sig
      val lex : Arg.symbol Strm.t -> Arg.t
   end
= ... implementation ...
```

The parser specification is:

```
ocaml
name Fun

terminal NUMBER of t
terminal PLUS
terminal TIMES
terminal LPAREN
terminal RPAREN

nonterminal Atom : t =
  1:NUMBER => number_atom
  LPAREN 1:Term RPAREN => paren_atom

nonterminal Factor : t =
  1:Atom => atom_factor
  1:Atom TIMES 2:Factor => times_factor

nonterminal Term : t =
  1:Factor => factor_term
  1:Factor PLUS 2:Term => plus_term

start Term
```

which generates the functor:

```
module Fun
   (Strm : Streamable.S)
   (Arg :
       sig
          type t

          val plus_term : t -> t -> t
          val factor_term : t -> t
          val times_factor : t -> t -> t
          val atom_factor : t -> t
          val paren_atom : t -> t
          val number_atom : t -> t

          type terminal =
             NUMBER of t
           | PLUS
           | TIMES
           | LPAREN
           | RPAREN

          val error : terminal Strm.t -> exn
       end)
   :
   sig
      val parse : Arg.terminal Strm.t -> Arg.t * Arg.terminal Strm.t
   end
= ... implementation ...
```

The calculator satisfies the interface:

```
val calc : char FStream.stream -> int
```

We assume that the lexer and parser functors reside in files `CalcLex.ml` and `CalcParse.ml` and thus reside in modules named `CalcLex` and `CalcParse`. We then assemble the calculator as follows:

```
module Terminal =
   struct
      type terminal =
         NUMBER of int
       | PLUS
       | TIMES
       | LPAREN
       | RPAREN
   end

module Lexer =
   CalcLex.Fun
   (Streamable.StreamStreamable)
   (struct
       open FStream
```

```
        open LexInfo
        open Terminal

        type symbol = char
        let ord = int_of_char

        type t = terminal front

        type self = { lex : char stream -> t }
        type info = (symbol list, symbol FStream.stream, self) LexInfo.t

        let number { matc; follow; self } =
          Cons (NUMBER (int_of_string (StringUtil.implode matc)),
                laz (lazy (self.lex follow)))

        let simple terminal { follow; self } =
          Cons (terminal, laz (lazy (self.lex follow)))

        let plus = simple PLUS
        let times = simple TIMES
        let lparen = simple LPAREN
        let rparen = simple RPAREN

        let whitespace { follow; self } =
          self.lex follow

        let eof _ = Nil
      end)

module Parser =
   CalcParse.Fun
   (Streamable.StreamStreamable)
   (struct
       type t = int

       let id x = x

       let number_atom = id
       let paren_atom = id
       let atom_factor = id
       let times_factor x y = x * y
       let factor_term = id
       let plus_term x y = x + y

       include Terminal

       let error _ = Failure "syntax error"
     end)

let calc strm = fst (Parser.parse (FStream.laz (lazy (Lexer.lex strm))))
```

5 A Haskell example

Here we show how to build the calculator example in Haskell. The lexer specification is:

```
haskell
name CalcLexFun
alphabet 128

set digit = (range '0 '9)
set whitechar = (or 32 9 10)  /* space, tab, lf */

function lex : t =
  (+ digit) => number
  '+ => plus
  '* => times
  '( => lparen
  ') => rparen
  (+ whitechar) => whitespace

  /* Stop at the first illegal character */
  epsilon => eof

monadic
```

which generates a module exporting:

```
data Arg stream monad symbol t =
   Arg { ord :: symbol -> Int,

         {- type arguments -}
         monad :: Proxy.Proxy monad,
         t :: Proxy.Proxy t,

         {- action arguments -}
         eof :: LexEngine.LexInfo stream symbol -> monad t,
         lparen :: LexEngine.LexInfo stream symbol -> monad t,
         number :: LexEngine.LexInfo stream symbol -> monad t,
         plus :: LexEngine.LexInfo stream symbol -> monad t,
         rparen :: LexEngine.LexInfo stream symbol -> monad t,
         times :: LexEngine.LexInfo stream symbol -> monad t,
         whitespace :: LexEngine.LexInfo stream symbol -> monad t }

lex :: LexEngine.Streamable stream monad
       => CalcLexFun.Arg stream monad symbol t -> stream symbol -> monad t
```

The parser specification is:

```
haskell
name CalcParseFun

terminal NUMBER of t
terminal PLUS
terminal TIMES
terminal LPAREN
terminal RPAREN

nonterminal Atom : t =
  1:NUMBER => number_atom
  LPAREN 1:Term RPAREN => paren_atom

nonterminal Factor : t =
  1:Atom => atom_factor
  1:Atom TIMES 2:Factor => times_factor

nonterminal Term : t =
  1:Factor => factor_term
  1:Factor PLUS 2:Term => plus_term

start Term

data Terminal

monadic
```

which generates a module exporting:

```
data Terminal t =
   NUMBER t
 | PLUS
 | TIMES
 | LPAREN
 | RPAREN

data Arg stream monad t =
   Arg { error :: stream (Terminal t) -> monad Control.Exception.SomeException,

         {- type arguments -}
         monad :: Proxy.Proxy monad,
         t :: Proxy.Proxy t,

         {- action arguments -}
         plus_term :: t -> t -> t,
         factor_term :: t -> t,
         times_factor :: t -> t -> t,
         atom_factor :: t -> t,
         paren_atom :: t -> t,
         number_atom :: t -> t }

parse :: ParseEngine.Streamable stream monad
         => CalcParseFun.Arg stream monad t -> stream (Terminal t)
               -> monad (t, stream (Terminal t))
```

We assume that the lexer and parser functors reside in files `CalcLexFun.hs` and `CalcParseFun.hs` and thus reside in modules named `CalcLexFun` and `CalcParseFun`. We then assemble the calculator as follows:

```
module Calc where

import Data.Proxy
import Data.Char as Char
import Control.Exception
import Util.Stream
import Util.LexEngine
import CalcLexFun as Lex
import CalcParseFun as Parse

type Term = Terminal Int


{- The lexer -}

simple :: Term -> LexInfo (Stream IO) Char -> IO (Front Term (Stream IO Term))
simple terminal info =
   do {
      t <- lazy (Calc.lex (follow info));
      return (Cons terminal t)
      }
```

```
lexarg =
   Lex.Arg
      {
      Lex.ord = Char.ord,

      Lex.monad = Proxy :: Proxy IO,
      Lex.t = Proxy :: Proxy (Front Term (Stream IO Term)),

      Lex.number =
         (\ info ->
             do {
                t <- lazy (Calc.lex (follow info));
                return (Cons (NUMBER (read (match info))) t)
                }),

      Lex.lparen = simple LPAREN,
      Lex.rparen = simple RPAREN,
      Lex.plus = simple PLUS,
      Lex.times = simple TIMES,

      Lex.whitespace =
         (\ info -> Calc.lex (follow info)),

      Lex.eof =
         (\ info -> return Nil )
      }


lex :: Stream IO Char -> IO (Front Term (Stream IO Term))
lex s = Lex.lex lexarg s


{- The parser -}

newtype SyntaxError = SyntaxError (Stream IO Term)
instance Show SyntaxError where
   show _ = "syntax error"
instance Exception SyntaxError


parsearg =
   Parse.Arg
      {
      Parse.error =
         (\ s -> return (toException (SyntaxError s))),

      Parse.monad = Proxy :: Proxy IO,
      Parse.t = Proxy :: Proxy Int,

      Parse.plus_term = (+),
      Parse.times_factor = (*),

      Parse.number_atom = id,
      Parse.paren_atom = id,
```

```
      Parse.atom_factor = id,
      Parse.factor_term = id
      }


calc :: Stream IO Char -> IO Int
calc s =
   do {
      s <- lazy (Calc.lex s);
      (x, _) <- Parse.parse parsearg s;
      return x
      }
```

A Installation

To install CM-Lex and CM-Yacc, obtain the source code either from the distribution at:

`www.cs.cmu.edu/~crary/cmtool/`

or from Github at `kcrary/cmtool`. (The project uses a submodule, so if you clone the project from Github, use the `--recursive` option.)

Then follow the instructions in the INSTALL file. You will need either MLton or Standard ML of New Jersey installed. This manual is included as `manual.pdf`.