Alex

Simon Marlow and the Alex developers

Feb 28, 2022

CONTENTS

Alex is a tool for generating lexical analysers in Haskell, given a description of the tokens to be recognised in the form of regular expressions. It is similar to the tool “lex” or “flex” for C/C++.

CHAPTER

ONE

ABOUT ALEX

Alex can always be obtained from its home page. The latest source code lives in the git repository on `GitHub`.

1.1 Reporting bugs in Alex

Please report bugs on the Alex issue tracker. There are no specific mailing lists for the discussion of Alex-related matters, but such topics should be fine on the Haskell Cafe mailing list.

1.2 License

1.3 About this documentation

This documentation is based on a DocBook documentation originally written by Chris Dornan, Isaac Jones, and Simon Marlow.

Converted to RST / Sphinx / readthedocs.org by Andreas Abel in February 2022.

CHAPTER

TWO

INTRODUCTION

Alex is a tool for generating lexical analysers in Haskell, given a description of the tokens to be recognised in the form of regular expressions. It is similar to the tools lex and flex for C/C++.

Alex takes a description of tokens based on regular expressions and generates a Haskell module containing code for scanning text efficiently. Alex is designed to be familiar to existing lex users, although it does depart from lex in a number of ways.

A sample specification would be the following:

```
{
module Main (main) where
}

%wrapper "basic"

$digit = 0-9            -- digits
$alpha = [a-zA-Z]       -- alphabetic characters

tokens :-

  $white+                       ;
  "--".*                        ;
  let                           { \s -> Let }
  in                            { \s -> In }
  $digit+                       { \s -> Int (read s) }
  [\=\+\-\*\/\(\)]              { \s -> Sym (head s) }
  $alpha [$alpha $digit \_ \']*  { \s -> Var s }

{
-- Each action has type :: String -> Token

-- The token type:
data Token
  = Let
  | In
  | Sym Char
  | Var String
  | Int Int
  deriving (Eq, Show)

main = do
```

(continues on next page)

(continued from previous page)

```
  s <- getContents
  print (alexScanTokens s)
}
```

The first few lines between the { and } provide a code scrap (some inlined Haskell code) to be placed directly in the output, the scrap at the top of the module is normally used to declare the module name for the generated Haskell module, in this case `Main`.

The next line, `%wrapper "basic"` controls what kind of support code Alex should produce along with the basic scanner. The `basic` wrapper selects a scanner that tokenises a `String` and returns a list of tokens. Wrappers are described fully in *The Interface to an Alex-generated lexer*.

The next two lines define the `$digit` and `$alpha` macros for use in the token definitions.

The '`tokens :-`' line ends the macro definitions and starts the definition of the scanner.

The scanner is specified as a series of token definitions where each token specification takes the form of

```
regexp   { code }
```

The meaning of this rule is "if the input matches <regexp>, then return <code>". The code part along with the braces can be replaced by simply '`;`', meaning that this token should be ignored in the input stream. As you can see, we've used this to ignore whitespace in our example.

Our scanner is set up so that the actions are all functions with type `String->Token`. When the token is matched, the portion of the input stream that it matched is passed to the appropriate action function as a `String`.

At the bottom of the file we have another code fragment, surrounded by braces `{ ... }`. In this fragment, we declare the type of the tokens, and give a `main` function that we can use for testing it; the `main` function just tokenises the input and prints the results to standard output.

Alex has kindly provided the following function which we can use to invoke the scanner:

```
alexScanTokens :: String -> [Token]
```

Alex arranges for the input stream to be tokenised, each of the action functions to be passed the appropriate `String`, and a list of `Token`s returned as the result. If the input stream is lazy, the output stream will also be produced lazily[1].

We have demonstrated the simplest form of scanner here, which was selected by the `%wrapper "basic"` line near the top of the file. In general, actions do not have to have type `String->Token`, and there's no requirement for the scanner to return a list of tokens.

With this specification in the file `Tokens.x`, Alex can be used to generate `Tokens.hs`:

```
$ alex Tokens.x
```

If the module needed to be placed in a different file, `Main.hs` for example, then the output filename can be specified using the `-o` option:

```
$ alex Tokens.x -o Main.hs
```

The resulting module is Haskell 98 compatible. It can also be readily used with a Happy parser.

[1] That is, unless you have any patterns that require a long lookahead.

CHAPTER

THREE

ALEX FILES

In this section we describe the layout of an Alex lexical specification.

We begin with the lexical syntax; elements of the lexical syntax are referred to throughout the rest of this documentation, so you may need to refer back to the following section several times.

3.1 Lexical syntax

Alex's lexical syntax is given below. It is written as a set of macro definitions using Alex's own syntax. These macros are used in the BNF specification of the syntax later on.

```
$digit      = [0-9]
$octdig     = [0-7]
$hexdig     = [0-9A-Fa-f]
$special    = [\.\;\,\$\|\*\+\?\#\~\-\{\}\(\)\[\]\^\/]
$graphic    = $printable # $white

@string     = \" ($graphic # \")* \"
@id         = [A-Za-z][A-Za-z'_]*
@smac       = '$' id
@rmac       = '@' id
@char       = ($graphic # $special) | @escape
@escape     = '\\' ($printable | 'x' $hexdig+ | 'o' $octdig+ | $digit+)
@code       = -- curly braces surrounding a Haskell code fragment
```

3.2 Syntax of Alex files

In the following description of the Alex syntax, we use an extended form of BNF, where optional phrases are enclosed in square brackets (`[ ... ]`), and phrases which may be repeated zero or more times are enclosed in braces (`{ ... }`). Literal text is enclosed in single quotes.

An Alex lexical specification is normally placed in a file with a `.x` extension. Alex source files are encoded in UTF-8, just like Haskell source files[2].

The overall layout of an Alex file is:

```
alex := [ @code ] [ wrapper ] [ encoding ] { macrodef } @id ':-' { rule } [ @code ]
```

[2] Strictly speaking, GHC source files.

The file begins and ends with optional code fragments. These code fragments are copied verbatim into the generated source file.

At the top of the file, the code fragment is normally used to declare the module name and some imports, and that is all it should do: don't declare any functions or types in the top code fragment, because Alex may need to inject some imports of its own into the generated lexer code, and it does this by adding them directly after this code fragment in the output file.

Next comes an optional directives section

The first kind of directive is a specification:

```
wrapper := '%wrapper' @string
```

wrappers are described in *Wrappers*. This can be followed by an optional encoding declaration:

```
encoding := '%encoding' @string
```

encodings are described in *Unicode and UTF-8*.

Additionally, you can specify a token type, a typeclass, or an action type (depending on what wrapper you use):

```
action type := '%action' @string
```

```
token type := '%token' @string
```

```
typeclass(es) := '%typeclass' @string
```

these are described in *Type Signatures and Typeclasses*.

3.2.1 Macro definitions

Next, the lexer specification can contain a series of macro definitions. There are two kinds of macros, character set macros, which begin with a `$`, and regular expression macros, which begin with a `@`. A character set macro can be used wherever a character set is valid (see *Syntax of character sets*), and a regular expression macro can be used wherever a regular expression is valid (see *Regular Expression*).

```
macrodef  :=  @smac '=' set
          |   @rmac '=' regexp
```

3.2.2 Rules

The rules are heralded by the sequence '`id :-`' in the file. It doesn't matter what you use for the identifier, it is just there for documentation purposes. In fact, it can be omitted, but the `:-` must be left in.

The syntax of rules is as follows:

```
rule       := [ startcodes ] token
            | startcodes '{' { token } '}'

token      := [ left_ctx ] regexp [ right_ctx ]  rhs

rhs        := @code | ';'
```

Each rule defines one token in the lexical specification. When the input stream matches the regular expression in a rule, the Alex lexer will return the value of the expression on the right hand side, which we call the action. The action can be any Haskell expression. Alex only places one restriction on actions: all the actions must have the same type. They can be values in a token type, for example, or possibly operations in a monad. More about how this all works is in *The Interface to an Alex-generated lexer*.

The action may be missing, indicated by replacing it with ';', in which case the token will be skipped in the input stream.

Alex will always find the longest match. For example, if we have a rule that matches whitespace:

```
$white+      ;
```

Then this rule will match as much whitespace at the beginning of the input stream as it can. Be careful: if we had instead written this rule as

```
$white*      ;
```

then it would also match the empty string, which would mean that Alex could never fail to match a rule!

When the input stream matches more than one rule, the rule which matches the longest prefix of the input stream wins. If there are still several rules which match an equal number of characters, then the rule which appears earliest in the file wins.

Contexts

Alex allows a left and right context to be placed on any rule:

```
left_ctx   := '^'
            | set '^'

right_ctx  := '$'
            | '/' regexp
            | '/' @code
```

The left context matches the character which immediately precedes the token in the input stream. The character immediately preceding the beginning of the stream is assumed to be '\n'. The special left-context '^' is shorthand for '\n^'.

Right context is rather more general. There are three forms:

/ regexp This right-context causes the rule to match if and only if it is followed in the input stream by text which matches <regexp>.

NOTE: this should be used sparingly, because it can have a serious impact on performance. Any time this rule *could* match, its right-context will be checked against the current input stream.

$ Equivalent to '/\n'.

/ { ... } This form is called a *predicate* on the rule. The Haskell expression inside the curly braces should have type:

```
{ ... } :: user       -- predicate state
        -> AlexInput  -- input stream before the token
        -> Int        -- length of the token
        -> AlexInput  -- input stream after the token
        -> Bool       -- True <=> accept the token
```

Alex will only accept the token as matching if the predicate returns `True`.

See *The Interface to an Alex-generated lexer* for the meaning of the `AlexInput` type. The `user` argument is available for passing into the lexer a special state which is used by predicates; to give this argument a value, the `alexScanUser` entry point to the lexer must be used (see *Basic interface*).

Start codes

Start codes are a way of adding state to a lexical specification, such that only certain rules will match for a given state.

A startcode is simply an identifier, or the special start code '`0`'. Each rule may be given a list of startcodes under which it applies:

```
startcode  := @id | '0'
startcodes := '<' startcode { ',' startcode } '>'
```

When the lexer is invoked to scan the next token from the input stream, the start code to use is also specified (see *The Interface to an Alex-generated lexer*). Only rules that mention this start code are then enabled. Rules which do not have a list of startcodes are available all the time.

Each distinct start code mentioned in the lexical specification causes a definition of the same name to be inserted in the generated source file, whose value is of type `Int`. For example, if we mentioned startcodes `foo` and `bar` in the lexical spec, then Alex will create definitions such as:

```
foo = 1
bar = 2
```

in the output file.

Another way to think of start codes is as a way to define several different (but possibly overlapping) lexical specifications in a single file, since each start code corresponds to a different set of rules. In concrete terms, each start code corresponds to a distinct initial state in the state machine that Alex derives from the lexical specification.

Here is an example of using startcodes as states, for collecting the characters inside a string:

```
<0>      ([^\"] | \n)*  ;
<0>      \"             { begin string }
<string> [^\"]          { stringchar }
<string> \"             { begin 0 }
```

When it sees a quotation mark, the lexer switches into the `string` state and each character thereafter causes a `stringchar` action, until the next quotation mark is found, when we switch back into the `0` state again.

From the lexer's point of view, the startcode is just an integer passed in, which tells it which state to start in. In order to actually use it as a state, you must have some way for the token actions to specify new start codes - *The Interface to an Alex-generated lexer* describes some ways this can be done. In some applications, it might be necessary to keep a *stack* of start codes, where at the end of a state we pop the stack and resume parsing in the previous state. If you want this functionality, you have to program it yourself.

CHAPTER

FOUR

REGULAR EXPRESSION

Regular expressions are the patterns that Alex uses to match tokens in the input stream.

4.1 Syntax of regular expressions

```
regexp  := rexp2 { '|' rexp2 }

rexp2   := rexp1 { rexp1 }

rexp1   := rexp0 [ '*' | '+' | '?' | repeat ]

rexp0   := set
         | @rmac
         | @string
         | '(' [ regexp ] ')'

repeat  := '{' $digit+ '}'
         | '{' $digit+ ',' '}'
         | '{' $digit+ ',' $digit+ '}'
```

The syntax of regular expressions is fairly standard, the only difference from normal lex-style regular expressions being that we allow the sequence `()` to denote the regular expression that matches the empty string.

Spaces are ignored in a regular expression, so feel free to space out your regular expression as much as you like, even split it over multiple lines and include comments. Literal whitespace can be included by surrounding it with quotes " ", or by escaping each whitespace character with \.

`set` Matches any of the characters in <set>. See *Syntax of character sets* for the syntax of sets.

`@foo` Expands to the definition of the appropriate regular expression macro.

`"..."` Matches the sequence of characters in the string, in that order.

`r*` Matches zero or more occurrences of <r>.

`r+` Matches one or more occurrences of <r>.

`r?` Matches zero or one occurrences of <r>.

`r{n}` Matches <n> occurrences of <r>.

`r{n,}` Matches <n> or more occurrences of <r>.

`r{n,m}` Matches between <n> and <m> (inclusive) occurrences of <r>.

4.2 Syntax of character sets

Character sets are the fundamental elements in a regular expression. A character set is a pattern that matches a single character. The syntax of character sets is as follows:

```
set     := set '#' set0
         | set0

set0    := @char [ '-' @char ]
         | '.'
         | @smac
         | '[' [^] { set } ']'
         | '~' set0
```

The various character set constructions are:

char The simplest character set is a single Unicode character. Note that special characters such as `[` and `.` must be escaped by prefixing them with \ (see the lexical syntax, *Lexical syntax*, for the list of special characters).

Certain non-printable characters have special escape sequences. These are: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`. Other characters can be represented by using their numerical character values (although this may be non-portable): `\x0A` is equivalent to `\n`, for example.

Whitespace characters are ignored; to represent a literal space, escape it with \.

char-char A range of characters can be expressed by separating the characters with a '-', all the characters with codes in the given range are included in the set. Character ranges can also be non-portable.

. The built-in set '`.`' matches all characters except newline (`\n`).

Equivalent to the set `[\x00-\x10ffff] # \n`.

set0 # set1 Matches all the characters in <set0> that are not in <set1>.

[sets] The union of <sets>.

[^sets] The complement of the union of the <sets>. Equivalent to '`. # [sets]`'.

~set The complement of <set>. Equivalent to '`. # set`'

A set macro is written as `$` followed by an identifier. There are some builtin character set macros:

$white Matches all whitespace characters, including newline.

Equivalent to the set `[\ \t\n\f\v\r]`.

$printable Matches all "printable characters". Currently this corresponds to Unicode code points 32 to 0x10ffff, although strictly speaking there are many non-printable code points in this region. In the future Alex may use a more precise definition of `$printable`.

Character set macros can be defined at the top of the file at the same time as regular expression macros (see *Regular Expression*). Here are some example character set macros:

```
$lls      = a-z                  -- little letters
$not_lls  = ~a-z                 -- anything but little letters
$ls_ds    = [a-zA-Z0-9]          -- letters and digits
$sym      = [ \! \@ \# \$ ]      -- the symbols !, @, #, and $
$sym_q_nl = [ \' \! \@ \# \$ \n ] -- the above symbols with ' and newline
$quotable = $printable # \'      -- any graphic character except '
$del      = \127                 -- ASCII DEL
```

CHAPTER

FIVE

THE INTERFACE TO AN ALEX-GENERATED LEXER

This section answers the question: "How do I include an Alex lexer in my program?"

Alex provides for a great deal of flexibility in how the lexer is exposed to the rest of the program. For instance, there's no need to parse a `String` directly if you have some special character-buffer operations that avoid the overheads of ordinary Haskell `String`s. You might want Alex to keep track of the line and column number in the input text, or you might wish to do it yourself (perhaps you use a different tab width from the standard 8-columns, for example).

The general story is this: Alex provides a basic interface to the generated lexer (described in the next section), which you can use to parse tokens given an abstract input type with operations over it. You also have the option of including a wrapper, which provides a higher-level abstraction over the basic interface; Alex comes with several wrappers.

5.1 Unicode and UTF-8

Lexer specifications are written in terms of Unicode characters, but Alex works internally on a UTF-8 encoded byte sequence.

Depending on how you use Alex, the fact that Alex uses UTF-8 encoding internally may or may not affect you. If you use one of the wrappers (below) that takes input from a Haskell `String`, then the UTF-8 encoding is handled automatically. However, if you take input from a `ByteString`, then it is your responsibility to ensure that the input is properly UTF-8 encoded.

None of this applies if you used the `--latin1` option to Alex or specify a Latin-1 encoding via a `%encoding` declaration. In that case, the input is just a sequence of 8-bit bytes, interpreted as characters in the Latin-1 character set.

The following (case-insenstive) encoding strings are currently supported:

`%encoding "latin-1"`; `%encoding "iso-8859-1"` Declare Latin-1 encoding as described above.

`%encoding "utf-8"`; `%encoding "utf8"` Declare UTF-8 encoding. This is the default encoding but it may be useful to explicitly declare this to make protect against Alex being called with the `--latin1` flag.

5.2 Basic interface

If you compile your Alex file without a `%wrapper` declaration, then you get access to the lowest-level API to the lexer. You must provide definitions for the following, either in the same module or imported from another module:

```
type AlexInput
alexGetByte       :: AlexInput -> Maybe (Word8,AlexInput)
alexInputPrevChar :: AlexInput -> Char
```

The generated lexer is independent of the input type, which is why you have to provide a definition for the input type yourself. Note that the input type needs to keep track of the *previous* character in the input stream; this is used for implementing patterns with a left-context (those that begin with `^` or `set^`). If you don't ever use patterns with a left-context in your lexical specification, then you can safely forget about the previous character in the input stream, and have `alexInputPrevChar` return `undefined`.

Alex will provide the following function:

```
alexScan :: AlexInput             -- The current input
         -> Int                   -- The "start code"
         -> AlexReturn action     -- The return value

data AlexReturn action
  = AlexEOF

  | AlexError
      !AlexInput     -- Remaining input

  | AlexSkip
      !AlexInput     -- Remaining input
      !Int           -- Token length

  | AlexToken
      !AlexInput     -- Remaining input
      !Int           -- Token length
      action         -- action value
```

Calling `alexScan` will scan a single token from the input stream, and return a value of type `AlexReturn`. The value returned is either:

`AlexEOF` The end-of-file was reached.

`AlexError` A valid token could not be recognised.

`AlexSkip` The matched token did not have an action associated with it.

`AlexToken` A token was matched, and the action associated with it is returned.

The `action` is simply the value of the expression inside `{...}` on the right-hand-side of the appropriate rule in the Alex file. Alex doesn't specify what type these expressions should have, it simply requires that they all have the same type, or else you'll get a type error when you try to compile the generated lexer.

Once you have the `action`, it is up to you what to do with it. The type of `action` could be a function which takes the `String` representation of the token and returns a value in some token type, or it could be a continuation that takes the new input and calls `alexScan` again, building a list of tokens as it goes.

This is pretty low-level stuff; you have complete flexibility about how you use the lexer, but there might be a fair amount of support code to write before you can actually use it. For this reason, we also provide a selection of wrappers that add some common functionality to this basic scheme. Wrappers are described in the next section.

There is another entry point, which is useful if your grammar contains any predicates (see *Contexts*):

```
alexScanUser
         :: user             -- predicate state
         -> AlexInput        -- The current input
         -> Int              -- The "start code"
         -> AlexReturn action
```

The extra argument, of some type `user`, is passed to each predicate.

5.3 Wrappers

To use one of the provided wrappers, include the following declaration in your file:

```
%wrapper "name"
```

where <name> is the name of the wrapper, eg. `basic`. The following sections describe each of the wrappers that come with Alex.

5.3.1 The "basic" wrapper

The basic wrapper is a good way to obtain a function of type `String -> [token]` from a lexer specification, with little fuss.

It provides definitions for `AlexInput`, `alexGetByte` and `alexInputPrevChar` that are suitable for lexing a `String` input. It also provides a function `alexScanTokens` which takes a `String` input and returns a list of the tokens it contains.

The `basic` wrapper provides no support for using startcodes; the initial startcode is always set to zero.

Here is the actual code included in the lexer when the basic wrapper is selected:

```
type AlexInput =
  ( Char      -- previous char
  , [Byte]    -- rest of the bytes for the current char
  , String    -- rest of the input string
  )

alexGetByte :: AlexInput -> Maybe (Byte, AlexInput)
alexGetByte (c, b:bs, s  ) = Just (b, (c, bs, s))
alexGetByte (c, []  , [] ) = Nothing
alexGetByte (_, []  , c:s) = case utf8Encode c of
                               b:bs -> Just (b, (c, bs, s))

alexInputPrevChar :: AlexInput -> Char
alexInputPrevChar (c, _, _) = c

-- alexScanTokens :: String -> [token]
alexScanTokens str = go ('\n', [], str)
  where
    go inp@(_,_bs,str) =
      case alexScan inp 0 of
        AlexEOF                -> []
        AlexSkip  inp' len     -> go inp'
        AlexToken inp' len act -> act (take len str) : go inp'
        AlexError _            -> error "lexical error"
```

The type signature for `alexScanTokens` is commented out, because the `token` type is unknown. All of the actions in your lexical specification should have type:

```
{ ... } :: String -> token
```

for some type `token`.

For an example of the use of the basic wrapper, see the file `examples/Tokens.x` in the Alex distribution.

5.3.2 The "posn" wrapper

The posn wrapper provides slightly more functionality than the basic wrapper: it keeps track of line and column numbers of tokens in the input text.

The posn wrapper provides the following, in addition to the straightforward definitions of `alexGetByte` and `alexInputPrevChar`:

```
data AlexPosn = AlexPn
  !Int            -- absolute character offset
  !Int            -- line number
  !Int            -- column number

type AlexInput =
  ( AlexPosn      -- current position,
  , Char          -- previous char
  , [Byte]        -- rest of the bytes for the current char
  , String        -- current input string
  )

-- alexScanTokens :: String -> [token]
alexScanTokens str = go (alexStartPos, '\n', [], str)
  where
    go inp@(pos, _, _, str) =
      case alexScan inp 0 of
        AlexEOF                -> []
        AlexSkip  inp' len     -> go inp'
        AlexToken inp' len act -> act pos (take len str) : go inp'
        AlexError (AlexPn _ line column, _, _, _) -> error $ unwords
          [ "lexical error at", show line, "line,", show column, "column" ]
```

The types of the token actions should be:

```
{ ... } :: AlexPosn -> String -> token
```

For an example using the `posn` wrapper, see the file `examples/Tokens_posn.x` in the Alex distribution.

5.3.3 The "monad" wrapper

The `monad` wrapper is the most flexible of the wrappers provided with Alex. It includes a state monad which keeps track of the current input and text position, and the startcode. It is intended to be a template for building your own monads - feel free to copy the code and modify it to build a monad with the facilities you need.

```
data AlexState = AlexState
  { alex_pos   :: !AlexPosn  -- position at current input location
  , alex_inp   :: String     -- the current input
  , alex_chr   :: !Char      -- the character before the input
  , alex_bytes :: [Byte]     -- rest of the bytes for the current char
  , alex_scd   :: !Int       -- the current startcode
  }

newtype Alex a = Alex { unAlex :: AlexState
                               -> Either String (AlexState, a) }
```

(continues on next page)

(continued from previous page)

```
instance Functor     Alex where ...
instance Applicative Alex where ...
instance Monad       Alex where ...

runAlex          :: String -> Alex a -> Either String a

type AlexInput =
  ( AlexPosn                -- current position,
  , Char                    -- previous char
  , [Byte]                  -- rest of the bytes for the current char
  , String                  -- current input string
  )

alexGetInput     :: Alex AlexInput
alexSetInput     :: AlexInput -> Alex ()

alexError        :: String -> Alex a

alexGetStartCode :: Alex Int
alexSetStartCode :: Int -> Alex ()
```

The `monad` wrapper expects that you define a variable `alexEOF` with the following signature:

```
alexEOF :: Alex result
```

To invoke a scanner under the `monad` wrapper, use `alexMonadScan`:

```
alexMonadScan :: Alex result
```

The token actions should have the following type:

```
type AlexAction result = AlexInput -> Int -> Alex result
{ ... }  :: AlexAction result
```

The Alex file must also define a function `alexEOF`, which will be executed on when the end-of-file is scanned:

```
alexEOF :: Alex result
```

The `monad` wrapper also provides some useful combinators for constructing token actions:

```
-- skip :: AlexAction result
skip input len = alexMonadScan

-- andBegin :: AlexAction result -> Int -> AlexAction result
(act `andBegin` code) input len = do alexSetStartCode code; act input len

-- begin :: Int -> AlexAction result
begin code = skip `andBegin` code

-- token :: (AlexInput -> Int -> token) -> AlexAction token
token t input len = return (t input len)
```

5.3.4 The "monadUserState" wrapper

The `monadUserState` wrapper is built upon the `monad` wrapper. It includes a reference to a type which must be defined in the user's program, `AlexUserState`, and a call to an initialization function which must also be defined in the user's program, `alexInitUserState`. It gives great flexibility because it is now possible to add any needed information and carry it during the whole lexing phase.

The generated code is the same as in the `monad` wrapper, except in 3 places:

1. The definition of the general state, which now refers to a type `AlexUserState` that must be defined in the Alex file.

```
data AlexState = AlexState
  { alex_pos   :: !AlexPosn      -- position at current input location
  , alex_inp   :: String         -- the current input
  , alex_chr   :: !Char          -- the character before the input
  , alex_bytes :: [Byte]         -- rest of the bytes for the current char
  , alex_scd   :: !Int           -- the current startcode
  , alex_ust   :: AlexUserState  -- AlexUserState will be defined in the user␣
↪program
  }
```

2. The initialization code, where a user-specified routine (`alexInitUserState`) will be called.

```
runAlex :: String -> Alex a -> Either String a
runAlex input (Alex f) = case f st of
    Left msg     -> Left msg
    Right (_, a) -> Right a
  where
    st = AlexState
         { alex_pos   = alexStartPos
         , alex_inp   = input
         , alex_chr   = '\n'
         , alex_bytes = []
         , alex_ust   = alexInitUserState
         , alex_scd   = 0
         }
```

3. Two helper functions (`alexGetUserState` and `alexSetUserState`) are defined.

```
alexGetUserState :: Alex AlexUserState
alexSetUserState :: AlexUserState -> Alex ()
```

Here is an example of code in the user's Alex file defining the type and function:

```
data AlexUserState = AlexUserState
  { lexerCommentDepth  :: Int
  , lexerStringValue   :: String
  }

alexInitUserState :: AlexUserState
alexInitUserState = AlexUserState
  { lexerCommentDepth  = 0
  , lexerStringValue   = ""
  }
```

(continues on next page)

(continued from previous page)

```
getLexerCommentDepth :: Alex Int
getLexerCommentDepth = lexerCommentDepth <$> alexGetUserState

setLexerCommentDepth :: Int -> Alex ()
setLexerCommentDepth ss = do
  ust <- alexGetUserState
  alexSetUserState ust{ lexerCommentDepth = ss }

getLexerStringValue :: Alex String
getLexerStringValue = lexerStringValue <$> alexGetUserState

setLexerStringValue :: String -> Alex ()
setLexerStringValue ss = do
  ust <- alexGetUserState
  alexSetUserState ust{ lexerStringValue = ss }

addCharToLexerStringValue :: Char -> Alex ()
addCharToLexerStringValue c = do
  ust <- alexGetUserState
  alexSetUserState ust{ lexerStringValue = c : lexerStringValue ust }
```

5.3.5 The "gscan" wrapper

The `gscan` wrapper is provided mainly for historical reasons: it exposes an interface which is very similar to that provided by Alex version 1.x. The interface is intended to be very general, allowing actions to modify the startcode, and pass around an arbitrary state value.

```
alexGScan :: StopAction state result -> state -> String -> result

type StopAction state result =
  AlexPosn -> Char -> String -> (Int, state) -> result
```

The token actions should all have this type:

```
{ ... }   :: AlexPosn                  -- token position
          -> Char                      -- previous character
          -> String                    -- input string at token
          -> Int                       -- length of token
          -> ((Int, state) -> result)  -- continuation
          -> (Int, state)              -- current (startcode, state)
          -> result
```

5.3.6 The bytestring wrappers

The `basic-bytestring`, `posn-bytestring` and `monad-bytestring` wrappers are variations on the `basic`, `posn` and `monad` wrappers that use lazy `ByteString`s as the input and token types instead of an ordinary `String`.

The point of using these wrappers is that `ByteString`s provide a more memory efficient representation of an input stream. They can also be somewhat faster to process. Note that using these wrappers adds a dependency on the `ByteString` modules, which live in the `bytestring` package (or in the `base` package in `ghc-6.6`)

As mentioned earlier (*Unicode and UTF-8*), Alex lexers internally process a UTF-8 encoded string of bytes. This means that the `ByteString` supplied as input when using one of the ByteString wrappers should be UTF-8 encoded (or use either the `--latin1` option or the `%encoding` declaration).

Do note that `token` provides a *lazy* `ByteString` which is not the most compact representation for short strings. You may want to convert to a strict `ByteString` or perhaps something more compact still. Note also that by default tokens share space with the input `ByteString` which has the advantage that it does not need to make a copy but it also prevents the input from being garbage collected. It may make sense in some applications to use `ByteString`'s `copy` function to unshare tokens that will be kept for a long time, to allow the original input to be collected.

The "basic-bytestring" wrapper

The `basic-bytestring` wrapper is the same as the `basic` wrapper but with lazy `ByteString` instead of `String`:

```
import           Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

data AlexInput = AlexInput
  { alexChar    :: {-# UNPACK #-} !Char     -- previous char
  , alexStr     :: !ByteString              -- current input string
  , alexBytePos :: {-# UNPACK #-} !Int64    -- bytes consumed so far
  }

alexGetByte       :: AlexInput -> Maybe (Char, AlexInput)

alexInputPrevChar :: AlexInput -> Char

-- alexScanTokens :: ByteString -> [token]
```

All of the actions in your lexical specification should have type:

```
{ ... } :: ByteString -> token
```

for some type `token`.

The "posn-bytestring" wrapper

The `posn-bytestring` wrapper is the same as the `posn` wrapper but with lazy `ByteString` instead of `String`:

```
import           Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

type AlexInput =
  ( AlexPosn    -- current position
  , Char        -- previous char
```

(continues on next page)

(continued from previous page)

```
  , ByteString  -- current input string
  , Int64       -- bytes consumed so far
  )

-- alexScanTokens :: ByteString -> [token]
```

All of the actions in your lexical specification should have type:

```
{ ... } :: AlexPosn -> ByteString -> token
```

for some type `token`.

The "monad-bytestring" wrapper

The `monad-bytestring` wrapper is the same as the `monad` wrapper but with lazy `ByteString` instead of `String`:

```
import           Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

data AlexState = AlexState
  { alex_pos  :: !AlexPosn    -- position at current input location
  , alex_bpos :: !Int64       -- bytes consumed so far
  , alex_inp  :: ByteString   -- the current input
  , alex_chr  :: !Char        -- the character before the input
  , alex_scd  :: !Int         -- the current startcode
  }

newtype Alex a = Alex { unAlex :: AlexState
                               -> Either String (AlexState, a) }

runAlex :: ByteString -> Alex a -> Either String a

type AlexInput =
  ( AlexPosn                  -- current position
  , Char                      -- previous char
  , ByteString                -- current input string
  , Int64                     -- bytes consumed so far
  )

-- token :: (AlexInput -> Int -> token) -> AlexAction token
```

All of the actions in your lexical specification have the same type as in the `monad` wrapper. It is only the types of the function to run the monad and the type of the `token` function that change.

The "monadUserState-bytestring" wrapper

The `monadUserState-bytestring` wrapper is the same as the `monadUserState` wrapper but with lazy `ByteString` instead of `String`:

```
import           Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

data AlexState = AlexState
  { alex_pos  :: !AlexPosn      -- position at current input location
  , alex_bpos :: !Int64         -- bytes consumed so far
  , alex_inp  :: ByteString     -- the current input
  , alex_chr  :: !Char          -- the character before the input
  , alex_scd  :: !Int           -- the current startcode
  , alex_ust  :: AlexUserState  -- AlexUserState will be defined in the user program
  }

newtype Alex a = Alex { unAlex :: AlexState
                               -> Either String (AlexState, a) }

runAlex :: ByteString -> Alex a -> Either String a

-- token :: (AlexInput -> Int -> token) -> AlexAction token
```

All of the actions in your lexical specification have the same type as in the `monadUserState` wrapper. It is only the types of the function to run the monad and the type of the `token` function that change.

5.4 Type Signatures and Typeclasses

The `%token`, `%typeclass`, and `%action` directives can be used to cause Alex to emit additional type signatures in generated code. This allows the use of typeclasses in generated lexers.

5.4.1 Generating Type Signatures with Wrappers

The `%token` directive can be used to specify the token type when any kind of `%wrapper` directive has been given. Whenever `%token` is used, the `%typeclass` directive can also be used to specify one or more typeclass constraints. The following shows a simple lexer that makes use of this to interpret the meaning of tokens using the `Read` typeclass:

```
%wrapper "basic"
%token "Token s"
%typeclass "Read s"

tokens :-

[a-zA-Z0-9]+ { mkToken }
[ \t\r\n]+   ;

{

data Token s = Tok s
```

(continues on next page)

(continued from previous page)

```
mkToken :: Read s => String -> Token s
mkToken = Tok . read

lex :: Read s => String -> [Token s]
lex = alexScanTokens

}
```

Multiple typeclasses can be given by separating them with commas, for example:

```
%typeclass "Read s, Eq s"
```

5.4.2 Generating Type Signatures without Wrappers

Type signatures can also be generated for lexers that do not use any wrapper. Instead of the `%token` directive, the `%action` directive is used to specify the type of a lexer action. The `%typeclass` directive can be used to specify the typeclass in the same way as with a wrapper. The following example shows the use of typeclasses with a "homegrown" monadic lexer:

```
{
{-# LANGUAGE FlexibleContexts #-}

module Lexer where

import Control.Monad.State
import qualified Data.Bits
import Data.Word

}

%action "AlexInput -> Int -> m (Token s)"
%typeclass "Read s, MonadState AlexState m"

tokens :-

[a-zA-Z0-9]+ { mkToken }
[ \t\n\r]+   ;

{

alexEOF :: MonadState AlexState m => m (Token s)
alexEOF = return EOF

mkToken :: (Read s, MonadState AlexState m) =>
           AlexInput -> Int -> m (Token s)
mkToken (_, _, _, s) len = return (Tok (read (take len s)))

data Token s = Tok s | EOF

lex :: (MonadState AlexState m, Read s) => String -> m (Token s)
lex input = alexMonadScan
```

(continues on next page)

(continued from previous page)

```
-- "Boilerplate" code from monad wrapper has been omitted

}
```

The `%token` directive may only be used with wrapper, and the `%action` can only be used when no wrapper is used.

The `%typeclass` directive cannot be given without the `%token` or `%action` directive.

CHAPTER

SIX

INVOKING ALEX

The command line syntax for Alex is entirely standard:

```
$ alex { option } file.x  { option }
```

Alex expects a single `file.x` to be named on the command line. By default, Alex will create `file.hs` containing the Haskell source for the lexer.

The options that Alex accepts are listed below:

`-o <file>; --outfile=<file>` Specifies the filename in which the output is to be placed. By default, this is the name of the input file with the `.x` suffix replaced by `.hs`.

`-i [<file>]; --info [<=file>]` Produces a human-readable rendition of the state machine (DFA) that Alex derives from the lexer, in <file> (default: `file.info` where the input file is `file.x`).

The format of the info file is currently a bit basic, and not particularly informative.

`-t [<dir>]; --template=<dir>` Look in <dir> for template files.

`-g; --ghc` Causes Alex to produce a lexer which is optimised for compiling with GHC. The lexer will be significantly more efficient, both in terms of the size of the compiled lexer and its runtime.

`-d; --debug` Causes Alex to produce a lexer which will output debugging messages as it runs.

`-l; --latin1` Disables the use of UTF-8 encoding in the generated lexer. This has two consequences:

- The Alex source file is still assumed to be UTF-8 encoded, but any Unicode characters outside the range 0-255 are mapped to Latin-1 characters by taking the code point modulo 256.
- The built-in macros `$printable` and '`.`' range over the Latin-1 character set, not the Unicode character set.

Note that this currently does not disable the UTF-8 encoding that happens in the "basic" wrappers, so `--latin1` does not make sense in conjunction with these wrappers (not that you would want to do that, anyway). Alternatively, a `%encoding "latin1"` declaration can be used inside the Alex source file to request a Latin-1 mapping. See also *Unicode and UTF-8* for more information about the `%encoding` declaration.

`-?; --help` Display help and exit.

`-V; --version` Output version information and exit. Note that for legacy reasons `-v` is supported, too, but the use of it is deprecated. `-v` will be used for verbose mode when it is actually implemented.