Alex User Guide

Chris Dornan

Isaac Jones

Simon Marlow

Alex User Guide

by Chris Dornan

by Isaac Jones

by Simon Marlow

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++.

Table of Contents

List of Figures

Chapter 1. About Alex

Alex can always be obtained from its home page (http://www.haskell.org/alex). The latest source code lives in the `fptools` CVS repository; instructions on accessing that repository are here (http://www.haskell.org/ghc/docs/latest/html/building/sec-cvs.html).

1.1. Release Notes for version 2.0

Alex has changed a *lot* between versions 1.x and 2.0. The following is supposed to be an exhaustive list of the changes:

1.1.1. Syntax changes

- Code blocks are now surrounded by `{...}` rather than `%{...%}`.
- Character-set macros ow begin with ‘`$`’ instead of ‘`^`’ and have multi-character names.
- Regular expression macros now begin with ‘`@`’ instead of ‘`%`’ and have multi-character names.
- Macro definitions are no longer surrounded by `{ ... }`.
- Rules are now of the form

  ```
  <c1,c2,...>  regex   { code }
  ```

  where `c1`, `c2` are startcodes, and `code` is an arbitrary Haskell expression.
- Regular expression syntax changes:
  - `()` is the empty regular expression (used to be ‘`$`’)
  - set complement can now be expressed as `[^sets]` (for similarity with lex regular expressions).
  - The `'abc'` form is no longer available, use `[abc]` instead.
  - ‘`^`’ and ‘`$`’ have the usual meanings: ‘`^`’ matches just after a ‘`\n`’, and ‘`$`’ matches just before a ‘`\n`’.
  - ‘`\n`’ is now the escape character, not ‘`^`’.
  - The form `"..."` means the same as the sequence of characters inside the quotes, the difference being that special characters do not need to be escaped inside `"..."`.
- Rules can have arbitrary predicates attached to them. This subsumes the previous left-context and right-context facilities (although these are still allowed as syntactic sugar).

1.1.2. Changes in the form of an Alex file

- Each file can now only define a single grammar. This change was made to simplify code generation. Multiple grammars can be simulated using startcodes, or split into separate modules.
- The programmer experience has been simplified, and at the same time made more flexible. See the Chapter 5 for details.
- You no longer need to import the `Alex` module.

1.1.3. Usage changes

The command-line syntax is quite different. See Chapter 6.

1.1.4. Implementation changes

- A more efficient table representation, coupled with standard table-compression techniques, are used to keep the size of the generated code down.
- When compiling a grammar with GHC, the -g switch causes an even faster and smaller grammer to be generated.
- Startcodes are implemented in a different way: each state corresponds to a different initial state in the DFA, so the scanner doesn't have to check the startcode when it gets to an accept state. This results in a larger, but quicker, scanner.

1.2. Reporting bugs in Alex

Please report bugs in Alex to `<simonmar@microsoft.com>`. There are no specific mailing lists for the discussion of Alex-related matters, but such topics should be fine on the Haskell (http://www.haskell.org/mailman/listinfo/haskell) and Haskell Cafe (http://www.haskell.org/mailman/listinfo/haskell-cafe) mailing lists.

1.3. License

Chapter 2. 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 exisiting lex users, although it does depart from lex in a number of ways.

Figure 2-1. A simple Alex specification.

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

A sample specification is given in Figure 2-1. 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 Chapter 5.

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 meaming of a 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 appriate `String`, and a list of `Token`s returned as the result. If the input stream is lazy, the output stream will also be produced lazilly[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 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 (http://www.haskell.org/happy/) parser.

Notes

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

Chapter 3. 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. The overall layout of an Alex file is:

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

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 wrapper specification:

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

wrappers are described in Section 5.2.

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 Section 4.2), and a regular expression macro can be used wherever a regular expression is valid (see Chapter 4).

```
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 identifer, 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 lexcial 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 Chapter 5.

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.

3.2.2.1. 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 Chapter 5 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 Section 5.1).

3.2.2.2. 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 identifer, 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 Chapter 5). 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 - Chapter 5 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 4. 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 Section 4.2 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 occurences of *`r`*.

`r+`

Matches one or more occurences of *`r`*.

`r?`

Matches zero or one occurences 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     := @char [ '-' @char ]
        | '.'
        |  @smac
        | '[' [^] { set } ']'
        | '~' set0
```

The various character set constructions are:

char

The simplest character set is a single character. Note that special characters such as `[` and `.` must be escaped by prefixing them with `\` (see the lexical syntax, Section 3.1, 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`).

`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.

`$printable`

Matches all printable characters (characters 32 to 126 in ASCII).

Character set macros can be defined at the top of the file at the same time as regular expression macros (see Chapter 4). 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 5. 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. 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
alexGetChar       :: AlexInput -> Maybe (Char,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"
         -> Maybe (          -- Nothing on error or EOF
                 AlexInput,  -- The remaining input
                 Int,        -- Length of this token
                 action      -- The action (an unknown type)
         )
```

Calling `alexScan` will scan a single token from the input stream, and return the remaining input stream, the length of the token, and the `action`.

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 Section 3.2.2.1):

```
alexScanUser
         :: user             -- predicate state
         -> AlexInput  -- The current input
         -> Int   -- The "start code"
         -> Maybe (          -- Nothing on error or EOF
                 AlexInput,  -- The remaining input
                 Int,        -- Length of this token
                 action      -- The action (an unknown type)
       )
```

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

5.2. 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.

5.2.1. 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`, `alexGetChar` 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
    String) -- current input string

alexGetChar :: AlexInput -> Maybe (Char,AlexInput)
alexGetChar (_, [])   = Nothing
alexGetChar (_, c:cs) = Just (c, (c,cs))

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

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

The type signature for `alexScanTokens` is commented out, because the `token` type is unkonwn. 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_basic.x` in the Alex distribution.

5.2.2. 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 `alexGetChar` and `alexInputPrevChar`:

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

```
type AlexInput = (AlexPosn, -- current position,
    Char, -- previous 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 -> []
      AlexError _ -> error "lexical error"
      AlexSkip  inp' len     -> go inp'
      AlexToken inp' len act -> act pos (take len str) : go inp'
```

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.

5.2.3. 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_scd :: !Int  -- the current startcode
    }

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

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

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

alexError        :: String -> Alex a

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

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

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

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

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

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

5.2.4. 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
```

Chapter 6. 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 then Haskell source for the lexer.

The options that Alex accepts are listed below:

`-d`
`--debug`

Causes Alex to produce a lexer which will output debugging messsages as it runs.

`-g`
`--ghc`

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

`-o file`
`--output-file=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.

`-v`
`--version`

Be more verbose. This currently doesn't do anything, but it might in the future.