Alex: A Lex for Haskell Programmers

Chris Dornan

29th September 1997

1 Introduction

The Alex package, like Lex, takes a description of tokens based on regular expressions and generates a program module for scanning text efficiently. The difference is that Alex generates Haskell modules rather than C/Ratfor source files. Although Alex takes after Lex, it is intended for Haskell programmers and so departs quite radically from Lex in some respects.

A sample specification is given in Figure 1. The first few lines between the `%{` and `%}` provide a code scrap (some inlined Haskell code) to be placed in the output. All such code fragments will be placed in order at the head of the module with the Alex-generated tables appearing at the end.

The next two lines define the `^d` and `^l` macros for use in the token definitions.

The `"tokens_lx"/"tokens_acts":-` line starts the definition of a scanner. It is generated in two parts: the main tables being placed in `token_lx` and the actions for each token being bound to `tokens_acts`.

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

> <*token-id*> `::=` *rexp*

If *token-id* is omitted then the token will be discarded (this is done for the first two white-space and comment token definitions), otherwise a corresponding action function for constructing the token must be given somewhere in the script. In this case the action functions are given in the code scraps to the right of the token definition but they could be specified anywhere. Here Alex differs from Lex; while each action must be named in a code scrap, the programmer has more flexibility in laying out the script. Like comments, code scraps may be placed anywhere in the module.

```
%{
module Tokens where

import Alex
%}


{ ^d = 0-9      }                       -- digits
{ ^l = [a-zA-Z] }                       -- alphabetic characters

"tokens_lx"/"tokens_acts":-

  <>     ::=  ^w+                       -- white space
  <>     ::=  ^-^-.*                    -- comments
  <let'> ::=  let                       %{ let' p s = Let p          %}
  <in'>  ::=  in                        %{ in'  p s = In  p          %}
  <int>  ::=  ^d+                       %{ int  p s = Int p (read s) %}
  <sym>  ::=  '=+-*/()'                 %{ sym  p s = Sym p (head s) %}
  <var>  ::=  ^l[^l^d^_^']*             %{ var  p s = Var p s        %}


%{
data Token =
        Let Posn                |
        In  Posn                |
        Sym Posn Char           |
        Var Posn String         |
        Int Posn Int            |
        Err Posn
        deriving (Eq,Text)

tokens:: String -> [Token]
tokens inp = scan tokens_scan inp

tokens_scan:: Scan Token
tokens_scan = load_scan (tokens_acts,stop_act) tokens_lx
        where
        stop_act p ""  = []
        stop_act p inp = [Err p]
%}
```

Figure 1: A simple Alex specification.

The action function for each token takes the text matched and its position and generates a token suitable for the parser, of type `Token` in this case.

The remaining lines define the `Token` data type and the scanner in two parts, `tokens_scan` which uses `load_scan` for amalgamating the token actions, stop action (for stopping the scanner) and token specification tables into a `Scan` structure, and `tokens`, the scanner, which simply passes the `Scan` structure to `scan`. `load_scan`, `Scan` and `scan` were imported from the `Scan` module that comes with the Alex distribution.

While delegating the task of assembling the scanner to the programmer may seem a bit bothersome, the effort is rewarded with a flexible and modular scheme for generating scanners.

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

```
alex Tokens.x
```

If the module needed to be placed in different file, `tkns.hs` for example, then a second file-name can be specified on the command line:

```
alex Tokens.x tkns.hs
```

The resulting module is Haskell 1.2 and Haskell 1.3 compatible. It can also be readily used with a Happy parser, with the catch that the `Err` token must be declared.

If the script were written with literate conventions then the `.lx` extension would be used instead of `.x`. (Literate scripts will be described in Section 2.5 on lexical syntax.)

2 Syntax

The syntax in this section is described with an extended BNF in which optional phrases are enclosed in square brackets ([...]) and repeated phrases in braces ({...}). The terminal symbols ide, tkn, ch, ech, cch smac, rmac and quot are are defined in Section 2.5 on lexical syntax.

2.1 Scripts, Macros and Scanners

Alex scripts contain a list of scanner specifications, optionally preceded by some global macro definitions. Each scanner consists of a header with the Haskell identifiers to be bound in the output module, a list of macro definitions and a

list of token definitions. Macros may also be specified on the right-hand-side of token definitions.

```
alex      →  { macdef } { scanner }

macdef    →  { smac = set | rmac = rexp }

scanner   →  ide [/ ide] :- { macdef } { def }
def       →  tkn ::= { macdef } ctx
ctx       →  { sc : } [set \] rexp [/ rexp]
sc        →  "0" | ide
```

Macros come in two flavours: regular expression macros and character set macros. A regular expression is more powerful than a character set but it can be used in less contexts. Regular expressions and character sets are described below.

Macros obey a static scoping discipline with each macro scoping over the construction it precedes. Thus macros at the head of the script scope over the whole script, macros preceding a scanner scope over the scanner and those on the right-hand-side of a token definition will only be effective for the token definition. Because they are statically scoped any macros mentioned in the body of a macro must be defined in its defining environment and its meaning is then fixed at the point of definition. For example, the definitions

```
{ ^g = ^a        }
{ ^a = [a-zA-Z] }
```

will bind `^g` to the contents of the `^a` macro (which must be defined) and rebind `^a` to match the alphabetic characters. After both definitions, `^g` will be bound to the original value of the `^a` macro.

The header line of a macro will usually give two identifiers: the first one for binding the tables containing the token specifications and the second for binding the action list. If the second identifier is omitted then the action list will not be generated and the programmer need not specify action functions for each named token; however, to get a useful scanner with `load_scan`, a hand-built action list will have to be supplied.

Leading context, trailing context and start codes may be specified with the `:`, `\`, `/` operators. These features will be described in Section 3.5 on context specifications.

A token identifier may be bound to more than one specification with the result

that all the definitions will be handled by the same action function.

2.2 Regular Expressions

```
rexp      →  rexp2 { | rexp2 }
rexp2     →  rexp1 { rexp1 }
rexp1     →  rexp0 [ * | + | ? | repeat ]
repeat    →  { digit [, [digit] ] }
rexp0     →  $ | rmac | set | ( rexp )

digit     →  0 | 1 | 2 | 3 | 4
          |  5 | 6 | 7 | 8 | 9
```

The regular expression syntax is similar to that of Lex, with the addition of `$` for ϵ, matching the empty string, and the %⟨*letter*⟩ syntax for regular-expression macros.

Here are some of the ways of repeating `a`s.

```
"example_rexps":-

  <a_star>  ::= $ | a+          -- = a*, zero or more as
  <a_plus>  ::= aa*             -- = a+, one or more as
  <a_quest> ::= $ | a           -- = a?, zero or one as
  <a_3>     ::= a{3}            -- = aaa
  <a_3_5>   ::= a{3,5}          -- = a{3}a?a?
  <a_3_>    ::= a{3,}           -- = a{3}a*
```

2.3 Sets of Characters

A set is a special form of regular expression that matches strings of length one. Here Alex differs markedly from Lex.

set	→	set_0 [# set_0]
set_0	→	~ set_0
	|	*chr* [- *chr*]
	|	smac
	|	[{*set*}]
	|	quot
chr	→	ch | ech | cch

The simplest set, *chr*, contains a single character. The letters and digits represent themselves while symbolic characters can be escaped with a ^. Any character can be generated with a ^ followed by its decimal code, though this is not portable.

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 union of a number of sets may be taken by enumerating them in square brackets ([...]), the complement of a set can be taken with ~ and the difference of two sets can be taken with the # operator. Finding a good use for [] is left as an exercise for the devious reader.

A quoted set of characters can be expressed by enclosing it in quotes (`...'). Note that the quoted set starts with a back-quote and finishes with a single-quote. A ' character may not be included in such sets.

A set macro is expressed by a . or by a ^ followed by a letter. The standard macros are listed in Figure 3. Most of them consist of bindings for the Haskell \⟨*letter*⟩ character escape codes. The ^w and ^p correspond to the prelude `isSpace` and `isPrint` prelude functions.

```
"example_sets":-

  <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> ::= ^p#^'            -- any graphic character except '
  <del>      ::= ^127             -- ASCII DEL
```

```
{ ^a = ^7             }   -- alarm
{ ^b = ^8             }   -- back space
{ ^t = ^9             }   -- form feed
{ ^n = ^10            }   -- newline
{ ^v = ^11            }   -- vertical tab
{ ^f = ^12            }   -- form feed
{ ^r = ^13            }   -- carriage return
{ ^w = [^t^n^v^f^r^ ] }   -- white space
{ ^p = ^32-^126       }   -- printable/graphic characters
{ .  = ^0-^255 # ^n   }   -- non-newline characters
```

Figure 2: The standard macros (for Unix sytems).

2.4 The Scanner's Alphabet

The characters accepted by a scanner are precisely those permitted by the regular expressions and the macro definitions. This is usually determined by the settings of the . macro. By default, . includes every character except newline but it could be redefined, for example, to exclude all the eight bit codes:

```
{ . = ^0-^127 # ^n }
```

This works because set complement, `~`⟨*set*⟩ is defined to mean `[.^n]#`⟨*set*⟩ so complemented sets would also exclude eight-bit characters.

Note that the above restriction of the . macro is unlikely to reduce the size of the tables used by Alex or speed up the scanners generated by it, in fact, the contrary, as the table formats were designed with the default configuration in mind.

2.5 Lexical Syntax

Alex supports the Haskell literate script convention as described in the Haskell report (version 1.2). See Figure 5 for an example literate script. If the name of the file containing the scripts ends in `.lx` then the lines that make up the script start with a `>`; the script is preprocessed by stripping out all the other lines and replacing the initial `>` at the start of each line with a space. This process is formalised in Section 3.3 where the scanner used to preprocess Alex scripts is given.

All white space appearing in the script is ignored, except where a space is quoted with `^` or `'...'`. Haskell-style line comments are introduced with `--` and code

```
{ ^s = ^w#^n      }                                    -- spaces + tabs, etc
{ ^d = 0-9        }                                    -- digits
{ ^a = a-z        }                                    -- lower-case alphas
{ ^A = A-Z        }                                    -- upper-case alphas
{ ^l = [^a^A]     }                                    -- alpha characters
{ ^i = [^l^d^_^'] }                                    -- identifier trailer

"alex_lx" :-

  <>     ::=  ^w+                                     -- white space
  <>     ::=  ^-^-.*                                  -- comments
  <code> ::=                                          -- code scraps:
        ^%^{ (.#^%|^%.#^})* ^%^}                      --   single-line scraps
        |  ^%^{^s*^n (((.#^%.*)?|^%(.#^}.*)?)^n)* ^%^}  --   multi-line scraps
        |  ^%^{^s*^n                                  --   multi-line scraps
              (((.#^ .*)?|^ (.#^%.*)?|^ ^%(.#^}.*)?)^n)*--      (literate
                     ^ ^%^}                           --       scripts)
  <zero> ::=  ^" 0 ^"                                 -- "0" start code
  <ide>  ::=  ^" ^a^i* ^"                             -- function identifier
  <tkn>  ::=  ^< (^a^i*)? ^>                          -- token identifier
  <bnd>  ::=  ^:^-                                    -- ":-"
  <prd>  ::=  ^:^:^=                                  -- "::="
  <spe>  ::=  '{=}:\/|*+?,$()#[]-'                    -- specials
  <ch>   ::=  [^l^d]                                  -- letter or digit
  <ech>  ::=  ^^ ^p#[^l^d]                            -- escaped symbols
  <cch>  ::=  ^^ ^d{1,3}                              -- character codes
  <smac> ::=  ^^ ^l | ^.                              -- set macros
  <rmac> ::=  ^% ^l                                   -- rexp macros
  <quot> ::=  ^' ^p#^'+ ^'                            -- quoted sets
```

Figure 3: The Alex scanner.

scraps are enclosed in %{ ...%} brackets. Otherwise,

ide	is a Haskell identifier in quotes ("...").
tkn	is an optional Haskell identifier in angle brackets (<...>).
ch	is a letter or digit.
ech	is a ^ followed by a symbolic character.
cch	is a ^ followed by a character code.
smac	is either a . or a ^ followed by a letter.
rmac	is a % followed by a letter.
quot	is a sequence of non-' characters in quotes ('...').

The lexical syntax is formalised by the Alex script in Figure 3.

As can be seen from the `<code>` token, code scraps come in two varieties: those that start and end with a newline and those that are contained on one line. If the code scrap starts with a newline then it must finish with the `%}` at the start of the line. (In fact, it may have a single space between the newline and the `%}`; this is to accomodate literate scripts where the `>` between the newline and the `%}` is converted to a space.) This means that the `%}` sequence may be used anywhere in the code scrap except at the start of the line and that the text from the first newline to the last newline can be copied without alteration into the output module.

The code in multi-line code scraps must follow the same layout conventions used for the tables generated by Alex, namely that all top-level defintions start in the left-hand column for ordinary scripts, column two for literate scripts.

Line code scraps may not contain the `%}` sequence anywhere in them. They will appear in the output in the left hand column for ordinary scripts, at column two for literate scripts.

3 General Scanners

3.1 The Alex Module

The `Alex` module contains the run-time interface. It is self-contained so only `Alex` and the Alex generated modules need to be added to programs using a Alex scanner.

The `Posn` data type is the first product of the `Alex` module. It provides a standard means of positioning tokens in the input stream.

```
data Posn = Pn Int Int Int  deriving (Eq,Text)

start_pos:: Posn
start_pos = Pn 0 1 1

eof_pos:: Posn
eof_pos = Pn (-1) (-1) (-1)
```

`Pn addr ln col` represents the location of a token found `addr` characters into the file on line `ln` and column `col`. In calculating the column position, it will be assumed that tab characters use eight-character tab stops. The first character of the file is located at `start_pos` and `eof_pos`, by convention, will represent the end of file.

The `Alex` module provides two packages for generating scanners from the tables

generated by Alex: the basic `Scan/load_scan/scan` package used for the `Token` module of Figure 1, and a more flexible `GScan/load_gscan/gscan` package.

The `scan` package generates simple scanners that convert input text to streams of tokens. The scanners are stateless as each token generated is a function of its textual content and location.

The token actions take the form of an association list associating each token name with an action function that constructs the token from the text matched and its location. The stop action is invoked when no more input can be tokenised; it takes the residual input and its position and generates the remaining stream of tokens, usually the empty list or an end-of-file token if the empty string is passed, an error token otherwise.

```
type Actions t = ([(String,TokenAction t)], StopAction t)

type TokenAction t = Posn -> String -> t

type StopAction t = Posn -> String -> [t]
```

`load_scan` combines the actions with the dump generated by Alex to produce a `Scan` structure that can be passed to `scan`. `scan` takes the scanner and the input text and generates a stream of tokens. It assumes that the text is at the start of the input with the position set to `start_pos` (see above) and sets the last character read to newline (the last character read is used to resolve leading context specifications); `scan'` can be used to override these defaults.

```
load_scan:: Actions t -> DFADump -> Scan t
scan:: Scan t -> String -> [t]
scan':: Scan t -> Posn -> Char -> String -> [t]
```

The `gscan` package generates general-purpose scanners for converting input text into a return type determined by the application. Access to the scanner's internal state, start codes and some application-specific state is provided.

The token actions take the form of an association list associating each token name with an action function that constructs the result from the length of the token, the scanner's state (including the remaining input from the start of the token) and a continuation function that scans the remaining input.

More specifically, each token action takes as arguments the position of the token, the last character read before the token (used to resolve leading context), the whole input from the start of the token, the length of the token, the continuation function and the visible state (as distinct from the scanner's internal state) including the current start code and the application specific state. The stop action is invoked when no more input can be scanned; it takes the same parameters as the token actions without the token length and the continuation

function.

```
type GScan s r = (DFA (GTokenAction s r), GStopAction s r)

type GActions s r = ([(String, GTokenAction s r)], GStopAction s r)

type GTokenAction s r =
        Posn -> Char -> String -> Int ->
                ((StartCode,s)->r) -> (StartCode,s) -> r

type GStopAction s r = Posn -> Char -> String -> (StartCode,s) -> r
```

`load_gscan` combines the actions with the dump generated by Alex to produce a `GScan` structure that can be passed to `gscan`. `gscan` takes the scanner, the application-specific state and the input text as parameters. It assumes that the text is at the start of the input with the position set to `start_pos` (see above) and sets the last character read to newline and the start code to 0; `gscan'` can be used to override these defaults.

```
load_gscan:: GActions s r -> DFADump -> GScan s r
gscan:: GScan s r -> s -> String -> r
gscan':: GScan s r -> Posn -> Char -> String -> (StartCode,s) -> r
```

Note that a token action can ignore its continuation function and call up `gscan'` with a different scanner to tokenise the rest of the input. This offers a more efficient alternative to start codes (see Section 3.5 on context specifications) for invoking alternate scanners on segments of the input.

3.2 Stateful Scanners

Some parsing constructions are best handled by making the scanner stateful, allowing the action functions to read and alter some state. A stateful scanner will instantiate the `s` parameter of `GScan` with the state type needed by the application and will make use of the `(StartCode,s)` argument of the action functions. (The `StartCode` component of the scanner's state will be dealt with in Section 3.5 on context specifications.)

Consider the problem of collecting the code scraps from a Alex script. The code scraps could have been included in the grammar, forcing them to appear at certain points in the script and complicating the grammar and parser. Instead, they are ignored by the parser and collected in the scanner's state, being passed back to the parser in an explicit end-of-file token.

A simple scanner illustrating this technique is given in Figure 4. It returns a stream of identifiers terminated with an end-of-file token containing all the

```
%{
import Alex
%}

"state_lx"/"state_acts":-

  <>     ::= ^w+
  <code> ::= ^%^{ (~^% | ^%~^})* ^%^}
  <ide>  ::= [A-Za-z]+


%{
code _ _ inp len cont (sc,frags) = cont (sc,frag:frags)
        where
        frag = take (len-4) (drop 2 inp)

ide _ _ inp len cont st = Ide (take len inp):cont st


data Token = Ide String | Eof String | Err


tokens:: String -> [Token]
tokens inp = gscan state_scan [] inp

state_scan:: GScan [String] [Token]
state_scan = load_gscan (state_acts,stop_act) state_lx
        where
        stop_act _ _ "" (_,frags) = [Eof (unlines(reverse frags))]
        stop_act _ _ _ _ = [Err]
%}
```

Figure 4: A stateful scanner.

accumulated code scraps on the input.

The idea for this technique, and another potential application of it, came from the Brisk scanner which constructs the symbol table, returning integer handles in the token stream and the symbol table in the end-of-file token.

3.3 Literate Scripts

The `scan` package only provides for actions that return a single token as part of a list of such tokens. Sometimes a more flexible format is required, such as a preprocessor that generates another stream of characters.

Alex itself uses such a preprocessor to deal with literate scripts. Recall that each line in a (Haskell) literate scripts is either a blank line, a code scrap line starting with a `>` or a comment line, and that comment lines and scrap lines must be separated by one or more blank lines. The script in Figure 5 defines three macros, `%b`, `%s` and `%c`, for recognising blank, scrap and comment lines (where each line *starts* with a newline character). Figure 5 is itself a literate script, of course.

If two newline characters are added to the front of the input then a valid literate script could be considered as a series of scraps and comments in which a scrap consists of a blank line followed by one or more scrap lines and a comment consists of a blank line followed by zero or more comment lines. Note that the initial lines in a series of blank lines will each be considered comments in this scheme.

The action for the `<scrap>` token replaces each of the `>` in column one with a space, appending the continuation onto the result. The `<comment>` action strips out everything except the newlines, appending its continuation. (The newlines from the comment scraps are retained in order to keep the line numbers synchronised with the original input.)

To construct the `literate` scanner, of type `String -> String`, we must remember to insert the dummy newlines onto the input, and to remove them again afterwards.

3.4 A Simple Preprocessor

A general scanner need not return a list at all. The scanner of Figure 6 is a schematic version of the C preprocessor that only supports `#include "foo"` preprocessor lines. The return type of the scanner, and therefore the continuation passed to the action functions, is `IO ()`, which it combines with the `>>` and `>>=` operators rather than the `:` and `++` operators for the list generating scanners.

3.5 Context Specifications

Alex retains the facilities of Lex for restricting token specifications to given contexts, albeit in a modified form. Leading context is specified with the \ operator, with its left operand being restricted to a *set* specification. The character to the left of the token must be matched by the specification but it is not included in the token.

```
>%{ import Alex %}

> "lit_lx"/"lit_acts":-

> { ^s = ^w#^n                  }
> { %b = ^n^s*                  }
> { %s = ^n^>.*                 }
> { %c = ^n(~[^>^w].*|^s+~^w.*) }

>   <scrap>   ::= %b%s+
>   <comment> ::= %b%c*


>%{
> scrap _ _ inp len cont st = strip len inp
>       where
>       strip 0 _ = cont st
>       strip (n+1) (c:rst) =
>               if c=='\n'
>                  then '\n':strip_nl n rst
>                  else c:strip n rst
>
>       strip_nl (n+1) ('>':rst) = ' ':strip n rst
>       strip_nl n rst = strip n rst

> comment _ _ inp len cont st = strip len inp
>       where
>       strip 0 _ = cont st
>       strip (n+1) (c:rst) = if c=='\n' then c:strip n rst else strip n rst


> literate:: String -> String
> literate inp = drop 2 (gscan lit_scan () ('\n':'\n':inp))

> lit_scan:: GScan () String
> lit_scan = load_gscan (lit_acts,stop_act) lit_lx
>       where
>       stop_act p _ "" st = []
>       stop_act p _ _  _  = error (msg ++ loc p ++ "\n")
>
>       msg  = "literate preprocessing error at "
>
>       loc (Pn _ l c) = "line " ++ show(l-2) ++ ", column " ++ show c
>%}
```

Figure 5: A preprocessor for literate scripts.

```
%{
import Alex
%}

"pp_lx"/"pp_acts":-

{ ^s = ^w#^n                }           -- spaces and tabs, etc.
{ ^f = [A-Za-z0-9‘~%-_.,/’] }           -- file-name character

  <inc> ::= ^#include^s+^"^f+^"^s*^n
  <txt> ::= .*^n

%{
inc p c inp len cont st = pp fn >> cont st
        where
        fn = (takeWhile (’"’/=) . tail . dropWhile isSpace . drop 8) inp

txt p c inp len cont st = putStr (take len inp) >> cont st

pp:: String -> IO ()
pp fn = readFile fn >>= \cts -> gscan pp_scan () cts

pp_scan:: GScan () (IO ())
pp_scan = load_gscan (pp_acts,stop_act) pp_lx
        where
        stop_act _ _ _ _ = return ()
%}
```

Figure 6: A scanner capable of I/O.

Trailing context is specified with a `/`, where the expression to the right can be a fully-fledged regular expression. Again, the input to the right of the token must match the regular expression but it is not included in the token.

Note that unlike Lex, the leading and trailing context do not contribute to the length of the token when choosing from a number of matching tokens. The only effect of the context specifications is to eliminate tokens that would otherwise match the input.

A token may be restricted to given 'start codes' by prefixing the token specification (to the right of the `::=`) with a `"foo":` where `foo` is the name of the start code; to restrict a definition to the `0` start code, prefix it with `"0":`; if several start codes are given then the scanner may be in any one of start codes for the token to be selected.

Each start code, `"foo"`, mentioned in the script will result in a definition like

```
foo = 1
```

being added to the output (where the integer is positive and distinct from those assigned to other start codes) so each start code must be a Haskell function identifier that is not otherwise bound in the output module.

By default, the scanner starts in start code `0`. To change to a different start code, an action function has only to call its continuation function with the `StartCode` component of its state argument set to a different value; this will immediately eliminate token definitions annotated with start codes that do not include the new start code.

Figure 7 gives a rather contrived scanner that illustrates several aspects of context specifications.

The first four tokens specifications match the same text, only differing in their context. `<only_ide>` will be selected if it is the only alphabetic string on the line, `<start_ide>` if it is at the start of the line, `<end_ide>` if it is at the end of a line, otherwise `<ide>`. Note that there is no question of the leading or trailing context elongating any of the tokens and interfering with their priority.

The `<tricky>` definition would not work properly in Lex, the final `x` in the specification being included in the token. Alex does not suffer from this problem.

The `<dot>` token is restricted to start code 0 so dot characters will be initially recognised, but they will be ignored after a `(` token has been read, as it changes the start code to `op`. Once a `)` has been read, the start code will revert to 0 and the dots will resume. On the other hand commas are restricted to the `op` start code so they will be initially disabled, appearing between open and close

```
%{
import Alex
%}


{ ^A = A-Z       }
{ %t = [^ ^t]*^n }

"ctx_lx"/"ctx_acts":-

  <only_ide>  ::=   ^n\^A+/%t           %{ only_ide  = tkn 0    %}
  <start_ide> ::=   ^n\^A+              %{ start_ide = tkn 1    %}
  <end_ide>   ::=      ^A+/%t           %{ end_ide   = tkn 2    %}
  <ide>       ::=      ^A+              %{ ide       = tkn 3    %}
  <tricky>    ::=      x*/x             %{ tricky    = tkn 4    %}
  <dot>       ::=  "0":^.               %{ dot       = tkn 5    %}
  <open>      ::=      ^(               %{ open      = start op %}
  <comma>     ::= "op":^,               %{ comma     = tkn 6    %}
  <close>     ::=      ^)               %{ close     = start 0  %}
  <>          ::=      [.^n]


%{
tkn n _ _ inp len cont st = Ide n (take len inp):cont st

start sc _ _ _ _ cont (_,s) = cont (sc,s)


data Tkn = Ide Int String

tokens:: String -> [Tkn]
tokens inp = gscan ctx_scan () inp

ctx_scan:: GScan () [Tkn]
ctx_scan = load_gscan (ctx_acts,stop_act) ctx_lx
        where
        stop_act _ _ "" _ = []
        stop_act _ _ _  _ = error "tokens"
%}
```

Figure 7: Specifying context in Alex.

brackets.

Acknowledgements

I would like to thank Tom Alardice for providing the original motivation for writing Alex, Henk Muller for suggestions and encouragement, Ian Holyer for the various discussions that have helped to shape it and Alastair Reid for feedback and suggestions.