Alex & Happy

LEXERS AND PARSERS IN HASKELL

Jyotirmoy Bhattacharya

Alex and Happy

Lexers and Parsers in Haskell

Jyotirmoy Bhattacharya

This book is for sale at http://leanpub.com/alexandhappy

This version was published on 2015-05-05

Contents

Preface

The full source code for the examples can be downloaded from Github[1]

This is a work-in-progress. Please send your comments and bug reports to jyotirmoy@jyotirmoy.net[2].

Cover image

Leafy seadragon[3]. Image by user lecates'[4] on Flickr[5]. Used under the terms of the CC BY-SA 2.0[6] license. The artwork for the cover page is released under a newer version of the same license and is included in the Github repository for this book.

[1] https://github.com/jmoy/alexhappy
[2] mailto:jyotirmoy@jyotirmoy.net
[3] http://en.wikipedia.org/wiki/Leafy_seadragon
[4] https://www.flickr.com/people/lecates/
[5] https://www.flickr.com/photos/lecates/290117655/
[6] https://creativecommons.org/licenses/by-sa/2.0/

1. Introduction

1.1 Lexers and parsers

Many programs take structured text as input. In the case of a compiler this text is a program written in a certain programming language. In the case of a analytics tool it might be a log file with entries in a particular format. In the case of a web server it may be a configuration file describing the different sites to be served. In all these cases the text first appears to the program as a simple sequence of bytes. It is the task of the program to interpret the bytes and discover the intended structure. That is the task of parsing.

Parsing itself is often divided into two phases. The first phase is *lexical analysis*, is carried out by a part of the program called the *lexical analyser* or *lexer* for short. This phase breaks up the sequence of bytes provided as input into a sequence of indivisible *tokens* while at the same time carrying out other tasks such as keeping track of line numbers of the source file and skipping whitespace and comments. The *parser* proper then takes these tokens and assembles them into larger structures according to rules given by a *grammar*.

Thus, given the following line of Haskell source code

```
module   Main where   -- the main module
```

the lexical analyser will produce the tokens `module`, `Main` and `where` while skipping the whitespaces and the comment. It may also mark `module` and `where` as keywords of the language. The parser then looks at this sequence of tokens and recognizes this as a statement declaring the module “Main”.

The advantage of having a separate lexer is that lexers are easy to write since the breaking a string of bytes into tokens usually does not require a understanding of the entire input at a time. Boundaries between tokens can usually be found by looking at a few characters at a time. At the same time by getting rid of extraneous data like whitespace and comments, the presence of a lexer simplifies the design and implementation of the parser since the parser can now deal with a more abstract representation of the input.

1.2 Alex and Happy

Alex and Happy are programs that generate other programs. Given a high-level description of the rules governing the language to be parsed, `alex` produces a lexical analyser and `happy` produces a parser. Alex and Happy are the intellectual descendants of the Unix programs Lex and YACC (Yet Another Compiler Compiler). But whereas the lexers and parsers generated by Lex and YACC were C, the programs generated by Alex and Happy are in Haskell and therefore can easily be used as a part of a larger Haskell program.

Installation

Alex and Happy are part of the Haskell Platform[1] and should have been installed when you installed the Haskell Platform. The can also be installed or updated using the Cabal package manager.

[1] http://www.haskell.org/platform/

2. A Basic Lexer

We begin with the simple task of writing a program that extracts all words from the standard input and prints each of them on a separate line to standard output. Here we define a word to mean a string of uppercase or lowercase letters from the English alphabet. Any non-letter character between words should be ignored.

Alex is a preprocesser. It takes as as input the description of a lexical analyser and produces a Haskell program that implements the analyser. An Alex file for this task above can be found in `wordcount/wordcount.x` can be found in this book's Github repository. Alex input files are usually given the extension `.x`. Invoking Alex on this file with the command

```
alex wordcount.x
```

produces the file `wordcount.hs` which in turn can be compiled with

```
ghc --make wordcount
```

Cabal also knows that it can get a file with an `.hs` extension by invoking `alex` on a file with the same name and an `.x` extension. If you have Alex files in your project you have to include `alex` among the `build-tools` and also include the `array` package as a dependency since the programs produced by `alex` use this package.

2.1 Structure of an Alex file

Initial code

Our `wordcount.x` file is composed of a number of components. At the beginning is a code fragment enclosed in braces which is copied

verbatim by Alex to its output

```
{
module Main(main) where
}
```

This is where you put the module declaration for the generated code. This is also where you need to put any import statements that you may need for the rest of your program. Do not put any other declarations here since Alex will place its own imports after this section.

The wrapper

The next line

```
%wrapper "basic"
```

specifies the kind of interface Alex should provide to the lexical analyser it generates. The "basic" wrapper is the simplest option. In this case Alex will provide a function `alexScanTokens::String->[Token]` whose argument is a string which contains the entire input and whose result is a list of tokens, where `Token` is a user-defined type. In later chapters we will discuss other interfaces that Alex can provide which are more flexible but which also require more work from the user.

Macro definitions

Next come definitions of macros.

```
$letter = [a-zA-Z]
$nonletter = [~$letter\n]
```

A macro is a shortcut specifying a set of characters or a regular expressions. The names of character set macros begin with `$` and those of regular expression macros begin with `@`.

In the definition of `$letter` the expression `[a-zA-Z]` on the right-hand side is a character set. The square brackets `[]` are the union operator, forming the union of a list of character sets given within the bracket. `a-z` and `A-Z` denote character ranges representing the lowercase and uppercase characters respectively whose union is the character set `$letter`.

In the definition of `$nonletter` the expression `~$letter` denotes the complement of the character set `$letter`. The universal set in Alex's negation operator does not contain the newline, so we specify it separately with the escape sequence `\n`.

Rules

After the definitions we specify the rules. The beginning of the rules section is market by the line `token :-`. The name `token` is purely decorative. Alex looks for only the `:-` to see where rules begin. In our example the rules are

```
tokens :-
  $nonletter+     ;
  $letter+            {id}
```

Each rule is a regular expression followed by action. Whenever a regular expression in a rule matches the input from the current position Alex carries out the corresponding action. If more than one rule matches the current input the longest match is preferred.

If there are multiple longest matches then the rule which comes earlier in the Alex file is preferred.

Actions can be of two types: either a bare `;` or some Haskell code. If the action is a bare `;` then Alex skips the input which matches the rule. In the case of an action which is Haskell code, what Alex does depends on the wrapper. For the basic wrapper the code for each action must be a function of the type `String->Token` which is called by Alex with the matched input to produce a token.

The first rule in our example says that sequences of non-letters must be skipped. Here `+` is the regular expression operator which matches one or more occurrences of the preceding regular expression.

The second rule similarly matches a sequence of one or more letters. Alex's rule for finding the longest match automatically ensures that it will extend a word as far as possible. The action in this case is the standard Haskell identity function `id` which returns its input unchanged. This implicitly defines our token data type to be `String` the value of a token is simply the source text corresponding the the word.

Final code segment

At the end of the Alex file is another code segment that is copied verbatim to the output file. We make use of it to define our `main` function.

```
{
main::IO ()
main = do
  s <- getContents
  let toks = alexScanTokens s
  mapM_ putStrLn toks
}
```

This reads in the the entire standard input into the `String` `s`. This is then passed to the Alex-generated function `alexScanTokens` which generates a list of tokens—in our case just a list of strings in which each element is a word found in our input. The final line of `main` prints each of these words in a separate line.

3. Monadic Parsers and Lexers

Lexers and parsers must often maintain state while they parse a text. One way of encoding stateful computations in Haskell is to use the `State` monad. We can do so using the monadic interface to Happy which allows parser actions to be of any monadic type, not just actions in the `State` monad. Alex can work with monadic actions even more easily since it allows its actions to be of any type. Finally, rather than requiring a list of tokens, Happy can provide a parser function which executes a monadic action to acquire a new token, thus easily interfacing with a monadic Alex parser.

The code for this chapter is in the folder `arith1/` in the Github repository of this book. The language we are trying to parse is a simple language of arithmetic and boolean expressions adapted for Pierce's *Types and Programming Languages*. There is one numeric literal `0` and two boolean literals `true` and `false`. The numeric functions `succ` and `pred` evaluate to the successor and predecessor of a number. The function `iszero` applied to a number evaluates to `true` or `false` depending on whether a number is `0` or not. Finally, the term `if [expr1] then [expr2] else [expr3]` evaluates to `[expr2]` if `[expr1]` evaluates to `true` and to `[expr3]` otherwise.

We want to provide a Read-Eval-Print Loop (a REPL) for expressions in this language. It will not do for such a program to die when faced with a lexical and syntax error. We expect it to print an error message and recover to accept the next expression. We will achieve this in this chapter by using Haskell's MonadError[1] class to handle errors gracefully.

[1] https://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-Error-Class.html

3.1 The Lexer

The prologue

```
{
module Lexer (Token(..),P,evalP,lexer) where
import Control.Monad.State
import Control.Monad.Error
import Data.Word
}
```

Alex rules

```
tokens :-
    $white+      ;
    true                        {TTrue}
    false               {TFalse}
    0                       {TZero}
   succ                       {TSucc}
   pred                       {TPred}
   if                       {TIf}
   then                       {TThen}
   else                       {TElse}
   iszero              {TIsZero}
```

Haskell code

First we define the token types.

```
{
data Token =
     TTrue
     | TFalse
     | TZero
     | TSucc
     | TPred
     | TIf
     | TThen
     | TElse
     | TIsZero
     | TEOF
     deriving (Eq,Show)
```

and the functions that must be provided to Alex's basic interface,

```
type AlexInput = [Word8]
alexGetByte :: AlexInput -> Maybe (Word8,AlexInput)
alexGetByte (b:bs) = Just (b,bs)
alexGetByte []     = Nothing

alexInputPrevChar :: AlexInput -> Char
alexInputPrevChar = undefined
```

Now comes our parser monad. `Control.Monad.Error` defines the type `Either String` to be an instance of the class `MonadError` which allows us to throw and catch exceptions. We will use this facility to demonstrate the graceful handling of parse errors. We apply the `StateT` monad transformer to the base error monad to allow us to store the current input to alex as state.

```
type P a = StateT AlexInput (Either String) a

evalP::P a -> AlexInput -> Either String a
evalP = evalStateT
```

Next we define an action in our monad which will produce a new token.

```
readToken::P Token
readToken = do
          s <- get
          case alexScan s 0 of
                    AlexEOF -> return TEOF
               AlexError _ -> throwError "!Lexical error"
                  AlexSkip inp' _ -> do
                         put inp'
                         readToken
                  AlexToken inp' _ tk -> do
                         put inp'
                         return tk
```

We have not provided any `wrapper` specification to Alex, so it provides us with the lowest level interface. The function `alexScan` takes an `AlexInput` and start code and tries to produce a token. We will discuss start codes in detail in a later chapter. For now we use the default startcode of `0`. `alexScan` can return four kinds of values.

- `AlexEOF` indicates the end of output. In this case we return the token `TEOF` which will indicate end of input to the parser.
- `AlexError` an error condition. Here we make use of the `throwError` action provided by the MonadError class to raise an exception within our monad.
- `AlexSkip` is an indication an indication that some input needs to be skipped over. `inp'` indicates the position in the input

from which scanning should continue. We use the `put` action provided by `StateT` to make a record of this. Then we call `readToken` again to try and find a token.

- `AlexToken` indicates that a token has been fount. Once again `inp'` is the new input position which we save. `tk` is the lexer action specified in the patter. For this lexer it is simply a value of the `Token` type which we return.

Ideally the Happy parser would take a monadic action like `readToken` and use it to fetch token. Instead, for historical performance reasons, it expects a different interface that we provide in the `lexer` function.

```
lexer::(Token -> P a)->P a
lexer cont = readToken >>= cont
```

Happy passes a *continuation* `cont` to the lexer, which represents the parsing task to be performed as a function of the next token read and expects it to be called with the actual next token. `lexer` does precisely this, using `readToken` to fetch the token and using the monadic bind operator `>>=` to pass it on to the provided continuation. We could have also written it as

```
lexer cont = do
  token <- readToken
  cont token
```

3.2 The Abstract Syntax Tree Type

The `AST` module provides an abstract syntax tree type for our language. It will be the job of the parser to build values of this type.

```
module AST where

data Term =
     STrue
     | SFalse
     | SZero
     | SIsZero Term
     | SSucc Term
     | SPred Term
     | SIfThen Term Term Term
     deriving Show
```

3.3 The Parser

The prologue

This is just the module declaration and imports.

```
{
module Parser(parse) where
import AST
import qualified Lexer as L
import Control.Monad.Error
}
```

Defining the types of lexer and parser

It is in this section that we tell Happy that we need a monadic parser and have a monadic lexer.

```
%monad{L.P}
%lexer{L.lexer}{L.TEOF}
%name parse
%tokentype{L.Token}
%error {parseError}
```

The declaration `%monad{L.P}` says that we need a monadic parser that operates in the `L.P` monad. The declaration `%lexer{L.lexer}{L.TEOF}` specifies that we have a monadic lexer `L.lexer` and `L.TEOF` is the token pattern that denotes end of input.

Defining tokens

```
%token
true          {L.TTrue}
false         {L.TFalse}
zero           {L.TZero}
iszero        {L.TIsZero}
succ               {L.TSucc}
pred               {L.TPred}
if               {L.TIf}
then               {L.TThen}
else               {L.TElse}
```

Grammar rules

```
%%

Term        :  true                                  {STrue}
|  false                          {SFalse}
|  zero                                {SZero}
|  iszero Term                        {SIsZero $2}
|  succ Term                        {SSucc $2}
|  pred Term                        {SPred $2}
|  if Term then Term else Term        {SIfThen $2 $4 $6}
```

The Error Handler

The error handling function `parseError` is now of the type `L.Token -> L.P a`. Happy requires that it be polymorhpic in type `a`. In our case it again uses the `throwError` action of `MonadError` to raise an exception within the monad.

```
{
parseError _ = throwError "!Parse Error"

}
```

3.4 The Driver

The driver code in `Main.hs` and `Evaluator.hs` provide a primitive REPL, taking one expression per line and printing the result of evaluating it. The interesting part as far as Alex and Happy are concerned are the following two lines in the function `myREPL` in `Main.hs`

```
      case L.evalP P.parse (encode s) of
        Right t -> putStrLn (output t)
        Left s -> putStrLn s
```

Because we are using `Either String` as an error monad, a lexing or parsing error which leads to our calling `throwError` results in `L.evalP` returning a `Left` value. In our case we just print in and continue on our way to read the next line. This is much better than the alternatives in earlier chapters of calling `error` and dying.

4. Significant Whitespace

We present a lexer for a Python-like language which uses whitespace for indicating block structure. The actual language is very simple: identifiers consisting of alphanumeric characters and underscores separated by whitespace. The amount of whitespace at the beginning of a line indicates block structure. For simplicity we do not convert between tabs and spaces, so you will get unpredictable answers if you mix the two.

While parsing we maintain a stack of indentation levels. When encountering a line with more indentation than the previous line we generate an "Indent" token to denote that a new block has started. When encountering a lines with less indentation than the previous line, or at the end of the input, we pop items from the stack until we can match the current indentation and emit "Dedent" tokens for each level of indentation popped. A parser for this language can treat the "Indent" and "Dedent" tokens like opening and closing braces for a C-like language.

Our lexer is going to be a monadic lexer of the type discussed in the last chapter. We will use the `State` monad to keep track of our indentation stack.

4.1 Imports

```
{
module Main (main) where
import Control.Monad.State
import Control.Monad
import Data.Word
import Codec.Binary.UTF8.String (encode)
}
```

4.2 Alex rules

```
tokens :-
       \n$white*               {startWhite}
       $white+                       ;
       [a-z0-9_]+               {name}
```

4.3 Datatypes and utility functions

```
{
data Token
     = TIndent
     | TDedent
     | TNewline
     | TName String
     | TEOF
     deriving (Eq,Show)

-- The functions that must be provided to Alex's basic \
interface
-- The input: last character, unused bytes, remaining s\
tring
data AlexInput = AlexInput Char [Word8] String
     deriving Show
alexGetByte :: AlexInput -> Maybe (Word8,AlexInput)
```

```
alexGetByte (AlexInput c (b:bs) s) = Just (b,AlexInput \
c bs s)
alexGetByte (AlexInput c [] [])    = Nothing
alexGetByte (AlexInput _ [] (c:s)) = case encode [c] of
                                          (b:bs) -> Just (b, Al\
exInput c bs s)

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

State

We use a simple list `indent_stack` as our stack of indentation levels. Multiple "Dedent" tokens may have to be yielded when the level of indentation decreases. `pending_tokens` keeps track of these.

```
data ParseState =
     ParseState {input::AlexInput,
                 indent_stack::[Int],
                 pending_tokens::[Token]}
                 deriving Show

initialState::String -> ParseState
initialState s = ParseState {   input = AlexInput '\n' \
[] s,
                                indent_stack = [1],
                                pending_tokens = []
                                }
```

The Parser monad

```
type P a = State ParseState a

evalP::P a -> String -> a
evalP m s= evalState m (initialState s)
```

4.4 Lexer actions

Process whitespace at the start of a line

We check if the number of whitespace characters at the start of a line is greater than the number for the indentation level at the top of the stack. If the current line has more indentation we push the number of whitespace characters in it into the stack and enqueue an "Indent" token. If the number of whitespace characters in the current line is less than that on the top of the stack, we pop the stack looking for a level where the number of whitespace characters matches those in this line. If we do find the level then we enqueue as many "Dedent" tokens as items popped from the stack. If a matching level cannot be found then an error is signalled.

```
startWhite::Int->String->P Token
startWhite n _ = do
           s<-get
           let is@(cur:_) = indent_stack s
           when (n>cur) $ do
              put s{indent_stack = n:is,pending_tokens \
= [TIndent]}
           when (n<cur) $ do
              let (pre,post@(top:_)) = span (> n) is
              if top == n then
                 put s{indent_stack = post,
                                    pending_tokens = ma\
p (const TDedent) pre}
              else
```

```
                error "Indents don't match"
          return TNewline
```

Process a word

```
name::n->String->P Token
name _ s = return (TName s)
```

Action to read a token

If there are pending tokens we return one of them without consuming input. Otherwise we scan the input. At the end of input we have to be careful to issue sufficient "Dedent" tokens to close any open blocks. We do this by the acting as if we have received a blank line with no indentation (`startWhite 1 ""`)

```
readToken::P Token
readToken = do
          s <- get
          case pending_tokens s of
               t:ts -> do
                         put s{pending_tokens = ts}
                         return t
               [] ->  case alexScan (input s) 0 of
                        AlexEOF -> do
                                     rval <- startWhite \
1 ""
                                     put s{pending_token\
s=(pending_tokens s)++[TEOF]}
                                     return rval
                        AlexError _ -> error "!Lexical e\
rror"
                        AlexSkip inp' _ -> do
                           put s{input = inp'}
```

```
                        readToken
                    AlexToken inp' n act -> do
                        let (AlexInput _ _ buf) = inp\
ut s
                        put s{input = inp'}
                        act n (take n buf)
```

4.5 Driver

```
readtoks::P [Token]
readtoks = do
            t<-readToken
            case t of
              TEOF -> return [t]
              _ -> do
                rest<- readtoks
                return (t:rest)

tokenize::String->[Token]
tokenize s =
        evalP readtoks s

main::IO()
main = do
     input <- getContents
     print (tokenize input)

}
```

5. Parsing comments and strings: Startcodes

In this chapter we discuss how to parse two features of programming language grammars: strings and comments.

Given the text

```
/* This whole section commented out
/* A sample program */
*/
"Bo\"bcat" cat cat cat
"/*Comment*/" cat cat /*cat*/
```

we expect our program to tell us that the person called `Bo"bcat` has three cats and the person called `/*Comment*/` has two cats. The names of the cat owners are given as strings delimited by double quote characters, with the escape sequence `\"` being used for a double quote within a name. `/*` and `*/` denote the beginning and end of comments. Comments can span multiple lines and nested comments allowed.

Comments and strings have very different rules of lexical analysis than the rest of the program. Most text within strings and comments are not interpreted. On the other hand escape sequences within strings are interpreted . The nesting of comments poses another problem since a regular expression cannot remember arbitrary depth of nesting and the depth of nested comments must be tracked elsewhere.

5.1 Token definitions

Startcodes are the solution provided by `alex` to this problem. A startcode is just a number with a set of these numbers being attached to each rule in an `alex` specification. When calling `alex` to find a token we specify the startcode to use and `alex` then considers only the rules belonging to that startcode.

A lexer and parser for the language of this chapter is in the `startcode/` folder of the Github repository of this book. It uses the monadic interface of `alex` and `happy` which is discussed in Chapter 3.

Let's look at the token defintions in `Lexer.x` first. The initial definitions are ordinary ones that skip whitespace and define the `cat` token:

```
<0>         $white+          ;
<0>         cat              {plainTok TCat}
```

Note the `<0>`. This specifies the startcode for which these rules apply. `0` is the default startcode which we will be using for the main program text. Beware of a common error: if you leave out the startcode at the beginning of a rule then that rule applies to all startcodes. This can bite you when you have a lexer with multiple start codes.

Now for the patterns for strings

```
<0>         \"               {beginString}
<stringSC>  \"               {endString}
<stringSC>  .                  {appendString}
<stringSC>  \\[nt\"]         {escapeString}
```

The first rule, which has startcode `0` matches the double quote in ordinary program text that starts a string. `stringSC` defines a new

start code. In the Haskell code generated by `alex`, `stringSC` will be defined with a numeric value which can be used in other parts of the program. The action `beginString`, whose implementation we will discuss below, ensures that later invocations of `alexScan` use the startcode `stringSC`.

The second rule, which also matches a double quote but in the context of the startcode `stringSC` recognized the end of the string. The action `endString` yields the token constructed out of the string and ensures that future invokations of `alexScan` happen with startcode `0`.

The last two rules accumulate characters in the string. The first matches ordinary characters and the second the escape sequences `\n`, `\t` and `\"`.

The patterns for the comments are similar:

```
<0,commentSC>             "/*"              {beginCommen\
t}
<commentSC> "*/"               {endComment}
<commentSC> [.\n]             ;
```

The first rule recognizes the beginning of a comment. The prefix `<0,commentSC>` specifies that it applies both in the middle of ordinary program text as well as in the middle of a comment when the start code is `commentSC`. This is to handle nested comments. The second rule handles the ending of a comment. Between then `beginComment` and `endComment` must keep track of the nesting level, with `endComment` switching back to start code `0` only when all levels of comments have been closed.

The last rule just ignores all characters in a comment. Remember that the catch-all character class `.` does not match newlines. Therefore we include an explicit `\n` to allow comments to span lines.

5.2 The state

We use the `State` monad as our parser/lexer monad. The state is of type `ParseState`which keeps track of the nesting level of comments and accumulate characters in strings. The accompanying type `AlexInput` keeps tracks of line numbers. The implementation of these types are in the file `LexParseCommon.hs`.

```
data AlexInput
  = AlexInput {
    aiprev::Char,
    aibytes::[Word8],
    airest::String,
    ailineno::Int}
  deriving Show

-- [...]
data ParseState =
     ParseState {input::AlexInput,
                 lexSC::Int,       --Lexer start code
                 commentDepth::Int,--Comment depth
                 stringBuf::String --Temporary storage \
for strings
                }
     deriving Show
```

5.3 The Lexer Actions

Let's now go back to the lexer actions in `Lexer.h`. The lexer actions have type

```
type LexAction = Int->String->P (Maybe Token)
```

with the arguments being the number of characters matched and the matching string. The result is `Maybe Token` since some actions, such as those marking the beginning of a string, or the accumulation of characters within a string, may not yield a token.

5.4 String lexing actions

`beginString` sets the startcode to `stringSC`. `appendString` and `escapeString` add characters to the beginning of `stringBuf`. `endString` sets the start code back to `0` and returns the string saved in `stringBuf` reversed (since the characters were added to the beginning of the string). The characters are added to the beginning rather than the end of the string as the former is more efficient. The actions `get` and `put` for reading and writing the state are provided by the `State` monad.

```
beginString::LexAction
beginString _ _ = do
  s <- get
  put s{lexSC = stringSC}
  return Nothing

appendString::LexAction
appendString _ (c:_) = do
  s <- get
  put s{stringBuf = c:(stringBuf s)}
  return Nothing

escapeString::LexAction
escapeString _ (_:c:_) = do
  let unesc =
        case c of
          'n' -> '\n'
          't' -> '\t'
```

```
          '"' -> '"'
  s <- get
  put s{stringBuf = unesc:(stringBuf s)}
  return Nothing

endString::LexAction
endString _ _ = do
  s <- get
  let buf = stringBuf s
  put s{lexSC = 0, stringBuf = ""}
  return $ Just $ TString (reverse buf)
```

5.5 Comment lexing actions

The implementation of the comment lexing actions is similar. The beginning of a comment changes the startcode to `commentSC` and increments `commentDepth`. The ending of a comment decrements `commentDepth`, setting the start code back to `0` when `commentDepth` reaches `0`.

```
beginComment::LexAction
beginComment _ _ = do
  s <- get
  put s {lexSC = commentSC,
         commentDepth = (commentDepth s)+1}
  return Nothing

endComment _ _ = do
  s <- get
  let cd = commentDepth s
  let sc' = if cd==1 then 0 else commentSC
  put s {lexSC=sc',commentDepth=cd-1}
  return Nothing
```

5.6 The driver

The monadic action `readToken` in `Lexer.x` provides the interface with `alex`. The following line calls `alexScan` with the startcode saved in the state

```
s <- get
case alexScan (input s) (lexSC s) of
```

When `alex` returns `AlexToken` indicating a match we call the corresponding action, looping back by calling ourselves if the action returns `Nothing` indicating that no token is generated.

```
-- [...]
    AlexToken inp' n act -> do
      let (AlexInput{airest=buf}) = input s
      put s{input = inp'}
      res <- act n (take n buf)
      case res of
        Nothing -> readToken
        Just t -> return t
```

The rest of the code is routine. `lexer` in `Lexer.h` wraps up `readToken` in the interface expected by `happy`. The `happy` parser itself is in `Parser.y`. `main` in `Main.hs` parses texts read in from the standard input.