A Lexer for Haskell in Haskell (first draft)

Thomas Hallgren

March 7, 2003

Abstract

We describe a lexer for Haskell in Haskell. The lexer is implemented in a modular way, reflecting the structure of the description of the lexical syntax in the Haskell report. The largest part – the token recognizing function – is generated from (a transliteration of) the lexical grammar in the Haskell report using a regular expression compiler. The speed of the lexer is comparable to that of a handwritten, monolithic lexer.

1 Introduction

One would have hoped that these days, the *syntax* is the *least* controversial part of a programming language specification, and that implementing a parser for the language would be a routine job, something that is not worth writing a scientific paper about. However, as witnessed by many lengthy discussions on the Haskell mailing list, this does not appear to be true for Haskell. The syntax is complex, and the specification in the Haskell report leaves room for many questions.

In spite of the complexities of the Haskell syntax, it appears that solving the problem of parsing Haskell has not been seen as prestigious enough that someone has been willing to invest the time to do it well.

Being inspired by *Typing Haskell in Haskell* [Jon99], it seems like a good idea to work toward *Parsing Haskell in Haskell*, i.e., building an executable specification of the Haskell syntax. In this paper, we give a first contribution toward that goal by creating

- *A Lexer for Haskell in Haskell* that is *simple*, *correct*, *efficient* and *reusable.*

Correctness means that we want the lexer to agree with the specification, that is with what is written in the Haskell report. *Simplicity* means that we want it to be easy to see that the implementation agrees with the specification. We want *efficiency*, so that the lexer is not merely a specification, but, like the specification of the Haskell module system presented in [DJH02], also a practical implementation that can be used in a Haskell compiler without slowing it down too much. We want *re-usability*, so that we can easily adapt the lexer to changes of the Haskell language (and there have been subtle changes in amost every(?) new version of the Haskell report), and so that the lexer can be used in tools other than compilers (for example, in an HTML renderer with syntax highlighting).

The goals are not independent: simplicity is likely to be good both for correctness and re-usability. In fact, becase the specification isn't formal, we cannot expect a formal proof of correctness, but instead rely on simplicity to convince ourselves that the implementation is correct.

2 Method

Most of the work was done during a warm and sunny summer week in 1999. The final parts, nested comments and layout processing, were added in 2001.

The goal was to use the (original) Haskell 98 Report [Je99] as the only source of information, preferably restricting attention to Appendix B, the syntax reference. The author tried to forget everything (he thought) he already knew about Haskell.

Appendix B consists of some segments of pseudo-formal notation, glued together and clarified by informal text. Unfortunately, the description of the lexical syntax is not self-contained. For example, it was necessary to consult Chapter 2, where you are referred to (the wrong section of) Chapter 5, to find a confirmation that qualified names are part of the lexical syntax (i.e., that spaces are not allowed in `M.x`).

Fortunately, these problems appear to be fixed in the revised Haskell 98 report [Jon03], and some confusing formulations have been removed.

3 Existing solutions

We have taken a quick look at the following Haskell implementations[1]:

Source	Lexer language	Lexer size
The hssource library [M$^+$]	Haskell 98	$\sim$ 1300 lines
NHC [NHC02]	Haskell 98	$\sim$ 1200 lines
GHC [GHC02]	nonstandard Haskell	$\sim$ 1300 lines
HBC [Aug98]	C	$\sim$ 1700 lines
Hugs [Hug02]	C	$\sim$ 2000 lines

In all cases, the lexer is handwritten, essentially as a large monolithic chunk of code, without much separation of concerns. As a result, it would presumably be difficult to verify the correctness of these lexers, and it has probably been difficult to adapt them to new versions of Haskell (1.2, 1.3, 1.4, 98, revised 98...) without introducing bugs. Only one of the goals stated in the introduction is achieved: efficiency.

Monolithic solutions are complex because of the many non-trivial subtask involved:

[1]The sizes given here should be taken with a grain of salt.

- removing nested comments, preserving position information, interacting with the parser to implement the layout rule. recognizing string literals, recognizing simple identifiers, recognizing qualified identifiers, recognizing keywords and reserved operators. ...

Preserving position information is not optional in Haskell: it is needed to implement the layout rule.

One tempting approach to deal with the complexity of a Haskell lexer is of course to split the lexer into a number of simpler passes. At first sight, it may seem possible to implement many of the subtasks mentioned above separately, but it turns out not to be so easy. We give a few examples below.

3.1 Hopeless attempts

===> TODO: *Skip this entire section!?*

Nested comments It seems easy to define a function to recognize nested comments; they are enclosed in `{- -}` brackets and have to be properly nested. However, doing this as a preprocessing pass, separately from the rest of the lexical analysis, seems impossible, because `{-` and `-}` do not always start and end comments:

```
code -- comment -} comment
code -- comment {- comment -} comment
code ----> code {- comment -} code
code "{-" code "-}" code
code " \"{-" code
```

To correctly identify nested comments, it appears that we also have to correctly identify one-line comments and string literals. Recognizing one-line comments also requires recognizing symbolic identifiers (an extra complexity that was introduced in Haskell 98). Recognizing string literals is nontrivial because of the escape sequences.

Qualified names At first it may seem that recognition of qualified names could be added as a post-processing step to a lexer that only recognizes simple names by simply gluing tokens together. However, in Haskell, a qualified name `M.x` is treated as *one* lexical symbol, and extra space is not allowed before or after the dot, so it seems difficult to get the post-processing right if the simple lexer discards white space. Even if white space is preserved, there are some additional pitfalls with a post-processing solution. For example, a simple lexer would presumably treat `M..` as two tokens: `M` followed by the reserved operator `..`. With qualified names, it should be treated as the operator `.` qualified by `M`. This could be handled as a special case, but then the post-processor solution does not seem like a particularly elegant solution anymore.

Keywords Keywords is another candidate for recognition in a post-processor. But keywords interact with qualified names, for example while `M.them` is a qualified name, `M.then` is three tokens, the last one being the keyword `then`.

4 A better solution

To achieve our goals of simplicity and correctness, it seems like a good idea to structure the implementation of the lexer according to the structure of the specification given in Appendix B of the Haskell 98 report (Chapter 9 in the revised report).

Appendix B of the Haskell report contains the following lexer related parts:

- B.2 The lexical syntax: this is a BNF-like description of what a Haskell program looks like on the lexical level.
- B.3 Layout: this section discusses several relevant issues:
  - How to compute the indentation of a lexeme,
  - How to insert the additional tokens $\langle n \rangle$ and $\{n\}$ to indicate layout.
  - Layout contexts and the function L that implements the layout rule.

Our implementation follows this structure, and below we describe the parts in turn. The reader is encouraged to have a copy of the Haskell report handy for reference.

4.1 The lexical syntax

From the lexical syntax in Appendix B.2, we construct a token recognizing function called `haskellLex`. As can be seen in Figure 1, the input is a string (a Haskell module to be parsed), and the output is a list of tokens, represented as pairs of values from the type `Token` that classifies tokens (see Appendix A), and a string of the characters that form the token. The function `haskellLex` preserves white space, so it has a very simple inverse: `concatMap snd`.

In previous lexers for Haskell, this part is typically implemented as a hand-written function that in addition to recognizing tokens also computes positions and discards white space. Since the function (`haskellLex`) preserves white space, we can deal with those issues separately. This solution also allows the function to be reused in applications where it is desirable to preserve white space.

While this separation of concerns makes `haskellLex` a much easier function to implement by hand, we have gone one step further: we have simply transliterated the lexical grammar, using a set of regular expression combinators, and automatically generated the function from the grammar using a regular expression compiler.

The transliterated lexical syntax is included in Appendix B, and the combinators used are explain below. While the regular expression compiler is not

```
module HsLex(haskellLex) where
import Char
import HsLexUtils(Token(..),nestedComment, ...)

type Lexer = Input -> Output
type Input = String
type Output = [(Token,String)]

haskellLex :: Lexer
haskellLex is = ...

...
```

Figure 1: The output of the lexer generator

```
-- This program generates the core of lexical analyzer for Haskell

import HaskellLexicalSyntax(program) -- The lexical syntax specification
import LexerGen(lexerGen)            -- The lexer generator

main = lexerGen "HsLex" "haskellLex" program
```

Figure 2: The lexer generator applied to the lexical syntax of Haskell

included here, its application to the Haskell lexical grammar is shown in Figure 2. Notice that it is the nonterminal *program* that specifies what a Haskell program is on the lexical level, and that is the nonterminal from which we generate our token recognizing function.

While the generated function `haskellLex` is too big to be included here, Appendix H shows what the regular expression compiler produces for a simpler regular expression.

The regular expressions combinators are summarized in Table 1. To specify output, our combinators include the combinator `o`, and the lexical grammar as been augmented to indicate what output to produce. So, the combinators do not specify pure regular expressions, but rather *transducers*, i.e., automata that have both input and output transitions. This is a slight deviation from standard text book methods, where output is represented by *final states* rather than output transitions.

The transducer expressions are compiled in the same way as regular expressions are typically compiled: an expression is converted to a NFA (nondeterministic finite automaton) and the NFA is converted to a DFA (deterministic finite automaton). The only difference is that some transitions denote output and some denote input.

In the Haskell function generated from the DFA, input transitions are given higher priority than output transitions. This means that as long as more input can be consumed, no output will be produced. This is how the *maximal much*

Operation	Text book	Haskell Report	Combinator
The empty string	ϵ		e
One character	`c`	`c`	`t 'c'`
Optional	$r?$	$[r]$	`opt` r
Sequence	r_1r_2	r_1r_2	r_1 `&` r_2
Alternative	$r_1\|r_2$	$r_1\|r_2$	r_1 `!` r_2
Zero or more	$r*$	$\{r\}$	`many` r
One or more	$r+$		`some` r
Difference		$r_{1<r_2>}$	
Output			`o` t

Table 1: Comparing our regular expression combinators to the notation used in the Haskell report and standard text book notation.

rule is implemented.

The Haskell report uses one nonstandard operation: difference $r_{1<r_2>}$. It is used to exclude keywords from the productions for identifiers, and in other similar situations. Our combinators do not include the difference operator[2], and we have simply omitted such exclusions from the grammar. This makes the grammar ambiguous, resulting in an automaton that, e.g., after seeing the string `then` could output either a `Varid` token or a `Reservedid` token. The ambiguity is resolved by choosing the earlier token in the token data type. The token type thus has to be in the `Ord` class, and the order has to be chosen so that the ambiguities are resolved correctly.

4.2 Computing source positions and removing white space

The output from the token recognizing function `haskellLex` is fed to a the function `addPos`, that adds source positions, and then to the function `rmSpace` that removes white space.

```
type Pos = (Int,Int) -- (row,column)
type PosToken = [(Token,(Pos,String)]

addPos    :: [(Token,String)] -> [PosToken]
rmSpace   :: [PosToken] -> [PosToken]
```

The implementation of these functions is included in Appendix D.

We compute both the horizontal and vertical position of every token. While this is more than we need to implement layout, accurate source positions are of course useful in error messages.

While the Haskell report specifies that tab stops are 8 characters apart, it does not say where the first tab stop is located. For the implementation of function `addPos`, we have, in apparent agreement with other Haskell implementations, assumed that the first tab stop is located in column 1.

[2]It is sad that the Haskell report doesn't say where one can find a description of how to implement it.

4.3 Adding layout indicator tokens

Adding the layout indicating tokens $\langle n \rangle$ and $\{n\}$, is implemented by the separate function `layoutPre`,

```
layoutPre :: [PosToken] -> [PosToken]
```

The implementation of `layoutPre` is included in Appendix E.

4.4 Layout contexts and the function L

The function L is responsible for implementing the layout rule, i.e., inserting `{` and `}` tokens that are implied by layout. It uses a *layout context stack* to do its job, and also needs cooperation with the parser.

If it weren't for the need for cooperation with the parser, the function L could have been implemented as a pure function of type `[PosToken] -> [PosToken]`, pretty much as it is presented in the Haskell report.

To implement the interaction with the parser, we use a parser monad that as part of its state has the remaining tokens, and the current layout context stack:

```
type LayoutContextStack = [Int]
type State = ([PosToken],LayoutContextStack)

-- The parser monad:
newtype PM a = PM (State->Either Error (a,State))
type Error = String
get :: PM State
set :: State -> PM ()
```

The parser (which is was inherited from the hssource library) interacts with the lexer through two functions:

```
token :: PM PosToken
popContext :: PM ()
```

The function `token` is called when the parser needs a new token from the lexer and `popContext` is called when the parser has determined that a `}` has to be inserted to avoid a syntax error. These two functions, which together implement the function L, are included in Appendix F. Most of function L appears as a auxiliary function in the definition of `token`.[3]

4.5 Additional pieces

Nested comments can not be described by regular expressions, so recognize them, the generated function `haskellLex` calls a handwritten function `nestedComment`, which is included in Appendix G, together with some other auxiliary functions that are called from `haskellLex`.

[3]Note that we had to deviate from the report and call it l, since L is not allowed as a function name in Haskell.

5 Putting it together

The part of the lexing that is independent of the parser is combined into a function called `lexerPass1`:

```
lexerPass1 :: String -> [PosToken]
lexerPass1 = layoutPre . rmSpace . addPos . haskellLex
```

The output from `lexerPass1`, together with an empty layout context stack, is used as the initial state when parsing a Haskell file:

```
parseFile :: Monad m => PM a -> String -> m a
parseFile (PM p) s =
  case p (lexerPass1 s,[]) of
    Left err    -> fail err
    Right (x,_) -> return x
```

6 Evaluation

The lexer described in this paper has been developed and used as part of a Haskell front-end in the Programatica project [Pro02]. It is based on code from the hssource library [M^{+}], so we mostly compare with the lexer from that library.

While the lexer has been replaced completely, the parser used in the Programatica Haskell front-end remains essentially the same as the one in the hssource library.

Below, we briefly discuss to what extent the goals stated in the introduction have been achieved.

6.1 Simplicity

Using size as a measure of simplicity:

- The old handwritten lexer for Haskell: 664 lines
- The regular expression compiler: 678 lines
- The transliterated lexical syntax for Haskell: 193 lines
- The new layout processing: 100 lines

Even when including the (reusable) regular expression compiler, the new lexer is not much larger than the old lexer. It is enough to implement two lexers to break even :-)

6.2 Efficiency

Regarding speed, the new lexer+parser seems to be 10-15 percent slower than the old one. The Haskell Prelude and Standard Libraries (2943 lines) are still parsed in less than one second (on a 600MHz Pentium III), so the speed is still more than adequate.

Regarding size, the generated DFA has 153 states, and the Haskell code for the token recognizing function is 5600 lines long. The lexer is thus considerably larger than the handwritten one (and too big to be included here). When compiled with `ghc-5.04.2 -O`, the size of the object code for the module `HsLex` is 529KB. As a comparison, our Haskell parser generated by Happy is 740KB, and the total size of our Haskell front-end is 5458KB.

6.3 Correctness

The original motivation for replacing the hssource library lexer with a completely new one was that we encountered several bugs in the hssource lexer, and because of the complexity of the code, fixing the bugs was rather tedious.

After the replacement, we have not had to pay much attention to the lexer. There has been a small number of changes to the Haskell report (regarding layout processing) and we have found it easy to adapt our lexer to those changes.

Our new lexer is more modular, the parts are simpler and the structure follows the structure of the specification in the Haskell report more closely, so it should be easier to convince oneself that the implementation is correct, and to maintain correctness when changes are made. In particular, to check that the transliteration of lexical syntax is correct, a visual inspection should be enough.

Apart from the correct transliteration of the lexical syntax, the correctness of the generated token recognizing function depends on two more factors:

- The correctness of the regular expression compiler, with respect to the text book methods it is based on [App98]. This is an independent issue.
- The agreement of the semantics implemented by the regular expression compiler with the intended semantics of the pseudo-formal specification in the Haskell report. Due the unfortunate informal style of Haskell report, we can not make any guarantees about this step.

6.4 Re-usability

The lexer has already been reused in two applications:

- An HTML renderer with syntax highlighting for Haskell [Hal02a]. This application generates HTML from the output of the function `haskellLex`, so the fact that it preserves white space was useful.
- A simple program called `stripcomments` [Hal02b] that removes comments and blank lines from Haskell source code. Someone asked for this on the

Haskell mailing list, and by re-using the functions `haskellLex`, `addPos` and `rmSpace`, this simple application was very easy to implement.

6.5 What about Unicode?

The Haskell report states that Haskell uses the Unicode character sets. Our lexer supports Unicode, provided that it is compiled with a Haskell compiler that supports Unicode. It also requires that functions in the standard library module `Char` support Unicode, since the nonterminals *uniWhite*, *uniSmall*, *uniLarge* and *uniDigit* in the lexical syntax are implemented by calls to the functions `isSpace`, `isLower`, `isUpper` and `isDigit`, respectively. As far as we know, the only Haskell compiler that implements this is HBC [Aug98].

7 Conclusion

7.1 Future work

The current solution is far from perfect. Possible improvements include:

- Add conversion of the escapes in character and string literals.
- See if size of the generated DFA can be reduced further...
- Make some of the handwritten parts more readable.
- Make the regular expression compiler more readable.
- Update it to the revised Haskell 98 report.
- Implement the difference operator, so that our lexical grammar can correspond even more accurately to the grammar in the report
- Since the lexical syntax is *not* specified by a regular expression, but a context-free grammar, it seems that a parser generator would be a more appropriate tool than a regular expression compiler...
- Extend the work to a complete *Parser for Haskell in Haskell*
- ...

Acknowledgements

Thanks to Mark P. Jones for suggesting many improvements.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998. ISBN 0-521-58274-1.

[Aug98] Lennart Augustsson. HBC. http://www.cs.chalmers.se/~augustss/hbc/hbc.html, 1998.

[DJH02] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification for the Haskell 98 Module System. In *Proceedings of the 2002 Haskell Workshop*, Pittsburgh, USA, October 2002. http://www.cse.ogi.edu/~diatchki/hsmod/.

[GHC02] The Glasgow Haskell Compiler, 2002. http://www.haskell.org/ghc/.

[Hal02a] Thomas Hallgren. Conversion of Haskell source code to HTML. http://www.cse.ogi.edu/~hallgren/h2h.html, 2002.

[Hal02b] Thomas Hallgren. stripcomments - a simple tools for removing comments and blank lines from Haskell programs. http://www.cse.ogi.edu/~hallgren/stripcomments/, 2002.

[Hug02] Hugs Online. http://www.haskell.org/hugs/, 2002.

[Je99] Simon Peyton Jones and John Hughes (editors). Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://www.haskell.org/definiton/`, February 1999.

[Jon99] M.P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Paris, France, September 1999. http://citeseer.nj.nec.com/article/jones99typing.html.

[Jon03] Simon Peton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, April 2003. ISBN 0521826144, http:/www.haskell.org/definition/.

[M$^+$] Simon Marlow et al. The hssource library. Distributed with GHC.

[NHC02] Nhc98. http://www.cs.york.ac.uk/fp/nhc98/, 2002.

[Pro02] The Programatica Project home page. http://www.cse.ogi.edu/PacSoft/projects/programatica/, 2002.

A The module HsTokens

```
module HsTokens where


-- Haskell token classifications:
```

```
data Token
  = Varid | Conid | Varsym | Consym
  | Reservedid | Reservedop  -- | Specialid
  | IntLit | FloatLit | CharLit | StringLit
  | Special
  | Qvarid | Qconid | Qvarsym | Qconsym
  | Whitespace

  | NestedCommentStart -- will cause a call to an external function
  | NestedComment
  | Commentstart  -- dashes
  | Comment -- what follows the dashes
  | ErrorToken
  | GotEOF
  | TheRest

  | Layout     -- for tagging braces inserted by layout processing
  | Indent Int -- <n>, to preceed first token on each line, see Haskell 98, B.3
  | Open Int   -- {n}, after let, where, do or of, if not followed by a "{"
  deriving (Show,Eq,Ord)
```

B Module HaskellLexicalSyntax

This module is a transcription of the Lexical Syntax given in appendix B.2 of the Haskell 98 Report, except for character set definitions, which are given in module HaskellChars (Appendix C).

The grammar below contains extra annotations (not found in the report) to allow the recognized strings to be paired with token classifications.

```
module HaskellLexicalSyntax(program) where
import List((\\))
import HaskellChars
import HsTokens
import RegExp

program = many (lexeme ! whitespace)

lexeme  = varid      & o Varid
        ! conid      & o Conid
        ! varsym     & o Varsym
        ! consym     & o Consym
        ! literal -- & o Literal
        ! a special  & o Special
        ! reservedop & o Reservedop
        ! reservedid & o Reservedid
--      ! specialid  & o Specialid -- recognized by the parser

        ! qvarid     & o Qvarid
        ! qconid     & o Qconid
        ! qvarsym    & o Qvarsym
        ! qconsym    & o Qconsym

literal = integer    & o IntLit
        ! float      & o FloatLit
        ! char       & o CharLit
        ! string     & o StringLit

whitechar = newline ! a vertab ! a formfeed ! a space ! a tab
newline   = a creturn & a linefeed
          ! a creturn ! a linefeed ! a formfeed

whitespace = some whitechar & o Whitespace
           ! comment & o Comment
           ! ncomment & o NestedComment
comment = dashes & o Commentstart & many (a cany) & newline
dashes = as "--" & many (aa "-")
opencom = as "{-"
ncomment = opencom & o NestedCommentStart
            -- handled by calling an external function

varid = (a small & many (a small ! a large ! a digit ! aa "'")) -- <reservedid>
conid = a large & many (a small ! a large ! a digit ! aa "'")
reservedid =
      as"case" ! as"class" ! as"data" ! as"default" ! as"deriving" ! as"do"
    ! as"else" ! as"if" ! as"import" ! as"in" ! as ! as"infix" ! as"infixl"
    ! as"infixr" ! as"instance" ! as"let" ! as"module" ! as"newtype"
    ! as"of" ! as"then" ! as"type" ! as"where" ! as"_"

--specialid = as"as" ! as"qualified" ! as"hiding"

varsym = (a symbol & many (a symbol ! aa ":")) -- <reservedop>
consym = (aa ":" & many (a symbol ! aa ":")) -- <reservedop>

reservedop = as ".." ! as ":" ! as"::" ! as "=" ! as "\\" ! as "|" ! as"<-"
           ! as "->" ! as "@" ! as "~" ! as "=>"

--specialop = aa "-" ! aa "!" -- recognized by the parser instead

modid = conid
optq = opt qual
qual = modid & aa "."
```

In the report, qvarid etc include both qualified and unqualified names, but here they denote qualified names only, this allows qualified and unqualified names to be distinguished in the parser.

```
qvarid = qual & varid
qconid = qual & conid
qvarsym = qual & varsym
```

```
qconsym = qual & consym

decimal = some (a digit)
octal = some (a octit)
hexadecimal = some (a hexit)

integer = decimal
        ! aa "0" & aa "Oo" & octal
        ! aa "0" & aa "Xx" & hexadecimal

float = decimal & aa "." & decimal & opt (aa "eE" & opt (aa "-+") & decimal)
char = aa "’" & (a (graphic \\ acs "’\\") ! a space ! escape{-<\&>-}) & aa "’"
string =
  aa "\"" & many (a (graphic \\ acs "\"\\") ! a space ! escape ! gap) & aa "\""

escape = aa "\\" & ( charesc ! ascii ! decimal !
                    aa "o" & octal ! aa "x" & hexadecimal )

charesc = aa "abfnrtv\\\"’&"

ascii = aa "^" & cntrl
      ! as"NUL" ! as"SOH" ! as"STX" ! as"ETX" ! as"EOT" ! as"ENQ" ! as"ACK"
      ! as"BEL" ! as"BS" ! as"HT" ! as"LF" ! as"VT" ! as"FF" ! as"CR" ! as"SO"
      ! as"SI"  ! as"DLE" ! as"DC1" ! as"DC2" ! as"DC3" ! as"DC4" ! as"NAK"
      ! as"SYN" ! as"ETB" ! as"CAN" ! as"EM" ! as"SUB" ! as"ESC" ! as"FS"
      ! as"GS" ! as"RS" ! as"US" ! as"SP" ! as"DEL"

cntrl = a ascLarge ! aa "@[\\]^_"

gap = aa "\\" & some whitechar & aa "\\"

--
aa = a . acs
as = ts . acs
```

C Module HaskellChars

This module collects the definitions from the Lexical Syntax in appendix B.3 of the (revised) Haskell 98 report that define sets of characters. These sets are referred to in the rest of the lexical syntax, which is given in module `HaskellLexicalSyntax` (Appendix B).

```
module HaskellChars where

data HaskellChar
  -- ASCII characters are represented by themselves
  = ASCII Char
  -- Non-ASCII characters are represented by the class they belong to
  | UniWhite   -- any UNIcode character defined as whitespace
  | UniSymbol  -- any Unicode symbol or punctuation
  | UniDigit   -- any Unicode numeric
  | UniLarge   -- any uppercase or titlecase Unicode letter
  | UniSmall   -- any Unicode lowercase letter
  deriving (Eq,Ord,Show)

acs = map ASCII

-- Character classifications:
special   = acs "(),;[]`{}"
creturn   = acs "\r"
linefeed  = acs "\LF"
vertab    = acs "\VT"
formfeed  = acs "\FF"
space     = acs " \xa0"
tab       = acs "\t"
uniWhite  = [UniWhite]

cany      = graphic++space++tab
graphic   = small++large++symbol++digit++special++acs ":\"'"
small     = ascSmall++uniSmall++acs "_"
ascSmall  = acs ['a'..'z']
uniSmall  = [UniSmall]
large     = ascLarge++uniLarge
ascLarge  = acs ['A'..'Z']
uniLarge  = [UniLarge]
symbol    = ascSymbol++uniSymbol
ascSymbol = acs "!#$%&*+./<=>?@\\^|-~"
uniSymbol = [UniSymbol]
digit     = ascDigit++uniDigit
ascDigit  = acs ['0'..'9']
uniDigit  = [UniDigit]
octit     = acs ['0'..'7']
hexit     = digit ++ acs ['A'..'F'] ++ acs ['a'..'f']
```

D Module HsLexerPass1

```
module HsLexerPass1 where
import HsLex(haskellLex)
import HsLexUtils
import HsLayoutPre(layoutPre)
import List(mapAccumL)

default(Int)
```

The function `lexerPass1` handles the part of lexical analysis that can be done independently of the parser, i.e., the tokenization and the addition of the extra layout tokens $\langle n\rangle$ and $\{n\}$, as specified in appendix B.3 of the Haskell 98 Report.

```
type LexerOutput = [(Token,(Pos,String))]
type Lexer = String -> LexerOutput

lexerPass1 :: Lexer
lexerPass1 = lexerPass1Only . lexerPass0

lexerPass1Only = layoutPre . rmSpace

rmSpace = filter (notWhite.fst)
  where
    notWhite t = t/=Whitespace &&
                 t/=Commentstart && t/=Comment &&
                 t/=NestedComment

-- Tokenize and add position information:

lexerPass0 :: Lexer
lexerPass0 = addPos . haskellLex

addPos = snd . mapAccumL pos startPos
  where
    pos p (t,r) = (nextPos p s,(t,(p,s)))
      where s = reverse r

type Pos = (Int,Int)
startPos = (1,1) :: Pos -- The first column is designated column 1, not 0.


nextPos :: Pos -> String -> Pos
nextPos = foldl nextPos1

nextPos1 :: Pos -> Char -> Pos
nextPos1 (y,x) c =
    case c of
      -- The characters newline, return, linefeed, and formfeed, all start
      -- a new line.
      '\n'  -> (y+1, 1)
      '\CR' -> (y+1, 1)
      '\LF' -> (y+1, 1)
      '\FF' -> (y+1, 1)
      -- Tab stops are 8 characters apart.
      -- A tab character causes the insertion of enough spaces to align the
      -- current position with the next tab stop.
      -- + (not in the report) the first tab stop is column 1.
      '\t'  -> (y, x+8 - (x-1) `mod` 8)
      _ -> (y, x+1)
```

E The module HsLayoutPre

This is an implementation of Haskell layout, as specified in appendix B.3 of the revised Haskell 98 report.

This module contains the layout preprocessor that inserts the extra $\langle n \rangle$ and $\{n\}$ tokens.

```
module HsLayoutPre(layoutPre) where
import HsTokens

layoutPre :: [(Token,((Int,Int),String))] -> [(Token,((Int,Int),String))]
layoutPre = indent . open

open = open1
```

If the first lexeme of a module is not `{` or `module`, then it is preceded by $\{n\}$ where n is the indentation of the lexeme.

```
open1 (t1@(Reservedid,(_,"module")):ts) = t1:open2 ts
open1 (t1@(Special,(_,"{")):ts)         = t1:open2 ts
open1 ts@((t,(p@(r,c),s)):_)            = (Open c,(p,"")):open2 ts
open1 []                                = []
```

If a let, where, do, or of keyword is not followed by the lexeme , the token n is inserted after the keyword, where n is the indentation of the next lexeme if there is one, or 0 if the end of file has been reached.

```
open2 (t1:ts1) | ltok t1 =
    case ts1 of
      t2@(_,(p@(r,c),_)):ts2 ->
        if notLBrace t2
        then t1:(Open c,(p,"")):open2 ts1
        else t1:t2:open2 ts2
      [] -> t1:(Open 0,(fst (snd t1),"")):[]
  where
    ltok (Reservedid,(_,s)) = s `elem` ["let","where","do","of"]
    ltok _ = False

    notLBrace (Special,(_,"{")) = False
    notLBrace _ = True
open2 (t:ts) = t:open2 ts
open2 [] = []
```

The first token on each line (not including tokens already annotated) is preceeded by $\langle n \rangle$, where n is the indentation of the token.

```
indent (t1@(Open _,((r,c),_)):ts) = t1:indent2 r ts
indent (t1@(t,(p@(r,c),s)):ts)    = (Indent c,(p,"")):t1:indent2 r ts
indent [] = []

indent2 r (t1@(_,((r',_),_)):ts) | r'==(r::Int) = t1:indent2 r ts
indent2 r ts = indent ts
```

F Module HsLexer

This module implements the part of the lexer that interacts with the Happy parser, i.e., the layout processing.

```
module HsLexer where
import ParseMonad
import HsTokens(Token(..))

lexer cont = cont =<< token
```

`popContexts`, together with the error handling in the Happy parser, implements the equation dealing with parse-error(t) in the definition of the function *L*, in appendix B.3 in the revised Haskell 98 report.

```
popContext =
    do (ts,m:ms) <- get
       if m/=0 then set (ts,ms) -- redudant test
         else fail "Grammar bug? Unbalanced implicit braces?"

token = uncurry l =<< get
  where
    -- Here is the rest of the function L in the report:
    -- The equations for cases when <n> is the first token:
    l ts0@((Indent n,(p,_)):ts) ms0@(m:ms)
        | m==n      = ok (semi p)    ts  ms0
        | n<m       = ok (vrcurly p) ts0 ms
    l ((Indent _,_):ts) ms = l ts ms
    -- The equations for cases when {n} is the first token:
    l ((Open n,(p,_)):ts) (m:ms) | n>m = ok (vlcurly p) ts (n:m:ms)
    l ((Open n,(p,_)):ts) []     | n>0 = ok (vlcurly p) ts [n]
    l ((Open n,(p,_)):ts) ms           = ok (vlcurly p)
                                            (vrcurly p:(Indent n,(p,"")):ts)
                                            (0:ms)
    -- Equations for explicit braces:
    l (t1@(Special,(_,"}")):ts) (0:ms) = ok t1 ts ms
    l (t1@(Layout, (_,"}")):ts) (0:ms) = ok t1 ts ms
    l (t1@(Special,(_,"}")):ts) ms     = fail "unexpected }"
    l (t1@(Special,(p,"{")):ts) ms     = ok t1 ts (0:ms)
    -- The equation for ordinary tokens:
    l (t:ts) ms = ok t ts ms
    -- Equations for end of file:
    l [] [] = return eoftoken
    l [] (m:ms) = if m/=0
                  then ok (vrcurly eof) [] ms
                  else fail "missing } at eof"

    ok t ts ctx = setreturn t (ts,ctx)

    vlcurly p = (Layout,(p,"{"))
    vrcurly p = (Layout,(p,"}"))
    semi p = (Special,(p,";"))
```

G Module HsLexutils

```
module HsLexUtils(module HsLexUtils,Token(..)) where
import HsTokens

gotEOF [] = []
gotEOF as = [(GotEOF, as)]

gotError as is =
  (ErrorToken, as):
  if null is then [(GotEOF,[])] else [(TheRest,reverse (take 80 is))]

-- Not reversing the token string here seems to save about 10% of the time
output token as cont = (token,as):cont


-- #ifndef __HBC__
isSymbol _ = False
-- #endif

nestedComment as is next = nest 0 as is
  where
    nest n as is =
      case is of
'-':'}':is -> if n==0
      then next gotError ('}':'-':as) is
      else nest (n-1) ('}':'-':as) is
        '{':'-':is -> nest (n+1) ('-':'{':as) is
c:is -> nest n (c:as) is
_ -> gotError as is -- EOF inside comment
```

H Sample output from the lexer generator

Running the following program,

```
import LexerGen(lexerGen)
import RegExp
import HaskellChars
import HsTokens

main = lexerGen "TstHsLex" "tstLex" r
  where
    r  = many (r1!r2!r3)
    r1 = some (aa 'a' ! aa 'b') & aa 'c' & o Varid
    r2 = some (aa 'a') & o Reservedid
```

```
         r3 = aa 'd' & o Special

         aa = t . ASCII -- recognize one ASCII character
```

produces the following Haskell module:

```
module TstHsLex (tstLex) where
import Char
import HsLexUtils

type Output = [(Token,String)]
type Input = String
type Acc = Input -- reversed
type Lexer = Input -> Output
type LexerState = (Acc->Lexer) -> Acc -> Lexer

tstLex :: Lexer
tstLex is = start1 is

start1 :: Lexer
start1 is = state1 (\ as is -> gotError as is) "" is
state1 :: LexerState
state1 err as [] = gotEOF as
state1 err as iis@(i:is) =
  case i of
    'b' -> state3 err (i:as) is
    'a' -> state4 err (i:as) is
    'd' -> state5 err (i:as) is
    _ -> err as iis

state2 :: LexerState
state2 err as is = output Varid as (start1 is)

state3 :: LexerState
state3 err as [] = err as []
state3 err as iis@(i:is) =
  case i of
    'c' -> state2 err (i:as) is
    'a' -> state3 err (i:as) is
    'b' -> state3 err (i:as) is
    _ -> err as iis

state4 :: LexerState
state4 err as [] = err as []
  where err _ _ = output Reservedid as (start1 [])
state4 err as iis@(i:is) =
  case i of
    'c' -> state2 err (i:as) is
    'b' -> state3 err (i:as) is
    'a' -> state4 err (i:as) is
    _ -> err as iis
  where err _ _ = output Reservedid as (start1 iis)

state5 :: LexerState
state5 err as is = output Special as (start1 is)
```