JFP **13** (1): 7–16, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803000418 Printed in the United Kingdom

Chapter 2

Lexical Structure

In this chapter, we describe the low-level lexical structure of Haskell. Most of the details may be skipped in a first reading.

2.1 Notational Conventions

These notational conventions are used for presenting syntax:

$[pattern]$	optional
$\{pattern\}$	zero or more repetitions
$(pattern)$	grouping
$pat_1 \mid pat_2$	choice
$pat_{\langle pat' \rangle}$	difference – elements generated by pat except those generated by pat'
`fibonacci`	terminal syntax in typewriter font

Because the syntax in this section describes *lexical* syntax, all whitespace is expressed explicitly; there is no implicit space between juxtaposed symbols. BNF-like syntax is used throughout, with productions having the form:

$$nonterm \quad \rightarrow \quad alt_1 \mid alt_2 \mid \ldots \mid alt_n$$

Care must be taken in distinguishing metalogical syntax such as $\mid$ and $[\ldots]$ from concrete terminal syntax (given in typewriter font) such as `|` and `[...]`, although usually the context makes the distinction clear.

Haskell uses the Unicode [15] character set. However, source programs are currently biased toward the ASCII character set used in earlier versions of Haskell.

This syntax depends upon properties of the Unicode characters as defined by the Unicode consortium. Haskell compilers are expected to make use of new versions of Unicode as they are made available.

2.2 Lexical Program Structure

$\mathit{program}$	$\to$	$\{\ \mathit{lexeme} \mid \mathit{whitespace}\ \}$
lexeme	$\to$	$\mathit{qvarid} \mid \mathit{qconid} \mid \mathit{qvarsym} \mid \mathit{qconsym}$
	$\mid$	$\mathit{literal} \mid \mathit{special} \mid \mathit{reservedop} \mid \mathit{reservedid}$
$\mathit{literal}$	$\to$	$\mathit{integer} \mid \mathit{float} \mid \mathit{char} \mid \mathit{string}$
$\mathit{special}$	$\to$	$\texttt{(} \mid \texttt{)} \mid \texttt{,} \mid \texttt{;} \mid \texttt{[} \mid \texttt{]} \mid \texttt{`} \mid \texttt{\{} \mid \texttt{\}}$
$\mathit{whitespace}$	$\to$	$\mathit{whitestuff}\ \{\mathit{whitestuff}\}$
$\mathit{whitestuff}$	$\to$	$\mathit{whitechar} \mid \mathit{comment} \mid \mathit{ncomment}$
$\mathit{whitechar}$	$\to$	$\mathit{newline} \mid \mathit{vertab} \mid \mathit{space} \mid \mathit{tab} \mid \mathit{uniWhite}$
$\mathit{newline}$	$\to$	$\mathit{return}\ \mathit{linefeed} \mid \mathit{return} \mid \mathit{linefeed} \mid \mathit{formfeed}$
return	$\to$	a carriage return
$\mathit{linefeed}$	$\to$	a line feed
vertab	$\to$	a vertical tab
$\mathit{formfeed}$	$\to$	a form feed
space	$\to$	a space
tab	$\to$	a horizontal tab
$\mathit{uniWhite}$	$\to$	any Unicode character defined as whitespace
$\mathit{comment}$	$\to$	$\mathit{dashes}\ [\ \mathit{any}_{\langle \mathit{symbol} \rangle}\ \{\mathit{any}\}\ ]\ \mathit{newline}$
dashes	$\to$	$\texttt{--}\ \{\texttt{-}\}$
$\mathit{opencom}$	$\to$	$\texttt{\{-}$
$\mathit{closecom}$	$\to$	$\texttt{-\}}$
$\mathit{ncomment}$	$\to$	$\mathit{opencom}\ \mathit{ANYseq}\ \{\mathit{ncomment}\ \mathit{ANYseq}\}\ \mathit{closecom}$
ANYseq	$\to$	$\{\mathit{ANY}\}_{\langle \{\mathit{ANY}\}\ (\ \mathit{opencom} \mid \mathit{closecom}\ )\ \{\mathit{ANY}\} \rangle}$
ANY	$\to$	$\mathit{graphic} \mid \mathit{whitechar}$
any	$\to$	$\mathit{graphic} \mid \mathit{space} \mid \mathit{tab}$
$\mathit{graphic}$	$\to$	$\mathit{small} \mid \mathit{large} \mid \mathit{symbol} \mid \mathit{digit} \mid \mathit{special} \mid \texttt{:} \mid \texttt{"} \mid \texttt{'}$
small	$\to$	$\mathit{ascSmall} \mid \mathit{uniSmall} \mid \texttt{\_}$
$\mathit{ascSmall}$	$\to$	$\texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z}$

uniSmall	→	any Unicode lowercase letter
large	→	*ascLarge* | *uniLarge*
ascLarge	→	`A` | `B` | ... | `Z`
uniLarge	→	any uppercase or titlecase Unicode letter
symbol	→	*ascSymbol* | *uniSymbol*$_{\langle special \mid \_ \mid : \mid " \mid ' \rangle}$
ascSymbol	→	`!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@`
	|	`\` | `^` | `|` | `-` | `~`
uniSymbol	→	any Unicode symbol or punctuation
digit	→	*ascDigit* | *uniDigit*
ascDigit	→	`0` | `1` | ... | `9`
uniDigit	→	any Unicode decimal digit
octit	→	`0` | `1` | ... | `7`
hexit	→	*digit* | `A` | ... | `F` | `a` | ... | `f`

Lexical analysis should use the "maximal munch" rule: at each point, the longest possible lexeme satisfying the *lexeme* production is read. So, although `case` is a reserved word, `cases` is not. Similarly, although = is reserved, == and ~= are not.

Any kind of *whitespace* is also a proper delimiter for lexemes.

Characters not in the category *ANY* are not valid in Haskell programs and should result in a lexing error.

2.3 Comments

Comments are valid whitespace.

An ordinary comment begins with a sequence of two or more consecutive dashes (e.g. `--`) and extends to the following newline. *The sequence of dashes must not form part of a legal lexeme.* For example, "`-->`" or "`|--`" do *not* begin a comment, because both of these are legal lexemes; however "`--foo`" does start a comment.

A nested comment begins with "`{-`" and ends with "`-}`". No legal lexeme starts with "`{-`"; hence, for example, "`{---`" starts a nested comment despite the trailing dashes.

The comment itself is not lexically analysed. Instead, the first unmatched occurrence of the string "`-}`" terminates the nested comment. Nested comments may be nested to any depth: any occurrence of the string "`{-`" within the nested comment starts a new nested comment, terminated by "`-}`". Within a nested comment, each "`{-`" is matched by a corresponding occurrence of "`-}`".

In an ordinary comment, the character sequences "`{-`" and "`-}`" have no special significance, and, in a nested comment, a sequence of dashes has no special significance.

Nested comments are also used for compiler pragmas, as explained in Chapter 11.

If some code is commented out using a nested comment, then any occurrence of `{-` or `-}` within a string or within an end-of-line comment in that code will interfere with the nested comments.

2.4 Identifiers and Operators

$$\begin{array}{lcl} \mathit{varid} & \rightarrow & (\mathit{small}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid \texttt{'}\ \})_{\langle \mathit{reservedid}\rangle} \\ \mathit{conid} & \rightarrow & \mathit{large}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid \texttt{'}\ \} \\ \mathit{reservedid} & \rightarrow & \texttt{case} \mid \texttt{class} \mid \texttt{data} \mid \texttt{default} \mid \texttt{deriving} \mid \texttt{do} \mid \texttt{else} \\ & \mid & \texttt{if} \mid \texttt{import} \mid \texttt{in} \mid \texttt{infix} \mid \texttt{infixl} \mid \texttt{infixr} \mid \texttt{instance} \\ & \mid & \texttt{let} \mid \texttt{module} \mid \texttt{newtype} \mid \texttt{of} \mid \texttt{then} \mid \texttt{type} \mid \texttt{where} \mid \texttt{\_} \end{array}$$

An identifier consists of a letter followed by zero or more letters, digits, underscores, and single quotes. Identifiers are lexically distinguished into two namespaces (Section 1.4): those that begin with a lower-case letter (variable identifiers) and those that begin with an upper-case letter (constructor identifiers). Identifiers are case sensitive: `name`, `naMe`, and `Name` are three distinct identifiers (the first two are variable identifiers, the last is a constructor identifier).

Underscore, "`_`", is treated as a lower-case letter, and can occur wherever a lower-case letter can. However, "`_`" all by itself is a reserved identifier, used as wild card in patterns. Compilers that offer warnings for unused identifiers are encouraged to suppress such warnings for identifiers beginning with underscore. This allows programmers to use "`_foo`" for a parameter that they expect to be unused.

$$\begin{array}{lcl} \mathit{varsym} & \rightarrow & (\ \mathit{symbol}\ \{\mathit{symbol} \mid \texttt{:}\}\ )_{\langle \mathit{reservedop} \mid \mathit{dashes}\rangle} \\ \mathit{consym} & \rightarrow & (\texttt{:}\ \{\mathit{symbol} \mid \texttt{:}\})_{\langle \mathit{reservedop}\rangle} \\ \mathit{reservedop} & \rightarrow & \texttt{..} \mid \texttt{:} \mid \texttt{::} \mid \texttt{=} \mid \texttt{\textbackslash} \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} \mid \texttt{@} \mid \texttt{\~{}} \mid \texttt{=>} \end{array}$$

Operator symbols are formed from one or more symbol characters, as defined above, and are lexically distinguished into two namespaces (Section 1.4):

- An operator symbol starting with a colon is a constructor.
- An operator symbol starting with any other character is an ordinary identifier.

Notice that a colon by itself, "`:`", is reserved solely for use as the Haskell list constructor; this makes its treatment uniform with other parts of list syntax, such as "`[]`" and "`[a,b]`".

Other than the special syntax for prefix negation, all operators are infix, although each infix operator can be used in a *section* to yield partially applied operators (see Section 3.5). All of the standard infix operators are just predefined symbols and may be rebound.

In the remainder of the report six different kinds of names will be used:

$varid$			(variables)
$conid$			(constructors)
$tyvar$	$\rightarrow$	$varid$	(type variables)
$tycon$	$\rightarrow$	$conid$	(type constructors)
$tycls$	$\rightarrow$	$conid$	(type classes)
$modid$	$\rightarrow$	$conid$	(modules)

Variables and type variables are represented by identifiers beginning with small letters, and the other four by identifiers beginning with capitals; also, variables and constructors have infix forms, the other four do not. Namespaces are also discussed in Section 1.4.

A name may optionally be *qualified* in certain circumstances by prepending them with a module identifier. This applies to variable, constructor, type constructor and type class names, but not type variables or module names. Qualified names are discussed in detail in Chapter 5.

$qvarid$	$\rightarrow$	$[modid$ `.`$]\ varid$
$qconid$	$\rightarrow$	$[modid$ `.`$]\ conid$
$qtycon$	$\rightarrow$	$[modid$ `.`$]\ tycon$
$qtycls$	$\rightarrow$	$[modid$ `.`$]\ tycls$
$qvarsym$	$\rightarrow$	$[modid$ `.`$]\ varsym$
$qconsym$	$\rightarrow$	$[modid$ `.`$]\ consym$

Since a qualified name is a lexeme, no spaces are allowed between the qualifier and the name. Sample lexical analyses are shown below.

This	Lexes as this	
`f.g`	`f . g`	(three tokens)
`F.g`	`F.g`	(qualified 'g')
`f..`	`f ..`	(two tokens)
`F..`	`F..`	(qualified '.')
`F.`	`F .`	(two tokens)

The qualifier does not change the syntactic treatment of a name; for example, `Prelude.+` is an infix operator with the same fixity as the definition of + in the Prelude (Section 4.4.2).

2.5 Numeric Literals

$decimal$	$\rightarrow$	$digit\{digit\}$
$octal$	$\rightarrow$	$octit\{octit\}$
$hexadecimal$	$\rightarrow$	$hexit\{hexit\}$

$$\begin{array}{lcl}
\mathit{integer} & \rightarrow & \mathit{decimal} \\
 & | & \texttt{0o}\ \mathit{octal} \mid \texttt{0O}\ \mathit{octal} \\
 & | & \texttt{0x}\ \mathit{hexadecimal} \mid \texttt{0X}\ \mathit{hexadecimal}
\end{array}$$

$$\begin{array}{lcl}
\mathit{float} & \rightarrow & \mathit{decimal}\ \texttt{.}\ \mathit{decimal}\ [\mathit{exponent}] \\
 & | & \mathit{decimal}\ \mathit{exponent} \\
\\
\mathit{exponent} & \rightarrow & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{decimal}
\end{array}$$

There are two distinct kinds of numeric literals: integer and floating. Integer literals may be given in decimal (the default), octal (prefixed by `0o` or `0O`) or hexadecimal notation (prefixed by `0x` or `0X`). Floating literals are always decimal. A floating literal must contain digits both before and after the decimal point; this ensures that a decimal point cannot be mistaken for another use of the dot character. Negative numeric literals are discussed in Section 3.4. The typing of numeric literals is discussed in Section 6.4.1.

2.6 Character and String Literals

$$\begin{array}{lcl}
\mathit{char} & \rightarrow & \texttt{'}\ (\mathit{graphic}_{\langle \texttt{'} \mid \texttt{\char`\\} \rangle} \mid \mathit{space} \mid \mathit{escape}_{\langle \texttt{\char`\\\&} \rangle})\ \texttt{'} \\
\mathit{string} & \rightarrow & \texttt{"}\ \{\mathit{graphic}_{\langle \texttt{"} \mid \texttt{\char`\\} \rangle} \mid \mathit{space} \mid \mathit{escape} \mid \mathit{gap}\}\ \texttt{"} \\
\mathit{escape} & \rightarrow & \texttt{\char`\\}\ (\ \mathit{charesc} \mid \mathit{ascii} \mid \mathit{decimal} \mid \texttt{o}\ \mathit{octal} \mid \texttt{x}\ \mathit{hexadecimal}\ ) \\
\mathit{charesc} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t} \mid \texttt{v} \mid \texttt{\char`\\} \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\mathit{ascii} & \rightarrow & \texttt{\char`\^}\mathit{cntrl} \mid \texttt{NUL} \mid \texttt{SOH} \mid \texttt{STX} \mid \texttt{ETX} \mid \texttt{EOT} \mid \texttt{ENQ} \mid \texttt{ACK} \\
 & | & \texttt{BEL} \mid \texttt{BS} \mid \texttt{HT} \mid \texttt{LF} \mid \texttt{VT} \mid \texttt{FF} \mid \texttt{CR} \mid \texttt{SO} \mid \texttt{SI} \mid \texttt{DLE} \\
 & | & \texttt{DC1} \mid \texttt{DC2} \mid \texttt{DC3} \mid \texttt{DC4} \mid \texttt{NAK} \mid \texttt{SYN} \mid \texttt{ETB} \mid \texttt{CAN} \\
 & | & \texttt{EM} \mid \texttt{SUB} \mid \texttt{ESC} \mid \texttt{FS} \mid \texttt{GS} \mid \texttt{RS} \mid \texttt{US} \mid \texttt{SP} \mid \texttt{DEL} \\
\mathit{cntrl} & \rightarrow & \mathit{ascLarge} \mid @ \mid \texttt{[} \mid \texttt{\char`\\} \mid \texttt{]} \mid \texttt{\char`\^} \mid \texttt{\_} \\
\mathit{gap} & \rightarrow & \texttt{\char`\\}\ \mathit{whitechar}\ \{\mathit{whitechar}\}\ \texttt{\char`\\}
\end{array}$$

Character literals are written between single quotes, as in `'a'`, and strings between double quotes, as in `"Hello"`.

Escape codes may be used in characters and strings to represent special characters. Note that a single quote `'` may be used in a string, but must be escaped in a character; similarly, a double quote `"` may be used in a character, but must be escaped in a string. `\` must always be escaped. The category *charesc* also includes portable representations for the characters "alert" (`\a`), "backspace" (`\b`), "form feed" (`\f`), "new line" (`\n`), "carriage return" (`\r`), "horizontal tab" (`\t`), and "vertical tab" (`\v`).

Escape characters for the Unicode character set, including control characters such as `\^X`, are also provided. Numeric escapes such as `\137` are used to designate the character with decimal representation 137; octal (e.g. `\o137`) and hexadecimal (e.g. `\x37`) representations are also allowed.

Consistent with the "maximal munch" rule, numeric escape characters in strings consist of all consecutive digits and may be of arbitrary length. Similarly, the one ambiguous ASCII escape code, `"\SOH"`, is parsed as a string of length 1. The escape character `\&` is provided as a "null character" to allow strings such as `"\137\&9"` and `"\SO\&H"` to be constructed (both of length two). Thus `"\&"` is equivalent to `""` and the character `'\&'` is disallowed. Further equivalences of characters are defined in Section 6.1.2.

A string may include a "gap" – two backslants enclosing white characters – which is ignored. This allows one to write long strings on more than one line by writing a backslant at the end of one line and at the start of the next. For example,

```
"Here is a backslant \\ as well as \137, \
    \a numeric escape character, and \^X, a control character."
```

String literals are actually abbreviations for lists of characters (see Section 3.7).

2.7 Layout

Haskell permits the omission of the braces and semicolons used in several grammar productions, by using *layout* to convey the same information. This allows both layout-sensitive and layout-insensitive styles of coding, which can be freely mixed within one program. Because layout is not required, Haskell programs can be straightforwardly produced by other programs.

The effect of layout on the meaning of a Haskell program can be completely specified by adding braces and semicolons in places determined by the layout. The meaning of this augmented program is now layout insensitive.

Informally stated, the braces and semicolons are inserted as follows. The layout (or "off-side") rule takes effect whenever the open brace is omitted after the keyword `where`, `let`, `do`, or `of`. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments). For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted). If the indentation of the non-brace lexeme immediately following a `where`, `let`, `do` or `of` is less than or equal to the current indentation level, then instead of starting a layout, an empty list "`{}`" is inserted, and layout processing occurs for the current level (i.e. insert a semicolon or close brace). A close brace is also inserted whenever the syntactic category containing the layout list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted. The layout rule matches only those open braces that it has inserted; an explicit open brace must be matched by an explicit close brace. Within these explicit open braces, *no* layout processing is performed for constructs outside the braces, even if a line is indented to the left of an earlier implicit open brace.

Section 9.3 gives a more precise definition of the layout rules.

Given these rules, a single newline may actually terminate several layout lists. Also, these rules permit:

```
f x = let a = 1; b = 2
          g y = exp2
       in exp1
```

making `a`, `b` and `g` all part of the same layout list.

As an example, Figure 2.1 shows a (somewhat contrived) module and Figure 2.2 shows the result of applying the layout rule to it. Note in particular: (a) the line beginning `}};pop`, where the termination of the previous line invokes three applications of the layout rule, corresponding to the depth (3) of the nested `where` clauses, (b) the close braces in the `where` clause nested within the tuple and `case` expression, inserted because the end of the tuple was detected, and (c) the close brace at the very end, inserted because of the column 0 indentation of the end-of-file token.

```
module AStack( Stack, push, pop, top, size ) where
data Stack a = Empty
             | MkStack a (Stack a)

push :: a -> Stack a -> Stack a
push x s = MkStack x s

size :: Stack a -> Int
size s = length (stkToLst s)  where
           stkToLst  Empty         = []
           stkToLst (MkStack x s)  = x:xs where xs = stkToLst s

pop :: Stack a -> (a, Stack a)
pop (MkStack x s)
  = (x, case s of r -> i r where i x = x)
                             -- (pop Empty) is an error

top :: Stack a -> a
top (MkStack x s) = x        -- (top Empty) is an error
```

Figure 2.1: A sample program

```
module AStack( Stack, push, pop, top, size ) where
{data Stack a = Empty
             | MkStack a (Stack a)

;push :: a -> Stack a -> Stack a
;push x s = MkStack x s

;size :: Stack a -> Int
;size s = length (stkToLst s)  where
           {stkToLst  Empty         = []
           ;stkToLst (MkStack x s)  = x:xs where {xs = stkToLst s

}};pop :: Stack a -> (a, Stack a)
;pop (MkStack x s)
  = (x, case s of {r -> i r where {i x = x}})
                             -- (pop Empty) is an error

;top :: Stack a -> a
;top (MkStack x s) = x        -- (top Empty) is an error
}
```

Figure 2.2: Sample program with layout expanded