JFP **13** (1): 125–138, January 2003.
DOI: 10.1017/S0956796803001114 Printed in the United Kingdom

Chapter 9

Syntax Reference

This chapter summarises the syntax of Haskell 98.

9.1 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

BNF-like syntax is used throughout, with productions having the form:

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

There are some families of nonterminals indexed by precedence levels (written as a superscript). Similarly, the nonterminals op, $varop$, and $conop$ may have a double index: a letter l, r, or n for left-, right- or nonassociativity and a precedence level. A precedence-level variable i ranges from 0 to 9; an associativity variable a varies over $\{l,\ r,\ n\}$. Thus, for example

$$aexp \rightarrow (\, exp^{i+1} \; qop^{(a,i)} \,)$$

actually stands for 30 productions, with 10 substitutions for i and 3 for a.

In both the lexical and the context-free syntax, there are some ambiguities that are to be resolved by making grammatical phrases as long as possible, proceeding from left to right (in shift-reduce parsing, resolving shift/reduce conflicts by shifting). In the lexical syntax, this is the "maximal munch" rule. In the context-free syntax, this means that conditionals, let-expressions, and lambda abstractions extend to the right as far as possible.

9.2 Lexical Syntax

program → { *lexeme* | *whitespace* }
lexeme → *qvarid* | *qconid* | *qvarsym* | *qconsym*
  | *literal* | *special* | *reservedop* | *reservedid*
literal → *integer* | *float* | *char* | *string*
special → (|) | , | ; | [|] | ` | { | }

whitespace → *whitestuff* {*whitestuff*}
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *vertab* | *space* | *tab* | *uniWhite*
newline → *return linefeed* | *return* | *linefeed* | *formfeed*
return → a carriage return
linefeed → a line feed
vertab → a vertical tab
formfeed → a form feed
space → a space
tab → a horizontal tab
uniWhite → any Unicode character defined as whitespace

comment → *dashes* [$any_{\langle symbol \rangle}$ {*any*}] *newline*
dashes → -- {-}
opencom → {-
closecom → -}
ncomment → *opencom ANYseq* {*ncomment ANYseq*} *closecom*
ANYseq → $\{ANY\}_{\langle \{ANY\} \; (\, opencom \; | \; closecom \,) \; \{ANY\} \rangle}$
ANY → *graphic* | *whitechar*
any → *graphic* | *space* | *tab*
graphic → *small* | *large* | *symbol* | *digit* | *special* | : | " | '

small → *ascSmall* | *uniSmall* | _
ascSmall → a | b | ... | z

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

uniSmall → any Unicode lowercase letter

large → *ascLarge* | *uniLarge*
ascLarge → `A` | `B` | ... | `Z`
uniLarge → any uppercase or titlecase Unicode letter
symbol → *ascSymbol* | *uniSymbol*⟨*special* | `_` | `:` | `"` | `'`⟩

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`

varid → (*small* {*small* | *large* | *digit* | `'` })⟨*reservedid*⟩
conid → *large* {*small* | *large* | *digit* | `'` }
reservedid → `case` | `class` | `data` | `default` | `deriving` | `do` | `else`
| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance`
| `let` | `module` | `newtype` | `of` | `then` | `type` | `where` | `_`

varsym → (*symbol* {*symbol* | `:`})⟨*reservedop* | *dashes*⟩
consym → (`:` {*symbol* | `:`})⟨*reservedop*⟩
reservedop → `..` | `:` | `::` | `=` | `\` | `|` | `<-` | `->` | `@` | `~` | `=>`

varid (variables)
conid (constructors)
tyvar → *varid* (type variables)
tycon → *conid* (type constructors)
tycls → *conid* (type classes)
modid → *conid* (modules)

qvarid → [*modid* `.`] *varid*
qconid → [*modid* `.`] *conid*
qtycon → [*modid* `.`] *tycon*
qtycls → [*modid* `.`] *tycls*
qvarsym → [*modid* `.`] *varsym*
qconsym → [*modid* `.`] *consym*

decimal → *digit*{*digit*}
octal → *octit*{*octit*}
hexadecimal → *hexit*{*hexit*}

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

$$
\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} \\
\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} \\
 & & \\
\mathit{char} & \rightarrow & \texttt{'}\ (\mathit{graphic}_{\langle \texttt{'} \mid \texttt{\textbackslash} \rangle} \mid \mathit{space} \mid \mathit{escape}_{\langle \texttt{\textbackslash\&} \rangle})\ \texttt{'} \\
\mathit{string} & \rightarrow & \texttt{"}\ \{\mathit{graphic}_{\langle \texttt{"} \mid \texttt{\textbackslash} \rangle} \mid \mathit{space} \mid \mathit{escape} \mid \mathit{gap}\}\ \texttt{"} \\
\mathit{escape} & \rightarrow & \texttt{\textbackslash}\ (\ \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{\textbackslash} \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\mathit{ascii} & \rightarrow & \texttt{\^{}}\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{\textbackslash} \mid \texttt{]} \mid \texttt{\^{}} \mid \texttt{\_} \\
\mathit{gap} & \rightarrow & \texttt{\textbackslash}\ \mathit{whitechar}\ \{\mathit{whitechar}\}\ \texttt{\textbackslash}
\end{array}
$$

9.3 Layout

Section 2.7 gives an informal discussion of the layout rule. This section defines it more precisely.

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

The effect of layout is specified in this section by describing how to add braces and semicolons to a laid-out program. The specification takes the form of a function L that performs the translation. The input to L is:

- A stream of lexemes as specified by the lexical syntax in the Haskell report, with the following additional tokens:
  - 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.
  - 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.
  - Where the start of a lexeme is preceded only by white space on the same line, this lexeme is preceded by $< n >$ where n is the indentation of the lexeme, provided that it is not, as a consequence of the first two rules, preceded by $\{n\}$. (NB: a string literal may span multiple lines – Section 2.6. So in the fragment

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

```
f = ("Hello \
        \Bill", "Jake")
```

There is no $< n >$ inserted before the `\Bill`, because it is not the beginning of a complete lexeme; nor before the `,`, because it is not preceded only by white space.)

- A stack of "layout contexts", in which each element is either:
  - Zero, indicating that the enclosing context is explicit (i.e. the programmer supplied the opening brace. If the innermost context is 0, then no layout tokens will be inserted until either the enclosing context ends or a new context is pushed.
  - A positive integer, which is the indentation column of the enclosing layout context.

The "indentation" of a lexeme is the column number of the first character of that lexeme; the indentation of a line is the indentation of its leftmost lexeme. To determine the column number, assume a fixed-width font with the following conventions:

- The characters $newline$, $return$, $linefeed$, and $formfeed$, all start a new line.
- The first column is designated column 1, not 0.
- 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.

For the purposes of the layout rule, Unicode characters in a source program are considered to be of the same, fixed, width as an ASCII character. However, to avoid visual confusion, programmers should avoid writing programs in which the meaning of implicit layout depends on the width of non-space characters.

The application

$$L\ tokens\ []$$

delivers a layout-insensitive translation of $tokens$, where $tokens$ is the result of lexically analysing a module and adding column-number indicators to it as described above. The definition of L is as follows, where we use ":" as a stream construction operator, and "[ ]" for the empty stream.

$$
\begin{array}{lcll}
L\ (< n >: ts)\ (m : ms) & = & \texttt{;}\ :\ (L\ ts\ (m : ms)) & \text{if } m = n \\
 & = & \texttt{\}}\ :\ (L\ (< n >: ts)\ ms) & \text{if } n < m \\
L\ (< n >: ts)\ ms & = & L\ ts\ ms & \\
 & & & \\
L\ (\{n\} : ts)\ (m : ms) & = & \texttt{\{}\ :\ (L\ ts\ (n : m : ms)) & \text{if } n > m\ (Note\ 1) \\
L\ (\{n\} : ts)\ [] & = & \texttt{\{}\ :\ (L\ ts\ [n]) & \text{if } n > 0\ (Note\ 1) \\
L\ (\{n\} : ts)\ ms & = & \texttt{\{}\ :\ \texttt{\}}\ :\ (L\ (< n >: ts)\ ms) & (Note\ 2) \\
 & & & \\
L\ (\texttt{\}} : ts)\ (0 : ms) & = & \texttt{\}}\ :\ (L\ ts\ ms) & (Note\ 3) \\
L\ (\texttt{\}} : ts)\ ms & = & \text{parse-error} & (Note\ 3) \\
 & & & \\
L\ (\texttt{\{} : ts)\ ms & = & \texttt{\{}\ :\ (L\ ts\ (0 : ms)) & (Note\ 4) \\
 & & & \\
L\ (t : ts)\ (m : ms) & = & \texttt{\}}\ :\ (L\ (t : ts)\ ms) & \text{if } m/ = 0 \text{ and parse-error}(t) \\
 & & & (Note\ 5) \\
L\ (t : ts)\ ms & = & t\ :\ (L\ ts\ ms) & \\
 & & & \\
L\ []\ [] & = & [] & \\
L\ []\ (m : ms) & = & \texttt{\}}\ :\ L\ []\ ms & \text{if } m \neq 0\ (Note\ 6)
\end{array}
$$

Note 1. A nested context must be further indented than the enclosing context ($n > m$). If not, L fails, and the compiler should indicate a layout error. An example is:

```
f x = let
         h y = let
  p z = z
               in p
      in h
```

Here, the definition of `p` is indented less than the indentation of the enclosing context, which is set in this case by the definition of `h`.

Note 2. If the first token after a `where` (say) is not indented more than the enclosing layout context, then the block must be empty, so empty braces are inserted. The $\{n\}$ token is replaced by $< n >$, to mimic the situation if the empty braces had been explicit.

Note 3. By matching against 0 for the current layout context, we ensure that an explicit close brace can only match an explicit open brace. A parse error results if an explicit close brace matches an implicit open brace.

Note 4. This clause means that all brace pairs are treated as explicit layout contexts, including labelled construction and update (Section 3.15). This is a difference between this formulation and Haskell 1.4.

Note 5. The side condition parse-error(t) is to be interpreted as follows: if the tokens generated so far by L together with the next token t represent an invalid prefix of the Haskell grammar,

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

and the tokens generated so far by L followed by the token "}" represent a valid prefix of the Haskell grammar, then parse-error(t) is true.

The test $m/ = 0$ checks that an implicitly-added closing brace would match an implicit open brace.

Note 6. At the end of the input, any pending close-braces are inserted. It is an error at this point to be within a non-layout context (i.e. $m = 0$).

If none of the rules given above matches, then the algorithm fails. It can fail for instance when the end of the input is reached, and a non-layout context is active, since the close brace is missing. Some error conditions are not detected by the algorithm, although they could be: for example `let }`.

Note 1 implements the feature that layout processing can be stopped prematurely by a parse error. For example

```
let x = e; y = x in e′
```

is valid, because it translates to

```
let { x = e; y = x } in e′
```

The close brace is inserted due to the parse error rule above. The parse-error rule is hard to implement in its full generality, because doing so involves fixities. For example, the expression

```
do a == b == c
```

has a single unambiguous (albeit probably type-incorrect) parse, namely

```
(do { a == b }) == c
```

because `(==)` is non-associative. Programmers are therefore advised to avoid writing code that requires the parser to insert a closing brace in such situations.

9.4 Literate Comments

The "literate comment" convention, first developed by Richard Bird and Philip Wadler for Orwell, and inspired in turn by Donald Knuth's "literate programming", is an alternative style for encoding Haskell source code. The literate style encourages comments by making them the default. A line in which ">" is the first character is treated as part of the program; all other lines are comment.

The program text is recovered by taking only those lines beginning with ">", and replacing the leading ">" with a space. Layout and comments apply exactly as described in Chapter 9 in the resulting text.

To capture some cases where one omits an ">" by mistake, it is an error for a program line to appear adjacent to a non-blank comment line, where a line is taken as blank if it consists only of whitespace.

By convention, the style of comment is indicated by the file extension, with "`.hs`" indicating a usual Haskell file and "`.lhs`" indicating a literate Haskell file. Using this style, a simple factorial program would be:

```
     This literate program prompts the user for a number
     and prints the factorial of that number:

> main :: IO ()

> main = do putStr "Enter a number: "
>           l <- readLine
>           putStr "n!= "
>           print (fact (read l))

  This is the factorial function.

> fact :: Integer -> Integer
> fact 0 = 1
> fact n = n * fact (n-1)
```

An alternative style of literate programming is particularly suitable for use with the LaTeX text processing system. In this convention, only those parts of the literate program that are entirely enclosed between `\begin{code}`...`\end{code}` delimiters are treated as program text; all other lines are comment. More precisely:

- Program code begins on the first line following a line that begins `\begin{code}`.
- Program code ends just before a subsequent line that begins `\end{code}` (ignoring string literals, of course).

It is not necessary to insert additional blank lines before or after these delimiters, though it may be stylistically desirable. For example,

```
\documentstyle{article}

\begin{document}

\section{Introduction}

This is a trivial program that prints the first 20 factorials.

\begin{code}
main :: IO ()
main =  print [ (n, product [1..n]) | n <- [1..20]]
\end{code}

\end{document}
```

This style uses the same file extension. It is not advisable to mix these two styles in the same file.

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

9.5 Context-Free Syntax

module	$\rightarrow$	`module` *modid* [*exports*] `where` *body*	
	$\vert$	*body*	
body	$\rightarrow$	`{` *impdecls* `;` *topdecls* `}`	
	$\vert$	`{` *impdecls* `}`	
	$\vert$	`{` *topdecls* `}`	
impdecls	$\rightarrow$	$impdecl_1$ `;` ... `;` $impdecl_n$	$(n \geq 1)$
exports	$\rightarrow$	`(` $export_1$ `,` ... `,` $export_n$ [`,`] `)`	$(n \geq 0)$
export	$\rightarrow$	*qvar*	
	$\vert$	*qtycon* [`(..)` $\vert$ `(` $cname_1$ `,` ... `,` $cname_n$ `)`]	$(n \geq 0)$
	$\vert$	*qtycls* [`(..)` $\vert$ `(` $qvar_1$ `,` ... `,` $qvar_n$ `)`]	$(n \geq 0)$
	$\vert$	`module` *modid*	
impdecl	$\rightarrow$	`import` [`qualified`] *modid* [`as` *modid*] [*impspec*]	
	$\vert$		(empty declaration)
impspec	$\rightarrow$	`(` $import_1$ `,` ... `,` $import_n$ [`,`] `)`	$(n \geq 0)$
	$\vert$	`hiding` `(` $import_1$ `,` ... `,` $import_n$ [`,`] `)`	$(n \geq 0)$
import	$\rightarrow$	*var*	
	$\vert$	*tycon* [`(..)` $\vert$ `(` $cname_1$ `,` ... `,` $cname_n$ `)`]	$(n \geq 0)$
	$\vert$	*tycls* [`(..)` $\vert$ `(` var_1 `,` ... `,` var_n `)`]	$(n \geq 0)$
cname	$\rightarrow$	*var* $\vert$ *con*	
topdecls	$\rightarrow$	$topdecl_1$ `;` ... `;` $topdecl_n$	$(n \geq 0)$
topdecl	$\rightarrow$	`type` *simpletype* `=` *type*	
	$\vert$	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
	$\vert$	`newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]	
	$\vert$	`class` [*scontext* `=>`] *tycls* *tyvar* [`where` *cdecls*]	
	$\vert$	`instance` [*scontext* `=>`] *qtycls* *inst* [`where` *idecls*]	
	$\vert$	`default` `(` $type_1$ `,` ... `,` $type_n$ `)`	$(n \geq 0)$
	$\vert$	*decl*	
decls	$\rightarrow$	`{` $decl_1$ `;` ... `;` $decl_n$ `}`	$(n \geq 0)$
decl	$\rightarrow$	*gendecl*	

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

$\quad\quad \mid \quad (funlhs \mid pat^0)\ rhs$

$cdecls \rightarrow \texttt{\{}\ cdecl_1\ \texttt{;}\ \ldots\ \texttt{;}\ cdecl_n\ \texttt{\}}$ $(n \geq 0)$
$cdecl \rightarrow gendecl$
$\quad\quad \mid \quad (funlhs \mid var)\ rhs$

$idecls \rightarrow \texttt{\{}\ idecl_1\ \texttt{;}\ \ldots\ \texttt{;}\ idecl_n\ \texttt{\}}$ $(n \geq 0)$
$idecl \rightarrow (funlhs \mid var)\ rhs$
$\quad\quad \mid$ (empty)

$gendecl \rightarrow vars\ \texttt{::}\ [context\ \texttt{=>}]\ type$ (type signature)
$\quad\quad \mid \quad fixity\ [integer]\ ops$ (fixity declaration)
$\quad\quad \mid$ (empty declaration)

$ops \rightarrow op_1\ \texttt{,}\ \ldots\ \texttt{,}\ op_n$ $(n \geq 1)$
$vars \rightarrow var_1\ \texttt{,}\ \ldots\texttt{,}\ var_n$ $(n \geq 1)$
$fixity \rightarrow \texttt{infixl} \mid \texttt{infixr} \mid \texttt{infix}$

$type \rightarrow btype\ [\texttt{->}\ type]$ (function type)

$btype \rightarrow [btype]\ atype$ (type application)

$atype \rightarrow gtycon$
$\quad\quad \mid \quad tyvar$
$\quad\quad \mid \quad \texttt{(}\ type_1\ \texttt{,}\ \ldots\ \texttt{,}\ type_k\ \texttt{)}$ (tuple type, $k \geq 2$)
$\quad\quad \mid \quad \texttt{[}\ type\ \texttt{]}$ (list type)
$\quad\quad \mid \quad \texttt{(}\ type\ \texttt{)}$ (parenthesized constructor)

$gtycon \rightarrow qtycon$
$\quad\quad \mid \quad \texttt{()}$ (unit type)
$\quad\quad \mid \quad \texttt{[]}$ (list constructor)
$\quad\quad \mid \quad \texttt{(->)}$ (function constructor)
$\quad\quad \mid \quad \texttt{(,}\{\texttt{,}\}\texttt{)}$ (tupling constructors)

$context \rightarrow class$
$\quad\quad \mid \quad \texttt{(}\ class_1\ \texttt{,}\ \ldots\ \texttt{,}\ class_n\ \texttt{)}$ $(n \geq 0)$
$class \rightarrow qtycls\ tyvar$
$\quad\quad \mid \quad qtycls\ \texttt{(}\ tyvar\ atype_1\ \ldots\ atype_n\ \texttt{)}$ $(n \geq 1)$
$scontext \rightarrow simpleclass$
$\quad\quad \mid \quad \texttt{(}\ simpleclass_1\ \texttt{,}\ \ldots\ \texttt{,}\ simpleclass_n\ \texttt{)}$ $(n \geq 0)$
$simpleclass \rightarrow qtycls\ tyvar$

$simpletype \rightarrow tycon\ tyvar_1\ \ldots\ tyvar_k$ $(k \geq 0)$

constrs	→	$constr_1$ | ... | $constr_n$	$(n \geq 1)$
constr	→	*con* [`!`] $atype_1$ ... [`!`] $atype_k$	(arity *con* $= k,\ k \geq 0$)
	|	(*btype* | `!` *atype*) *conop* (*btype* | `!` *atype*)	(infix *conop*)
	|	*con* `{` $fielddecl_1$ `,` ... `,` $fielddecl_n$ `}`	$(n \geq 0)$
newconstr	→	*con atype*	
	|	*con* `{` *var* `::` *type* `}`	
fielddecl	→	*vars* `::` (*type* | `!` *atype*)	
deriving	→	`deriving` (*dclass* | `(`$dclass_1$ `,` ... `,` $dclass_n$`)`)	$(n \geq 0)$
dclass	→	*qtycls*	

inst	→	*gtycon*	
	|	`(` *gtycon* $tyvar_1$ ... $tyvar_k$ `)`	($k \geq 0$, *tyvars* distinct)
	|	`(` $tyvar_1$ `,` ... `,` $tyvar_k$ `)`	($k \geq 2$, *tyvars* distinct)
	|	`[` *tyvar* `]`	
	|	`(` $tyvar_1$ `->` $tyvar_2$ `)`	$tyvar_1$ and $tyvar_2$ distinct

funlhs	→	*var apat* { *apat* }
	|	$pat^{i+1}\ varop^{(a,i)}\ pat^{i+1}$
	|	$lpat^{i}\ varop^{(\mathrm{l},i)}\ pat^{i+1}$
	|	$pat^{i+1}\ varop^{(\mathrm{r},i)}\ rpat^{i}$
	|	`(` *funlhs* `)` *apat* { *apat* }
rhs	→	`=` *exp* [`where` *decls*]
	|	*gdrhs* [`where` *decls*]
gdrhs	→	*gd* `=` *exp* [*gdrhs*]
gd	→	`|` exp^{0}

exp	→	exp^{0} `::` [*context* `=>`] *type*	(expression type signature)
	|	exp^{0}	
exp^{i}	→	exp^{i+1} [$qop^{(\mathrm{n},i)}\ exp^{i+1}$]	
	|	$lexp^{i}$	
	|	$rexp^{i}$	
$lexp^{i}$	→	($lexp^{i}$ | exp^{i+1}) $qop^{(\mathrm{l},i)}\ exp^{i+1}$	
$lexp^{6}$	→	`-` exp^{7}	
$rexp^{i}$	→	$exp^{i+1}\ qop^{(\mathrm{r},i)}$ ($rexp^{i}$ | exp^{i+1})	
exp^{10}	→	`\` $apat_1$ ... $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)
	|	`let` *decls* `in` *exp*	(let expression)
	|	`if` *exp* `then` *exp* `else` *exp*	(conditional)
	|	`case` *exp* `of` `{` *alts* `}`	(case expression)

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

	$\vert$	`do` { $stmts$ }	(do expression)
	$\vert$	$fexp$	
$fexp$	$\rightarrow$	$[fexp]\ aexp$	(function application)
$aexp$	$\rightarrow$	$qvar$	(variable)
	$\vert$	$gcon$	(general constructor)
	$\vert$	$literal$	
	$\vert$	(exp)	(parenthesized expression)
	$\vert$	(exp_1 , ... , exp_k)	(tuple, $k \geq 2$)
	$\vert$	[exp_1 , ... , exp_k]	(list, $k \geq 1$)
	$\vert$	[exp_1 $[$, $exp_2]$ .. $[exp_3]$]	(arithmetic sequence)
	$\vert$	[exp $\vert$ $qual_1$, ... , $qual_n$]	(list comprehension, $n \geq 1$)
	$\vert$	(exp^{i+1} $qop^{(a,i)}$)	(left section)
	$\vert$	($lexp^{i}$ $qop^{(l,i)}$)	(left section)
	$\vert$	($qop^{(a,i)}_{\langle - \rangle}$ exp^{i+1})	(right section)
	$\vert$	($qop^{(r,i)}_{\langle - \rangle}$ $rexp^{i}$)	(right section)
	$\vert$	$qcon$ { $fbind_1$, ... , $fbind_n$ }	(labeled construction, $n \geq 0$)
	$\vert$	$aexp_{\langle qcon \rangle}$ { $fbind_1$, ... , $fbind_n$ }	(labeled update, $n \geq 1$)
$qual$	$\rightarrow$	pat `<-` exp	(generator)
	$\vert$	`let` $decls$	(local declaration)
	$\vert$	exp	(guard)
$alts$	$\rightarrow$	alt_1 ; ... ; alt_n	($n \geq 1$)
alt	$\rightarrow$	pat `->` exp $[$`where` $decls]$	
	$\vert$	pat $gdpat$ $[$`where` $decls]$	
	$\vert$		($empty\ alternative$)
$gdpat$	$\rightarrow$	gd `->` exp $[$ $gdpat$ $]$	
$stmts$	$\rightarrow$	$stmt_1$ ... $stmt_n$ exp $[$;$]$	($n \geq 0$)
$stmt$	$\rightarrow$	exp ;	
	$\vert$	pat `<-` exp ;	
	$\vert$	`let` $decls$;	
	$\vert$	;	($empty\ statement$)
$fbind$	$\rightarrow$	$qvar$ = exp	
pat	$\rightarrow$	var + $integer$	(successor pattern)

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press

	$\mid$	pat^{0}	
pat^{i}	$\rightarrow$	$pat^{i+1}\ [qconop^{(\mathrm{n},i)}\ pat^{i+1}]$	
	$\mid$	$lpat^{i}$	
	$\mid$	$rpat^{i}$	
$lpat^{i}$	$\rightarrow$	$(lpat^{i} \mid pat^{i+1})\ qconop^{(\mathrm{l},i)}\ pat^{i+1}$	
$lpat^{6}$	$\rightarrow$	$\texttt{-}\ (integer \mid float)$	(negative literal)
$rpat^{i}$	$\rightarrow$	$pat^{i+1}\ qconop^{(\mathrm{r},i)}\ (rpat^{i} \mid pat^{i+1})$	
pat^{10}	$\rightarrow$	$apat$	
	$\mid$	$gcon\ apat_{1}\ \ldots\ apat_{k}$	(arity $gcon = k,\ k \geq 1$)

$apat$	$\rightarrow$	$var\ [\ @\ apat]$	(as pattern)
	$\mid$	$gcon$	(arity $gcon = 0$)
	$\mid$	$qcon\ \texttt{\{}\ fpat_{1}\ \texttt{,}\ \ldots\ \texttt{,}\ fpat_{k}\ \texttt{\}}$	(labeled pattern, $k \geq 0$)
	$\mid$	$literal$	
	$\mid$	$\texttt{\_}$	(wildcard)
	$\mid$	$\texttt{(}\ pat\ \texttt{)}$	(parenthesized pattern)
	$\mid$	$\texttt{(}\ pat_{1}\ \texttt{,}\ \ldots\ \texttt{,}\ pat_{k}\ \texttt{)}$	(tuple pattern, $k \geq 2$)
	$\mid$	$\texttt{[}\ pat_{1}\ \texttt{,}\ \ldots\ \texttt{,}\ pat_{k}\ \texttt{]}$	(list pattern, $k \geq 1$)
	$\mid$	$\texttt{\~{}}\ apat$	(irrefutable pattern)
$fpat$	$\rightarrow$	$qvar\ \texttt{=}\ pat$	

$gcon$	$\rightarrow$	$\texttt{()}$	
	$\mid$	$\texttt{[]}$	
	$\mid$	$\texttt{(,}\{\texttt{,}\}\texttt{)}$	
	$\mid$	$qcon$	

var	$\rightarrow$	$varid \mid \texttt{(}\ varsym\ \texttt{)}$	(variable)
$qvar$	$\rightarrow$	$qvarid \mid \texttt{(}\ qvarsym\ \texttt{)}$	(qualified variable)
con	$\rightarrow$	$conid \mid \texttt{(}\ consym\ \texttt{)}$	(constructor)
$qcon$	$\rightarrow$	$qconid \mid \texttt{(}\ gconsym\ \texttt{)}$	(qualified constructor)
$varop$	$\rightarrow$	$varsym \mid \texttt{`}\ varid\ \texttt{`}$	(variable operator)
$qvarop$	$\rightarrow$	$qvarsym \mid \texttt{`}\ qvarid\ \texttt{`}$	(qualified variable operator)
$conop$	$\rightarrow$	$consym \mid \texttt{`}\ conid\ \texttt{`}$	(constructor operator)
$qconop$	$\rightarrow$	$gconsym \mid \texttt{`}\ qconid\ \texttt{`}$	(qualified constructor operator)
op	$\rightarrow$	$varop \mid conop$	(operator)
qop	$\rightarrow$	$qvarop \mid qconop$	(qualified operator)
$gconsym$	$\rightarrow$	$\texttt{:} \mid qconsym$	

https://doi.org/10.1017/S0956796803001114 Published online by Cambridge University Press