Chapter 10
Syntax Reference

10.1 Notational Conventions

These notational conventions are used for presenting syntax:

[*pattern*]	optional
{*pattern*}	zero or more repetitions
(*pattern*)	grouping
pat_1 | 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:

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

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.

10.2 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 ANY seq* {*ncomment ANY seq*} *closecom*
ANY seq	→	$\{ANY\}_{\langle \{ANY\}\ (\ opencom \mid closecom\ )\ \{ANY\} \rangle}$

ANY → *graphic* | *whitechar*
any → *graphic* | *space* | *tab*
graphic → *small* | *large* | *symbol* | *digit* | *special* | `"` | `'`

small → *ascSmall* | *uniSmall* | `_`
ascSmall → `a` | `b` | … | `z`
uniSmall → any Unicode lowercase letter

large → *ascLarge* | *uniLarge*
ascLarge → `A` | `B` | … | `Z`
uniLarge → any uppercase or titlecase Unicode letter
symbol → *ascSymbol* | $uniSymbol_{\langle special \mid \_ \mid " \mid ' \rangle}$

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`

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

varsym → $(\ symbol_{\langle \texttt{:} \rangle}\ \{symbol\}\ )_{\langle reservedop \mid dashes \rangle}$
consym → $(\ \texttt{:}\ \{symbol\})_{\langle reservedop \rangle}$
reservedop → `..` | `:` | `::` | `=` | `\` | `|` | `<-` | `->` | `@` | `~` | `=>`

varid (variables)
conid (constructors)
tyvar → *varid* (type variables)
tycon → *conid* (type constructors)
tycls → *conid* (type classes)
modid → {*conid* `.`} *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*}

integer → *decimal*
| `0o` *octal* | `0O` *octal*

	∣	`0x` *hexadecimal* ∣ `0X` *hexadecimal*
float	→	*decimal* `.` *decimal* [*exponent*]
	∣	*decimal exponent*
exponent	→	(`e` ∣ `E`) [`+` ∣ `-`] *decimal*
char	→	`'` ($graphic_{\langle ' \mid \backslash \rangle}$ ∣ *space* ∣ $escape_{\langle \backslash \& \rangle}$) `'`
string	→	`"` {$graphic_{\langle " \mid \backslash \rangle}$ ∣ *space* ∣ *escape* ∣ *gap*} `"`
escape	→	`\` (*charesc* ∣ *ascii* ∣ *decimal* ∣ `o` *octal* ∣ `x` *hexadecimal*)
charesc	→	`a` ∣ `b` ∣ `f` ∣ `n` ∣ `r` ∣ `t` ∣ `v` ∣ `\` ∣ `"` ∣ `'` ∣ `&`
ascii	→	`^`*cntrl* ∣ `NUL` ∣ `SOH` ∣ `STX` ∣ `ETX` ∣ `EOT` ∣ `ENQ` ∣ `ACK`
	∣	`BEL` ∣ `BS` ∣ `HT` ∣ `LF` ∣ `VT` ∣ `FF` ∣ `CR` ∣ `SO` ∣ `SI` ∣ `DLE`
	∣	`DC1` ∣ `DC2` ∣ `DC3` ∣ `DC4` ∣ `NAK` ∣ `SYN` ∣ `ETB` ∣ `CAN`
	∣	`EM` ∣ `SUB` ∣ `ESC` ∣ `FS` ∣ `GS` ∣ `RS` ∣ `US` ∣ `SP` ∣ `DEL`
cntrl	→	*ascLarge* ∣ `@` ∣ `[` ∣ `\` ∣ `]` ∣ `^` ∣ `_`
gap	→	`\` *whitechar* {*whitechar*} `\`

10.3 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

    ```
    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.

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

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, 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 $m \neq 0$ 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.

10.4 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 comments.

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 10 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 comments. 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}

\chapter{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.

10.5 Context-Free Syntax

module → `module` *modid* [*exports*] `where` *body*
| *body*
body → `{` *impdecls* `;` *topdecls* `}`
| `{` *impdecls* `}`
| `{` *topdecls* `}`

impdecls → *impdecl*$_1$ `;` … `;` *impdecl*$_n$ ($n \geq 1$)

exports → `(` *export*$_1$ `,` … `,` *export*$_n$ [`,`] `)` ($n \geq 0$)

export → *qvar*
| *qtycon* [`(..)` | `(` *cname*$_1$ `,` … `,` *cname*$_n$ `)`] ($n \geq 0$)
| *qtycls* [`(..)` | `(` *qvar*$_1$ `,` … `,` *qvar*$_n$ `)`] ($n \geq 0$)
| `module` *modid*

impdecl → `import` [`qualified`] *modid* [`as` *modid*] [*impspec*]
| (empty declaration)

impspec → `(` *import*$_1$ `,` … `,` *import*$_n$ [`,`] `)` ($n \geq 0$)
| `hiding` `(` *import*$_1$ `,` … `,` *import*$_n$ [`,`] `)` ($n \geq 0$)

import → *var*
| *tycon* [`(..)` | `(` *cname*$_1$ `,` … `,` *cname*$_n$ `)`] ($n \geq 0$)
| *tycls* [`(..)` | `(` *var*$_1$ `,` … `,` *var*$_n$ `)`] ($n \geq 0$)
cname → *var* | *con*

topdecls → *topdecl*$_1$ `;` … `;` *topdecl*$_n$ ($n \geq 0$)
topdecl → `type` *simpletype* `=` *type*
| `data` [*context* `=>`] *simpletype* [`=` *constrs*] [*deriving*]
| `newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]
| `class` [*scontext* `=>`] *tycls* *tyvar* [`where` *cdecls*]
| `instance` [*scontext* `=>`] *qtycls* *inst* [`where` *idecls*]
| `default` `(`*type*$_1$ `,` … `,` *type*$_n$`)` ($n \geq 0$)
| `foreign` *fdecl*
| *decl*

decls → `{` *decl*$_1$ `;` … `;` *decl*$_n$ `}` ($n \geq 0$)
decl → *gendecl*
| (*funlhs* | *pat*) *rhs*

cdecls → `{` *cdecl*$_1$ `;` … `;` *cdecl*$_n$ `}` ($n \geq 0$)
cdecl → *gendecl*
| (*funlhs* | *var*) *rhs*

idecls	→	`{` $idecl_1$ `;` … `;` $idecl_n$ `}`	($n \geq 0$)
idecl	→	(*funlhs* $\mid$ *var*) *rhs*	
	$\mid$		(empty)
gendecl	→	*vars* `::` [*context* `=>`] *type*	(type signature)
	$\mid$	*fixity* [*integer*] *ops*	(fixity declaration)
	$\mid$		(empty declaration)
ops	→	op_1 `,` … `,` op_n	($n \geq 1$)
vars	→	var_1 `,` …, var_n	($n \geq 1$)
fixity	→	`infixl` $\mid$ `infixr` $\mid$ `infix`	
type	→	*btype* [`->` *type*]	(function type)
btype	→	[*btype*] *atype*	(type application)
atype	→	*gtycon*	
	$\mid$	*tyvar*	
	$\mid$	`(` $type_1$ `,` … `,` $type_k$ `)`	(tuple type, $k \geq 2$)
	$\mid$	`[` *type* `]`	(list type)
	$\mid$	`(` *type* `)`	(parenthesized constructor)
gtycon	→	*qtycon*	
	$\mid$	`()`	(unit type)
	$\mid$	`[]`	(list constructor)
	$\mid$	`(->)`	(function constructor)
	$\mid$	`(,{,})`	(tupling constructors)
context	→	*class*	
	$\mid$	`(` $class_1$ `,` … `,` $class_n$ `)`	($n \geq 0$)
class	→	*qtycls tyvar*	
	$\mid$	*qtycls* `(` *tyvar* $atype_1$ … $atype_n$ `)`	($n \geq 1$)
scontext	→	*simpleclass*	
	$\mid$	`(` $simpleclass_1$ `,` … `,` $simpleclass_n$ `)`	($n \geq 0$)
simpleclass	→	*qtycls tyvar*	
simpletype	→	*tycon* $tyvar_1$ … $tyvar_k$	($k \geq 0$)
constrs	→	$constr_1$ `|` … `|` $constr_n$	($n \geq 1$)
constr	→	*con* [`!`] $atype_1$ … [`!`] $atype_k$	(arity *con* = k, $k \geq 0$)
	$\mid$	(*btype* $\mid$ `!` *atype*) *conop* (*btype* $\mid$ `!` *atype*)	(infix *conop*)
	$\mid$	*con* `{` $fielddecl_1$ `,` … `,` $fielddecl_n$ `}`	($n \geq 0$)
newconstr	→	*con atype*	
	$\mid$	*con* `{` *var* `::` *type* `}`	
fielddecl	→	*vars* `::` (*type* $\mid$ `!` *atype*)	
deriving	→	`deriving` (*dclass* $\mid$ `(`$dclass_1$`,` … `,` $dclass_n$`)`)	($n \geq 0$)
dclass	→	*qtycls*	
inst	→	*gtycon*	
	$\mid$	`(` *gtycon* $tyvar_1$ … $tyvar_k$ `)`	($k \geq 0$, *tyvars* distinct)
	$\mid$	`(` $tyvar_1$ `,` … `,` $tyvar_k$ `)`	($k \geq 2$, *tyvars* distinct)
	$\mid$	`[` *tyvar* `]`	
	$\mid$	`(` $tyvar_1$ `->` $tyvar_2$ `)`	$tyvar_1$ and $tyvar_2$ distinct

fdecl	→	`import` *callconv* [*safety*] *impent var* `::` *ftype*	(define variable)
	|	`export` *callconv expent var* `::` *ftype*	(expose variable)
callconv	→	`ccall` | `stdcall` | `cplusplus`	(calling convention)
	|	`jvm` | `dotnet`	
	|	**system-specific calling conventions**	
impent	→	[*string*]	(see Section 8.5.1)
expent	→	[*string*]	(see Section 8.5.1)
safety	→	`unsafe` | `safe`	
ftype	→	*frtype*	
	|	*fatype* → *ftype*	
frtype	→	*fatype*	
	|	`()`	
fatype	→	*qtycon* $atype_1 \ldots atype_k$	($k \geq 0$)
funlhs	→	*var apat* { *apat* }	
	|	*pat varop pat*	
	|	(*funlhs*) *apat* { *apat* }	
rhs	→	= *exp* [`where` *decls*]	
	|	*gdrhs* [`where` *decls*]	
gdrhs	→	*guards* = *exp* [*gdrhs*]	
guards	→	| $guard_1, \ldots, guard_n$	($n \geq 1$)
guard	→	*pat* <- *infixexp*	(pattern guard)
	|	`let` *decls*	(local declaration)
	|	*infixexp*	(boolean guard)
exp	→	*infixexp* `::` [*context* =>] *type*	(expression type signature)
	|	*infixexp*	
infixexp	→	*lexp qop infixexp*	(infix operator application)
	|	- *infixexp*	(prefix negation)
	|	*lexp*	
lexp	→	\ $apat_1 \ldots apat_n$ -> *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)
	|	`do` { *stmts* }	(do expression)
	|	*fexp*	
fexp	→	[*fexp*] *aexp*	(function application)
aexp	→	*qvar*	(variable)
	|	*gcon*	(general constructor)
	|	*literal*	
	|	(*exp*)	(parenthesized expression)
	|	(exp_1 , … , exp_k)	(tuple, $k \geq 2$)
	|	[exp_1 , … , exp_k]	(list, $k \geq 1$)
	|	[exp_1 [, exp_2] `..` [exp_3]]	(arithmetic sequence)
	|	[*exp* | $qual_1$, … , $qual_n$]	(list comprehension, $n \geq 1$)
	|	(*infixexp qop*)	(left section)
	|	($qop_{\langle - \rangle}$ *infixexp*)	(right section)
	|	*qcon* { $fbind_1$, … , $fbind_n$ }	(labeled construction, $n \geq 0$)

	|	$aexp_{\langle qcon \rangle}$ `{` *fbind*$_1$ `,` … `,` *fbind*$_n$ `}`	(labeled update, $n \geq 1$)
qual	→	*pat* `<-` *exp*	(generator)
	|	`let` *decls*	(local declaration)
	|	*exp*	(guard)
alts	→	*alt*$_1$ `;` … `;` *alt*$_n$	($n \geq 1$)
alt	→	*pat* `->` *exp* [`where` *decls*]	
	|	*pat gdpat* [`where` *decls*]	
	|		(*empty alternative*)
gdpat	→	*guards* `->` *exp* [*gdpat*]	
stmts	→	*stmt*$_1$ … *stmt*$_n$ *exp* [`;`]	($n \geq 0$)
stmt	→	*exp* `;`	
	|	*pat* `<-` *exp* `;`	
	|	`let` *decls* `;`	
	|	`;`	(*empty statement*)
fbind	→	*qvar* `=` *exp*	
pat	→	*lpat qconop pat*	(infix constructor)
	|	*lpat*	
lpat	→	*apat*	
	|	`-` (*integer* | *float*)	(negative literal)
	|	*gcon apat*$_1$ … *apat*$_k$	(arity *gcon* $= k$, $k \geq 1$)
apat	→	*var* [`@` *apat*]	(as pattern)
	|	*gcon*	(arity *gcon* $= 0$)
	|	*qcon* `{` *fpat*$_1$ `,` … `,` *fpat*$_k$ `}`	(labeled pattern, $k \geq 0$)
	|	*literal*	
	|	`_`	(wildcard)
	|	`(` *pat* `)`	(parenthesized pattern)
	|	`(` *pat*$_1$ `,` … `,` *pat*$_k$ `)`	(tuple pattern, $k \geq 2$)
	|	`[` *pat*$_1$ `,` … `,` *pat*$_k$ `]`	(list pattern, $k \geq 1$)
	|	`~` *apat*	(irrefutable pattern)
fpat	→	*qvar* `=` *pat*	
gcon	→	`()`	
	|	`[]`	
	|	`(,{,})`	
	|	*qcon*	
var	→	*varid* | `(` *varsym* `)`	(variable)
qvar	→	*qvarid* | `(` *qvarsym* `)`	(qualified variable)
con	→	*conid* | `(` *consym* `)`	(constructor)
qcon	→	*qconid* | `(` *gconsym* `)`	(qualified constructor)
varop	→	*varsym* | `` ` `` *varid* `` ` ``	(variable operator)
qvarop	→	*qvarsym* | `` ` `` *qvarid* `` ` ``	(qualified variable operator)
conop	→	*consym* | `` ` `` *conid* `` ` ``	(constructor operator)
qconop	→	*gconsym* | `` ` `` *qconid* `` ` ``	(qualified constructor operator)
op	→	*varop* | *conop*	(operator)
qop	→	*qvarop* | *qconop*	(qualified operator)

gconsym → `:` | *qconsym*

10.6 Fixity Resolution

The following is an example implementation of fixity resolution for Haskell expressions. Fixity resolution also applies to Haskell patterns, but patterns are a subset of expressions so in what follows we consider only expressions for simplicity.

The function `resolve` takes a list in which the elements are expressions or operators, i.e. an instance of the *infixexp* non-terminal in the context-free grammar. It returns either `Just e` where `e` is the resolved expression, or `Nothing` if the input does not represent a valid expression. In a compiler, of course, it would be better to return more information about the operators involved for the purposes of producing a useful error message, but the `Maybe` type will suffice to illustrate the algorithm here.

```
import Control.Monad

type Prec   = Int
type Var    = String

data Op = Op String Prec Fixity
  deriving (Eq,Show)

data Fixity = Leftfix | Rightfix | Nonfix
  deriving (Eq,Show)

data Exp = Var Var | OpApp Exp Op Exp | Neg Exp
  deriving (Eq,Show)

data Tok = TExp Exp | TOp Op | TNeg
  deriving (Eq,Show)

resolve :: [Tok] -> Maybe Exp
resolve tokens = fmap fst $ parseNeg (Op "" (-1) Nonfix) tokens
  where
    parseNeg :: Op -> [Tok] -> Maybe (Exp,[Tok])
    parseNeg op1 (TExp e1 : rest)
       = parse op1 e1 rest
    parseNeg op1 (TNeg : rest)
       = do guard (prec1 < 6)
            (r, rest') <- parseNeg (Op "-" 6 Leftfix) rest
            parse op1 (Neg r) rest'
       where
          Op _ prec1 fix1 = op1

    parse :: Op -> Exp -> [Tok] -> Maybe (Exp, [Tok])
    parse _   e1 [] = Just (e1, [])
    parse op1 e1 (TOp op2 : rest)
       -- case (1): check for illegal expressions
       | prec1 == prec2 && (fix1 /= fix2 || fix1 == Nonfix)
       = Nothing

       -- case (2): op1 and op2 should associate to the left
       | prec1 > prec2 || (prec1 == prec2 && fix1 == Leftfix)
       = Just (e1, TOp op2 : rest)

       -- case (3): op1 and op2 should associate to the right
       | otherwise
       = do (r,rest') <- parseNeg op2 rest
            parse op1 (OpApp e1 op2 r) rest'
       where
         Op _ prec1 fix1 = op1
         Op _ prec2 fix2 = op2
```

The algorithm works as follows. At each stage we have a call

```
parse op1 E1 (op2 : tokens)
```

which means that we are looking at an expression like

```
E0 ‘op1‘ E1 ‘op2‘ ...     (1)
```

(the caller holds E0). The job of `parse` is to build the expression to the right of `op1`, returning the expression and any remaining input.

There are three cases to consider:

1. if `op1` and `op2` have the same precedence, but they do not have the same associativity, or they are declared to be nonfix, then the expression is illegal.
2. If `op1` has a higher precedence than `op2`, or `op1` and `op2` should left-associate, then we know that the expression to the right of `op1` is `E1`, so we return this to the caller.
3. Otherwise, we know we want to build an expression of the form `E1 ‘op2‘ R`. To find `R`, we call `parseNeg op2 tokens` to compute the expression to the right of `op2`, namely `R` (more about `parseNeg` below, but essentially if `tokens` is of the form `(E2 : rest)`, then this is equivalent to `parse op2 E2 rest`). Now, we have `E0 ‘op1‘ (E1 ‘op2‘ R) ‘op3‘ ...` where `op3` is the next operator in the input. This is an instance of (1) above, so to continue we call parse, with the new `E1 == (E1 ‘op2‘ R)`.

To initialise the algorithm, we set `op1` to be an imaginary operator with precedence lower than anything else. Hence `parse` will consume the whole input, and return the resulting expression.

The handling of the prefix negation operator, `-`, complicates matters only slightly. Recall that prefix negation has the same fixity as infix negation: left-associative with precedence 6. The operator to the left of `-`, if there is one, must have precedence lower than 6 for the expression to be legal. The negation operator itself may left-associate with operators of the same fixity (e.g. `+`). So for example `-a + b` is legal and resolves as `(-a) + b`, but `a + -b` is illegal.

The function `parseNeg` handles prefix negation. If we encounter a negation operator, and it is legal in this position (the operator to the left has precedence lower than 6), then we proceed in a similar way to case (3) above: compute the argument to `-` by recursively calling `parseNeg`, and then continue by calling `parse`.

Note that this algorithm is insensitive to the range and resolution of precedences. There is no reason in principle that Haskell should be limited to integral precedences in the range 1 to 10; a larger range, or fractional values, would present no additional difficulties.