Part I

The Haskell 98 Language

JFP **13** (1): i–xii, 1–6, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803000315 Printed in the United Kingdom

Chapter 1

Introduction

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages.

This book defines the syntax for Haskell programs and an informal abstract semantics for the meaning of such programs. We leave as implementation dependent the ways in which Haskell programs are to be manipulated, interpreted, compiled, etc. This includes such issues as the nature of programming environments and the error messages returned for undefined programs (i.e. programs that formally evaluate to $\perp$).

1.1 Program Structure

In this section, we describe the abstract syntactic and semantic structure of Haskell, as well as how it relates to the organization of the rest of the report.

1. At the topmost level a Haskell program is a set of *modules*, described in Chapter 5. Modules provide a way to control namespaces and to re-use software in large programs.
2. The top level of a module consists of a collection of *declarations*, of which there are several kinds, all described in Chapter 4. Declarations define things such as ordinary values, datatypes, type classes, and fixity information.

3. At the next lower level are *expressions*, described in Chapter 3. An expression denotes a *value* and has a *static type*; expressions are at the heart of Haskell programming "in the small."

4. At the bottom level is Haskell's *lexical structure*, defined in Chapter 2. The lexical structure captures the concrete representation of Haskell programs in text files.

This book proceeds bottom-up with respect to Haskell's syntactic structure.

The chapters not mentioned above are Chapter 6, which describes the standard built-in datatypes and classes in Haskell, and Chapter 7, which discusses the I/O facility in Haskell (i.e. how Haskell programs communicate with the outside world). Also, there are several chapters describing the Prelude, the concrete syntax, literate programming, the specification of derived instances, and pragmas supported by most Haskell compilers.

Examples of Haskell program fragments in running text are given in typewriter font:

```
let x = 1
    z = x+y
in  z+1
```

"Holes" in program fragments representing arbitrary pieces of Haskell code are written in italics, as in `if` e_1 `then` e_2 `else` e_3. Generally, the italicized names are mnemonic, such as e for expressions, d for declarations, t for types, etc.

1.2 The Haskell Kernel

Haskell has adopted many of the convenient syntactic structures that have become popular in functional programming. In this Report, the meaning of such syntactic sugar is given by translation into simpler constructs. If these translations are applied exhaustively, the result is a program written in a small subset of Haskell that we call the Haskell *kernel*.

Although the kernel is not formally specified, it is essentially a slightly sugared variant of the lambda calculus with a straightforward denotational semantics. The translation of each syntactic structure into the kernel is given as the syntax is introduced. This modular design facilitates reasoning about Haskell programs and provides useful guidelines for implementors of the language.

1.3 Values and Types

An expression evaluates to a *value* and has a static *type*. Values and types are not mixed in Haskell. However, the type system allows user-defined datatypes of various sorts, and permits not only parametric polymorphism (using a traditional Hindley-Milner type structure) but also *ad hoc* polymorphism, or *overloading* (using *type classes*).

Errors in Haskell are semantically equivalent to $\perp$. Technically, they are not distinguishable from nontermination, so the language includes no mechanism for detecting or acting upon errors. However, implementations will probably try to provide useful information about errors (see Section 3.1).

1.4 Namespaces

There are six kinds of names in Haskell: those for *variables* and *constructors* denote values; those for *type variables*, *type constructors*, and *type classes* refer to entities related to the type system; and *module names* refer to modules. There are two constraints on naming:

1. Names for variables and type variables are identifiers beginning with lowercase letters or underscore; the other four kinds of names are identifiers beginning with uppercase letters.
2. An identifier must not be used as the name of a type constructor and a class in the same scope.

These are the only constraints; for example, `Int` may simultaneously be the name of a module, class, and constructor within a single scope.

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
$pat_{(pat')}$	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 | and [...] 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

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* [$any_{(symbol)}$ {*any*}] *newline*
dashes	→	`--` {`-`}
opencom	→	`{-`
closecom	→	`-}`
ncomment	→	*opencom ANYseq* {*ncomment ANYseq*} *closecom*
ANYseq	→	$\{ANY\}_{(\{ANY\}\ (\ opencom\ \mid\ closecom\ )\ \{ANY\})}$
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*$_{\langle special \mid \_ \mid : \mid " \mid ' \rangle}$
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.

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

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	→	*varid*	(type variables)
tycon	→	*conid*	(type constructors)
tycls	→	*conid*	(type classes)
modid	→	*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	→	[*modid* `.`] *varid*
qconid	→	[*modid* `.`] *conid*
qtycon	→	[*modid* `.`] *tycon*
qtycls	→	[*modid* `.`] *tycls*
qvarsym	→	[*modid* `.`] *varsym*
qconsym	→	[*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	→	*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*

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

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*} `\`

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.