Report on the Programming Language

Haskell

A Non-strict, Purely Functional Language

Version 1.3

May 1, 1996

John Peterson[1] [editor]
Kevin Hammond[2] [editor]
Lennart Augustsson[3]
Brian Boutel[4]
Warren Burton[5]
Joseph Fasel[6]
Andrew D. Gordon[7]
John Hughes[3]
Paul Hudak[1]
Thomas Johnsson[3]
Mark Jones[9]
Simon Peyton Jones[8]
Alastair Reid[1]
Philip Wadler[8]

Authors' affiliations: (1) Yale University, (2) University of St. Andrews, (3) Chalmers University of Technology, (4) Victoria University of Wellington, (5) Simon Fraser University, (6) Los Alamos National Laboratory, (7) University of Cambridge, (8) University of Glasgow, (9) University of Nottingham

Contents

Preface
(May 1, 1996)

> *"Some half dozen persons have written technically on combinatory logic, and most of these, including ourselves, have published something erroneous. Since some of our fellow sinners are among the most careful and competent logicians on the contemporary scene, we regard this as evidence that the subject is refractory. Thus fullness of exposition is necessary for accuracy; and excessive condensation would be false economy here, even more than it is ordinarily."*
>
> Haskell B. Curry and Robert Feys
> in the Preface to *Combinatory Logic* [2], May 31, 1956

In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages. This document describes the result of that committee's efforts: a purely functional programming language called Haskell, named after the logician Haskell B. Curry whose work provides the logical basis for much of ours.

Goals

The committee's primary goal was to design a language that satisfied these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

The committee hopes that Haskell can serve as a basis for future research in language design. We hope that extensions or variants of the language may appear, incorporating experimental features.

This Report

This report is the official specification of the Haskell language and should be suitable for writing programs and building implementations. It is *not* a tutorial on programming in Haskell such as the 'Gentle Introduction' [5], so some familiarity with functional languages is assumed.

Version 1.3 of the report was unveiled in 1996. It corrects some minor errors in the 1.2 report and adds important new functionality, especially for input/output. This new functionality is summarized in the following section. Unlike earlier versions of Haskell, version 1.3 is described in two separate documents: the Haskell Language Report (this document) and the Haskell Library Report[9].

Highlights

Libraries

For the first time, we distinguish between Prelude and Library entities. Entities defined by the Prelude, a module named `Prelude`, are in scope unless explicitly hidden. Entities defined in library modules are in scope *only* if that module is explicitly imported. The library modules specified by Haskell are described in the Haskell Library Report.

Monadic I/O

Monadic I/O has proven to be more general and in many respects simpler than the stream-based I/O system used in Haskell 1.2. Here are the highlights of the I/O definition.

- We define a *monadic programming model* for Haskell. Expressions of type `IO` a denote computations that may engage in I/O before returning an answer of type a.
- The `IO` monad admits computations that fail and recovers from such failures.
- We define a new type of *handles*, to mediate I/O operations on files and other I/O devices. Handles are part of the I/O library.
- We define input polling and input of characters. In contrast, Haskell 1.2 represented character input as a single `String` (that is, a lazy list of characters), containing all the characters available for input throughout the program execution.
- Monadic I/O provides an extensible framework capable of incorporating advanced operating system and GUI interfaces in libraries.
- Monadic programming has been made more readable through the introduction of a special `do` syntax.

Constructor Classes

Constructor classes are a natural generalization of the original Haskell type system, supporting polymorphism over type constructors. For example, the monadic operators used by the I/O system have been generalized using constructor classes to arbitrary monads just as `(+)` has been generalized to arbitrary numeric types using type classes.

New Datatype Features

A number of enhancements have been made to Haskell type declarations. These include:

- Strictness annotations allow structures to be represented in a more efficient manner.
- The components of a constructor may be labeled using field names. Selection, construction, and update operations which reference fields by name rather than position are now available.
- The `newtype` declaration defines a type which renames an existing datatype without changing the underlying object representation. Unlike type synonyms, types defined by `newtype` are distinct from their definition.

Improvements in the Module System

A number of substantial changes to the module system have been made. Instead of renaming, qualified names are used to resolve name conflicts. All names are now redefinable; there is no longer a `PreludeCore` module containing names that cannot be reused. Interface files are no longer specified by this report; all issues of separate compilation are now left up to the implementation.

The n+k Pattern Controversy

For technical reasons, many people feel that n+k patterns are an incongruous language design feature that should be eliminated from Haskell. On the other hand, they serve as a vehicle for teaching introductory programming, in particular recursion over natural numbers. Alternatives to n+k patterns have been explored, but are too premature to include in Haskell 1.3. Thus the 1.3 committee decided to retain this feature at present but to discourage the use of n+k patterns by Haskell users. This feature may be altered or removed in future versions of Haskell and should be avoided. Implementors are encouraged to provide a mechanism for users to selectively enable or disable n+k patterns.

Haskell Resources

We welcome your comments, suggestions, and criticisms on the language or its presentation in the report. A common mailing list for technical discussion of Haskell uses the following electronic mail addresses:

- `haskell@dcs.gla.ac.uk` forwards mail to all subscribers of the Haskell list.
- `majordomo@dcs.gla.ac.uk` is used to add and remove subscribers from the mailing list. To subscribe or unsubscribe send messages of the form:

  ```
  subscribe haskell
  unsubscribe haskell
  ```

  You may wish to subscribe or remove a mailing address other than the reply-to address contained in your mail message. These commands may include an explicit email address:

  ```
  subscribe haskell bjm@wotsamatta.edu
  ```

  Please do not send subscription requests direct to the mailing list.
- Each implementation has an email address for discussions of specific Haskell systems. Please send questions and comments regarding these directly to the associated groups instead of the global Haskell community.

Web pages for Haskell, which includes an on-line version of this report, a tutorial, extensions to Haskell, information about upgrading programs from prior Haskell versions, and information about Haskell implementations can be found at the following sites:

- `http://www.cs.yale.edu/HTML/YALE/CS/haskell/yale-fp.html`
- `http://www.dcs.gla.ac.uk/fp/software/ghc`
- `http://www.cs.chalmers.se/Haskell`
- `http://www.cs.nott.ac.uk/Research/fpg/haskell.html`

Acknowledgements

We heartily thank these people for their useful contributions to this report: Richard Bird, Stephen Blott, Tom Blenko, Duke Briscoe, Magnus Carlsson, Chris Clack, Guy Cousineau, Tony Davie, Chris Fasel, Pat Fasel, Andy Gill, Cordy Hall, Thomas Hallgren, Bob Hiromoto, Nic Holt, Ian Holyer, Randy Hudson, Simon B. Jones, Stef Joosten, Mike Joy, Stefan Kahrs, Kent Karlsson, Richard Kelsey, Siau-Cheng Khoo, Amir Kishon, John Launchbury, Mark Lillibridge, Sandra Loosemore, Olaf Lubeck, Jim Mattson, Erik Meijer, Randy Michelsen, Rick Mohr, Arthur Norman, Nick North, Paul Otto, Larne Pekowsky, Rinus Plasmeijer, Ian Poole, John Robson, Colin Runciman, Patrick Sansom, Lauren Smith, Raman Sundaresh, Satish Thatte, Tom Thomson, Pradeep Varma, Tony Warnock, Stuart Wray, and Bonnie Yantis. We are especially grateful to past members of the Haskell committee—Arvind, Jon Fairbairn, Maria M. Guzman, Dick Kieburtz, Rishiyur Nikhil, Mike Reeve, David Wise, and Jonathan Young—for the major contributions they have made to previous versions of this report, which we have been able to build upon, and for their support for this latest revision of Haskell. We also thank those who have participated in the lively discussions about Haskell on the FP and Haskell mailing lists.

Finally, aside from the important foundational work laid by Church, Rosser, Curry, and others on the lambda calculus, we wish to acknowledge the influence of many noteworthy programming languages developed over the years. Although it is difficult to pinpoint the origin of many ideas, we particularly wish to acknowledge the influence of Lisp (and its modern-day incarnations Common Lisp and Scheme); Landin's ISWIM; APL; Backus's FP [1]; ML and Standard ML; Hope and Hope$^+$; Clean; Id; Gofer; Sisal; and Turner's series of languages culminating in Miranda.[1] Without these forerunners Haskell would not have been possible.

[1]Miranda is a trademark of Research Software Ltd.

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 lazy functional languages.

This report 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 Section 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 Section 4. Declarations define things such as ordinary values, datatypes, type classes, and fixity information.
3. At the next lower level are *expressions*, described in Section 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 Section 2. The lexical structure captures the concrete representation of Haskell programs in text files.

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

The sections not mentioned above are Section 6, which describes the standard built-in datatypes and classes in Haskell, and Section 7, which discusses the I/O facility in Haskell (i.e. how Haskell programs communicate with the outside world). Also, there are several appendices 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 all cases, their formal semantics can be given via translation into a proper subset of Haskell called the Haskell *kernel*. 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. Of course, implementations will probably try to provide useful information about errors.

1.4 Namespaces

Haskell provides a lexical syntax for infix *operators* (either functions or constructors). To emphasize that operators are bound to the same things as identifiers, and to allow the two to be used interchangeably, there is a simple way to convert between the two: any function or constructor identifier may be converted into an operator by enclosing it in backquotes, and any operator may be converted into an identifier by enclosing it in parentheses. For example, `x + y` is equivalent to `(+) x y`, and `f x y` is the same as ``x `f` y``. These lexical matters are discussed further in Section 2.

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 three constraints on naming:

1. Names for variables and type variables are identifiers beginning with lowercase letters; the other four kinds of names are identifiers beginning with uppercase letters.

2. Constructor operators are operators beginning with "`:`"; variable operators are operators not beginning with "`:`".

3. 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.

1.5 Layout

In the syntax given in the rest of the report, *declaration lists* are always preceded by the keyword `where`, `let`, `do`, or `of`, and are enclosed within curly braces (`{ }`) with the individual declarations separated by semicolons (`;`). For example, the syntax of a `let` expression is:

$$\texttt{let \{ } decl_1 \texttt{ ; } decl_2 \texttt{ ; } \ldots \texttt{ ; } decl_n \ [\texttt{;}] \texttt{ \} in } exp$$

Haskell permits the omission of the braces and semicolons by using *layout* to convey the same information. This allows both layout-sensitive and -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 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 declaration list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the declaration 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.

Given these rules, a single newline may actually terminate several declaration 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 declaration list.

To facilitate the use of layout at the top level of a module (an implementation may allow several modules may reside in one file), the keyword `module` and the end-of-file token are assumed to occur in column 0 (whereas normally the first column is 1). Otherwise, all top-level declarations would have to be indented.

See also Section B.3.

As an example, Figure 1 shows a (somewhat contrived) module and Figure 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.

When comparing indentations for standard Haskell programs, a fixed-width font with this tab convention is assumed: tab stops are 8 characters apart (with the first tab stop in column 9), and a tab character causes the insertion of enough spaces (always ≥ 1) to align the current position with the next tab stop. Particular implementations may alter this rule to accommodate variable-width fonts and alternate tab conventions, but standard Haskell programs must observe this rule.

```
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 -> Integer
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 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 -> Integer
;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: Sample program with layout expanded

2 Lexical Structure

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

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:

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

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 Latin-ISO-8859-1[6] character set. However, source programs are currently biased toward the ASCII character set used in earlier versions of Haskell.

2.2 Lexical Program Structure

program → { *lexeme* | *whitespace* }
lexeme → *varid* | *conid* | *varsym* | *consym* | *literal* | *special* | *reservedop* | *reservedid*
literal → *integer* | *float* | *char* | *string*
special → (|) | , | ; | [|] | _ | ` | { | }

whitespace → *whitestuff* {*whitestuff*}
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *vertab* | *formfeed* | *space* | *tab* | *nonbrkspc*
newline → a newline (system dependent)
space → a space
tab → a horizontal tab
vertab → a vertical tab

formfeed → a form feed
nonbrkspc → a non-breaking space
comment → `--` {*any*} *newline*
ncomment → `{-` *ANYseq* {*ncomment ANYseq*} `-}`
ANYseq → $\{ANY\}_{\langle \{ANY\}\ (\ \texttt{\{-}\ \mid\ \texttt{-\}}\ )\ \{ANY\} \rangle}$
ANY → *any* | *newline* | *vertab* | *formfeed*
any → *graphic* | *space* | *tab* | *nonbrkspc*
graphic → *large* | *small* | *digit* | *symbol* | *special* | `:` | `"` | `'`

small → *ASCsmall* | *ISOsmall*
ASCsmall → `a` | `b` | ... | `z`
ISOsmall → à | á | â | ã | ä | å | æ | ç | è | é | ê | ë
| ì | í | î | ï | ð | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | ý | *thorn* | ÿ | ß

large → *ASClarge* | *ISOlarge*
ASClarge → `A` | `B` | ... | `Z`
ISOlarge → À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë
| Ì | Í | Î | Ï | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
| Ù | Ú | Û | Ü | Ý | *Thorn*

symbol → *ASCsymbol* | *ISOsymbol*
ASCsymbol → `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@`
| `\` | `^` | `|` | `-` | `~`
ISOsymbol → ¡ | ¢ | £ | *currency* | ¥ | ¦ | § | ¨ | © | ª | «
| ¬ | – | ® | ¯ | ° | ± | ² | ³ | ´ | µ | ¶
| · | ¸ | ¹ | º | » | ¼ | ½ | ¾ | ¿ | × | ÷

digit → `0` | `1` | ... | `9`
octit → `0` | `1` | ... | `7`
hexit → *digit* | `A` | ... | `F` | `a` | ... | `f`

Characters not in the category *ANY* are not valid in Haskell programs and should result in a lexing error. Comments are valid whitespace. An ordinary comment begins with two consecutive dashes (`--`) and extends to the following newline. A nested comment begins with `{-` and ends with `-}`; it can be between any two lexemes. All character sequences not containing `{-` nor `-}` are ignored within a nested comment. Nested comments may be nested to any depth: any occurrence of `{-` 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, the sequence `--` has no special significance. Nested comments are used for compiler pragmas, as explained in Appendix E.

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.

2.3 Identifiers and Operators

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

An identifier consists of a letter followed by zero or more letters, digits, underscores, and single quotes. Identifiers are lexically distinguished into two classes: 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). Some identifiers, here indicated by *specialid*, have special meanings in certain contexts but can be used as ordinary identifiers.

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

Operator symbols are formed from one or more symbol characters, as defined above, and are lexically distinguished into two classes: those that start with a colon (constructors) and those that do not (functions). Some operators, here indicated by *specialop*, have special meanings in certain contexts but can be used as ordinary operators.

The sequence `--` immediately terminates a symbol; thus `+--+` parses as the symbol `+` followed by a comment.

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.

Although `case` is a reserved word, `cases` is not. Similarly, although `=` is reserved, `==` and `~=` are not. At each point, the longest possible lexeme is read, using a context-independent deterministic lexical analysis (i.e. no lookahead beyond the current character is required). Any kind of *whitespace* is also a proper delimiter for lexemes.

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.

External names 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 Section 5.1.2.

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

2.4 Numeric Literals

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

integer	$\rightarrow$	*decimal*
	|	`0o` *octal* | `0O` *octal*
	|	`0x` *hexadecimal* | `0X` *hexadecimal*
float	$\rightarrow$	*decimal* `.` *decimal*[(`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 6.3.1.

2.5 Character and String Literals

char	$\rightarrow$	`'` (*graphic*$_{\langle \texttt{'} \mid \texttt{\\} \rangle}$ | *space* | *escape*$_{\langle \texttt{\\\&} \rangle}$) `'`
string	$\rightarrow$	`"` {*graphic*$_{\langle \texttt{"} \mid \texttt{\\} \rangle}$ | *space* | *escape* | *gap*} `"`
escape	$\rightarrow$	`\` (*charesc* | *ascii* | *decimal* | `o` *octal* | `x` *hexadecimal*)
charesc	$\rightarrow$	`a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | `'` | `&`
ascii	$\rightarrow$	`^`*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	$\rightarrow$	*ASClarge* | `@` | `[` | `\` | `]` | `^` | `_`
gap	$\rightarrow$	`\` *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 ISO-8859-1 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. Numeric escapes that are out-of-range of the ISO standard are undefined and thus non-portable.

Consistent with the "consume longest lexeme" 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).

3 Expressions

In this section, we describe the syntax and informal semantics of Haskell *expressions*, including their translations into the Haskell kernel, where appropriate. Except in the case of `let` expressions, these translations preserve both the static and dynamic semantics. Some of the names and symbols used in the syntax are not reserved. These are indicated by the 'special' productions in the lexical syntax. Examples include `!` (used only in `data` declarations) and `as` (used in `import` declarations).

Free variables and constructors used in these translations refer to entities defined by the Prelude. To avoid clutter, we use `True` instead of `Prelude.True` or `map` instead of `Prelude.map`. (`Prelude.True` is a *qualified name* as described in Section 5.1.2.)

In the syntax that follows, 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 non-associativity 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

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

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

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

Item	Associativity
simple terms, parenthesized terms	–
irrefutable patterns (`~`)	–
as-patterns (`@`)	right
function application	left
`do`, `if`, `let`, lambda(`\`), `case` (leftwards)	right
`case` (rightwards)	right
infix operators, prec. 9	as defined
...	...
infix operators, prec. 0	as defined
function types (`->`)	right
contexts (`=>`)	–
type constraints (`::`)	–
`do`, `if`, `let`, lambda(`\`) (rightwards)	right
sequences (`..`)	–
generators (`<-`)	–
grouping (`,`)	n-ary
guards (`|`)	–
case alternatives (`->`)	–
definitions (`=`)	–
separation (`;`)	n-ary

Table 1: Precedence of expressions, patterns, definitions (highest to lowest)

|	`[` exp `|` $qual_1$ `,` ... `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
|	`(` exp^{i+1} $qop^{(a,i)}$ `)`	(left section)
|	`(` $qop^{(a,i)}$ exp^{i+1} `)`	(right section)
|	$qcon$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
|	$aexp_{\{qcon\}}$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled update, $n \geq 1$)

As an aid to understanding this grammar, Table 1 shows the relative precedence of expressions, patterns and definitions, plus an extended associativity. – indicates that the item is non-associative.

The grammar is ambiguous regarding the extent of lambda abstractions, let expressions, and conditionals. The ambiguity is resolved by the metarule that each of these constructs extends as far to the right as possible. As a consequence, each of these constructs has two precedences, one to its left, which is the precedence used in the grammar; and one to its right which is obtained via the metarule. See the sample parses below.

Expressions involving infix operators are disambiguated by the operator's fixity (see Section 5.6). Consecutive unparenthesised operators with the same precedence must both be either left or right associative to avoid a syntax error. Given an unparenthesised expression "$x\ qop^{(a,i)}\ y\ qop^{(b,j)}\ z$", parentheses must be added around either "$x\ qop^{(a,i)}\ y$" or "$y\ qop^{(b,j)} z$" when $i = j$ unless $a = b =$ l or $a = b =$ r.

Negation is the only prefix operator in Haskell; it has the same precedence as the infix `-` operator defined in the Prelude (see Figure 2, page 62).

The separation of function arrows from case alternatives solves the ambiguity which otherwise arises when an unparenthesised function type is used in an expression, such as the guard in a case expression.

Sample parses are shown below.

This	Parses as
`f x + g y`	`(f x) + (g y)`
`- f x + y`	`(- (f x)) + y`
`let { ... } in x + y`	`let { ... } in (x + y)`
`z + let { ... } in x + y`	`z + (let { ... } in (x + y))`
`f x y :: Int`	`(f x y) :: Int`
`\ x -> a+b :: Int`	`\ x -> ((a+b) :: Int)`

For the sake of clarity, the rest of this section shows the syntax of expressions without their precedences.

3.1 Errors

Errors during expression evaluation, denoted by $\perp$, are indistinguishable from non-termination. Since Haskell is a lazy language, all Haskell types include $\perp$. That is, a value of any type may be bound to a computation that, when demanded, results in an error. When evaluated, errors cause immediate program termination and cannot be caught by the user. The Prelude provides two functions to directly cause such errors:

```
error     :: String -> a
undefined :: a
```

A call to `error` terminates execution of the program and returns an appropriate error indication to the operating system. It should also display the string in some system-dependent manner. When `undefined` is used, the error message is created by the compiler.

Translations of Haskell expressions use `error` and `undefined` to explicitly indicate where execution time errors may occur. The actual program behavior when an error occurs is up to the implementation. The messages passed to the `error` function in these translations are only suggestions; implementations may choose to display more or less information when an error occurs.

3.2 Variables, Constructors, and Operators

aexp	$\rightarrow$	*qvar*	(variable)
	|	*gcon*	(general constructor)
	|	*literal*	
gcon	$\rightarrow$	`()`	
	|	`[]`	
	|	`(,{,})`	
	|	*qcon*	
qvar	$\rightarrow$	*qvarid* | `(` *qvarsym* `)`	(qualified variable)
qcon	$\rightarrow$	*qconid* | `(` *qconsym* `)`	(qualified constructor)

Alphanumeric operators are formed by enclosing an identifier between grave accents (backquotes). Any variable or constructor may be used as an operator in this way. If *fun* is an identifier (either variable or constructor), then an expression of the form *fun x y* is equivalent to *x* \`*fun*\` *y*. If no fixity declaration is given for \`*fun*\` then it defaults to highest precedence and left associativity (see Section 5.6).

Similarly, any symbolic operator may be used as a (curried) variable or constructor by enclosing it in parentheses. If *op* is an infix operator, then an expression or pattern of the form *x op y* is equivalent to `(`*op*`)` *x y*.

Qualified names may only be used to reference an imported variable or constructor (see Section 5.1.2) but not in the definition of a new variable or constructor. Thus

```
let F.x = 1 in F.x   -- invalid
```

incorrectly uses a qualifier in the definition of `x`, regardless of the module containing this definition. Qualification does not affect the nature of an operator: `F.+` is an infix operator just as `+` is.

Special syntax is used to name some constructors for some of the built-in types, as found in the production for *gcon* and *literal*. These are described in Section 6.1.

An integer literal represents the application of the function `fromInteger` to the appropriate value of type `Integer`. Similarly, a floating point literal stands for an application of `fromRational` to a value of type `Rational` (that is, `Ratio Integer`).

Translation: The integer literal i is equivalent to `fromInteger` i, where `fromInteger` is a method in class `Num` (see Section 6.3.1).
The floating point literal f is equivalent to `fromRational` (n `Ratio.%` d), where `fromRational` is a method in class `Fractional` and `Ratio.%` constructs a rational from two integers, as defined in the `Ratio` library. The integers n and d are chosen so that $n/d = f$.

3.3 Curried Applications and Lambda Abstractions

fexp	$\rightarrow$	[*fexp*] *aexp*	(function application)
exp	$\rightarrow$	`\` $apat_1$ ... $apat_n$ `->` *exp*	

Function application is written e_1 e_2. Application associates to the left, so the parentheses may be omitted in `(f x) y`. Because e_1 could be a data constructor, partial applications of data constructors are allowed.

Lambda abstractions are written `\` p_1 ... p_n `->` e, where the p_i are *patterns*. An expression such as `\x:xs->x` is syntactically incorrect, and must be rewritten as `\(x:xs)->x`.

The set of patterns must be *linear*—no variable may appear more than once in the set.

Translation: The lambda abstraction `\` p_1 ... p_n `->` e is equivalent to

`\` x_1 ... x_n `-> case (`x_1 `,` ... `,` x_n`) of (`p_1 `,` ... `,` p_n`) ->` e

where the x_i are new identifiers. Given this translation combined with the semantics of case expressions and pattern matching described in Section 3.17.3, if the pattern fails to match, then the result is $\perp$.

3.4 Operator Applications

exp	$\rightarrow$	exp_1 *qop* exp_2	
	|	`-` *exp*	(*prefix negation*)

The form e_1 *qop* e_2 is the infix application of binary operator *qop* to expressions e_1 and e_2.

The special form `-`e denotes prefix negation, the only prefix operator in Haskell, and is syntax for `negate` (e). The binary `-` operator does not necessarily refer to the definition of `-` in the Prelude; it may be rebound by the module system. However, unary `-` will always refer to the `negate` function defined in the Prelude. There is no link between the local meaning of the `-` operator and unary negation.

Prefix negation has the same precedence as the infix operator `-` defined in the Prelude (see Figure 2, page 62). Because `e1-e2` parses as an infix application of the binary operator `-`, one must write `e1(-e2)` for the alternative parsing. Similarly, `(-)` is syntax for `(\ x y -> x-y)`, as with any infix operator, and does not denote `(\ x -> -x)`—one must use `negate` for that.

Translation: e_1 *op* e_2 is equivalent to `(`*op*`)` e_1 e_2. `-`e is equivalent to `negate` (e).

3.5 Sections

aexp	$\rightarrow$	`(` *exp qop* `)`
	$\mid$	`(` *qop exp* `)`

Sections are written as `(` *op e* `)` or `(` *e op* `)`, where *op* is a binary operator and *e* is an expression. Sections are a convenient syntax for partial application of binary operators.

The normal rules of syntactic precedence apply to sections; for example, `(*a+b)` is syntactically invalid, but `(+a*b)` and `(*(a+b))` are valid. Syntactic associativity, however, is not taken into account in sections; thus, `(a+b+)` must be written `((a+b)+)`.

Because `-` is treated specially in the grammar, `(-` *exp*`)` is not a section, but an application of prefix negation, as described in the preceding section. However, there is a `subtract` function defined in the Prelude such that `(subtract` *exp*`)` is equivalent to the disallowed section. The expression `(+ (-` *exp*`))` can serve the same purpose.

> **Translation:** For binary operator *op* and expression *e*, if *x* is a variable that does not occur free in *e*, the section `(`*op e*`)` is equivalent to `\` *x* `->` *x op e*, and the section `(`*e op*`)` is equivalent to `(`*op*`)` *e*.

3.6 Conditionals

exp	$\rightarrow$	`if` exp_1 `then` exp_2 `else` exp_3

A *conditional expression* has the form `if` e_1 `then` e_2 `else` e_3 and returns the value of e_2 if the value of e_1 is `True`, e_3 if e_1 is `False`, and $\perp$ otherwise.

> **Translation:** `if` e_1 `then` e_2 `else` e_3 is equivalent to:
>
> `case` e_1 `of { True ->` e_2 `; False ->` e_3 `}`
>
> where `True` and `False` are the two nullary constructors from the type `Bool`, as defined in the Prelude.

3.7 Lists

aexp	$\rightarrow$	`[` exp_1 `,` ... `,` exp_k `]`	$(k \geq 1)$

Lists are written `[`e_1`,` ... `,` e_k`]`, where $k \geq 1$; the empty list is written `[]`. Standard operations on lists are given in the Prelude (see Appendix A, notably Section A.1).

Translation: `[`e_1 `,` ... `,` e_k`]` is equivalent to

$$e_1 \texttt{ : (} e_2 \texttt{ : (... (} e_k \texttt{ : []))))}$$

where `:` and `[]` are constructors for lists, as defined in the Prelude (see Section 6.1.3). The types of e_1 through e_k must all be the same (call it t), and the type of the overall expression is `[`t`]` (see Section 4.1.1).

3.8 Tuples

aexp → `(` exp_1 `,` ... `,` exp_k `)` $(k \geq 2)$

Tuples are written `(`e_1 `,` ... `,` e_k`)`, and may be of arbitrary length $k \geq 2$. Standard operations on tuples are given in the Prelude (see Appendix A).

Translation: `(`e_1 `,` ... `,` e_k`)` for $k \geq 2$ is an instance of a k-tuple as defined in the Prelude, and requires no translation. If t_1 through t_k are the types of e_1 through e_k, respectively, then the type of the resulting tuple is `(`t_1 `,` ... `,` t_k`)` (see Section 4.1.1).

3.9 Unit Expressions and Parenthesized Expressions

aexp → `()`
| `(` *exp* `)`

The form `(`e`)` is simply a *parenthesized expression*, and is equivalent to e. The *unit expression* `()` has type `()` (see Section 4.1.1); it is the only member of that type apart from $\perp$ (it can be thought of as the "nullary tuple")—see Section 6.1.5.

Translation: `(`e`)` is equivalent to e.

3.10 Arithmetic Sequences

aexp → `[` exp_1 [`,` exp_2] `..` [exp_3] `]`

The form `[`e_1 `,` e_2 `..` e_3`]` denotes an *arithmetic sequence* from e_1 in increments of $e_2 - e_1$ of values not greater than e_3 (if the increment is nonnegative) or not less than e_3 (if the increment is negative). Thus, the resulting list is empty if the increment is nonnegative and e_3 is less than e_1 or if the increment is negative and e_3 is greater than e_1. If the increment is zero, an infinite list of e_1s results if e_3 is not less than e_1. If e_3 is omitted, the result is an infinite list, unless the element type is finite, in which case the implied limit is the greatest value of the type if the increment is nonnegative, or the least value, otherwise.

The forms `[`e_1 `..` e_3`]` and `[`e_1 `..]` are similar to those above, but with an implied increment of one.

Arithmetic sequences may be defined over any type in class `Enum`, including `Char`, `Int`, and `Integer` (see Figure 5 , page 66 and Section 4.3.3). For example, `['a'..'z']` denotes the list of lowercase letters in alphabetical order.

Translation: Arithmetic sequences satisfy these identities:

`[` e_1 `.. ]`	=	`enumFrom` e_1
`[` e_1`,`e_2 `.. ]`	=	`enumFromThen` e_1 e_2
`[` e_1`..`e_3 `]`	=	`enumFromTo` e_1 e_3
`[` e_1`,`e_2`..`e_3 `]`	=	`enumFromThenTo` e_1 e_2 e_3

where `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo` are class methods in the class `Enum` as defined in the Prelude (see Figure 5 , page 66).

3.11 List Comprehensions

aexp	→	`[` *exp* `|` *qual*$_1$ `,` ... `,` *qual*$_n$ `]`	(list comprehension, $n \geq 1$)
qual	→	*pat* `<-` *exp*	
	|	`let` *decllist*	
	|	*exp*	

A *list comprehension* has the form `[` e `|` q_1 `,` ... `,` q_n `]`, $n \geq 1$, where the q_i qualifiers are either

- *generators* of the form p `<-` e, where p is a pattern (see Section 3.17) of type t and e is an expression of type `[`t`]`
- *guards*, which are arbitrary expressions of type `Bool`
- *local bindings* which provide new definitions for use in the generated expression e or subsequent guards and generators.

Such a list comprehension returns the list of elements produced by evaluating e in the successive environments created by the nested, depth-first evaluation of the generators in the qualifier list. Binding of variables occurs according to the normal pattern matching rules (see Section 3.17), and if a match fails then that element of the list is simply skipped over. Thus:

```
[ x |  xs   <- [ [(1,2),(3,4)], [(5,4),(3,2)] ],
      (3,x) <- xs ]
```

yields the list `[4,2]`. If a qualifier is a guard, it must evaluate to `True` for the previous pattern match to succeed. As usual, bindings in list comprehensions can shadow those in outer scopes; for example:

```
[ x | x <- x, x <- x ]  =  [ z | y <- x, z <- y]
```

Translation: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

```
[ e | b ]              =  if b then [e] else []
[ e | let decllist ]   =  let decllist in [e]
[ e | q1 , q2 ]        =  concat [ [ e | q2 ] | q1 ]
[ e | p <- l ]         =  let ok p  =  True
                              ok _  =  False
                          in map (\p -> e) (filter ok l)
```

where e ranges over expressions, p ranges over patterns, l ranges over list-valued expressions, b ranges over boolean expressions, q_1 and q_2 range over non-empty lists of qualifiers, and `ok` is a new identifier not appearing in e, p, or l. These equations uniquely define list comprehensions. `True`, `False`, `map`, `concat`, and `filter` are all as defined in the Prelude.

As indicated by the translation of list comprehensions, variables bound by `let` have fully polymorphic types while those defined by `<-` are lambda bound and are thus monomorphic (see Section 4.5.3).

3.12 Let Expressions

exp → `let` *decllist* `in` *exp*

Let expressions have the general form `let {` d_1 `;` ... `;` d_n `} in` e, and introduce a nested, lexically-scoped, mutually-recursive list of declarations (`let` is often called `letrec` in other languages). The scope of the declarations is the expression e and the right hand side of the declarations. Declarations are described in Section 4. Pattern bindings are matched lazily; an implicit `~` makes these patterns irrefutable. For example,

```
let (x,y) = undefined in e
```

does not cause an execution-time error until `x` or `y` are evaluated.

Translation: The dynamic semantics of the expression `let {` d_1 `; ... ;` d_n `} in` e_0 are captured by this translation: After removing all type signatures, each declaration d_i is translated into an equation of the form p_i `=` e_i, where p_i and e_i are patterns and expressions respectively, using the translation in Section 4.4.2. Once done, these identities hold, which may be used as a translation into the kernel:

`let {`p_1 `=` e_1`; ...;` p_n `=` e_n`} in` e_0	=	`let (~`p_1`,...,~`p_n`) = (`e_1`,...,`e_n`) in` e_0
`let` p `=` e_1 `in` e_0	=	`case` e_1 `of ~`p `->` e_0 where no variable in p appears free in e_1
`let` p `=` e_1 `in` e_0	=	`let` p `= fix ( \ ~`p `->` e_1`) in` e_0

where `fix` is the least fixpoint operator. Note the use of the irrefutable patterns in the second and third rules. This translation does not preserve the static semantics because the use of `case` precludes a fully polymorphic typing of the bound variables. The static semantics of the bindings in a `let` expression are described in Section 4.4.2.

3.13 Case Expressions

exp	$\rightarrow$	`case` *exp* `of {` *alts* [`;`] `}`	
alts	$\rightarrow$	alt_1 `; ... ;` alt_n	$(n \geq 1)$
alt	$\rightarrow$	*pat* `->` *exp* [`where` *decllist*]	
	$\vert$	*pat gdpat* [`where` *decllist*]	
gdpat	$\rightarrow$	*gd* `->` *exp* [*gdpat*]	
gd	$\rightarrow$	`|` exp^0	

A *case expression* has the general form

`case` e `of {` p_1 $match_1$ `; ... ;` p_n $match_n$ `}`

where each $match_i$ is of the general form

`|` g_{i1} `->` e_{i1}
`...`
`|` g_{im_i} `->` e_{im_i}
`where` $decllist_i$

Each alternative p_i $match_i$ consists of a pattern p_i and its matches, $match_i$, which consists of pairs of guards g_{ij} and bodies e_{ij} (expressions), as well as optional bindings ($decllist_i$) that scope over all of the guards and expressions of the alternative. An alternative of the form

pat `->` *exp* `where` *decllist*

is treated as shorthand for:

pat `| True ->` *expr*
`where` *decllist*

A case expression must have at least one alternative and each alternative must have at least one body. Each body must have the same type, and the type of the whole expression is that type.

A case expression is evaluated by pattern matching the expression e against the individual alternatives. The matches are tried sequentially, from top to bottom. The first successful match causes evaluation of the corresponding alternative body, in the environment of the case expression extended by the bindings created during the matching of that alternative and by the $decllist_i$ associated with that alternative. If no match succeeds, the result is $\perp$. Pattern matching is described in Section 3.17, with the formal semantics of case expressions in Section 3.17.3.

3.14 Do Expressions

exp	→	`do {` *stmts* [`;`]`}`	(do expression)
stmts	→	*exp* [`;` *stmts*]	
	|	*pat* `<-` *exp* `;` *stmts*	
	|	`let` *decllist* `;` *stmts*	

A *do expression* provides a more readable syntax for monadic programming.

Translation: Do expressions satisfy these identities, which may be used as a translation into the kernel:

`do {`e`}`	=	e
`do {`e`;`*stmts*`}`	=	e `>> do {`*stmts*`}`
`do {`p `<-` e`;` *stmts*`}`	=	e `>>= \`p `-> do {`*stmts*`}` where p is failure-free

`do {`p `<-` e`;` *stmts*`}`	=	`let ok` p `= do {`*stmts*`}`
		 `ok _ = zero`
		 `in` e `>>= ok`
		 where p is not failure-free

`do {let` decllist`;` *stmts*`}`	=	`let` decllist `in do {`*stmts*`}`

`>>`, `>>=`, and `zero` are operations in the classes `Monad` and `MonadZero`, as defined in the Prelude.

A failure-free pattern is one that can only be refuted by $\perp$. Failure-free patterns are defined as follows:

- All irrefutable patterns are failure-free (irrefutable patterns are described in Section 3.17.1).
- If C is the only constructor in its type, then $C\ p_1\ \ldots p_n$ is failure-free when each of the p_i is failure free.

- If pattern p is failure-free, then the pattern v`@`p is failure-free.

This translation requires a monad in class `MonadZero` if any pattern bound by `<-` is not failure-free. Otherwise, only class methods from `Monad` are generated. Type errors resulting from patterns which are not failure-free can be corrected by using `~` to force the pattern to be failure-free.

As indicated by the translation of `do`, variables bound by `let` have fully polymorphic types while those defined by `<-` are lambda bound and are thus monomorphic.

3.15 Datatypes with Field Labels

A datatype declaration may optionally include field labels for some or all of the components of the type (see Section 4.2.1). Readers unfamiliar with datatype declarations in Haskell may wish to read Section 4.2.1 first. These field labels can be used to construct, select from, and update fields in a manner that is independent of the overall structure of the datatype.

Different datatypes cannot share common field labels in the same scope. A field label can be used at most once in a constructor. Within a datatype, however, a field name can be used in more than one constructor provided the field has the same typing in all constructors.

3.15.1 Field Selection

$aexp \rightarrow qvar$

Field names are used as selector functions. When used as a variable, a field name serves as a function which extracts the field from an object. Selectors are top level bindings and so they may be shadowed by local variables but cannot conflict with other top level bindings of the same name. This shadowing only affects selector functions; in other record constructs, field labels cannot be confused with ordinary variables.

Translation: A field label f introduces a selector function defined as:

f `x` = `case x of {` $C_1\ p_{11}\ \ldots\ p_{1k}$ `->` e_1 `;` ... `;` $C_n\ p_{n1}\ \ldots\ p_{nk}$ `->` e_n `}`
where $C_1 \ldots C_n$ are all the constructors of the datatype containing a field labeled with f, p_{ij} is `y` when f labels the jth component of C_i or `_` otherwise, and e_i is `y` when some field in C_i has a label of f or `undefined` otherwise.

3.15.2 Construction Using Field Labels

$aexp$	$\rightarrow$	$qcon$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
$fbind$	$\rightarrow$	$var \mid var$ `=` exp	

A constructor with labeled fields may be used to construct a value in which the components are specified by name rather than by position. Unlike the braces used in declaration lists, these are not subject to layout; the `{` and `}` characters must be explicit. (This is also true of field updates and field patterns.) Construction using field names is subject to the following constraints:

- Only field labels declared with the specified constructor may be mentioned.
- A field name may not be mentioned more than once.
- Fields not mentioned are initialized to $\perp$.
- When the `=` *exp* is omitted and there is a variable with the same name as the field label in scope, the field is initialized to the value of that variable.
- A compile-time error occurs when any strict fields (fields whose declared types are prefixed by `!`) are omitted during construction. Strict fields are discussed in Section 4.2.1.

Translation: In the binding f `=` v, the field f labels v. Any binding f that omits the `=` v is expanded to f `=` f.

$$C \texttt{ \{ } bs \texttt{ \}} \quad = \quad C\ (pick^C_1\ bs\ \texttt{undefined})\ \ldots\ (pick^C_k\ bs\ \texttt{undefined})$$

k is the arity of C.
The auxiliary function $pick^C_i\ bs\ d$ is defined as follows:

> If the ith component of a constructor C has the field name f, and if $f = v$ appears in the binding list bs, then $pick^C_i\ bs\ d$ is v. Otherwise, $pick^C_i\ bs\ d$ is the default value d.

3.15.3 Updates Using Field Labels

$$aexp \quad \rightarrow \quad aexp_{\langle qcon\rangle} \texttt{ \{ } fbind_1 \texttt{ , } \ldots \texttt{ , } fbind_n \texttt{ \}} \qquad \text{(labeled update, } n \geq 1\text{)}$$

Values belonging to a datatype with field names may be non-destructively updated. This creates a new value in which the specified field values replace those in the existing value. Updates are restricted in the following ways:

- All labels must be taken from the same datatype.
- At least one constructor must define all of the labels mentioned in the update.
- No label may be mentioned more than once.
- An execution error occurs when the value being updated does not contain all of the specified labels.

- When the `=` *exp* is omitted, the field is updated to the value of the variable in scope with the same name as the field label.

Translation: Using the prior definition of *pick*,

$$
\begin{array}{lll}
e \; \texttt{\{} \; bs \; \texttt{\}} & = & \texttt{case}\ e\ \texttt{of} \\
 & & \quad C_1\ v_1\ \ldots\ v_{k_1}\ \texttt{->}\ C\ (pick^C_1\ bs\ v_1)\ \ldots\ (pick^C_k\ bs\ v_{k_1}) \\
 & & \quad \ldots \\
 & & \quad C_j\ v_1\ \ldots\ v_{k_j}\ \texttt{->}\ C\ (pick^C_1\ bs\ v_1)\ \ldots\ (pick^C_k\ bs\ v_{k_j}) \\
 & & \quad \texttt{\_ -> error "Update error"}
\end{array}
$$

where $\{C_1, \ldots, C_j\}$ is the set of constructors containing all labels in b, and k_i is the arity of C_i.

Here are some examples using labeled fields:

```
data T    = C1 {f1,f2 :: Int}
          | C2 {f1 :: Int,
                f3,f4 :: Char}
```

Expression	Translation
`C1 {f1 = 3}`	`C1' 3 undefined`
`C2 {f1 = 1, f4 = 'A', f3 = 'B'}`	`C2' 1 'B' 'A'`
`x {f1 = 1}`	`case x of C1' _ f2 -> C1' 1 f2` `           C2' _ f3 f4 -> C2' 1 f3 f4`

The field `f1` is common to both constructors in `T`. The constructors `C1'` and `C2'` are 'hidden constructors', see the translation in Section 4.2.1. A compile-time error will result if no single constructor defines the set of field names used in an update, such as `x {f2 = 1, f3 = 'x'}`.

3.16 Expression Type-Signatures

exp → *exp* `::` [*context* `=>`] *type*

Expression type-signatures have the form *e* `::` *t*, where *e* is an expression and *t* is a type (Section 4.1.1); they are used to type an expression explicitly and may be used to resolve ambiguous typings due to overloading (see Section 4.3.4). The value of the expression is just that of *exp*. As with normal type signatures (see Section 4.4.1), the declared type may be more specific than the principal type derivable from *exp*, but it is an error to give a type that is more general than, or not comparable to, the principal type.

3.17 Pattern Matching

Patterns appear in lambda abstractions, function definitions, pattern bindings, list comprehensions, do expressions, and case expressions However, the first five of these ultimately translate into case expressions, so defining the semantics of pattern matching for case expressions is sufficient.

3.17.1 Patterns

Patterns have this syntax:

pat	$\rightarrow$	*var* `+` *integer*	(successor pattern)
pat | pat^{0}			
pat^{i}	$\rightarrow$	pat^{i+1} [$qconop^{(\mathrm{n},i)}$ pat^{i+1}]	
	|	$lpat^{i}$	
	|	$rpat^{i}$	
$lpat^{i}$	$\rightarrow$	($lpat^{i}$ | pat^{i+1}) $qconop^{(\mathrm{l},i)}$ pat^{i+1}	
$lpat^{6}$	$\rightarrow$	`-` (*integer* | *float*)	(negative literal)
$rpat^{i}$	$\rightarrow$	pat^{i+1} $qconop^{(\mathrm{r},i)}$ ($rpat^{i}$ | pat^{i+1})	
pat^{10}	$\rightarrow$	*apat*	
	|	*gcon* $apat_1$ ... $apat_k$	(arity *gcon* $= k$, $k \geq 1$)
apat	$\rightarrow$	*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	$\rightarrow$	*var* `=` *pat*	
	|	*var*	

The arity of a constructor must match the number of sub-patterns associated with it; one cannot match against a partially-applied constructor.

All patterns must be *linear* —no variable may appear more than once.

Patterns of the form *var*`@`*pat* are called *as-patterns*, and allow one to use *var* as a name for the value being matched by *pat*. For example,

```
case e of { xs@(x:rest) -> if x==0 then rest else xs }
```

is equivalent to:

```
let { xs = e } in
  case xs of { (x:rest) -> if x == 0 then rest else xs }
```

Patterns of the form `_` are *wildcards* and are useful when some part of a pattern is not referenced on the right-hand-side. It is as if an identifier not used elsewhere were put in its place. For example,

```
case e of { [x,_,_]  ->  if x==0 then True else False }
```

is equivalent to:

```
case e of { [x,y,z]  ->  if x==0 then True else False }
```

In the pattern matching rules given below we distinguish two kinds of patterns: an *irrefutable pattern* is: a variable, a wildcard, N *apat* where N is a constructor defined by `newtype` and *apat* is irrefutable (see Section 4.2.3), *var*`@`*apat* where *apat* is irrefutable, or of the form `~`*apat* (whether or not *apat* is irrefutable). All other patterns are *refutable*.

3.17.2 Informal Semantics of Pattern Matching

Patterns are matched against values. Attempting to match a pattern can have one of three results: it may *fail*; it may *succeed*, returning a binding for each variable in the pattern; or it may *diverge* (i.e. return $\perp$). Pattern matching proceeds from left to right, and outside to inside, according to these rules:

1. Matching a value v against the irrefutable pattern *var* always succeeds and binds *var* to v. Similarly, matching v against the irrefutable pattern `~`*apat* always succeeds. The free variables in *apat* are bound to the appropriate values if matching v against *apat* would otherwise succeed, and to $\perp$ if matching v against *apat* fails or diverges. (Binding does *not* imply evaluation.)

   Matching any value against the wildcard pattern `_` always succeeds and no binding is done.

   Operationally, this means that no matching is done on an irrefutable pattern until one of the variables in the pattern is used. At that point the entire pattern is matched against the value, and if the match fails or diverges, so does the overall computation.

2. Matching a value *con* v against the pattern *con pat*, where *con* is a constructor defined by `newtype`, is equivalent to matching v against the pattern *pat*. That is, constructors associated with `newtype` serve only to change the type of a value.

3. Matching $\perp$ against a refutable pattern always diverges.

4. Matching a non-$\perp$ value can occur against three kinds of refutable patterns:

   (a) Matching a non-$\perp$ value against a pattern whose outermost component is a constructor defined by `data` fails if the value being matched was created by a different constructor. If the constructors are the same, the result of the match is the result of matching the sub-patterns left-to-right against the components of the data value: if all matches succeed, the overall match succeeds; the first to fail or diverge causes the overall match to fail or diverge, respectively.

   (b) Numeric literals are matched using the overloaded `==` function. The behavior of numeric patterns depends entirely on the definition of `==` for the type of object being matched.

(c) Matching a non-$\perp$ value x against a pattern of the form n+k (where n is a variable and k is a positive integer literal) succeeds if $x \geq k$, resulting in the binding of n to $x - k$, and fails if $x < k$. The behavior of n+k patterns depends entirely on the underlying definitions of `>=`, `fromInteger`, and `-` for the type of the object being matched.

5. Matching against a constructor using labeled fields is the same as matching ordinary constructor patterns except that the fields are matched in the order they are named in the field list. All fields listed must be declared by the constructor; fields may not be named more than once. Fields not named by the pattern are ignored (matched against `_`).

6. The result of matching a value v against an as-pattern *var*`@`*apat* is the result of matching v against *apat* augmented with the binding of *var* to v. If the match of v against *apat* fails or diverges, then so does the overall match.

Aside from the obvious static type constraints (for example, it is a static error to match a character against a boolean), these static class constraints hold: an integer literal pattern can only be matched against a value in the class `Num` and a floating literal pattern can only be matched against a value in the class `Fractional`. and a n+k pattern can only be matched against a value in the class `Integral`.

Many people feel that n+k patterns should not be used. These patterns may be removed or changed in future versions of Haskell. Compilers should support a flag which disables the use of these patterns.

Here are some examples:

1. If the pattern `[1,2]` is matched against `[0,⊥]`, then `1` *fails* to match against `0`, and the result is a failed match. But if `[1,2]` is matched against `[⊥,0]`, then attempting to match `1` against $\perp$ causes the match to *diverge*.

2. These examples demonstrate refutable vs. irrefutable matching:

```
(\ ~(x,y) -> 0) ⊥     ⇒    0
(\  (x,y) -> 0) ⊥     ⇒    ⊥

(\ ~[x] -> 0) []    ⇒    0
(\ ~[x] -> x) []    ⇒    ⊥

(\ ~[x,~(a,b)] -> x) [(0,1),⊥]    ⇒    (0,1)
(\ ~[x, (a,b)] -> x) [(0,1),⊥]    ⇒    ⊥

(\  (x:xs) -> x:x:xs) ⊥   ⇒   ⊥
(\ ~(x:xs) -> x:x:xs) ⊥   ⇒   ⊥:⊥:⊥
```

Additional examples illustrating some of the subtleties of pattern matching may be found in Section 4.2.3.

```
(a)  case e of { alts } = (\v -> case v of { alts }) e
     where v is a completely new variable
(b)  case v of { p1  match1;  ... ; pn  matchn }
     =  case v of { p1  match1 ;
                    _  -> ... case v of {
                                  pn  matchn
                                  _  -> error "No match" }...}
      where each matchi has the form:
       | gi,1  -> ei,1  ;  ...  ;  | gi,mi  -> ei,mi  where { declsi }
(c)  case v of { p | g1 -> e1  ; ...
                   | gn -> en where { decls }
                  _       -> e' }
     = case e' of
       {y ->  (where y is a completely new variable)
        case v of {
              p -> let { decls } in
                     if g1 then e1 ... else if gn then en else y
              _ -> y }}
(d)  case v of { ~p -> e; _ -> e' }
     = (\x'1 ... x'n -> e1 ) (case v of { p-> x1 }) ... (case v of { p -> xn})
     where e1 = e [x'1/x1, ..., x'n/xn]
     x1, ..., xn are all the variables in p; x'1, ..., x'n are completely new variables
(e)  case v of { x@p -> e; _ -> e' }
     =  case v of { p -> ( \ x -> e ) v ; _ -> e' }
(f)  case v of { _ -> e; _ -> e' } = e
```

Figure 3: Semantics of Case Expressions, Part 1

Top level patterns in case expressions and the set of top level patterns in function or pattern bindings may have zero or more associated *guards*. A guard is a boolean expression that is evaluated only after all of the arguments have been successfully matched, and it must be true for the overall pattern match to succeed. The environment of the guard is the same as the right-hand-side of the case-expression alternative, function definition, or pattern binding to which it is attached.

The guard semantics have an obvious influence on the strictness characteristics of a function or case expression. In particular, an otherwise irrefutable pattern may be evaluated because of a guard. For example, in

```
f ~(x,y,z) [a] | a==y = 1
```

both `a` and `y` will be evaluated by a standard definition of `==`.

3.17.3 Formal Semantics of Pattern Matching

The semantics of all pattern matching constructs other than `case` expressions are defined by giving identities that relate those constructs to `case` expressions. The semantics of `case` expressions themselves are in turn given as a series of identities, in Figures 3–4. Any implementation should behave so that these identities hold; it is not expected that it will use them directly, since that would generate rather inefficient code.

In Figures 3–4: e, e' and e_i are expressions; g and g_i are boolean-valued expressions; p and p_i are patterns; v, x, and x_i are variables; K and K' are algebraic datatype (`data`) constructors (including tuple constructors); N is a `newtype` constructor;

and k is a character, string, or numeric literal.

Rule (b) matches a general source-language `case` expression, regardless of whether it actually includes guards—if no guards are written, then `True` is substituted for the guards $g_{i,j}$ in the $match_i$ forms. Subsequent identities manipulate the resulting `case` expression into simpler and simpler forms.

Rule (h) in Figure 4 involves the overloaded operator `==`; it is this rule that defines the meaning of pattern matching against overloaded constants.

These identities all preserve the static semantics. Rules (d), (e), and (j) use a lambda rather than a `let`; this indicates that variables bound by `case` are monomorphically typed (Section 4.1.3).

(g) $\texttt{case}\ v\ \texttt{of \{}\ K\ p_1 \ldots p_n\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$= \texttt{case}\ v\ \texttt{of \{}$
$\quad K\ x_1 \ldots x_n\ \texttt{->}\ \texttt{case}\ x_1\ \texttt{of \{}$
$\qquad p_1\ \texttt{->}\ \ldots\ \texttt{case}\ x_n\ \texttt{of \{}\ p_n\ \texttt{->}\ e\ \texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}\ \ldots$
$\qquad \texttt{\_ ->}\ e'\ \texttt{\}}$
$\quad \texttt{\_ ->}\ e'\ \texttt{\}}$
at least one of $p_1, \ldots, p_n$ is not a variable; $x_1, \ldots, x_n$ are new variables

(h) $\texttt{case}\ v\ \texttt{of \{}\ k\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}} = \texttt{if (}v\ \texttt{==}\ k\texttt{) then}\ e\ \texttt{else}\ e'$

(i) $\texttt{case}\ v\ \texttt{of \{}\ x\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}} = \texttt{case}\ v\ \texttt{of \{}\ x\ \texttt{->}\ e\ \texttt{\}}$

(j) $\texttt{case}\ v\ \texttt{of \{}\ x\ \texttt{->}\ e\ \texttt{\}} = \texttt{(\textbackslash}\ x\ \texttt{->}\ e\ \texttt{)}\ v$

(k) $\texttt{case}\ N\ v\ \texttt{of \{}\ N\ p\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$= \texttt{case}\ v\ \texttt{of \{}\ p\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
where N is a `newtype` constructor

(l) $\texttt{case}\ \bot\ \texttt{of \{}\ N\ p\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}} = \texttt{case}\ \bot\ \texttt{of \{}\ p\ \texttt{->}\ e\ \texttt{\}}$
where N is a `newtype` constructor

(m) $\texttt{case}\ v\ \texttt{of \{}\ K\ \texttt{\{}\ f_1\ \texttt{=}\ p_1\ \texttt{,}\ f_2\ \texttt{=}\ p_2\ \texttt{,}\ \ldots\texttt{\}}\ \texttt{->}\ e\ \texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$= \texttt{case}\ e'\ \texttt{of \{}$
$\quad y\ \texttt{->}$
$\quad \texttt{case}\ v\ \texttt{of \{}$
$\qquad K\ \texttt{\{}\ f_1\ \texttt{=}\ p_1\ \texttt{\} ->}$
$\qquad\quad \texttt{case}\ v\ \texttt{of \{}\ K\ \texttt{\{}\ f_2\ \texttt{=}\ p_2\ \texttt{,}\ \ldots\ \texttt{\} ->}\ e\ \texttt{;}\ \texttt{\_ ->}\ y\ \texttt{\};}$
$\qquad\quad \texttt{\_ ->}\ y\ \texttt{\}\}}$
where f_1, f_2, ... are fields of constructor K; y is a new variable

(n) $\texttt{case}\ v\ \texttt{of \{}\ K\ \texttt{\{}\ f\ \texttt{=}\ p\texttt{\}}\ \texttt{->}\ e\ \texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$= \texttt{case}\ v\ \texttt{of \{}$
$\quad K\ p_1\ \ldots\ p_n\ \texttt{->}\ e\ \texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
where p_i is p if f labels the ith component of K, `_` otherwise

(o) $\texttt{case (}K'\ e_1\ \ldots\ e_m\texttt{) of \{}\ K\ x_1\ \ldots\ x_n\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}} = e'$
where K and K' are distinct `data` constructors of arity n and m, respectively

(p) $\texttt{case (}K\ e_1\ \ldots\ e_n\texttt{) of \{}\ K\ x_1\ \ldots\ x_n\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$= \texttt{case}\ e_1\ \texttt{of \{}\ x'_1\ \texttt{->}\ \ldots\ \texttt{case}\ e_n\ \texttt{of \{}\ x'_n\ \texttt{->}\ e[x'_1/x_1 \ldots x'_n/x_n]\ \texttt{\}}\ldots\texttt{\}}$
where K is a constructor of arity n; $x'_1 \ldots x'_n$ are completely new variables

(q) $\texttt{case}\ e_0\ \texttt{of \{}\ x\texttt{+}k\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$= \texttt{if}\ e_0\ \texttt{>=}\ k\ \texttt{then let \{}x'\ \texttt{=}\ e_0\texttt{-}k\texttt{\} in}\ e[x'/x]\ \texttt{else}\ e'$ (x' is a new variable)

Figure 4: Semantics of Case Expressions, Part 2

4 Declarations and Bindings

In this section, we describe the syntax and informal semantics of Haskell *declarations*.

module	→	`module` *modid* [*exports*] `where` *body*	
	|	*body*	
body	→	`{` [*impdecls* `;`] [[*fixdecls* `;`] *topdecls* [`;`]] `}`	
	|	`{` *impdecls* [`;`] `}`	
topdecls	→	*topdecl*$_1$ `;` ... `;` *topdecl*$_n$	$(n \geq 0)$
topdecl	→	`type` *simpletype* `=` *type*	
	|	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
	|	`newtype` [*context* `=>`] *simpletype* `=` *con atype* [*deriving*]	
	|	`class` [*context* `=>`] *simpleclass* [`where {` *cbody* [`;`] `}`]	
	|	`instance` [*context* `=>`] *qtycls inst* [`where {` *valdefs* [`;`] `}`]	
	|	`default (`*type*$_1$ `,` ... `,` *type*$_n$`)`	$(n \geq 0)$
	|	*decl*	
decls	→	*decl*$_1$ `;` ... `;` *decl*$_n$	$(n \geq 0)$
decl	→	*signdecl*	
	|	*valdef*	
decllist	→	`{` *decls* [`;`] `}`	
signdecl	→	*vars* `::` [*context* `=>`] *type*	
vars	→	*var*$_1$ `,` ..., *var*$_n$	$(n \geq 1)$

The declarations in the syntactic category *topdecls* are only allowed at the top level of a Haskell module (see Section 5), whereas *decls* may be used either at the top level or in nested scopes (i.e. those within a `let` or `where` construct).

For exposition, we divide the declarations into three groups: user-defined datatypes, consisting of `type`, `newtype`, and `data` declarations (Section 4.2); type classes and overloading, consisting of `class`, `instance`, and `default` declarations (Section 4.3); and nested declarations, consisting of value bindings and type signatures (Section 4.4).

Haskell has several primitive datatypes that are "hard-wired" (such as integers and floating-point numbers), but most "built-in" datatypes are defined with normal Haskell code, using normal `type` and `data` declarations. These "built-in" datatypes are described in detail in Section 6.1.

4.1 Overview of Types and Classes

Haskell uses a traditional Hindley-Milner polymorphic type system to provide a static type semantics [3, 4], but the type system has been extended with *type* and *constructor* classes (or just *classes*) that provide a structured way to introduce *overloaded* functions.

A `class` declaration (Section 4.3.1) introduces a new *type class* and the overloaded operations that must be supported by any type that is an instance of that class. An `instance` declaration (Section 4.3.2) declares that a type is an *instance* of a class and includes the definitions of the overloaded operations—called *class methods*—instantiated on the named type.

For example, suppose we wish to overload the operations `(+)` and `negate` on types `Int` and `Float`. We introduce a new type class called `Num`:

```
class Num a  where          -- simplified class declaration for Num
  (+)    :: a -> a -> a
  negate :: a -> a
```

This declaration may be read "a type `a` is an instance of the class `Num` if there are (overloaded) class methods `(+)` and `negate`, of the appropriate types, defined on it."

We may then declare `Int` and `Float` to be instances of this class:

```
instance Num Int  where     -- simplified instance of Num Int
  x + y       =  addInt x y
  negate x    =  negateInt x

instance Num Float  where   -- simplified instance of Num Float
  x + y       =  addFloat x y
  negate x    =  negateFloat x
```

where `addInt`, `negateInt`, `addFloat`, and `negateFloat` are assumed in this case to be primitive functions, but in general could be any user-defined function. The first declaration above may be read "`Int` is an instance of the class `Num` as witnessed by these definitions (i.e. class methods) for `(+)` and `negate`."

More examples of type and constructor classes can be found in the papers by Jones [7] or Wadler and Blott [11]. The term 'type class' was used to describe the original Haskell 1.0 type system; 'constructor class' was used to describe an extension to the original type classes. There is no longer any reason to use two different terms: in this report, 'type class' includes both the original Haskell type classes and the constructor classes introduced by Jones.

4.1.1 Syntax of Types

type	→	*btype* [`->` *type*]	(function type)
btype	→	[*btype*] *atype*	(type application)
atype	→	*gtycon*	
	|	*tyvar*	
	|	`(` $type_1$ `,` ... `,` $type_k$ `)`	(tuple type, $k \geq 2$)
	|	`[` *type* `]`	(list type)
	|	`(` *type* `)`	(parenthesised constructor)

gtycon	→	*qtycon*	
	|	`()`	(unit type)
	|	`[]`	(list constructor)
	|	`(->)`	(function constructor)
	|	`(,{,})`	(tupling constructors)

The syntax for Haskell type expressions is given above. Just as data values are built using data constructors, type values are built from *type constructors*. As with data constructors, the names of type constructors start with uppercase letters.

To ensure that they are valid, type expressions are classified into different *kinds* which take one of two possible forms:

- The symbol $*$ represents the kind of all nullary type constructors.
- If κ_1 and κ_2 are kinds, then $\kappa_1 \rightarrow \kappa_2$ is the kind of types that take a type of kind κ_1 and return a type of kind κ_2.

The main forms of type expression are as follows:

1. Type variables, written as identifiers beginning with a lowercase letter. The kind of a variable is determined implicitly by the context in which it appears.
2. Type constructors. Most type constructors are written as identifiers beginning with an uppercase letter. For example:
   - `Char`, `Int`, `Integer`, `Float`, `Double` and `Bool` are type constants with kind $*$.
   - `Maybe` and `IO` are unary type constructors, and treated as types with kind $* \rightarrow *$.
   - A datatype declaration `data T ...` adds the type constructor `T` to the type vocabulary. The kind of `T` is determined by kind inference.

   Special syntax is provided for some type constructors:

   - The *trivial type* is written as `()` and has kind $*$. It denotes the "nullary tuple" type, and has exactly one value, also written `()` (see Sections 3.9 and 6.1.5).
   - The *function type* is written as `(->)` and has kind $* \rightarrow * \rightarrow *$.
   - The *list type* is written as `[]` and has kind $* \rightarrow *$.
   - The *tuple types* are written as `(,)`, `(,,)`, and so on. Their kinds are $* \rightarrow * \rightarrow *$, $* \rightarrow * \rightarrow * \rightarrow *$, and so on.

   Use of the `(->)` and `[]` constants is described in more detail below.
3. Type application. If t_1 is a type of kind $\kappa_1 \rightarrow \kappa_2$ and t_2 is a type of kind κ_1, then $t_1\ t_2$ is a type expression of kind κ_2.
4. A *parenthesised type*, having form `(`t`)`, is identical to the type t.

For example, the type expression `IO a` can be understood as the application of a constant, `IO`, to the variable `a`. Since the `IO` type constructor has kind $* \rightarrow *$, it follows that both the variable `a` and the whole expression, `IO a`, must have kind $*$. In general, a process of *kind inference* (see Section 4.6) is needed to determine appropriate kinds for user-defined datatypes, type synonyms, and classes.

Special syntax is provided to allow certain type expressions to be written in a more traditional style:

1. A *function type* has the form t_1 `->` t_2, which is equivalent to the type `(->)` t_1 t_2. Function arrows associate to the right.
2. A *tuple type* has the form `(`t_1 `,` ... `,` t_k`)` where $k \geq 2$, which is equivalent to the type `(,...,)` t_1 ... t_k where there are $k-1$ commas between the parenthesis. It denotes the type of k-tuples with the first component of type t_1, the second component of type t_2, and so on (see Sections 3.8 and 6.1.4).
3. A *list type* has the form `[`t`]` which is equivalent to the type `[]` t. It denotes the type of lists with elements of type t (see Sections 3.7 and 6.1.3).

Although the tuple, list, and function types have special syntax, they are not different from user-defined types with equivalent functionality.

Expressions and types have a consistent syntax. If t_i is the type of expression or pattern e_i, then the expressions `(\` e_1 `->` e_2`)`, `[`e_1`]`, and `(`e_1`,`e_2`)` have the types `(`t_1 `->` t_2`)`, `[`t_1`]`, and `(`t_1`,`t_2`)`, respectively.

With one exception, the type variables in a Haskell type expression are all assumed to be universally quantified; there is no explicit syntax for universal quantification [3]. For example, the type expression `a -> a` denotes the type $\forall\, a.\ a \rightarrow a$. For clarity, however, we often write quantification explicitly when discussing the types of Haskell programs.

The exception referred to is that of the distinguished type variable in a class declaration (Section 4.3.1).

4.1.2 Syntax of Class Assertions and Contexts

context	$\rightarrow$	*class*	
	|	(*class*$_1$, ... , *class*$_n$)	$(n \geq 1)$
class	$\rightarrow$	*qtycls tyvar*	
simpleclass	$\rightarrow$	*tycls tyvar*	
tycls	$\rightarrow$	*conid*	
tyvar	$\rightarrow$	*varid*	

A *class assertion* has form *qtycls tyvar*, and indicates the membership of the parameterized type *tyvar* in the class *qtycls*. A class identifier begins with an uppercase letter.

A *context* consists of one or more class assertions, and has the general form

$$(C_1\ u_1 ,\ \ldots,\ C_n\ u_n)$$

where $C_1 ,\ \ldots,\ C_n$ are class identifiers, and $u_1 ,\ \ldots,\ u_n$ are type variables; the parentheses may be omitted when $n = 1$. In general, we use c to denote a context and we write c `=>` t to indicate the type t restricted by the context c. The context c must only contain type variables referenced in t. For convenience, we write c `=>` t even if the context c is empty, although in this case the concrete syntax contains no `=>`.

4.1.3 Semantics of Types and Classes

In this subsection, we provide informal details of the type system. (Wadler and Blott [11] and Jones [7] discuss type and constructor classes, respectively, in more detail.)

The Haskell type system attributes a *type* to each expression in the program. In general, a type is of the form $\forall\ \overline{u}.\ c\ \Rightarrow\ t$, where $\overline{u}$ is a set of type variables $u_1 ,\ \ldots,\ u_n$. In any such type, any of the universally-quantified type variables u_i which are free in c must also be free in t. Furthermore, the context c must be of the form given above in Section 4.1.2; that is, it must have the form ($C_1\ u_1 ,\ \ldots,\ C_n\ u_n$) where $C_1 ,\ \ldots,\ C_n$ are class identifiers, and $u_1 ,\ \ldots,\ u_n$ are type variables.

The type of an expression e depends on a *type environment* that gives types for the free variables in e, and a *class environment* that declares which types are instances of which classes (a type becomes an instance of a class only via the presence of an `instance` declaration or a `deriving` clause).

Types are related by a generalization order (specified below); the most general type that can be assigned to a particular expression (in a given environment) is called its *principal type*. Haskell's extended Hindley-Milner type system can infer the principal type of all expressions, including the proper use of overloaded class methods (although certain ambiguous overloadings could arise, as described in Section 4.3.4). Therefore, explicit typings (called *type signatures*) are usually optional (see Sections 3.16 and 4.4.1).

The type $\forall\ \overline{u}.\ c_1\ \Rightarrow\ t_1$ is *more general than* the type $\forall\ \overline{w}.\ c_2\ \Rightarrow\ t_2$ if and only if there is a substitution S whose domain is $\overline{u}$ such that:

- t_2 is identical to $S(t_1)$.
- Whenever c_2 holds in the class environment, $S(c_1)$ also holds.

The main point about contexts above is that, given the type $\forall\ \overline{u}.\ c\ \Rightarrow\ t$, the presence of $C\ u_i$ in the context c expresses the constraint that the type variable u_i may be instantiated as t' within the type expression t only if t' is a member of the class C. For example, consider the function `double`:

```
double x = x + x
```

The most general type of `double` is $\forall\ a.$ `Num` $a\ \Rightarrow\ a\ \rightarrow\ a$. `double` may be applied to values of type `Int` (instantiating a to `Int`), since `Int` is an instance of the class `Num`. However, `double` may not be applied to values of type `Char`, because `Char` is not an instance of class `Num`.

4.2 User-Defined Datatypes

In this section, we describe algebraic datatypes (`data` declarations), renamed datatypes (`newtype` declarations), and type synonyms (`type` declarations). These declarations may only appear at the top level of a module.

4.2.1 Algebraic Datatype Declarations

topdecl	→	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
simpletype	→	*tycon* $tyvar_1$ … $tyvar_k$	
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* `{` $fielddecl_1$ `,` … `,` $fielddecl_n$ `}`	$(n \geq 1)$
fielddecl	→	*vars* `::` (*type* | `!` *atype*)	
deriving	→	`deriving` (*dclass* | `(`$dclass_1$ `,` … `,` $dclass_n$`)`)$(n \geq 0)$	
dclass	→	*qtycls*	

The precedence for *constr* is the same as that for expressions—normal constructor application has higher precedence than infix constructor application (thus `a : Foo a` parses as `a : (Foo a)`).

An algebraic datatype declaration introduces a new type and constructors over that type and has the form:

$$\texttt{data}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ K_1\ t_{11}\ \ldots\ t_{1k_1}\ \texttt{|}\ \cdots\ \texttt{|}\ K_n\ t_{n1}\ \ldots\ t_{nk_n}$$

where c is a context. This declaration introduces a new type constructor T with constituent data constructors $K_1, \ldots, K_n$ whose types are given by:

$$K_i :: \forall\ u_1\ \ldots\ u_k.\ c_i \Rightarrow t_{i1} \rightarrow \cdots \rightarrow t_{ik_i} \rightarrow (T\ u_1\ \ldots\ u_k)$$

where c_i is the largest subset of c that constrains only those type variables free in the types $t_{i1}, \ldots, t_{ik_i}$. The type variables u_1 through u_k must be distinct and may appear in c and the t_{ij}; it is a static error for any other type variable to appear in c or on the right-hand-side. The new type constant T has a kind of the form $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow *$ where the kinds κ_i of the argument variables u_i are determined by kind inference as described in Section 4.6. This means that T may be used in type expressions with anywhere between 0 and k arguments.

For example, the declaration

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

introduces a type constructor `Set` of kind $* \rightarrow *$, and constructors `NilSet` and `ConsSet` with types

$$\texttt{NilSet} :: \forall\ a.\ \texttt{Set}\ a$$
$$\texttt{ConsSet} :: \forall\ a.\ \texttt{Eq}\ a \Rightarrow a \rightarrow \texttt{Set}\ a \rightarrow \texttt{Set}\ a$$

In the example given, the overloaded type for `ConsSet` ensures that `ConsSet` can only be applied to values whose type is an instance of the class `Eq`. The context in the `data` declaration has no other effect whatsoever.

The visibility of a datatype's constructors (i.e. the "abstractness" of the datatype) outside of the module in which the datatype is defined is controlled by the form of the datatype's name in the export list as described in Section 5.5.

The optional `deriving` part of a `data` declaration has to do with *derived instances*, and is described in Section 4.3.3.

Labeled Fields

A data constructor of arity k creates an object with k components. These components are normally accessed positionally as arguments to the constructor in expressions or patterns. For large datatypes it is useful to assign *field labels* to the components of a data object. This allows a specific field to be referenced independently of its location within the constructor.

A constructor definition in a `data` declaration using the `{ }` syntax assigns labels to the components of the constructor. Constructors using field labels may be freely mixed with constructors without them. A constructor with associated field labels may still be used as an ordinary constructor; features using labels are simply a shorthand for operations using an underlying positional constructor. The arguments to the positional constructor occur in the same order as the labeled fields. For example, the declaration

```
data C = F { f1,f2 :: Int, f3 :: Bool}
```

defines a type and constructor identical to the one produced by

```
data C = F Int Int Bool
```

Operations using field labels are described in Section 3.15. A `data` declaration may use the same field label in multiple constructors as long as the typing of the field is the same in all cases after type synonym expansion. A label cannot be shared by more than one type in scope. Field names share the top level namespace with ordinary variables and class methods and must not conflict with other top level names in scope.

Strictness Flags

Whenever a data constructor is applied, each argument to the constructor is evaluated if and only if the corresponding type in the algebraic datatype declaration has a strictness flag (`!`).

Translation: A declaration of the form

$$\texttt{data}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ \ldots\ \texttt{|}\ K\ s_1\ \ldots\ s_n\ \texttt{|}\ \ldots$$

where each s_i is either of the form $\texttt{!}\,t_i$ or t_i, is replaced by a declaration of the form

$$\texttt{data}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ \ldots\ \texttt{|}\ K'\ t_1\ \ldots\ t_n\ \texttt{|}\ \ldots$$

each occurrence of K in a pattern is replaced by K', where K' is a 'hidden constructor' not directly available to the programmer. Each occurrence of K in an expression is replaced with

$$\texttt{(\textbackslash}\ x_1\ \ldots\ x_n\ \texttt{->}\ (\ ((K'\ op_1\ x_1)\ op_2\ x_2)\ \ldots\ )\ op_n\ x_n)$$

where op_i is the lazy apply function `$` if s_i is of the form t_i, and op_i is the strict apply function `` `strict` `` (see Section 6.2.7) if s_i is of the form `!` t_i.

Strictness flags may require the explicit inclusion of an `Eval` context in a `data` declaration (see Section 6.2.7). This occurs precisely when the context of a `strict` function used in the above translation propagates to a type variable. For example, in

```
data (Eval a) => Pair a b = MakePair !a b
```

the class assertion `(Eval a)` is required by the use of `strict` in the translation of the constructor `MakePair`. This context must be explicitly supplied by the programmer. The `Eval` context may be implied by a more general one; for example, the `Num` class includes `Eval` as a superclass to avoid mentioning `Eval` in the following:

```
data (Integral a) => Rational a = !a :% !a  -- Rational library
```

4.2.2 Type Synonym Declarations

topdecl	→	`type` *simpletype* `=` *type*
simpletype	→	*tycon* $tyvar_1$ ... $tyvar_k$

A type synonym declaration introduces a new type that is equivalent to an old type. It has the form

$$\texttt{type}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ t$$

which introduces a new type constructor, T. The type $(T\ t_1\ \ldots t_k)$ is equivalent to the type $t[t_1/u_1,\ \ldots,\ t_k/u_k]$. The type variables u_1 through u_k must be distinct and are scoped only over t; it is a static error for any other type variable to appear in t. The kind of the new type constructor T is of the form $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow \kappa$ where the kinds κ_i of the arguments u_i and κ of the right hand side t are determined by kind inference as described in Section 4.6. For example, the following definition can be used to provide an alternative way of writing the list type constructor:

```
type List = []
```

Type constructor symbols T introduced by type synonym declarations cannot be partially applied; it is a static error to use T without the full number of arguments.

Although recursive and mutually recursive datatypes are allowed, this is not so for type synonyms, *unless an algebraic datatype intervenes*. For example,

```
type Rec a   =  [Circ a]
data Circ a  =  Tag [Rec a]
```

is allowed, whereas

```
type Rec a   =  [Circ a]         -- invalid
type Circ a  =  [Rec a]          --
```

is not. Similarly, `type Rec a = [Rec a]` is not allowed.

4.2.3 Datatype Renamings

Type synonyms are a strictly syntactic mechanism to make type signatures more readable. A synonym and its definition are completely interchangeable.

topdecl	→	`newtype` [*context* `=>`] *simpletype* `=` *con atype* [*deriving*]
simpletype	→	*tycon* $tyvar_1$ ... $tyvar_k$

A declaration of the form

$$\texttt{newtype}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ N\ t$$

is valid if and only if

$$\texttt{data}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ N\ !t$$

is valid (see Section 4.2.1). The type `(` $T\ u_1 \ldots u_k$ `)` renames the datatype t.

A `newtype` declaration introduces a new type whose representation is the same as an existing type. It differs from a type synonym in that it creates a distinct type which must be explicitly coerced to or from the original type. The constructor N in an expression coerces a value from type t to type `(` $T\ u_1\ \ldots\ u_k$ `)`. Using N in a pattern coerces a value from type `(` $T\ u_1\ \ldots\ u_k$`)` to type t. These coercions are free of any execution time overhead; `newtype` does not change the underlying representation of an object.

New instances (see Section 4.3.2) can be defined for a type defined by `newtype` but may not be defined for a type synonym. A type created by `newtype` differs from an algebraic datatype in that the representation of an algebraic datatype has an extra level of indirection. This difference makes access to the representation less efficient. The difference is reflected in different rules for pattern matching (see Section 3.17). Unlike algebraic datatypes, the newtype constructor N is *unlifted*, so that $N\ \bot$ is the same as $\bot$.

The following examples clarify the differences between `data` (algebraic datatypes), `type` (type synonyms), and `newtype` (renaming types.) Given the declarations

```
data D1 = D1 Int
data D2 = D2 !Int
type S = Int
newtype N = N Int
d1 (D1 i) = 42
d2 (D2 i) = 42
s i = 42
n (N i) = 42
```

the expressions `( d1` $\bot$ `)`, `( d2` $\bot$ `)` and `(d2 (D2`$\bot$ `) )` are all equivalent to $\bot$, whereas `( n` $\bot$ `)`, `( n ( N` $\bot$ `) )`, `( d1 ( D1` $\bot$ `) )` and `( s` $\bot$ `)` are all equivalent to `42`. In particular, `( N` $\bot$ `)` is equivalent to $\bot$ while `( D1` $\bot$ `)` is not equivalent to $\bot$.

The optional deriving part of a `newtype` declaration is treated in the same way as the deriving component of a `data` declaration; see Section 4.3.3.

4.3 Type Classes and Overloading

4.3.1 Class Declarations

topdecl	$\rightarrow$	`class` [*context* `=>`] *simpleclass* [`where {` *cbody* [;] `}`]	
cbody	$\rightarrow$	[*cmethods* [; *cdefaults*]]	
cmethods	$\rightarrow$	$signdecl_1$; ... ; $signdecl_n$	$(n \geq 1)$
cdefaults	$\rightarrow$	$valdef_1$; ... ; $valdef_n$	$(n \geq 1)$

A *class declaration* introduces a new class and the operations (*class methods*) on it. A class declaration has the general form:

`class` c `=>` $C\ u$ `where {` v_1 `::` c_1 `=>` t_1 ; ... ; v_n `::` c_n `=>` t_n ;
$valdef_1$; ... ; $valdef_m$ `}`

This introduces a new class name C; the type variable u is scoped only over the class method signatures in the class body. The context c specifies the superclasses of C, as described below; the only type variable that may be referred to in c is u. The class declaration introduces new *class methods* $v_1, \ldots, v_n$, whose scope extends outside the `class` declaration, with types:

$$v_i :: \forall u, \overline{w}.\ (C u, c_i) \Rightarrow t_i$$

The t_i must mention u; they may mention type variables $\overline{w}$ other than u, and the type of v_i is polymorphic in both u and $\overline{w}$. The c_i may constrain only $\overline{w}$; in particular, the c_i may not constrain u. For example:

```
class Foo a where
      op :: Num b => a -> b -> a
```

Here the type of `op` is $\forall\ a,\ b.$ (`Foo` a, `Num` b) $\Rightarrow\ a\ \rightarrow\ b\ \rightarrow\ a$.

Default class methods for any of the v_i may be included in the `class` declaration as a normal *valdef*; no other definitions are permitted. The default class method for v_i is used if no binding for it is given in a particular `instance` declaration (see Section 4.3.2).

Class methods share the top level namespace with variable bindings and field names; they must not conflict with other top level bindings in scope. That is, a class method can not have the same name as a top level definition, a field name, or another class method.

A `class` declaration with no `where` part may be useful for combining a collection of classes into a larger one that inherits all of the class methods in the original ones. For example:

```
class  (Read a, Show a) => Textual a
```

In such a case, if a type is an instance of all superclasses, it is not *automatically* an instance of the subclass, even though the subclass has no immediate class methods. The `instance` declaration must be given explicitly with no `where` part.

The superclass relation must not be cyclic; i.e. it must form a directed acyclic graph.

4.3.2 Instance Declarations

topdecl	→	`instance` [*context* `=>`] *qtycls* *inst* [`where` `{` *valdefs* [`;`] `}`]	
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
valdefs	→	$valdef_1$ `;` ... `;` $valdef_n$	($n \geq 0$)

An *instance declaration* introduces an instance of a class. Let

`class` c `=>` C u `where {` *cbody* `}`

be a `class` declaration. The general form of the corresponding instance declaration is:

`instance` c' `=>` C $(T\ u_1\ \ldots\ u_k)$ `where {` d `}`

where $k \geq 0$ and T is not a type synonym. The constructor being instanced, $(T\ u_1\ \ldots\ u_k)$, is a type constructor applied to simple type variables $u_1, \ldots u_k$, which must be distinct. This prohibits instance declarations such as:

```
instance C (a,a) where ...
instance C (Int,a) where ...
instance C [[a]] where ...
```

The constructor $(T\ u_1\ \ldots\ u_k)$ must have an appropriate kind for the class C; this can be determined using kind inference as described in Section 4.6. The declarations d may contain bindings only for the class methods of C. The declarations may not contain any type signatures since the class method signatures have already been given in the `class` declaration.

If no binding is given for some class method then the corresponding default class method in the `class` declaration is used (if present); if such a default does not exist then the class method of this instance is bound to `undefined` and no compile-time error results.

An `instance` declaration that makes the type T to be an instance of class C is called a *C-T instance declaration* and is subject to these static restrictions:

- A type may not be declared as an instance of a particular class more than once in the program.
- The class and type must have the same kind.
- Assume that the type variables in the instance type ($T\ u_1\ \ldots\ u_k$) satisfy the constraints in the instance context c'. Under this assumption, the following two conditions must also be satisfied:

  1. The constraints expressed by the superclass context $c[(T\ u1\ \ldots\ uk)/u]$ of C must be satisfied. In other words, T must be an instance of each of C's superclasses and the contexts of all superclass instances must be implied by c'.
  2. Any constraints on the type variables in the instance type that are required for the class method declarations in d to be well-typed must also be satisfied.

  In fact, except in pathological cases it is possible to infer from the instance declaration the most general instance context c' satisfying the above two constraints, but it is nevertheless mandatory to write an explicit instance context.

The following illustrates the restrictions imposed by superclass instances:

```
class Foo a => Bar a where ...

instance (Eq a, Show a) => Foo [a] where ...

instance Num a => Bar [a] where ...
```

This is perfectly valid. Since `Foo` is a superclass of `Bar`, the second instance declaration is only valid if `[a]` is an instance of `Foo` under the assumption `Num a`. The first instance declaration does indeed say that `[a]` is an instance of `Foo` under this assumption, because `Eq` and `Show` are superclasses of `Num`.

If the two instance declarations instead read like this:

```
instance Num a => Foo [a] where ...

instance (Eq a, Show a) => Bar [a] where ...
```

then the program would be invalid. The second instance declaration is valid only if `[a]` is an instance of `Foo` under the assumptions `(Eq a, Show a)`. But this does not hold, since `[a]` is only an instance of `Foo` under the stronger assumption `Num a`.

Further examples of `instance` declarations may be found in Appendix A.

4.3.3 Derived Instances

As mentioned in Section 4.2.1, `data` and `newtype` declarations contain an optional `deriving` form. If the form is included, then *derived instance declarations* are automatically generated for the datatype in each of the named classes. These instances are subject to the same restrictions as user-defined instances. When deriving a class C for a type T, instances for all superclasses of C must exist for T, either via an explicit `instance` declaration or by including the superclass in the `deriving` clause.

Derived instances provide convenient commonly-used operations for user-defined datatypes. For example, derived instances for datatypes in the class `Eq` define the operations `==` and `/=`, freeing the programmer from the need to define them.

The only classes in the Prelude for which derived instances are allowed are `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`, all defined in Figure 5, page 66. The precise details of how the derived instances are generated for each of these classes are provided in Appendix D, including a specification of when such derived instances are possible. Instances of class `Eval` are always implicitly derived for algebraic datatypes. The class `Eval` is may not be explicitly listed in a deriving form or defined by an explicit instance declaration. Classes defined by the standard libraries may also be derivable.

A static error results if it is not possible to derive an `instance` declaration over a class named in a `deriving` form. For example, not all datatypes can properly support class methods in `Enum`. It is also a static error to give an explicit `instance` declaration for a class that is also derived.

If the `deriving` form is omitted from a `data` or `newtype` declaration, then *no* instance declarations (except for `Eval`) are derived for that datatype; that is, omitting a `deriving` form is equivalent to including an empty deriving form: `deriving ()`.

4.3.4 Defaults for Overloaded Numeric Operations

topdecl → `default` (*type*$_1$, ... , *type*$_n$) $(n \geq 0)$

A problem inherent with Haskell-style overloading is the possibility of an ambiguous type. For example, using the `read` and `show` functions defined in Appendix D, and supposing that just `Int` and `Bool` are members of `Read` and `Show`, then the expression

```
let x = read "..." in show x    -- invalid
```

is ambiguous, because the types for `show` and `read`,

$$\begin{array}{lll} \texttt{show} & :: & \forall\, a.\, \texttt{Show}\ a \Rightarrow a \rightarrow \texttt{String} \\ \texttt{read} & :: & \forall\, a.\, \texttt{Read}\ a \Rightarrow \texttt{String} \rightarrow a \end{array}$$

could be satisfied by instantiating `a` as either `Int` in both cases, or `Bool`. Such expressions are considered ill-typed, a static error.

We say that an expression `e` is *ambiguously overloaded* if, in its type $\forall\, \overline{u}.\ c \Rightarrow t$, there is a type variable u in $\overline{u}$ which occurs in c but not in t. Such types are invalid.

For example, the earlier expression involving `show` and `read` is ambiguously overloaded since its type is $\forall\, a.\ \texttt{Show}\ a,\ \texttt{Read}\ a \Rightarrow \texttt{String}$.

Overloading ambiguity can only be circumvented by input from the user. One way is through the use of *expression type-signatures* as described in Section 3.16. For example, for the ambiguous expression given earlier, one could write:

```
let x = read "..." in show (x::Bool)
```

which disambiguates the type.

Occasionally, an otherwise ambiguous expression needs to be made the same type as some variable, rather than being given a fixed type with an expression type-signature. This is the purpose of the function `asTypeOf` (Appendix A): x `` `asTypeOf` `` y has the value of x, but x and y are forced to have the same type. For example,

```
approxSqrt x = encodeFloat 1 (exponent x `div` 2) `asTypeOf` x
```

(See Section 6.3.6.)

Ambiguities in the class `Num` are most common, so Haskell provides another way to resolve them—with a *default declaration*:

$$\texttt{default (}t_1\texttt{ , ... , }t_n\texttt{)}$$

where $n \geq 0$, and each t_i must be a monotype for which `Num` t_i holds. In situations where an ambiguous type is discovered, an ambiguous type variable is defaultable if at least one of its classes is a numeric class (that is, `Num` or a subclass of `Num`) and if all of its classes are defined in the Prelude or a standard library (Figures 6–7, pages 73–74 show the numeric classes, and Figure 5, page 66, shows the classes defined in the Prelude.) Each defaultable variable is replaced by the first type in the default list that is an instance of all the ambiguous variable's classes. It is a static error if no such type is found.

Only one default declaration is permitted per module, and its effect is limited to that module. If no default declaration is given in a module then it assumed to be:

```
default (Int, Double)
```

The empty default declaration `default ()` must be given to turn off all defaults in a module.

4.4 Nested Declarations

The following declarations may be used in any declaration list, including the top level of a module.

4.4.1 Type Signatures

signdecl → *vars* `::` [*context* `=>`] *type*

A type signature specifies types for variables, possibly with respect to a context. A type signature has the form:

$$v_1\texttt{, ..., }v_n\texttt{ :: }c\texttt{ => }t$$

which is equivalent to asserting v_i :: c => t for each i from 1 to n. Each v_i must have a value binding in the same declaration list that contains the type signature; i.e. it is invalid to give a type signature for a variable bound in an outer scope. Moreover, it is invalid to give more than one type signature for one variable.

As mentioned in Section 4.1.1, every type variable appearing in a signature is universally quantified over that signature, and hence the scope of a type variable is limited to the type signature that contains it. For example, in the following declarations

```
f :: a -> a
f x = x :: a                          -- invalid
```

the `a`'s in the two type signatures are quite distinct. Indeed, these declarations contain a static error, since `x` does not have type $\forall\ a.\ a$. (The type of `x` is dependent on the type of `f`; there is currently no way in Haskell to specify a signature for a variable with a dependant type; this is explained in Section 4.5.3.)

If a given program includes a signature for a variable f, then each use of f is treated as having the declared type. It is a static error if the same type cannot also be inferred for the defining occurrence of f.

If a variable f is defined without providing a corresponding type signature declaration, then each use of f outside its own declaration group (see Section 4.5) is treated as having the corresponding inferred, or *principal* type . However, to ensure that type inference is still possible, the defining occurrence, and all uses of f within its declaration group must have the same monomorphic type (from which the principal type is obtained by generalization, as described in Section 4.5.2).

For example, if we define

```
sqr x  =  x*x
```

then the principal type is `sqr` :: $\forall\ a.$ `Num` $a \Rightarrow a \rightarrow a$, which allows applications such as `sqr 5` or `sqr 0.1`. It is also valid to declare a more specific type, such as

```
sqr :: Int -> Int
```

but now applications such as `sqr 0.1` are invalid. Type signatures such as

```
sqr :: (Num a, Num b) => a -> b      -- invalid
sqr :: a -> a                        -- invalid
```

are invalid, as they are more general than the principal type of `sqr`.

Type signatures can also be used to support *polymorphic recursion*. The following definition is pathological, but illustrates how a type signature can be used to specify a type more general than the one that would be inferred:

```
data T a  =  K (T Int) (T a)
f         :: T a -> a
f (K x y) =  if f x == 1 then f y else undefined
```

If we remove the signature declaration, the type of `f` will be inferred as `T Int -> Int` due to the first recursive call for which the argument to `f` is `T Int`. Polymorphic recursion allows the user to supply the more general type signature, `T a -> a`.

4.4.2 Function and Pattern Bindings

decl	$\rightarrow$	*valdef*
valdef	$\rightarrow$	*lhs* `=` *exp* [`where` *decllist*]
	$\vert$	*lhs gdrhs* [`where` *decllist*]
lhs	$\rightarrow$	pat^{0}
	$\vert$	*funlhs*
funlhs	$\rightarrow$	*var apat* { *apat* }
	$\vert$	$pat^{i+1}\ varop^{(a,i)}\ pat^{i+1}$
	$\vert$	$lpat^{i}\ varop^{(\mathrm{l},i)}\ pat^{i+1}$
	$\vert$	$pat^{i+1}\ varop^{(\mathrm{r},i)}\ rpat^{i}$
gdrhs	$\rightarrow$	*gd* `=` *exp* [*gdrhs*]
gd	$\rightarrow$	`|` exp^{0}

We distinguish two cases within this syntax: a *pattern binding* occurs when *lhs* is *pat*; otherwise, the binding is called a *function binding*. Either binding may appear at the top-level of a module or within a `where` or `let` construct.

Function bindings. A function binding binds a variable to a function value. The general form of a function binding for variable x is:

$$
\begin{array}{llll}
x & p_{11} & \ldots\ p_{1k} & match_1 \\
\ldots & & & \\
x & p_{n1} & \ldots\ p_{nk} & match_n
\end{array}
$$

where each p_{ij} is a pattern, and where each $match_i$ is of the general form:

`=` e `where {` $decls$ `}`

or

$$
\begin{array}{lll}
\texttt{|}\ g_{i1} & \texttt{=}\ e_{i1} \\
\ldots & \\
\texttt{|}\ g_{im_i} & \texttt{=}\ e_{im_i} \\
 & \texttt{where \{}\ decls_i\ \texttt{\}}
\end{array}
$$

and where $n \geq 1$, $1 \leq i \leq n$, $m_i \geq 1$. The former is treated as shorthand for a particular case of the latter, namely:

`| True =` e `where {` $decls$ `}`

Note that all clauses defining a function must be contiguous, and the number of patterns in each clause must be the same. The set of patterns corresponding to each match must be *linear*—no variable is allowed to appear more than once in the entire set.

Alternative syntax is provided for binding functional values to infix operators. For example, these two function definitions are equivalent:

```
plus x y z = x+y+z
x `plus` y = \ z -> x+y+z
```

Translation: The general binding form for functions is semantically equivalent to the equation (i.e. simple pattern binding):

$$
\begin{array}{ll}
x\ x_1\ x_2\ \ldots\ x_k\ \texttt{=}\ \texttt{case}\ \texttt{(}x_1\texttt{,}\ \ldots\texttt{,}\ x_k\texttt{)}\ \texttt{of} & \texttt{(}p_{11}\texttt{,}\ \ldots\texttt{,}\ p_{1k}\texttt{)}\ match_1 \\
& \ldots \\
& \texttt{(}p_{m1}\texttt{,}\ \ldots\texttt{,}\ p_{mk}\texttt{)}\ match_m
\end{array}
$$

where the x_i are new identifiers.

Pattern bindings. A pattern binding binds variables to values. A *simple* pattern binding has form $p\ =\ e$. The pattern p is matched "lazily" as an irrefutable pattern , as if there were an implicit `~` in front of it. See the translation in Section 3.12.

The *general* form of a pattern binding is $p\ match$, where a *match* is the same structure as for function bindings above; in other words, a pattern binding is:

$$
\begin{array}{lll}
p & \texttt{|}\ g_1 & \texttt{=}\ e_1 \\
 & \texttt{|}\ g_2 & \texttt{=}\ e_2 \\
 & \ldots & \\
 & \texttt{|}\ g_m & \texttt{=}\ e_m \\
 & \multicolumn{2}{l}{\texttt{where \{}\ decls\ \texttt{\}}}
\end{array}
$$

Translation: The pattern binding above is semantically equivalent to this simple pattern binding:

$$
\begin{array}{lll}
p & \texttt{=} & \texttt{let}\ decls\ \texttt{in} \\
 & & \texttt{if}\ g_1\ \texttt{then}\ e_1\ \texttt{else} \\
 & & \texttt{if}\ g_2\ \texttt{then}\ e_2\ \texttt{else} \\
 & & \ldots \\
 & & \texttt{if}\ g_m\ \texttt{then}\ e_m\ \texttt{else error "Unmatched pattern"}
\end{array}
$$

4.5 Static Semantics of Function and Pattern Bindings

The static semantics of the function and pattern bindings of a `let` expression or `where` clause are discussed in this section.

4.5.1 Dependency Analysis

In general the static semantics are given by the normal Hindley-Milner inference rules. A *dependency analysis transformation* is first performed to enhance polymorphism. Two variables bound by value declarations are in the same *declaration group* if either

1. they are bound by the same pattern binding, or
2. their bindings are mutually recursive (perhaps via some other declarations which are also part of the group).

Application of the following rules causes each `let` or `where` construct (including the `where` defining the top level bindings in a module) to bind only the variables of a single declaration group, thus capturing the required dependency analysis:[2]

1. The order of declarations in `where`/`let` constructs is irrelevant.
2. `let` {d_1 ; d_2} `in` e = `let` {d_1} `in` (`let` {d_2} `in` e)
   (when no identifier bound in d_2 appears free in d_1)

4.5.2 Generalization

The Hindley-Milner type system assigns types to a `let`-expression in two stages. First, the right-hand side of the declaration is typed, giving a type with no universal quantification. Second, all type variables which occur in this type are universally quantified unless they are associated with bound variables in the type environment; this is called *generalization*. Finally, the body of the `let`-expression is typed.

For example, consider the declaration

```
f x = let g y = (y,y)
      in ...
```

The type of `g`'s definition is $a \rightarrow (a, a)$. The generalization step attributes to `g` the polymorphic type $\forall\ a.\ \ a \rightarrow (a, a)$, after which the typing of the "`...`" part can proceed.

When typing overloaded definitions, all the overloading constraints from a single declaration group are collected together, to form the context for the type of each variable declared in the group. For example, in the definition:

```
f x = let g1 x y = if x>y then show x else g2 y x
          g2 p q = g1 q p
      in ...
```

The types of the definitions of `g1` and `g2` are both $a \rightarrow a \rightarrow$ `String`, and the accumulated constraints are `Ord` a (arising from the use of `>`), and `Show` a (arising from the use of `show`).

[2] A similar transformation is described in Peyton Jones' book [10].

The type variables appearing in this collection of constraints are called the *constrained type variables*.

The generalization step attributes to both `g1` and `g2` the type

$$\forall\ a.\ (\texttt{Ord}\ a,\ \texttt{Show}\ a)\ \Rightarrow\ a\ \rightarrow\ a\ \rightarrow\ \texttt{String}$$

Notice that `g2` is overloaded in the same way as `g1` even though the occurrences of `>` and `show` are in the definition of `g1`.

If the programmer supplies explicit type signatures for more than one variable in a declaration group, the contexts of these signatures must be identical up to renaming of the type variables.

As mentioned in Section 4.1.3, the context of a type may constrain only type variables. Consider, for example, the definition:

```
f xs y  =  xs == [y]
```

Its type is given by

```
f :: Eq a => [a] -> a -> Bool
```

and not

```
f :: Eq [a] => [a] -> a -> Bool
```

Even though the equality is taken at the list type, the context must be simplified, using the instance declaration for `Eq` on lists, before generalization. If no such instance is in scope, a static error occurs.

At generalization, the context of the generalized type must be reducible. That is, class constraints must apply only to type variables, not more general type expressions. For example, the following example is invalid:

```
f x = show (return x)
```

The type of `return` is `Monad m => a -> m a`; the type of `show` is `Show a => a -> String`. The type of `f` should be `(Monad m, Show (m a)) => a -> String`. Since the context `Show (m a)` cannot be further reduced, generalization results in a static error.

4.5.3 Monomorphism

Sometimes it is not possible to generalize over all the type variables used in the type of the definition. For example, consider the declaration

```
f x = let g y z = ([x,y], z)
      in ...
```

In an environment where `x` has type a, the type of `g`'s definition is $a\ \rightarrow\ b\ \rightarrow\ (\texttt{[}a\texttt{]},b)$. The generalization step attributes to `g` the type $\forall\ b.\ \ a\ \rightarrow\ b\ \rightarrow\ (\texttt{[}a\texttt{]},b)$; only b can be universally quantified because a occurs in the type environment. We say that the type of `g` is *monomorphic in the type variable* a.

The effect of such monomorphism is that the first argument of all applications of `g` must be of a single type. For example, it would be valid for the "`...`" to be

```
(g True, g False)
```

(which would, incidentally, force `x` to have type `Bool`) but invalid for it to be

```
(g True, g 'c')
```

In general, a type $\forall\ \overline{u}.\ c\ \Rightarrow\ t$ is said to be *monomorphic* in the type variable a if a is free in $\forall\ \overline{u}.\ c\ \Rightarrow\ t$.

It is worth noting that the explicit type signatures provided by Haskell are not powerful enough to express types which include monomorphic type variables. For example, we cannot write

```
f x = let
          g :: a -> b -> ([a],b)
          g y z = ([x,y], z)
        in ...
```

because that would claim that `g` was polymorphic in both `a` and `b` (Section 4.4.1). In this program, `g` can only be given a type signature if its first argument is restricted to a type not involving type variables; for example

```
          g :: Int -> b -> ([Int],b)
```

This signature would also cause `x` to have type `Int`.

4.5.4 The Monomorphism Restriction

Haskell places certain extra restrictions on the generalization step, beyond the standard Hindley-Milner restriction described above, which further reduces polymorphism in particular cases.

The monomorphism restriction depends on the binding syntax of a variable. Recall that a variable is bound by either a *function binding* or a *pattern binding*, and that a *simple* pattern binding is a pattern binding in which the pattern consists of only a single variable (Section 4.4.2).

Two rules define the monomorphism restriction:

Rule 1. We say that a given declaration group is *unrestricted* if and only if:

- **(a):** every variable in the group is bound by a function binding or a simple pattern binding, *and*
- **(b):** an explicit type signature is given for every variable in the group which is bound by simple pattern binding.

The usual Hindley-Milner restriction on polymorphism is that only type variables free in the environment may be generalized. In addition, *the constrained type variables of a restricted declaration group may not be generalized* in the generalization step for that group. (Recall that a type variable is constrained if it must belong to some type class; see Section 4.5.2.)

Rule 2. The type of a variable exported from a module must be completely polymorphic; that is, it must not have any free type variables. It follows from Rule 1 that if all top-level declaration groups are unrestricted, then Rule 2 is automatically satisfied.

Rule 1 is required for two reasons, both of which are fairly subtle. First, it prevents computations from being unexpectedly repeated. For example, `genericLength` is a standard function (in library `List`) whose type is given by

```
genericLength :: Num a => [b] -> a
```

Now consider the following expression:

```
let { len = genericLength xs } in (len, len)
```

It looks as if `len` should be computed only once, but without Rule 1 it might be computed twice, once at each of two different overloadings. If the programmer does actually wish the computation to be repeated, an explicit type signature may be added:

```
let { len :: Num a => a; len = genericLength xs } in (len, len)
```

When non-simple pattern bindings are used, the types inferred are always monomorphic in their constrained type variables, irrespective of whether a type signature is provided. For example, in

```
(f,g) = ((+),(-))
```

both `f` and `g` are monomorphic regardless of any type signatures supplied for `f` or `g`.

Rule 1 also prevents ambiguity. For example, consider the declaration group

```
[(n,s)] = reads t
```

Recall that `reads` is a standard function whose type is given by the signature

```
reads :: (Read a) => String -> [(a,String)]
```

Without Rule 1, `n` would be assigned the type $\forall\ a.$ `Read` $a\ \Rightarrow\ a$ and `s` the type $\forall\ a.$ `Read` $a \Rightarrow$ `String`. The latter is an invalid type, because it is inherently ambiguous. It is not possible to determine at what overloading to use `s`. Rule 1 makes `n` and `s` monomorphic in a.

Lastly, Rule 2 is required because there is no way to enforce monomorphic use of an exported binding, except by performing type inference on modules outside the current module.

The monomorphism rule has a number of consequences for the programmer. Anything defined with function syntax usually generalizes as a function is expected to. Thus in

```
f x y = x+y
```

the function `f` may be used at any overloading in class `Num`. There is no danger of recomputation here. However, the same function defined with pattern syntax:

```
f = \x -> \y -> x+y
```

requires a type signature if `f` is to be fully overloaded. Many functions are most naturally defined using simple pattern bindings; the user must be careful to affix these with type signatures to retain full overloading. The standard prelude contains many examples of this:

```
sum             :: (Num a) => [a] -> a
sum             =  foldl (+) 0
```

4.6 Kind Inference

This section describes the rules that are used to perform *kind inference*, i.e. to calculate a suitable kind for each type constructor and class appearing in a given program.

The first step in the kind inference process is to arrange the set of datatype, synonym, and class definitions into dependency groups. This can be achieved in much the same way as the dependency analysis for value declarations that was described in Section 4.5. For example, the following program fragment includes the definition of a datatype constructor `D`, a synonym `S` and a class `C`, all of which would be included in the same dependency group:

```
data C a => D a = Foo (S a)
type S a = [D a]
class C a where
    bar :: a -> D a -> Bool
```

The kinds of variables, constructors, and classes within each group are determined using standard techniques of type inference and kind-preserving unification [7]. For example, in the definitions above, the parameter `a` appears as an argument of the function constructor `(->)` in the type of `bar` and hence must have kind $*$. It follows that both `D` and `S` must have kind $* \rightarrow *$ and that every instance of class `C` must have kind $*$.

It is possible that some parts of an inferred kind may not be fully determined by the corresponding definitions; in such cases, a default of $*$ is assumed. For example, we could assume an arbitrary kind κ for the `a` parameter in each of the following examples:

```
data App f a = A (f a)
data Tree a  = Leaf | Fork (Tree a) (Tree a)
```

This would give kinds $(\kappa \rightarrow *) \rightarrow \kappa \rightarrow *$ and $\kappa \rightarrow *$ for `App` and `Tree`, respectively, for any kind κ, and would require an extension to allow polymorphic kinds. Instead, using the default binding $\kappa = *$, the actual kinds for these two constructors are $(* \rightarrow *) \rightarrow * \rightarrow *$ and $* \rightarrow *$, respectively.

Defaults are applied to each dependency group without consideration of the ways in which particular type constructor constants or classes are used in later dependency groups or elsewhere in the program. For example, adding the the following definition to those above do not influence the kind inferred for `Tree` (by changing it to $(* \rightarrow *) \rightarrow *$, for instance), and instead generates a static error because the kind of `[]`, $* \rightarrow *$, does not match the kind $*$ that is expected for an argument of `Tree`:

```
type FunnyTree = Tree []      -- invalid
```

This is important because it ensures that each constructor and class are used consistently with the same kind whenever they are in scope.

5 Modules

A module defines a collection of values, datatypes, type synonyms, classes, etc. (see Section 4) in an environment created by a set of *imports*, resources brought into scope from other modules, and *exports* some of these resources, making them available to other modules. We use the term *entity* to refer to a value, type, or classes defined in, imported into, or perhaps exported from a module.

A Haskell *program* is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in module `Main`, and `main` must have type `IO ()` (see Section 7).

Modules may reference other modules via explicit `import` declarations, each giving the name of a module to be imported and specifying its entities to be imported. Modules may be mutually recursive.

The name-space for modules is flat, with each module being associated with a unique module name (which are Haskell identifiers beginning with a capital letter; i.e. *modid*). There is one distinguished module, `Prelude`, which is imported into all programs by default (see Section 5.3), plus a set of standard library modules which may be imported as required (see the Haskell Library Report[9]).

5.1 Module Structure

A module defines a mutually recursive scope containing declarations for value bindings, data types, type synonyms, classes, etc. (see Section 4).

module	→	`module` *modid* [*exports*] `where` *body*	
	|	*body*	
body	→	`{` [*impdecls* `;`] [[*fixdecls* `;`] *topdecls* [`;`]] `}`	
	|	`{` *impdecls* [`;`] `}`	
modid	→	*conid*	
impdecls	→	$impdecl_1$ `;` … `;` $impdecl_n$	$(n \geq 1)$
topdecls	→	$topdecl_1$ `;` … `;` $topdecl_n$	$(n \geq 0)$

A module begins with a header: the keyword `module`, the module name, and a list of entities (enclosed in round parentheses) to be exported. The header is followed by an optional list of `import` declarations that specify modules to be imported, optionally restricting the imported bindings. This is followed by an optional list of fixity declarations and the module body. The module body is simply a list of top-level declarations (*topdecls*), as described in Section 4.

An abbreviated form of module is permitted which consists only of the module body. If this is used, the header is assumed to be '`module Main(main) where`'. If the first lexeme in the abbreviated module is not a `{`, then the layout rule applies for the top level of the module.

5.1.1 Export Lists

exports	$\rightarrow$	`(` *export*$_1$ `,` ... `,` *export*$_n$ [`,`] `)`	$(n \geq 0)$
export	$\rightarrow$	*qvar*	
	$\mid$	*qtycon* [`(..)` $\mid$ `(` *qcname*$_1$ `,` ... `,` *qcname*$_n$ `)`]	$(n \geq 1)$
	$\mid$	*qtycls* [`(..)` $\mid$ `(` *qvar*$_1$ `,` ... `,` *qvar*$_n$ `)`]	$(n \geq 0)$
	$\mid$	`module` *modid*	
qcname	$\rightarrow$	*qvar* $\mid$ *qcon*	

An *export list* identifies the entities to be exported by a module declaration. A module implementation may only export an entity that it declares, or that it imports from some other module. If the export list is omitted, all values, types and classes defined in the module are exported, *but not those that are imported.*

Entities in an export list may be named as follows:

1. Ordinary values, whether declared in the module body or imported, may be named by giving the name of the value as a *qvarid*. Operators should be enclosed in parentheses to turn them into *qvarid*'s.

2. An algebraic datatype T declared by a `data` or `newtype` declaration may be named in one of three ways:

   - The form T names the type *but not the constructors or field names.* The ability to export a type without its constructors allows the construction of abstract datatypes (see Section 5.5).
   - The form T`(`*qcname*$_1$`,` ... `,`*qcname*$_n$`)`, where all and only the constructors and field names are listed without duplications, names the type and all its constructors and field names.
   - The abbreviated form T`(..)` names the type and all its constructors and field names.

   Data constructors and field names cannot be named in export lists in any other way.

3. A type synonym T declared by a `type` declaration may be named by the form T.

4. A class C with operations $f_1, \ldots, f_n$ declared in a `class` declaration may be named in one of three ways:

   - The form C names the class *but not the class methods.*
   - The form C`(`f_1`,` ... `,`f_n`)`, where all and only the class methods in that class are listed without duplications, names the class and all its methods.
   - The abbreviated form C`(..)` names the class and all its methods.

   Class methods may not be named in export lists in any other way.

5. The set of all entities brought into scope from a module m by one or more unqualified `import` declarations may be named by the form '`module` m', which is equivalent to listing all of the entities imported from the module. For example:

```
module Queue( module Stack, enqueue, dequeue ) where
    import Stack
    ...
```

   Here the module `Queue` uses the module name `Stack` in its export list to abbreviate all the entities imported from `Stack`.

6. A module can name its own local definitions in its export list using its name in the '`module` m' syntax. For example:

```
module Mod1(module Mod1, module Mod2) where
import Mod2
import Mod3
```

   Here module `Mod1` exports all local definitions as well as those from imported from `Mod2` but not those imported from `Mod3`.

The qualifier (Section 5.1.2) on a name only identifies the module an entity is imported from; this may be different from the module in which the entity is defined. For example, if module A exports B.c, this is referenced as '`A.c`', not '`A.B.c`'. In consequence, names in export lists must remain distinct after qualifiers are removed. For example:

```
module A ( B.f, C.f, g, B.g ) where   -- an invalid module
import qualified B(f,g)
import qualified C(f)
g = True
```

There are name clashes in the export list between `B.f` and `C.f` and between `g` and `B.g` even though there are no name clashes within module `A`.

5.1.2 Import Declarations

impdecl	→	`import` [`qualified`] *modid* [`as` *modid*] [*impspec*]	
impspec	→	`(` $import_1$ `,` ... `,` $import_n$ [`,`] `)`	$(n \geq 0)$
	|	`hiding (` $import_1$ `,` ... `,` $import_n$ [`,`] `)`	$(n \geq 0)$
import	→	*var*	
	|	*tycon* [`(..)` | `(` $cname_1$ `,` ... `,` $cname_n$ `)`]	$(n \geq 1)$
	|	*tycls* [`(..)` | `(` var_1 `,` ... `,` var_n `)`]	$(n \geq 0)$
cname	→	*var* | *con*	

The entities exported by a module may be brought into scope in another module with an `import` declaration at the beginning of the module. The `import` declaration names the module to be imported and optionally specifies the entities to be imported. A single module may be imported by more than one `import` declaration. Imported names serve as top level declarations: they scope over the entire body of the module but may be shadowed by local non-top-level bindings. The effect of multiple `import` declarations is cumulative: an entity is in scope if it named by any of the `import` declarations in a module. The ordering of imports is irrelevant.

Exactly which entities are to be imported can be specified in one of three ways:

1. The imported entities can be specified explicitly by listing them in parentheses. Items in the list have the same form as those in export lists, except qualifiers are not permitted and the '`module` *modid*' entity is not permitted.

   The list must name only entities exported by the imported module. The list may be empty, in which case nothing except the instances are imported.

2. Entities can be excluded by using the form `hiding(`$import_1$ `,` ... `,` $import_n$ `)`, which specifies that all entities exported by the named module should be imported except for those named in the list. The effect of multiple `import` declarations is strictly cumulative: hiding an entity on one import declaration does not prevent the same entity from being imported by another import from the same module.

3. Finally, if *impspec* is omitted then all the entities exported by the specified module are imported.

When an import declaration uses the `qualified` keyword, the names brought into scope must be prefixed by the name of the imported module (or a local alias, if an `as` clause is present). A *qualified name* is written as *modid*`.`*name*. This allows full programmer control of the unqualified namespace: a locally defined entity can share the same name as a qualified import:

```
module Ring where
import qualified Prelude  -- All Prelude names must be qualified

l1 + l2 = l1 ++ l2        -- This + differs from the one in the Prelude
l1 * l2 = nub (l1 + l2)

succ = (Prelude.+ 1)
```

The qualifier does not change the syntactic treatment of a name: `Prelude.+` is an infix operator with the same fixity as the definition of `+` in the Prelude. Qualifiers may be applied to names imported by an unqualified import; this allows a qualified import to be replaced with an unqualified one without forcing changes in the references to the imported names.

Imported modules may be assigned a local alias in the importing module using the `as` clause. For example, in

```
import qualified Complex as C
```

entities must be referenced using '`C.`' as a qualifier instead of '`Complex.`'. This also allows a different module to be substituted for `Complex` without changing the qualifiers used for the imported module. It is an error for more than one module in scope to use the same qualifier. Qualifiers can only be used for imported entities: locally defined names within a module may not include a qualifier.

Since qualifier names are part of the lexical syntax, no spaces are allowed between the qualifier and the name. Sample parses 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)

It may be that a particular entity is imported into a module by more than one route — for example, because it is exported by two modules, both of which are imported by a third module. Benign name clashes of this form are allowed, but it is a static error for two different entities to have the same name. When two entities have the same name, they are considered to be the same object if and only if they are defined by the same module. Two different qualified names may refer to the same entity; the name of the importing module does not affect the identity of an entity.

It is an error for two different entities to have the same name. This is valid:

```
module A
import B(f)
import qualified C(f)
```

as long as only one imported `f` is unqualified and `f` is not defined at the top level of `A`. Qualifiers are the only way to resolve name clashes between imported entities.

5.1.3 Importing and Exporting Instance Declarations

Instance declarations cannot be explicitly named on import or export lists. All instances in scope within a module are *always* exported and any import brings *all* instances in from the imported module. Thus, an instance declaration is in scope if and only if a chain of `import` declarations leads to the module containing the instance declaration. For example, `import M()` would not bring any new names in scope from module `M`, but would bring in any instance visible in `M`.

5.2 Closure

Every module in a Haskell program must be *closed*. That is, every name explicitly mentioned by the source code must be either defined locally or imported from another module. Entities which the compiler requires for type checking or other compile time analysis need not be imported if they are not mentioned by name. The Haskell compilation system is responsible for finding any information needed for compilation without the help of the programmer. That is, the import of a variable `x` does not require that the datatypes and classes in the signature of `x` be brought into the module along with `x` unless these entities are referenced by name in the user program. The Haskell system silently imports any information which must accompany an entity for type checking or any other purposes. Such entities need not even be explicitly exported: the following program is valid even though `T` does not escape `M1`:

```
module M1(x) where
data T = T
x = T

module M2 where
import M1(x)
y = x
```

In this example, there is no way to supply an explicit type signature for `y` since `T` is not in scope. Whether or not `T` is explicitly exported, module `M2` knows enough about `T` to correctly type check the program.

The type of an exported entity is unaffected by non-exported type synonyms. For example, in

```
module M(x) where
type T = Int
x :: T
x = 1
```

the type of `x` is both `T` and `Int`; these are interchangeable even when `T` is not in scope. That is, the definition of `T` is available to any module which encounters it whether or not the name `T` is in scope. The only reason to export `T` is to allow other modules to refer it by name; the type checker find the definition of `T` if needed whether or not it is exported.

5.3 Standard Prelude

Many of the features of Haskell are defined in Haskell itself as a library of standard datatypes, classes, and functions, called the "Standard Prelude." In Haskell, the Prelude is contained in the the module `Prelude`. There are also many predefined library modules which provide less frequently used functions and types. For example, arrays, tables, and most of the input/output are all part of the standard libraries. These are defined in the Haskell Library Report[9], a separate document. Separating libraries from the Prelude has the advantage of reducing the size and complexity of the Prelude, allowing it to be more easily assimilated, and increasing the space of useful names available to the programmer.

Prelude and library modules differ from other modules in that their semantics (but not their implementation) are a fixed part of the Haskell language definition. This means, for example, that a compiler may optimize calls to functions in the Prelude without being concerned that a future change to the program will alter the semantics of the Prelude function.

5.3.1 The `Prelude` Module

The `Prelude` module is imported automatically into all modules as if by the statement '`import Prelude`', if and only if it is not imported with an explicit `import` declaration. This provision for explicit import allows values defined in the Prelude to be hidden from the unqualified name space. The `Prelude` module is always available as a qualified import: an implicit '`import qualified Prelude`' is part of every module and names prefixed by '`Prelude.`' can always be used to refer to entities in the Prelude.

The semantics of the entities in `Prelude` is specified by an implementation of `Prelude` written in Haskell, given in Appendix A. Some datatypes (such as `Int`) and functions (such as `Int` addition) cannot be specified directly in Haskell. Since the treatment of such entities depends on the implementation, they are not formally defined in the appendix. The implementation of `Prelude` is also incomplete in its treatment of tuples: there should be an infinite family of tuples and their instance declarations, but the implementation only gives a scheme.

5.3.2 Shadowing Prelude Names

The rules about the Prelude have been cast so that it is possible to use Prelude names for nonstandard purposes; however, every module that does so must have an `import` declaration that makes this nonstandard usage explicit. For example:

```
module A where
import Prelude hiding (null)
null x = []
```

Module `A` redefines `null`, but it must indicate this by importing `Prelude` without `null`. Furthermore, `A` exports `null`, but every module that imports `null` unqualified from `A` must also hide `null` from `Prelude` just as `A` does. Thus there is little danger of accidentally shadowing Prelude names.

It is possible to construct and use a different module to serve in place of the Prelude. Other than the fact that it is implicitly imported, the Prelude is an ordinary Haskell module; it is special only in that some objects in the Prelude are referenced by special syntactic constructs. Redefining names used by the Prelude does not affect the meaning of these special constructs. For example, in

```
module B where
import qualified Prelude
import MyPrelude
      ...
```

`B` imports nothing from `Prelude`, but the explicit `import qualified Prelude` declaration prevents the automatic import of `Prelude`. `import MyPrelude` brings the non-standard prelude into scope. As before, the standard prelude names are hidden explicitly. Special syntax, such as lists or tuples, always refers to prelude entities: there is no way to redefine the meaning of `[x]` in terms of a different implementation of lists.

5.4 Separate Compilation

Depending on the Haskell implementation used, separate compilation of mutually recursive modules may require that imported modules contain additional information so that they may be referenced before they are compiled. Explicit type signatures for all exported values may be necessary to deal with mutual recursion. The precise details of separate compilation are not defined by this report.

5.5 Abstract Datatypes

The ability to export a datatype without its constructors allows the construction of abstract datatypes (ADTs). For example, an ADT for stacks could be defined as:

```
module Stack( StkType, push, pop, empty ) where
        data StkType a = EmptyStk | Stk a (StkType a)
        push x s = Stk x s
        pop (Stk _ s) = s
        empty = EmptyStk
```

Modules importing `Stack` cannot construct values of type `StkType` because they do not have access to the constructors of the type.

It is also possible to build an ADT on top of an existing type by using a `newtype` declaration. For example, stacks can be defined with lists:

```
module Stack( StkType, push, pop, empty ) where
        newtype StkType a = Stk [a]
        push x (Stk s) = Stk (x:s)
        pop (Stk (x:s)) = Stk s
        empty = Stk []
```

5.6 Fixity Declarations

fixdecls	$\rightarrow$	*fix*$_1$; ... ; *fix*$_n$	$(n \geq 1)$
fix	$\rightarrow$	`infixl` [*digit*] *ops*	
	$\vert$	`infixr` [*digit*] *ops*	
	$\vert$	`infix` [*digit*] *ops*	
ops	$\rightarrow$	*op*$_1$, ... , *op*$_n$	$(n \geq 1)$
op	$\rightarrow$	*varop* $\vert$ *conop*	

A fixity declaration gives the fixity and binding precedence of a set of operators. Fixity declarations must appear only at the start of a module and may only be given for identifiers defined in that module. Fixity declarations cannot subsequently be overridden, and an identifier can only have one fixity definition.

There are three kinds of fixity, non-, left- and right-associativity (`infix`, `infixl`, and `infixr`, respectively), and ten precedence levels, 0 to 9 inclusive (level 0 binds least tightly, and level 9 binds most tightly). If the *digit* is omitted, level 9 is assumed. Any operator lacking a fixity declaration is assumed to be `infixl 9` (See Section 3 for more on the use of fixities). Table 2 lists the fixities and precedences of the operators defined in the Prelude.

Prec-edence	Left associative operators	Non-associative operators	Right associative operators
9	`!!`		`.`
8			`^`, `^^`, `**`
7	`*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` ``		
6	`+`, `-`		
5		`\\`	`:`, `++`
4		`==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` ``, `` `notElem` ``	
3			`&&`
2			`||`
1			`>>`, `>>=`
0			`$`, `` `seq` ``

Table 2: Precedences and fixities of prelude operators

Fixity is a property of the name of an identifier or operator: the same fixity attaches to every occurrence of an operator name in a module, whether at the top level or rebound at an inner level. For example:

```
module Foo
import Bar
infix 3 `op`

f x = ... where p `op` q = ...
```

Here `` `op` `` has fixity 3 wherever it is in scope, provided `Bar` does not export the identifier `op`. If `Bar` does export `op`, then the example becomes invalid, because the fixity (or lack thereof) of `op` is defined in `Bar` (or wherever `Bar` imported `op` from). If `op` is imported as a qualified name from `Bar`, no conflict may occur: the fixity of a qualified name does not affect unqualified uses of the same name.

6 Predefined Types and Classes

The Haskell Prelude contains predefined classes, types, and functions which are implicitly imported into every Haskell program. In this section, we describe the types and classes found in the Prelude. Most functions are not described in detail here as they can easily be understood from their definitions as given in Appendix A. Other predefined types such as arrays, complex numbers, and rationals are defined in the Haskell Library Report.

6.1 Standard Haskell Types

These types are defined by the Haskell Prelude. Numeric types are described in Section 6.3. When appropriate, the Haskell definition of the type is given. Some definitions may not be completely valid on syntactic grounds but they faithfully convey the meaning of the underlying type.

6.1.1 Booleans

```
data  Bool  =  False | True deriving
                             (Read, Show, Eq, Ord, Enum, Bounded)
```

The boolean type `Bool` is an enumeration.The basic boolean functions are `&&` (and), `||` (or), and `not`. The name `otherwise` is defined as `True` to make guarded expressions more readable.

6.1.2 Characters and Strings

The character type `Char` is an enumeration and consists of 256 values, conforming to the ISO 8859-1 standard [6]. The lexical syntax for characters is defined in Section 2.5; character literals are nullary constructors in the datatype `Char`. Type `Char` is an instance of the classes `Read`, `Show`, `Eq`, `Ord`, `Enum`, and `Bounded`. The `toEnum` and `fromEnum` functions, standard functions over bounded enumerations, map characters onto `Int` values in the range [0 , 255].

Note that ASCII control characters each have several representations in character literals: numeric escapes, ASCII mnemonic escapes, and the `\^`X notation. In addition, there are the following equivalences: `\a` and `\BEL`, `\b` and `\BS`, `\f` and `\FF`, `\r` and `\CR`, `\t` and `\HT`, `\v` and `\VT`, and `\n` and `\LF`.

A *string* is a list of characters:

```
type  String  =  [Char]
```

Strings may be abbreviated using the lexical syntax described in Section 2.5. For example, `"A string"` abbreviates

```
[ 'A',' ','s','t','r', 'i','n','g']
```

6.1.3 Lists

```
data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
```

Lists are an algebraic datatype of two constructors, although with special syntax, as described in Section 3.7. The first constructor is the null list, written '`[]`' ("nil"), and the second is '`:`' ("cons"). The module `PreludeList` (see Appendix A.1) defines many standard list functions. Arithmetic sequences and list comprehensions, two convenient syntaxes for special kinds of lists, are described in Sections 3.10 and 3.11, respectively. Lists are an instance of classes `Read`, `Show`, `Eq`, `Ord`, `Monad`, `MonadZero`, and `MonadPlus`.

6.1.4 Tuples

Tuples are algebraic datatypes with special syntax, as defined in Section 3.8. Each tuple type has a single constructor. There is no upper bound on the size of a tuple. However, some Haskell implementations may restrict the size of tuples and limit the instances associated with larger tuples. The Prelude and libraries define tuple functions such as `zip` for tuples up to a size of 7. All tuples are instances of `Eq`, `Ord`, `Bounded`, `Read`, and `Show`. Classes defined in the libraries may also supply instances for tuple types. The constructor for a tuple is written by omitting the expressions surrounding the commas: thus `(x,y)` and `(,) x y` produce the same value. The following functions are defined for pairs (2-tuples): `fst`, `snd`, `curry`, and `uncurry`. Similar functions are not predefined for larger tuples.

6.1.5 The Unit Datatype

```
data  () = () deriving (Eq, Ord, Bounded, Enum, Read, Show)
```

The unit datatype `()` has one non-$\perp$ member, the nullary constructor `()`. See also Section 3.9.

6.1.6 The Void Datatype

```
data Void
```

The `Void` has no constructors; only $\perp$ is an instance of this type.

6.1.7 Function Types

Functions are an abstract type: no constructors directly create functional values. Functions are an instance of the `Show` class but not `Read`. The following simple functions are found the Prelude: `id`, `const`, `(.)`, `flip`, `($)`, and `until`.

6.1.8 The IO and IOError Types

The `IO` type serves as a tag for operations (actions) which interact with the outside world. The `IO` type is abstract: no constructors are visible to the user. `IO` is an instance of the `Monad` and `Show` classes. Section 7 describes I/O operations.

`IOError` is an abstract type representing errors raised by I/O operations. It is an instance of `Show` and `Eq`. Values of this type are constructed by the various I/O functions and are not presented in any further detail in this report. The Library Report contains many other I/O functions.

6.1.9 Other Types

```
data  Maybe a     =  Nothing | Just a   deriving (Eq, Ord, Read, Show)
data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)
data  Ordering    =  LT | EQ | GT deriving
                                  (Eq, Ord, Bounded, Enum, Read, Show)
```

The `Maybe` type is an instance of classes `Functor`, `Monad`, `MonadZero` and `MonadPlus`. The `Ordering` type is used by `compare` in the class `Ord`. The functions `maybe` and `either` are found in the Prelude.

6.2 Standard Haskell Classes

Figure 5 shows the hierarchy of Haskell classes defined in the Prelude and the Prelude types which are instances of these classes. The `Void` type is not mentioned in this figure since it is not a member of any classes.

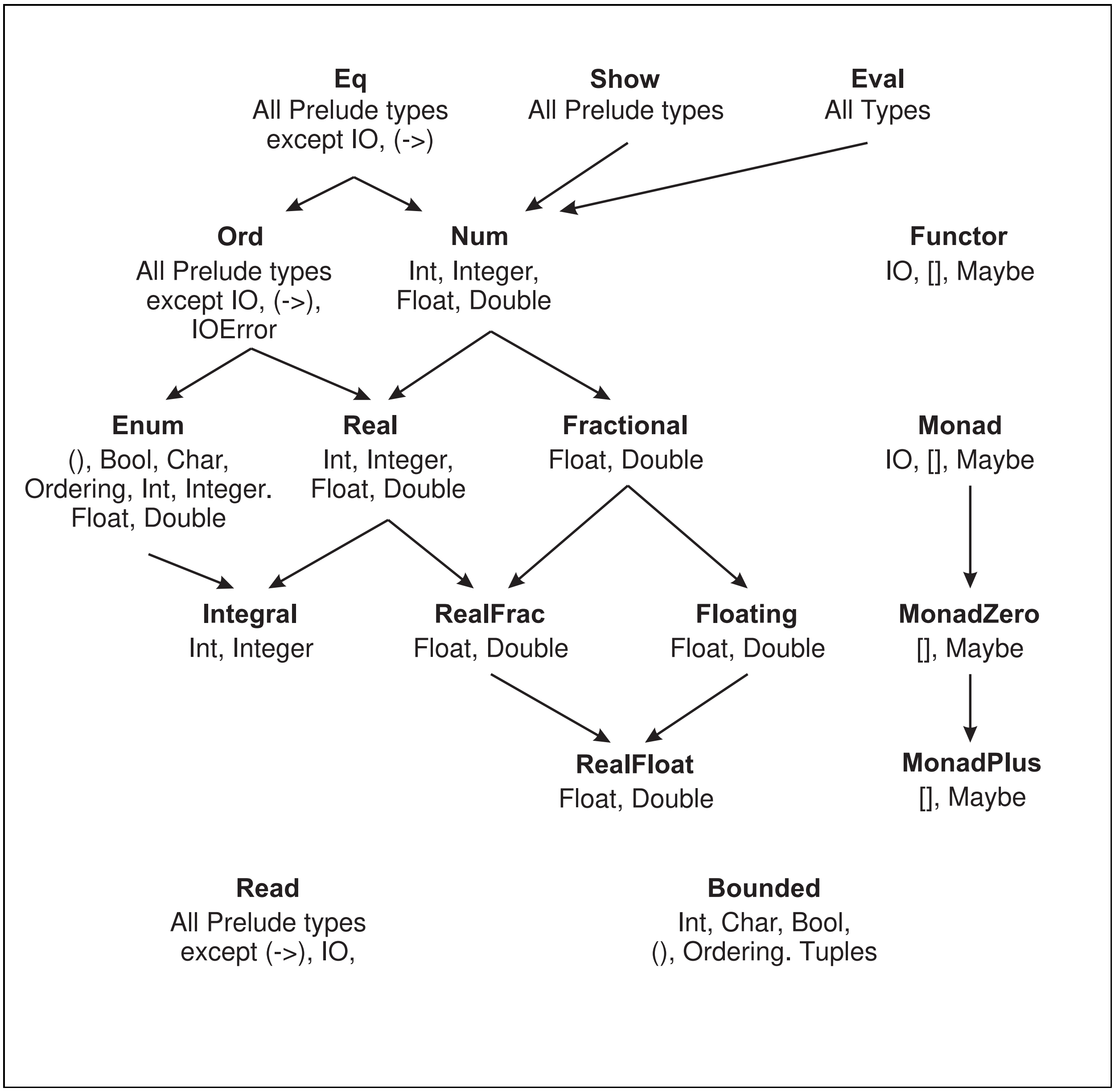

Figure 5: Standard Haskell Classes

6.2.1 The Eq Class

```
class  Eq a  where
        (==), (/=)  ::  a -> a -> Bool
        x /= y    =  not (x == y)
```

All basic datatypes except for functions and `IO` are instances of this class. Instances of `Eq` can be derived for any user-defined datatype whose constituents are also instances of `Eq`.

6.2.2 The Ord Class

```
class  (Eq a) => Ord a  where
        compare                 :: a -> a -> Ordering
        (<), (<=), (>=), (>)    :: a -> a -> Bool
        max, min                :: a -> a -> a

    compare x y
          | x == y    = EQ
          | x <= y    = LT
          | otherwise = GT

    x <= y            = compare x y /= GT
    x <  y            = compare x y == LT
    x >= y            = compare x y /= LT
    x >  y            = compare x y == GT

-- note that (min x y, max x y) = (x,y) or (y,x)
    max x y | x >= y    =  x
            | otherwise =  y
    min x y | x <  y    =  x
            | otherwise =  y
```

The `Ord` class is used for totally ordered datatypes. All basic datatypes except for functions and `IO` are instances of this class. Instances of `Ord` can be derived for any user-defined datatype whose constituent types are in `Ord`. The declared order of the constructors in the data declaration determines the ordering in derived `Ord` instances. The `Ordering` datatype allows a single comparison to determine the precise ordering of two objects. The defaults allow a user to create an `Ord` instance either with a type-specific `compare` function or with type-specific `==` and `<=` functions.

6.2.3 The Read and Show Classes

```
type  ReadS a = String -> [(a,String)]
type  ShowS   = String -> String

class  Read a  where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]

class  Show a  where
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS
```

The `Read` and `Show` classes are used to convert values to or from strings. Derived instances of `Read` and `Show` replicate the style in which a constructor is declared: infix constructors and field names are used on input and output. Strings produced by `showsPrec` are usually readable by `readsPrec`. Functions and the `IO` type are not in `Read`.

For convenience, the Prelude provides the following auxiliary functions:

```
reads           :: (Read a) => ReadS a
reads           =  readsPrec 0

shows           :: (Show a) => a -> ShowS
shows           =  showsPrec 0

read            :: (Read a) => String -> a
read s          =  case [x | (x,t) <- reads s, ("","") <- lex t] of
                        [x] -> x
                        []  -> error "PreludeText.read: no parse"
                        _   -> error "PreludeText.read: ambiguous parse"

show            :: (Show a) => a -> String
show x          =  shows x ""
```

`shows` and `reads` use a default precedence of 0. The `show` function returns a `String` instead of a `ShowS`; the `read` function reads input from a string which must be completely consumed by the input process. The `lex` function used by `read` is also part of the Prelude.

6.2.4 The Enum Class

```
class  (Ord a) => Enum a        where
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]             -- [n..]
    enumFromThen        :: a -> a -> [a]        -- [n,n'..]
    enumFromTo          :: a -> a -> [a]        -- [n..m]
    enumFromThenTo      :: a -> a -> a -> [a]   -- [n,n'..m]

    enumFromTo n m      =  takeWhile (<= m) (enumFrom n)
    enumFromThenTo n n' m
                        =  takeWhile (if n' >= n then (<= m) else (>= m))
                                     (enumFromThen n n')
```

Class `Enum` defines operations on sequentially ordered types. The `toEnum` and `fromEnum` functions map values from a type in `Enum` onto `Int`. These functions are not meaningful for all instances of `Enum`: floating point values or `Integer` may not be mapped onto an `Int`. An runtime error occurs if either `toEnum` or `fromEnum` is given a value not mappable to the result type. Instances of `Enum` may be derived for any enumeration type (types whose constructors have no fields). There are also `Enum` instances for floats.

6.2.5 Monadic Classes

```
class  Functor f  where
    map     :: (a -> b) -> (f a -> f b)

class  Monad m  where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a

class  (Monad m) => MonadZero m  where
    zero    :: m a

class  (MonadZero m) => MonadPlus m  where
   (++)         :: m a -> m a -> m a
```

These classes define the basic monadic operations. See Section 7 for more information about monads. The monadic classes serve to organize a set of operations common to a number of related types. These types are all *container types*: that is, they contain a value or values of another type. (To be precise, types in these classes must have kind $* \rightarrow *$.) In the Prelude, lists, `Maybe`, and `IO` are all predefined container types.

The `Functor` class is used for types which can be mapped over. Lists, `IO`, and `Maybe` are in this class. The `IO` type, `Maybe`, and lists are instances of `Monad`. The `do` syntax provides a more readable notation for the operators in `Monad`. Both lists and `Maybe` are instances of the `MonadZero` class. The `MonadPlus` class provides a 'monadic addition' operator: `++`. In the Prelude, `Maybe` and lists are in this class. For lists, `++` defines concatenation. For `Maybe`, the `++` function returns the first non-empty value (if any).

Instances of these classes should satisfy the following laws:

```
map id                        =  id
map (f . g)                   =  map f . map g
map f xs                      =  xs >>= return . f
return a >>= k                =  k a
m >>= return                  =  m
m >>= (\x -> k x >>= h))      =  (m >>= k) >>= h
m >> zero                     =  zero
zero >>= m                    =  zero
m ++ zero                     =  m
zero ++ m                     =  m
```

All instances defined in the Prelude satisfy these laws.

The Prelude provides the following auxiliary functions:

```
accumulate      :: Monad m => [m a] -> m [a]
sequence        :: Monad m => [m a] -> m ()
mapM            :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_           :: Monad m => (a -> m b) -> [a] -> m ()
guard           :: MonadZero m => Bool -> m ()
```

6.2.6 The Bounded Class

```
class  Bounded a  where
    minBound, maxBound :: a
```

The `Bounded` class is used to name the upper and lower limits of a type. `Ord` is not a superclass of `Bounded` since types that are not totally ordered may also have upper and lower bounds. The types `Int`, `Char`, `Bool`, `()`, `Ordering`, and all tuples are instances of `Bounded`. The `Bounded` class may be derived for any enumeration type; `minBound` is the first constructor listed in the `data` declaration and `maxBound` is the last. `Bounded` may also be derived for single-constructor datatypes whose constituent types are in `Bounded`.

6.2.7 The Eval Class

```
class Eval a where
  strict      :: (a -> b) -> a -> b
  seq         :: a -> b -> b
  strict f x  =  x `seq` f x
```

Class `Eval` is a special class for which no instances may be explicitly defined. An `Eval` instance is *implicitly* derived for every datatype. Functions as well as all other built-in types are in `Eval`. (As a consequence, $\perp$ is not the same as `\x ->` $\perp$ since `seq` can be used to distinguish them.)

The functions `seq` and `strict` are defined by the equations:

$$
\begin{aligned}
\texttt{seq}\ \bot\ b &= \bot \\
\texttt{seq}\ a\ b &= b,\ \mathit{if}\ a \neq \bot \\
\texttt{strict}\ f\ x &= \texttt{seq}\ x\ (f\ x)
\end{aligned}
$$

These functions are usually introduced to improve performance by avoiding unneeded laziness. Strict datatypes (see Section 4.2.1) are defined in terms of the `strict` function. This class explicitly marks functions and types which employ polymorphic strictness.

The `Eval` instance for a type `T` with a constructor `C` implicitly derived by the compiler is:

```
instance Eval T where
  x `seq` y = case x of
                C -> y
                _ -> y  -- catches any other constructors in T
```

The `case` is used to force evaluation of the first argument to ‘seq‘ before returning the second argument. The constructor mentioned by `seq` is arbitrary: any constructor from `T` can be used.

6.3 Numbers

Haskell provides several kinds of numbers; the numeric types and the operations upon them have been heavily influenced by Common Lisp and Scheme. Numeric function names and operators are usually overloaded, using several type classes with an inclusion relation shown in Figure 5, page 66. The class `Num` of numeric types is a subclass of `Eq`, since all numbers may be compared for equality; its subclass `Real` is also a subclass of `Ord`, since the other comparison operations apply to all but complex numbers (defined in the `Complex` library). The class `Integral` contains both fixed- and arbitrary-precision integers; the class `Fractional` contains all non-integral types; and the class `Floating` contains all floating-point types, both real and complex.

The Prelude defines only the most basic numeric types: fixed sized integers (`Int`), arbitrary precision integers (`Integer`), single precision floating (`Float`), and double precision floating (`Double`). Other numeric types such as rationals and complex numbers are defined in libraries. In particular, the type `Rational` is a ratio of two `Integer` values, as defined in the `Rational` library.

The default floating point operations defined by the Haskell Prelude do not conform to current language independent arithmetic (LIA) standards. These standards require considerable more complexity in the numeric structure and have thus been relegated to a library. Some, but not all, aspects of the IEEE standard floating point standard have been accounted for in class `RealFloat`.

Table 3 lists the standard numeric types. The type `Int` covers at least the range $[-2^{29}, 2^{29}-1]$. As `Int` is an instance of the `Bounded` class, `maxBound` and `minBound` can be used to determine the exact `Int` range defined by an implementation. `Float` is

Type	Class	Description
`Integer`	`Integral`	Arbitrary-precision integers
`Int`	`Integral`	Fixed-precision integers
`(Integral a) => Ratio a`	`RealFrac`	Rational numbers
`Float`	`RealFloat`	Real floating-point, single precision
`Double`	`RealFloat`	Real floating-point, double precision
`(RealFloat a) => Complex a`	`Floating`	Complex floating-point

Table 3: Standard Numeric Types

implementation-defined; it is desirable that this type be at least equal in range and precision to the IEEE single-precision type. Similarly, `Double` should cover IEEE double-precision. The results of exceptional conditions (such as overflow or underflow) on the fixed-precision numeric types are undefined; an implementation may choose error ($\perp$, semantically), a truncated value, or a special value such as infinity, indefinite, etc.

The standard numeric classes and other numeric functions defined in the Prelude are shown in Figures 6–7. Figure 5 shows the class dependencies and built-in types which are instances of the numeric classes.

6.3.1 Numeric Literals

The syntax of numeric literals is given in Section 2.4. An integer literal represents the application of the function `fromInteger` to the appropriate value of type `Integer`. Similarly, a floating literal stands for an application of `fromRational` to a value of type `Rational` (that is, `Ratio Integer`). Given the typings:

```
fromInteger  :: (Num a) => Integer -> a
fromRational :: (Fractional a) => Rational -> a
```

integer and floating literals have the typings `(Num a) => a` and `(Fractional a) => a`, respectively. Numeric literals are defined in this indirect way so that they may be interpreted as values of any appropriate numeric type. See Section 4.3.4 for a discussion of overloading ambiguity.

6.3.2 Arithmetic and Number-Theoretic Operations

The infix class methods `(+)`, `(*)`, `(-)`, and the unary function `negate` (which can also be written as a prefix minus sign; see section 3.4) apply to all numbers. The class methods `quot`, `rem`, `div`, and `mod` apply only to integral numbers, while the class method `(/)` applies only to fractional ones. The `quot`, `rem`, `div`, and `mod` class methods satisfy these laws:

```
(x `quot` y)*y + (x `rem` y) == x
(x `div`  y)*y + (x `mod` y) == x
```

```
class  (Eq a, Show a, Eval a) => Num a  where
    (+), (-), (*)       :: a -> a -> a
    negate              :: a -> a
    abs, signum         :: a -> a
    fromInteger         :: Integer -> a

class  (Num a, Ord a) => Real a  where
    toRational          ::  a -> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer

class  (Num a) => Fractional a  where
    (/)                 :: a -> a -> a
    recip               :: a -> a
    fromRational        :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a
```

Figure 6: Standard Numeric Classes and Related Operations, Part 1

'`quot`' is integer division truncated toward zero, while the result of '`div`' is truncated toward negative infinity. The `quotRem` class method takes a dividend and a divisor as arguments and returns a (quotient, remainder) pair; `divMod` is defined similarly:

```
quotRem x y  =  (x `quot` y, x `rem` y)
divMod  x y  =  (x `div` y, x `mod` y)
```

Also available on integral numbers are the even and odd predicates:

```
even x       =  x `rem` 2 == 0
odd          =  not . even
```

Finally, there are the greatest common divisor and least common multiple functions: `gcd` x y is the greatest integer that divides both x and y. `lcm` x y is the smallest positive integer that both x and y divide.

```
class  (Real a, Fractional a) => RealFrac a  where
    properFraction      :: (Integral b) => a -> (b,a)
    truncate, round     :: (Integral b) => a -> b
    ceiling, floor      :: (Integral b) => a -> b

class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix          :: a -> Integer
    floatDigits         :: a -> Int
    floatRange          :: a -> (Int,Int)
    decodeFloat         :: a -> (Integer,Int)
    encodeFloat         :: Integer -> Int -> a
    exponent            :: a -> Int
    significand         :: a -> a
    scaleFloat          :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                        :: a -> Bool

fromIntegral            :: (Integral a, Num b) => a -> b
gcd, lcm                :: (Integral a) => a -> a-> a
(^)                     :: (Num a, Integral b) => a -> b -> a
(^^)                    :: (Fractional a, Integral b) => a -> b -> a

fromRealFrac            :: (RealFrac a, Fractional b) => a -> b

atan2                   :: (RealFloat a) => a -> a -> a
```

Figure 7: Standard Numeric Classes and Related Operations, Part 2

6.3.3 Exponentiation and Logarithms

The one-argument exponential function `exp` and the logarithm function `log` act on floating-point numbers and use base e. `logBase` a x returns the logarithm of x in base a. `sqrt` returns the principal square root of a floating-point number. There are three two-argument exponentiation operations: `(^)` raises any number to a nonnegative integer power, `(^^)` raises a fractional number to any integer power, and `(**)` takes two floating-point arguments. The value of x`^0` or x`^^0` is `1` for any x, including zero; `0**`y is undefined.

6.3.4 Magnitude and Sign

A number has a *magnitude* and a *sign*. The functions `abs` and `signum` apply to any number and satisfy the law:

```
abs x * signum x == x
```

For real numbers, these functions are defined by:

```
abs x     | x >= 0  = x
          | x <  0  = -x

signum x  | x >  0  = 1
          | x == 0  = 0
          | x <  0  = -1
```

6.3.5 Trigonometric Functions

The circular and hyperbolic sine, cosine, and tangent functions and their inverses are provided for floating-point numbers. A version of arctangent taking two real floating-point arguments is also provided: For real floating x and y, `atan2` y x differs from `atan` (y`/`x) in that its range is $(-\pi, \pi]$ rather than $(-\pi/2, \pi/2)$ (because the signs of the arguments provide quadrant information), and that it is defined when x is zero.

The precise definition of the above functions is as in Common Lisp, which in turn follows Penfield's proposal for APL [8]. See these references for discussions of branch cuts, discontinuities, and implementation.

6.3.6 Coercions and Component Extraction

The `ceiling`, `floor`, `truncate`, and `round` functions each take a real fractional argument and return an integral result. `ceiling` x returns the least integer not less than x, and `floor` x, the greatest integer not greater than x. `truncate` x yields the integer nearest x between 0 and x, inclusive. `round` x returns the nearest integer to x, the even integer if x is equidistant between two integers.

The function `properFraction` takes a real fractional number x and returns a pair comprising x as a proper fraction: an integral number with the same sign as x and a fraction with the same type and sign as x and with absolute value less than 1. The `ceiling`, `floor`, `truncate`, and `round` functions can be defined in terms of this one.

Two functions convert numbers to type `Rational`: `toRational` returns the rational equivalent of its real argument with full precision; `approxRational` takes two real fractional arguments x and ϵ and returns the simplest rational number within ϵ of x, where a rational p/q in reduced form is *simpler* than another p'/q' if $|p| \leq |p'|$ and $q \leq q'$. Every real interval contains a unique simplest rational; in particular, note that 0/1 is the simplest rational of all.

The class methods of class `RealFloat` allow efficient, machine-independent access to the components of a floating-point number. The functions `floatRadix`, `floatDigits`, and `floatRange` give the parameters of a floating-point type: the radix of the representation, the number of digits of this radix in the significand, and the lowest and highest values the exponent may assume, respectively. The function `decodeFloat` applied to a real floating-point number returns the significand expressed as an `Integer` and an appropriately scaled exponent (an `Int`). If `decodeFloat x` yields (m`,`n), then `x` is equal in value to mb^n, where b is the floating-point radix, and furthermore, either m and n are both zero or else $b^{d-1} \leq m < b^d$, where d is the value of `floatDigits x`. `encodeFloat` performs the inverse of this transformation. The functions `significand` and `exponent` together provide the same information as `decodeFloat`, but rather than an `Integer`, `significand x` yields a value of the same type as `x`, scaled to lie in the open interval $(-1, 1)$. `exponent 0` is zero. `scaleFloat` multiplies a floating-point number by an integer power of the radix.

The functions `isNaN`, `isInfinite`, `isDenormalized`, `isNegativeZero`, and `isIEEE` all support numbers represented using the IEEE standard. For non-IEEE floating point numbers, these may all return false.

Also available are the following coercion functions:

```
fromIntegral :: (Integral a, Num b) => a -> b
fromRealFrac :: (RealFrac a, Fractional b) => a -> b
```

7 Basic Input/Output

The I/O system in Haskell is purely functional, yet has all of the expressive power found in conventional programming languages. To achieve this, Haskell uses a *monad* to integrate I/O operations into a purely functional context.

The I/O monad used by Haskell mediates between the *values* natural to a functional language and the *actions* which characterize I/O operations and imperative programming in general. The order of evaluation of expressions in Haskell is constrained only by data dependencies; an implementation has a great deal of freedom in choosing this order. Actions, however, must be ordered in a well-defined manner for program execution – and I/O in particular – to be meaningful. Haskell's I/O monad provides the user with a way to specify the sequential chaining of actions, and an implementation is obliged to preserve this order.

The term *monad* comes from a branch of mathematics known as *category theory*. From the perspective of a Haskell programmer, however, it is best to think of a monad as an *abstract datatype*. In the case of the I/O monad, the abstract values are the *actions* mentioned above. Some operations are primitive actions, corresponding to conventional I/O operations. Special operations (methods in the class `Monad`, see Section 6.2.5) sequentially compose actions, corresponding to sequencing operators (such as the semi-colon) in imperative languages. Finally, the *hidden implementation* can be thought of as the *system state*; i.e. the state of the world.

7.1 Standard I/O Functions

Although Haskell provides fairly sophisticated I/O facilities, as defined in the `IO` library, it is possible to write many Haskell programs using only the few simple functions which are exported from the Prelude, and which are described in this section.

Output Functions These functions write to the standard output device (this is normally the user's terminal).

```
putChar      :: Char -> IO ()
putStr       :: String -> IO ()
putStrLn     :: String -> IO ()  -- adds a newline
print        :: Show a => a -> IO ()
```

The `print` function outputs a value of any printable type to the standard output device (this is normally the user's terminal). Printable types are those which are instances of class `Show`; `print` converts values to strings for output using the `show` operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2 could be written as:

```
main = print ([(n, 2^n) | n <- [0..19]])
```

Input Functions These functions read input from the standard input device (normally the user's terminal).

```
getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String
interact     :: (String -> String) -> IO ()
readIO       :: Read a => String -> IO a
readLine     :: Read a => IO a
```

Both `getChar` and `getLine` raise an exception on end-of-file; the `IOError` value associated with end-of-file is defined in a library. The `getContents` operation returns all user input as a single string which is read lazily as it is needed. The `interact` function takes a function of type `String->String` as its argument. The entire input from the standard input device (normally the user's terminal) is passed to this function as its argument, and the resulting string is output on the standard output device. The `readIO` function is similar to `read` except that it signals parse failure to the I/O monad instead of terminating the program. The `readLine` function combines `getLine` and `readIO`.

By default, these input functions echo to standard output. Functions in the I/O library provide full control over echoing.

The following program simply removes all non-ASCII characters from its standard input and echoes the result on its standard output. (The `isAscii` function is defined in a library.)

```
main = interact (filter isAscii)
```

Files These functions operate on files of characters. Files are named by strings using some implementation-specific method to resolve strings as file names.

The `writeFile` and `appendFile` functions write or append the string, their second argument, to the file, their first argument. The `readFile` function reads a file and returns the contents of the file as a string. The file is read lazily, on demand, as with `getContents`.

```
type FilePath =  String

writeFile     ::  FilePath -> String            -> IO ()
appendFile    ::  FilePath -> String            -> IO ()
readFile      ::  FilePath                      -> IO String
```

Note that `writeFile` and `appendFile` write a literal string to a file. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first.

```
main = appendFile "squares" (show [(x,x*x) | x <- [0,0.1..2]])
```

7.2 Sequencing I/O Operations

The two monadic binding functions, methods in the `Monad` class, are used to compose a series of I/O operations. The `>>` function is used where the result of the first operation is uninteresting, for example when it is `()`. The `>>=` operation passes the result of the first operation as an argument to the second operation.

```
(>>=)        ::  IO a    -> (a -> IO b)        -> IO b
(>>)         ::  IO a    -> IO b               -> IO b
```

For example,

```
main = readFile "input-file"                       >>= \ s ->
       writeFile "output-file" (filter isAscii s)  >>
       putStr "Filtering successful\n"
```

is similar to the previous example using `interact`, but takes its input from `"input-file"` and writes its output to `"output-file"`. A message is printed on the standard output before the program completes.

The `do` notation allows programming in a more imperative syntactic style. A slightly more elaborate version of the previous example would be:

```
main = do
        putStr "Input file: "
        ifile <- getLine
        putStr "Output file: "
        ofile <- getLine
        s <- readFile ifile
        writeFile ofile (filter isAscii s)
        putStr "Filtering successful\n"
```

The `return` function is used to define the result of an I/O operation. For example, `getLine` is defined in terms of `getChar`, using `return` to define the result the monad:

```
getLine :: IO String
getLine = do c <- getChar
             if c == '\n' then return ""
                          else do s <- getLine
                                  return (c:s)
```

7.3 Exception Handling in the I/O Monad

The I/O monad includes a simple exception handling system. Any I/O operation may raise an exception instead of returning a result. Exceptions in the I/O monad are represented by values of type `IOError`. This is an abstract type: its constructors are hidden from the user. The `IO` library defines functions which construct and examine `IOError` values. The only

Prelude function which creates an `IOError` value is `userError`. User error values include an string describing the error.

Exceptions are raised and caught using the following functions:

```
fail             ::  IOError -> IO a
catch            ::  IO a    -> (IOError -> IO a) -> IO a
```

The `fail` function raises an exception; the `catch` function establishes a handler which receives any exception raised in the action protected by `catch`. An exception is caught by the most recent handler established by `catch`. These handlers are not selective: all exceptions are caught. Exception propagation must be explicitly provided in a handler by re-raising any unwanted exceptions. For example, in

```
f = catch g (\e -> if IO.isEofError e then return [] else fail e)
```

the function `f` returns `[]` when an end-of-file exception occurs in `g`; otherwise, the exception is propagated to the next outer handler. The `isEofError` function is part of `IO` library.

When an exception propagates outside the main program, the Haskell system prints the associated `IOError` value and exits the program.

The exceptions raised by the I/O functions in the Prelude are defined in the Library Report.

A Standard Prelude

In this appendix the entire Haskell prelude is given. It is organized into a root module and three sub-modules. Primitives which are not definable in Haskell, indicated by names starting with `prim`, are defined in a system dependent manner in module `PreludeBuiltin` and are not shown here. Instance declarations which simply bind primitives to class methods are omitted. Some of the more verbose instances with obvious functionality have been left out for the sake of brevity.

Declarations for special types such as `Integer`, `()`, or `(->)` are included in the Prelude for completeness even though the declaration may be incomplete or syntacticly invalid.

```
module Prelude (
    module PreludeList, module PreludeText, module PreludeIO,
    Bool(False, True),
    Maybe(Nothing, Just),
    Either(Left, Right),
    Ordering(LT, EQ, GT),
    Char, String, Int, Integer, Float, Double, IO, Void,
--  List type: []((:), [])
--  Tuple types: (,), (,,), etc.
--  Trivial type: ()
--  Functions: (->)
    Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
    Bounded(minBound, maxBound),
    Eval(seq, strict),
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Real(toRational),
    Integral(quot, rem, div, mod, quotRem, divMod, toInteger),
    Fractional((/), recip, fromRational),
    Floating(pi, exp, log, sqrt, (**), logBase, sin, cos, tan,
             asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh),
    RealFrac(properFraction, truncate, round, ceiling, floor),
    RealFloat(floatRadix, floatDigits, floatRange, decodeFloat,
              encodeFloat, exponent, significand, scaleFloat, isNaN,
              isInfinite, isDenormalized, isIEEE, isNegativeZero),
    Monad((>>=), (>>), return),
    MonadZero(zero),
    MonadPlus((++)),
    Functor(map),
    succ, pred,
    mapM, mapM_, guard, accumulate, sequence, filter, concat, applyM,
    maybe,
    (&&), (||), not, otherwise,
    subtract, even, odd, gcd, lcm, (^), (^^),
    fromIntegral, fromRealFrac, atan2,
    fst, snd, curry, uncurry, id, const, (.), flip, ($), until,
    asTypeOf, error, undefined ) where

import PreludeBuiltin  -- Contains all `prim' values
import PreludeList
import PreludeText
import PreludeIO
import Ratio(Ratio, Rational, (%), numerator, denominator)
```

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infixr 5  ++
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixr 1  >>, >>=
infixr 0  $, `seq`

-- Standard types, classes, instances and related functions

-- Equality and Ordered classes

class  Eq a  where
    (==), (/=)          :: a -> a -> Bool

    x /= y              =  not (x == y)

class  (Eq a) => Ord a  where
    compare             :: a -> a -> Ordering
    (<), (<=), (>=), (>):: a -> a -> Bool
    max, min            :: a -> a -> a

-- An instance of Ord should define either compare or <=
-- Using compare can be more efficient for complex types.
    compare x y
            | x == y    = EQ
            | x <= y    = LT
            | otherwise = GT

    x <= y              = compare x y /= GT
    x <  y              = compare x y == LT
    x >= y              = compare x y /= LT
    x >  y              = compare x y == GT

    max x y | x >= y    =  x
            | otherwise =  y
    min x y | x <  y    =  x
            | otherwise =  y
```

```
-- Enumeration and Bounded classes

class  (Ord a) => Enum a        where
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]             -- [n..]
    enumFromThen        :: a -> a -> [a]        -- [n,n'..]
    enumFromTo          :: a -> a -> [a]        -- [n..m]
    enumFromThenTo      :: a -> a -> a -> [a]   -- [n,n'..m]

    enumFromTo n m      =  takeWhile (<= m) (enumFrom n)
    enumFromThenTo n n' m
                        =  takeWhile (if n' >= n then (<= m) else (>= m))
                                     (enumFromThen n n')

succ, pred              :: Enum a => a -> a
succ                    =  toEnum . (+1) . fromEnum
pred                    =  toEnum . (subtract 1) . fromEnum

class  Bounded a  where
    minBound, maxBound :: a

-- Numeric classes

class  (Eq a, Show a, Eval a) => Num a  where
    (+), (-), (*)       :: a -> a -> a
    negate              :: a -> a
    abs, signum         :: a -> a
    fromInteger         :: Integer -> a

    x - y               =  x + negate y

class  (Num a, Ord a) => Real a  where
    toRational          ::  a -> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer

    n `quot` d          =  q  where (q,r) = quotRem n d
    n `rem` d           =  r  where (q,r) = quotRem n d
    n `div` d           =  q  where (q,r) = divMod n d
    n `mod` d           =  r  where (q,r) = divMod n d
    divMod n d          =  if signum r == - signum d then (q-1, r+d) else qr
                           where qr@(q,r) = quotRem n d
```

```
class  (Num a) => Fractional a  where
    (/)                 :: a -> a -> a
    recip               :: a -> a
    fromRational        :: Rational -> a

    recip x             =  1 / x

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

    x ** y              =  exp (log x * y)
    logBase x y         =  log y / log x
    sqrt x              =  x ** 0.5
    tan  x              =  sin  x / cos  x
    tanh x              =  sinh x / cosh x

class  (Real a, Fractional a) => RealFrac a  where
    properFraction      :: (Integral b) => a -> (b,a)
    truncate, round     :: (Integral b) => a -> b
    ceiling, floor      :: (Integral b) => a -> b

    truncate x          =  m  where (m,_) = properFraction x

    round x             =  let (n,r) = properFraction x
                               m     = if r < 0 then n - 1 else n + 1
                           in case signum (abs r - 0.5) of
                                -1 -> n
                                0  -> if even n then n else m
                                1  -> m

    ceiling x           =  if r > 0 then n + 1 else n
                           where (n,r) = properFraction x

    floor x             =  if r < 0 then n - 1 else n
                           where (n,r) = properFraction x
```

```
class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix          :: a -> Integer
    floatDigits         :: a -> Int
    floatRange          :: a -> (Int,Int)
    decodeFloat         :: a -> (Integer,Int)
    encodeFloat         :: Integer -> Int -> a
    exponent            :: a -> Int
    significand         :: a -> a
    scaleFloat          :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                        :: a -> Bool

    exponent x          =  if m == 0 then 0 else n + floatDigits x
                           where (m,n) = decodeFloat x

    significand x       =  encodeFloat m (- floatDigits x)
                           where (m,_) = decodeFloat x

    scaleFloat k x      =  encodeFloat m (n+k)
                           where (m,n) = decodeFloat x

-- Numeric functions

subtract        :: (Num a) => a -> a -> a
subtract        =  flip (-)

even, odd       :: (Integral a) => a -> Bool
even n          =  n `rem` 2 == 0
odd             =  not . even

gcd             :: (Integral a) => a -> a -> a
gcd 0 0         =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y         =  gcd' (abs x) (abs y)
                   where gcd' x 0  =  x
                         gcd' x y  =  gcd' y (x `rem` y)

lcm             :: (Integral a) => a -> a -> a
lcm _ 0         =  0
lcm 0 _         =  0
lcm x y         =  abs ((x `quot` (gcd x y)) * y)

(^)             :: (Num a, Integral b) => a -> b -> a
x ^ 0           =  1
x ^ n | n > 0   =  f x (n-1) x
                   where f _ 0 y = y
                         f x n y = g x n  where
                                   g x n | even n  = g (x*x) (n `quot` 2)
                                         | otherwise = f x (n-1) (x*y)
_ ^ _           = error "Prelude.^: negative exponent"
```

```
(^^)            :: (Fractional a, Integral b) => a -> b -> a
x ^^ n          =  if n >= 0 then x^n else recip (x^(-n))

fromIntegral    :: (Integral a, Num b) => a -> b
fromIntegral    =  fromInteger . toInteger

fromRealFrac    :: (RealFrac a, Fractional b) => a -> b
fromRealFrac    =  fromRational . toRational

atan2           :: (RealFloat a) => a -> a -> a
atan2 y x       =  case (signum y, signum x) of
                        ( 0, 1) ->  0
                        ( 1, 0) ->  pi/2
                        ( 0,-1) ->  pi
                        (-1, 0) -> -pi/2
                        ( _, 1) ->  atan (y/x)
                        ( _,-1) ->  atan (y/x) + pi
                        ( 0, 0) ->  error "Prelude.atan2: atan2 of origin"

-- Monadic classes

class  Functor f  where
    map         :: (a -> b) -> f a -> f b

class  Monad m  where
    (>>=)       :: m a -> (a -> m b) -> m b
    (>>)        :: m a -> m b -> m b
    return      :: a -> m a

    m >> k      =  m >>= \_ -> k

class  (Monad m) => MonadZero m  where
    zero        :: m a

class  (MonadZero m) => MonadPlus m where
   (++)         :: m a -> m a -> m a

accumulate      :: Monad m => [m a] -> m [a]
accumulate      =  foldr mcons (return [])
                   where mcons p q = p >>= \x -> q >>= \y -> return (x:y)

sequence        :: Monad m => [m a] -> m ()
sequence        =  foldr (>>) (return ())

mapM            :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as       =  accumulate (map f as)

mapM_           :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as      =  sequence (map f as)
```

```
guard            :: MonadZero m => Bool -> m ()
guard p          =  if p then return () else zero

-- This subsumes the list-based filter function.

filter           :: MonadZero m => (a -> Bool) -> m a -> m a
filter p         =  applyM (\x -> if p x then return x else zero)

-- This subsumes the list-based concat function.

concat           :: MonadPlus m => [m a] -> m a
concat           =  foldr (++) zero

applyM           :: Monad m => (a -> m b) -> m a -> m b
applyM f x       =  x >>= f

-- Eval Class

class Eval a where
   seq         :: a -> a -> b
   strict      :: (a -> b) -> a -> b
   strict f x  =  x `seq` f x

-- Trivial type

data  ()  =  ()  deriving (Eq, Ord, Enum, Bounded)

-- Function type

data a -> b  -- No constructor for functions is exported.

-- Empty type

data Void      -- No constructor for Void is exported.  Import/Export
               -- lists must use Void instead of Void(..) or Void()

-- Boolean type

data  Bool  =  False | True     deriving (Eq, Ord, Enum, Read, Show, Bounded)

-- Boolean functions

(&&), (||)              :: Bool -> Bool -> Bool
True  && x              =  x
False && _              =  False
True  || _              =  True
False || x              =  x

not                     :: Bool -> Bool
not True                =  False
not False               =  True
```

```
otherwise               :: Bool
otherwise               =  True

-- Character type

data Char = ... 'a' | 'b' ... -- 265 ISO values

instance  Eq Char  where
    c == c'             =  fromEnum c == fromEnum c'

instance  Ord Char  where
    c <= c'             =  fromEnum c <= fromEnum c'

instance  Enum Char  where
    toEnum              =  primIntToChar
    fromEnum            =  primCharToInt
    enumFrom c          =  map toEnum [fromEnum c .. fromEnum (maxBound::Char)]
    enumFromThen c c'   =  map toEnum [fromEnum c,
                                       fromEnum c' .. fromEnum lastChar]
                           where lastChar :: Char
                                 lastChar | c' < c    = minBound
                                          | otherwise = maxBound

instance  Bounded Char  where
    minBound            =  '\0'
    maxBound            =  '\255'

type  String = [Char]

-- Maybe type

data  Maybe a  =  Nothing | Just a      deriving (Eq, Ord, Read, Show)

maybe                   :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing       =  n
maybe n f (Just x)      =  f x

instance  Functor Maybe  where
    map f Nothing       = Nothing
    map f (Just a)      = Just (f x)

instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= k      = Nothing
    return              = Just

instance  MonadZero Maybe  where
    zero                = Nothing
```

```
instance  MonadPlus Maybe  where
    Nothing ++ ys       = ys
    xs      ++ ys       = xs

-- Either type

data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)

either                  :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)     =  f x
either f g (Right y)    =  g y

-- IO type

data  IO a  -- abstract

instance  Functor IO where
   map f x              = x >>= (return . f)

instance  Monad IO  where ...

-- Ordering type

data Ordering = LT | EQ | GT  deriving (Eq, Ord, Enum, Read, Show, Bounded)

-- Standard numeric types.  The data declarations for these types cannot
-- be expressed directly in Haskell since the constructor lists would be
-- far too large.

data Int  =  minBound ... -1 | 0 | 1 ... maxBound
instance  Eq       Int  where ...
instance  Ord      Int  where ...
instance  Num      Int  where ...
instance  Real     Int  where ...
instance  Integral Int  where ...
instance  Enum     Int  where ...
instance  Bounded  Int  where ...

data Integer = ... -1 | 0 | 1 ...
instance  Eq       Integer  where ...
instance  Ord      Integer  where ...
instance  Num      Integer  where ...
instance  Real     Integer  where ...
instance  Integral Integer  where ...
instance  Enum     Integer  where ...
```

```
data Float
instance  Eq         Float  where ...
instance  Ord        Float  where ...
instance  Num        Float  where ...
instance  Real       Float  where ...
instance  Fractional Float  where ...
instance  Floating   Float  where ...
instance  RealFrac   Float  where ...
instance  RealFloat  Float  where ...

data Double
instance  Eq         Double  where ...
instance  Ord        Double  where ...
instance  Num        Double  where ...
instance  Real       Double  where ...
instance  Fractional Double  where ...
instance  Floating   Double  where ...
instance  RealFrac   Double  where ...
instance  RealFloat  Double  where ...

-- The Enum instances for Floats and Doubles are slightly unusual.
-- The ‘toEnum’ function truncates numbers to Int.  The definitions
-- of enumFrom and enumFromThen allow floats to be used in arithmetic
-- series: [0,0.1 .. 1.0].  However, roundoff errors make these somewhat
-- dubious.  This example may have either 10 or 11 elements, depending on
-- how 0.1 is represented.

instance  Enum Float  where
    toEnum              =  fromIntegral
    fromEnum            =  fromInteger . truncate   -- may overflow
    enumFrom            =  numericEnumFrom
    enumFromThen        =  numericEnumFromThen

instance  Enum Double  where
    toEnum              =  fromIntegral
    fromEnum            =  fromInteger . truncate   -- may overflow
    enumFrom            =  numericEnumFrom
    enumFromThen        =  numericEnumFromThen

numericEnumFrom         :: (Real a) => a -> [a]
numericEnumFromThen     :: (Real a) => a -> a -> [a]
numericEnumFrom         =  iterate (+1)
numericEnumFromThen n m =  iterate (+(m-n)) n

-- Lists

data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
```

```
instance Functor [] where
    map f []             =  []
    map f (x:xs)         =  f x : map f xs

instance  Monad []  where
    m >>= k             = concat (map k m)
    return x            = [x]

instance  MonadZero []  where
    zero                = []

instance  MonadPlus []  where
    xs ++ ys            =  foldr (:) ys xs

-- Tuples

data  (a,b)   =  (a,b)   deriving (Eq, Ord, Bounded)
data  (a,b,c) =  (a,b,c) deriving (Eq, Ord, Bounded)

-- component projections for pairs:
-- (NB: not provided for triples, quadruples, etc.)
fst                     :: (a,b) -> a
fst (x,y)               =  x

snd                     :: (a,b) -> b
snd (x,y)               =  y

-- curry converts an uncurried function to a curried function;
-- uncurry converts a curried function to a function on pairs.
curry                   :: ((a, b) -> c) -> a -> b -> c
curry f x y             =  f (x, y)

uncurry                 :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p             =  f (fst p) (snd p)

-- Functions

-- Standard value bindings

-- identity function
id                      :: a -> a
id x                    =  x

-- constant function
const                   :: a -> b -> a
const x _               =  x

-- function composition
(.)                     :: (b -> c) -> (a -> b) -> a -> c
f . g                   =  \ x -> f (g x)
```

```
-- flip f  takes its (first) two arguments in the reverse order of f.
flip                    :: (a -> b -> c) -> b -> a -> c
flip f x y              =  f y x

-- right-associating infix application operator (useful in continuation-
-- passing style)
($)                     :: (a -> b) -> a -> b
f $ x                   =  f x

-- until p f  yields the result of applying f until p holds.
until                   :: (a -> Bool) -> (a -> a) -> a -> a
until p f x | p x       =  x
            | otherwise =  until p f (f x)

-- asTypeOf is a type-restricted version of const.  It is usually used
-- as an infix operator, and its typing forces its first argument
-- (which is usually overloaded) to have the same type as the second.
asTypeOf                :: a -> a -> a
asTypeOf                =  const

-- error stops execution and displays an error message

error                   :: String -> a
error                   =  primError

-- It is expected that compilers will recognize this and insert error
-- messages which are more appropriate to the context in which undefined
-- appears.

undefined               :: a
undefined               =  error "Prelude.undefined"
```

A.1 Prelude PreludeList

```
-- Standard list functions

module PreludeList(
   head, last, tail, init, null, length, (!!),
   foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
   iterate, repeat, replicate, cycle,
   take, drop, splitAt, takeWhile, dropWhile, span, break,
   lines, words, unlines, unwords, reverse, and, or,
   any, all, elem, notElem, lookup,
   sum, product, maximum, minimum, concatMap,
   zip, zip3, zipWith, zipWith3, unzip, unzip3)
  where

import qualified Char(isSpace)

infixl 9  !!
infix  4 `elem`, `notElem`

-- head and tail extract the first element and remaining elements,
-- respectively, of a list, which must be non-empty.  last and init
-- are the dual functions working from the end of a finite list,
-- rather than the beginning.

head                    :: [a] -> a
head (x:_)              =  x
head []                 =  error "PreludeList.head: empty list"

last                    :: [a] -> a
last [x]                =  x
last (_:xs)             =  last xs
last []                 =  error "PreludeList.last: empty list"

tail                    :: [a] -> [a]
tail (_:xs)             =  xs
tail []                 =  error "PreludeList.tail: empty list"

init                    :: [a] -> [a]
init [x]                =  []
init (x:xs)             =  x : init xs
init []                 =  error "PreludeList.init: empty list"

null                    :: [a] -> Bool
null []                 =  True
null (_:_)              =  False
```

```
-- length returns the length of a finite list as an Int; it is an instance
-- of the more general genericLength, the result type of which may be
-- any kind of number.
length                  :: [a] -> Int
length []               =  0
length (_:l)            =  1 + length l

-- List index (subscript) operator, 0-origin
(!!)                    :: [a] -> Int -> a
(x:_)  !! 0             =  x
(_:xs) !! n | n > 0     =  xs !! (n-1)
(_:_)  !! _             =  error "PreludeList.!!: negative index"
[]     !! _             =  error "PreludeList.!!: index too large"

-- foldl, applied to a binary operator, a starting value (typically the
-- left-identity of the operator), and a list, reduces the list using
-- the binary operator, from left to right:
--  foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
-- foldl1 is a variant that has no starting value argument, and  thus must
-- be applied to non-empty lists.  scanl is similar to foldl, but returns
-- a list of successive reduced values from the left:
--      scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
-- Note that  last (scanl f z xs) == foldl f z xs.
-- scanl1 is similar, again without the starting element:
--      scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]

foldl                   :: (a -> b -> a) -> a -> [b] -> a
foldl f z []            =  z
foldl f z (x:xs)        =  foldl f (f z x) xs

foldl1                  :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)         =  foldl f x xs
foldl1 _ []             =  error "PreludeList.foldl1: empty list"

scanl                   :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs            =  q : (case xs of
                                []   -> []
                                x:xs -> scanl f (f q x) xs)

scanl1                  :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs)         =  scanl f x xs
scanl1 _ []             =  error "PreludeList.scanl1: empty list"
```

```
-- foldr, foldr1, scanr, and scanr1 are the right-to-left duals of the
-- above functions.

foldr                   :: (a -> b -> b) -> b -> [a] -> b
foldr f z []            =  z
foldr f z (x:xs)        =  f x (foldr f z xs)

foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 f [x]            =  x
foldr1 f (x:xs)         =  f x (foldr1 f xs)
foldr1 _ []             =  error "PreludeList.foldr1: empty list"

scanr                   :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []           =  [q0]
scanr f q0 (x:xs)       =  f x q : qs
                           where qs@(q:_) = scanr f q0 xs

scanr1                  :: (a -> a -> a) -> [a] -> [a]
scanr1 f  [x]           =  [x]
scanr1 f  (x:xs)        =  f x q : qs
                           where qs@(q:_) = scanr1 f xs
scanr1 _ []             =  error "PreludeList.scanr1: empty list"

-- iterate f x returns an infinite list of repeated applications of f to x:
-- iterate f x == [x, f x, f (f x), ...]
iterate                 :: (a -> a) -> a -> [a]
iterate f x             =  x : iterate f (f x)

-- repeat x is an infinite list, with x the value of every element.
repeat                  :: a -> [a]
repeat x                =  xs where xs = x:xs

-- replicate n x is a list of length n with x the value of every element
replicate               :: Int -> a -> [a]
replicate n x           =  take n (repeat x)

-- cycle ties a finite list into a circular one, or equivalently,
-- the infinite repetition of the original list.  It is the identity
-- on infinite lists.

cycle                   :: [a] -> [a]
cycle xs                =  xs' where xs' = xs ++ xs'
```

```
-- take n, applied to a list xs, returns the prefix of xs of length n,
-- or xs itself if n > length xs.  drop n xs returns the suffix of xs
-- after the first n elements, or [] if n > length xs.  splitAt n xs
-- is equivalent to (take n xs, drop n xs).

take                   :: Int -> [a] -> [a]
take 0 _               =  []
take _ []              =  []
take n (x:xs) | n > 0  =  x : take (n-1) xs
take _     _           =  error "PreludeList.take: negative argument"

drop                   :: Int -> [a] -> [a]
drop 0 xs              =  xs
drop _ []              =  []
drop n (_:xs) | n > 0  =  drop (n-1) xs
drop _     _           =  error "PreludeList.drop: negative argument"

splitAt                   :: Int -> [a] -> ([a],[a])
splitAt 0 xs              =  ([],xs)
splitAt _ []              =  ([],[])
splitAt n (x:xs) | n > 0  =  (x:xs',xs'') where (xs',xs'') = splitAt (n-1) xs
splitAt _     _           =  error "PreludeList.splitAt: negative argument"

-- takeWhile, applied to a predicate p and a list xs, returns the longest
-- prefix (possibly empty) of xs of elements that satisfy p.  dropWhile p xs
-- returns the remaining suffix.  Span p xs is equivalent to
-- (takeWhile p xs, dropWhile p xs), while break p uses the negation of p.

takeWhile               :: (a -> Bool) -> [a] -> [a]
takeWhile p []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile               :: (a -> Bool) -> [a] -> [a]
dropWhile p []          =  []
dropWhile p xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  xs

span, break             :: (a -> Bool) -> [a] -> ([a],[a])
span p []               =  ([],[])
span p xs@(x:xs')
            | p x       =  (x:xs',xs'') where (xs',xs'') = span p xs
            | otherwise =  (xs,[])
break p                 =  span (not . p)
```

```
-- lines breaks a string up into a list of strings at newline characters.
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.

lines                 :: String -> [String]
lines ""              =  []
lines s               =  let (l, s') = break (== '\n') s
                         in  l : case s' of
                                      []      -> []
                                      (_:s'') -> lines s''

words                 :: String -> [String]
words s               =  case dropWhile Char.isSpace s of
                              "" -> []
                              s' -> w : words s''
                                    where (w, s'') =
                                      break Char.isSpace s'

unlines               :: [String] -> String
unlines               =  concatMap (++ "\n")

unwords               :: [String] -> String
unwords []            =  ""
unwords ws            =  foldr1 (\w s -> w ++ ' ':s) ws

-- reverse xs returns the elements of xs in reverse order.  xs must be finite.
reverse               :: [a] -> [a]
reverse               =  foldl (flip (:)) []

-- and returns the conjunction of a Boolean list.  For the result to be
-- True, the list must be finite; False, however, results from a False
-- value at a finite index of a finite or infinite list.  or is the
-- disjunctive dual of and.
and, or               :: [Bool] -> Bool
and                   =  foldr (&&) True
or                    =  foldr (||) False

-- Applied to a predicate and a list, any determines if any element
-- of the list satisfies the predicate.  Similarly, for all.
any, all              :: (a -> Bool) -> [a] -> Bool
any p                 =  or . map p
all p                 =  and . map p
```

```
-- elem is the list membership predicate, usually written in infix form,
-- e.g., x `elem` xs.  notElem is the negation.
elem, notElem           :: (Eq a) => a -> [a] -> Bool
elem x                  =  any (== x)
notElem x               =  all (not . (/= x))

-- lookup key assocs looks up a key in an association list.
lookup                  :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []           =  Nothing
lookup key ((x,y):xys)
    | key == x          =  Just y
    | otherwise         =  lookup key xys

-- sum and product compute the sum or product of a finite list of numbers.
sum, product            :: (Num a) => [a] -> a
sum                     =  foldl (+) 0
product                 =  foldl (*) 1

-- maximum and minimum return the maximum or minimum value from a list,
-- which must be non-empty, finite, and of an ordered type.
maximum, minimum        :: (Ord a) => [a] -> a
maximum []              =  error "PreludeList.maximum: empty list"
maximum xs              =  foldl1 max xs

minimum []              =  error "PreludeList.minimum: empty list"
minimum xs              =  foldl1 min xs

concatMap               :: (a -> [b]) -> [a] -> [b]
concatMap f             =  concat . map f

-- zip takes two lists and returns a list of corresponding pairs.  If one
-- input list is short, excess elements of the longer list are discarded.
-- zip3 takes three lists and returns a list of triples.  Zips for larger
-- tuples are in the List library

zip                     :: [a] -> [b] -> [(a,b)]
zip                     =  zipWith (,)

zip3                    :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3                    =  zipWith3 (,,)

-- The zipWith family generalises the zip family by zipping with the
-- function given as the first argument, instead of a tupling function.
-- For example, zipWith (+) is applied to two lists to produce the list
-- of corresponding sums.

zipWith                 :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs) =  z a b : zipWith z as bs
zipWith _ _ _           =  []
```

```
zipWith3                :: (a->b->c->d) -> [a]->[b]->[c]->[d]
zipWith3 z (a:as) (b:bs) (c:cs)
                        =  z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _        =  []

-- unzip transforms a list of pairs into a pair of lists.

unzip                   :: [(a,b)] -> ([a],[b])
unzip                   =  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

unzip3                  :: [(a,b,c)] -> ([a],[b],[c])
unzip3                  =  foldr (\(a,b,c) ~(as,bs,cs) -> (a:as,b:bs,c:cs))
                                 ([],[],[])
```

A.2 Prelude PreludeText

```
module  PreludeText (
        ReadS, ShowS,
        Read(readsPrec, readList),
        Show(showsPrec, showList),
        reads, shows, show, read, lex,
        showChar, showString, readParen, showParen ) where

-- The omitted instances can be implemented in standard Haskell but
-- they have been omitted for the sake of brevity

import Char( isSpace, isAlpha, isDigit, isAlphanum, isHexDigit )

type  ReadS a   = String -> [(a,String)]
type  ShowS     = String -> String

class  Read a  where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]

    readList    = readParen False (\r -> [pr | ("[",s)  <- lex r,
                                               pr       <- readl s])
                  where readl  s = [([],t)   | ("]",t)  <- lex s] ++
                                   [(x:xs,u) | (x,t)    <- reads s,
                                               (xs,u)   <- readl' t]
                        readl' s = [([],t)   | ("]",t)  <- lex s] ++
                                   [(x:xs,v) | (",",t)  <- lex s,
                                               (x,u)    <- reads t,
                                               (xs,v)   <- readl' u]

class  Show a  where
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS

    showList [] = showString "[]"
    showList (x:xs)
                = showChar '[' . shows x . showl xs
                  where showl []     = showChar ']'
                        showl (x:xs) = showString ", " . shows x . showl xs

reads           :: (Read a) => ReadS a
reads           =  readsPrec 0

shows           :: (Show a) => a -> ShowS
shows           =  showsPrec 0
```

```
read            :: (Read a) => String -> a
read s          =  case [x | (x,t) <- reads s, ("","") <- lex t] of
                        [x] -> x
                        []  -> error "PreludeText.read: no parse"
                        _   -> error "PreludeText.read: ambiguous parse"

show            :: (Show a) => a -> String
show x          =  shows x ""

showChar        :: Char -> ShowS
showChar        =  (:)

showString      :: String -> ShowS
showString      =  (++)

showParen       :: Bool -> ShowS -> ShowS
showParen b p   =  if b then showChar '(' . p . showChar ')' else p

readParen       :: Bool -> ReadS a -> ReadS a
readParen b g   =  if b then mandatory else optional
                   where optional r  = g r ++ mandatory r
                         mandatory r = [(x,u) | ("(",s) <- lex r,
                                                (x,t)   <- optional s,
                                                (")",u) <- lex t    ]

-- This lexer is not completely faithful to the Haskell lexical syntax.
-- Current limitations:
--    Qualified names are not handled properly
--    A `--' does not terminate a symbol
--    Octal and hexidecimal numerics are not recognized as a single token

lex                   :: ReadS String
lex ""                = [("","")]
lex (c:s) | isSpace c = lex (dropWhile isSpace s)
lex ('\'':s)          = [('\'':ch++"'", t) | (ch,'\'':t)  <- lexLitChar s,
                                               ch /= "'"                ]
lex ('"':s)           = [('"':str, t)      | (str,t) <- lexString s]
                        where
                        lexString ('"':s) = [("\"",s)]
                        lexString s = [(ch++str, u)
                                              | (ch,t)  <- lexStrItem s,
                                                (str,u) <- lexString t  ]

                        lexStrItem ('\\':'&':s) = [("\\&",s)]
                        lexStrItem ('\\':c:s) | isSpace c
                            = [("\\&",t) | '\\':t <- [dropWhile isSpace s]]
                        lexStrItem s            = lexLitChar s
```

```
lex (c:s) | isSingle c = [([c],s)]
          | isSym c    = [(c:sym,t)       | (sym,t) <- [span isSym s]]
          | isAlpha c  = [(c:nam,t)       | (nam,t) <- [span isIdChar s]]
          | isDigit c  = [(c:ds++fe,t)    | (ds,s)  <- [span isDigit s],
                                            (fe,t)  <- lexFracExp s     ]
          | otherwise  = []    -- bad character
             where
              isSingle c =  c `elem` ",;()[]{}_`"
              isSym c    =  c `elem` "!@#$%&*+./<=>?\\^|:-~"
              isIdChar c =  isAlphanum c || c `elem` "_'"

              lexFracExp ('.':s) = [('.':ds++e,u) | (ds,t) <- lexDigits s,
                                                    (e,u)  <- lexExp t]
              lexFracExp s       = [("",s)]

              lexExp (e:s) | e `elem` "eE"
                       = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
                                                 (ds,u) <- lexDigits t] ++
                         [(e:ds,t)   | (ds,t) <- lexDigits s]
              lexExp s = [("",s)]

lexDigits               :: ReadS String
lexDigits               =  nonnull isDigit

nonnull                 :: (Char -> Bool) -> ReadS String
nonnull p s             =  [(cs,t) | (cs@(_:_),t) <- [span p s]]

lexLitChar              :: ReadS String
lexLitChar ('\\':s)     =  [('\\':esc, t) | (esc,t) <- lexEsc s]
        where
        lexEsc (c:s)     | c `elem` "abfnrtv\\\"'" = [([c],s)]
        lexEsc s@(d:_)   | isDigit d               = lexDigits s
        lexEsc _                                   = []
lexLitChar (c:s)        =  [([c],s)]
lexLitChar ""           =  []

instance  Show Int  where ...
instance  Read Int  where ...

instance  Show Integer  where ...
instance  Read Integer  where ...

instance  Show Float  where ...
instance  Read Float  where ...

instance  Show Double  where ...
instance  Read Double  where ...
```

```
instance  Show ()  where
    showsPrec p () = showString "()"

instance Read () where
    readsPrec p    = readParen False
                            (\r -> [((),t) | ("(",s) <- lex r,
                                             (")",t) <- lex s ] )

instance  Show Char  where ...
instance  Read Char  where ...

instance  (Show a) => Show [a]  where
    showsPrec p         = showList

instance  (Read a) => Read [a]  where
    readsPrec p         = readList

-- Tuples

instance  (Show a, Show b) => Show (a,b)  where
    showsPrec p (x,y) = showChar '(' . shows x . showChar ',' .
                                       shows y . showChar ')'

instance  (Read a, Read b) => Read (a,b)  where
    readsPrec p = readParen False
                            (\r -> [((x,y), w) | ("(",s) <- lex r,
                                                 (x,t)   <- reads s,
                                                 (",",u) <- lex t,
                                                 (y,v)   <- reads u,
                                                 (")",w) <- lex v ] )

-- et cetera

-- Functions

instance  Show (a -> b)  where
    showsPrec p f  =  showString "<<function>>"

instance  Show (IO a)  where
    showsPrec p f  =  showString "<<IO action>>"
```

A.3 Prelude PreludeIO

```
module PreludeIO (
      FilePath, IOError, fail, userError, catch,
      putChar, putStr, putStrLn, print,
      getChar, getLine, getContents, interact,
      readFile, writeFile, appendFile, readIO, readLn
    ) where

import PreludeBuiltin

type FilePath   =  String

data IOError    -- The internals of this type are system dependant

instance Show IOError where ...
instance Eq IOError where ...

fail            ::  IOError -> IO a
fail            =   primFail

userError       ::  String  -> IOError
userError       =   primUserError

catch           ::  IO a    -> (IOError -> IO a) -> IO a
catch           =   primCatch

putChar         :: Char -> IO ()
putChar         =  primPutChar

putStr          :: String -> IO ()
putStr s        =  mapM_ s putChar

putStrLn        :: String -> IO ()
putStrLn s      =  do putStr s
                      putStr "\n"

print           :: Show a => IO ()
print x         =  putStrLn (show x)

getChar         :: IO Char
getChar         =  primGetChar

getLine         :: IO String
getLine         =  do c <- getChar
                      if c == '\n' then return "" else
                         do s <- getLine
                            return (c:s)
```

```
getContents     :: IO String
getContents     =  primGetContents

interact        ::  (String -> String) -> IO ()
interact f      =   do s <- getContents
                       putStr (f s)

readFile        :: FilePath -> IO String
readFile        =  primReadFile

writeFile       :: FilePath -> String -> IO ()
writeFile       =  primWriteFile

appendFile      :: FilePath -> String -> IO ()
appendFile      =  primAppendFile

readIO          :: Read a => String -> IO a
  -- raises an exception instead of an error
readIO s        =  case [x | (x,t) <- reads s, ("","") <- lex t] of
                        [x] -> return x
                        []  -> fail (userError "PreludeIO.readIO: no parse")
                        _   -> fail (userError
                                      "PreludeIO.readIO: ambiguous parse")

readLn          :: Read a => IO a
readLn          =  do l <- getLine
                      r <- readIO l
                      return r
```

B Syntax

B.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:

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

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

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

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 "consume longest lexeme" rule. In the context-free syntax, this means that conditionals, let-expressions, and lambda abstractions extend to the right as far as possible.

B.2 Lexical Syntax

program	→	{ *lexeme* | *whitespace* }
lexeme	→	*varid* | *conid* | *varsym* | *consym* | *literal* | *special* | *reservedop* | *reservedid*
literal	→	*integer* | *float* | *char* | *string*
special	→	`(` | `)` | `,` | `;` | `[` | `]` | `_` | `` ` `` | `{` | `}`
whitespace	→	*whitestuff* {*whitestuff*}
whitestuff	→	*whitechar* | *comment* | *ncomment*
whitechar	→	*newline* | *vertab* | *formfeed* | *space* | *tab* | *nonbrkspc*

newline → a newline (system dependent)
space → a space
tab → a horizontal tab
vertab → a vertical tab
formfeed → a form feed
nonbrkspc → a non-breaking space
comment → `--` {*any*} *newline*
ncomment → `{-` *ANYseq* {*ncomment ANYseq*} `-}`
ANYseq → $\{ANY\}_{\langle \{ANY\}\ (\ \texttt{\{-}\ \mid\ \texttt{-\}}\ )\ \{ANY\} \rangle}$
ANY → *any* | *newline* | *vertab* | *formfeed*
any → *graphic* | *space* | *tab* | *nonbrkspc*
graphic → *large* | *small* | *digit* | *symbol* | *special* | `:` | `"` | `'`

small → *ASCsmall* | *ISOsmall*
ASCsmall → a | b | ... | z
ISOsmall → à | á | â | ã | ä | å | æ | ç | è | é | ê | ë
| ì | í | î | ï | đ | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | ý | *thorn* | ÿ | ß

large → *ASClarge* | *ISOlarge*
ASClarge → A | B | ... | Z
ISOlarge → À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë
| Ì | Í | Î | Ï | Đ | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
| Ù | Ú | Û | Ü | Ý | *Thorn*

symbol → *ASCsymbol* | *ISOsymbol*
ASCsymbol → ! | # | $ | % | & | * | + | . | / | < | = | > | ? | @
| \ | ^ | `|` | - | ~
ISOsymbol → ¡ | ¢ | £ | *currency* | ¥ | ¦ | § | ¨ | © | ª | «
| ¬ | – | ® | ¯ | ° | ± | ² | ³ | ´ | µ | ¶
| · | ¸ | ¹ | º | » | ¼ | ½ | ¾ | ¿ | × | ÷

digit → 0 | 1 | ... | 9
octit → 0 | 1 | ... | 7
hexit → *digit* | A | ... | F | a | ... | f

varid → $(small\ \{small \mid large \mid digit \mid \texttt{'} \mid \texttt{\_}\})_{\langle reservedid \rangle}$
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`
specialid → `as` | `qualified` | `hiding`

$$
\begin{array}{lcll}
\mathit{varsym} & \rightarrow & (\ \mathit{symbol}\ \{\mathit{symbol} \mid \texttt{:}\}\ )_{\langle \mathit{reservedop}\rangle} & \\
\mathit{consym} & \rightarrow & (\texttt{:}\ \{\mathit{symbol} \mid \texttt{:}\})_{\langle \mathit{reservedop}\rangle} & \\
\mathit{reservedop} & \rightarrow & \texttt{..} \mid \texttt{::} \mid \texttt{=} \mid \backslash \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} \mid \ @ \mid \texttt{\~{}} \mid \texttt{=>} & \\
\mathit{specialop} & \rightarrow & \texttt{-} \mid \texttt{!} & \\
 & & & \\
\mathit{varid} & & & \text{(variables)} \\
\mathit{conid} & & & \text{(constructors)} \\
\mathit{tyvar} & \rightarrow & \mathit{varid} & \text{(type variables)} \\
\mathit{tycon} & \rightarrow & \mathit{conid} & \text{(type constructors)} \\
\mathit{tycls} & \rightarrow & \mathit{conid} & \text{(type classes)} \\
\mathit{modid} & \rightarrow & \mathit{conid} & \text{(modules)} \\
 & & & \\
\mathit{qvarid} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{varid} & \\
\mathit{qconid} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{conid} & \\
\mathit{qtycon} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{tycon} & \\
\mathit{qtycls} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{tycls} & \\
\mathit{qvarsym} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{varsym} & \\
\mathit{qconsym} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{consym} & \\
 & & & \\
\mathit{decimal} & \rightarrow & \mathit{digit}\{\mathit{digit}\} & \\
\mathit{octal} & \rightarrow & \mathit{octit}\{\mathit{octit}\} & \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\{\mathit{hexit}\} & \\
 & & & \\
\mathit{integer} & \rightarrow & \mathit{decimal} & \\
 & \mid & \texttt{0o}\ \mathit{octal} \mid \texttt{0O}\ \mathit{octal} & \\
 & \mid & \texttt{0x}\ \mathit{hexadecimal} \mid \texttt{0X}\ \mathit{hexadecimal} & \\
\mathit{float} & \rightarrow & \mathit{decimal}\ \texttt{.}\ \mathit{decimal}[(\texttt{e} \mid \texttt{E})[\texttt{-} \mid \texttt{+}]\mathit{decimal}] & \\
 & & & \\
\mathit{char} & \rightarrow & \texttt{'}\ (\mathit{graphic}_{\langle \texttt{'} \mid \backslash\rangle} \mid \mathit{space} \mid \mathit{escape}_{\langle \backslash\texttt{\&}\rangle})\ \texttt{'} & \\
\mathit{string} & \rightarrow & \texttt{"}\ \{\mathit{graphic}_{\langle \texttt{"} \mid \backslash\rangle} \mid \mathit{space} \mid \mathit{escape} \mid \mathit{gap}\}\ \texttt{"} & \\
\mathit{escape} & \rightarrow & \backslash\ (\ \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 \backslash \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} & \\
 & \mid & \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} & \\
 & \mid & \texttt{DC1} \mid \texttt{DC2} \mid \texttt{DC3} \mid \texttt{DC4} \mid \texttt{NAK} \mid \texttt{SYN} \mid \texttt{ETB} \mid \texttt{CAN} & \\
 & \mid & \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 \backslash \mid \texttt{]} \mid \texttt{\^{}} \mid \texttt{\_} & \\
\mathit{gap} & \rightarrow & \backslash\ \mathit{whitechar}\ \{\mathit{whitechar}\}\ \backslash &
\end{array}
$$

B.3 Layout

Definitions: The indentation of a lexeme is the column number indicating the start 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 this tab convention: tab stops are 8 characters apart, and a tab character causes the insertion of enough spaces to align the current position with the next tab stop.

In the syntax given in the rest of the report, *declaration lists* are always preceded by the keyword `where`, `let`, `do`, or `of`, and are enclosed within curly braces (`{ }`) with the individual declarations separated by semicolons (`;`). For example, the syntax of a `let` expression is:

$$\texttt{let \{ } decl_1 \texttt{ ; } decl_2 \texttt{ ; } \ldots \texttt{ ; } decl_n \texttt{ [;] \} in } exp$$

Haskell permits the omission of the braces and semicolons by using *layout* to convey the same information. This allows both layout-sensitive and -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 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 declaration list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the declaration 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.

Given these rules, a single newline may actually terminate several declaration 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 declaration list.

To facilitate the use of layout at the top level of a module (an implementation may allow several modules may reside in one file), the keyword `module` and the end-of-file token are assumed to occur in column 0 (whereas normally the first column is 1). Otherwise, all top-level declarations would have to be indented.

Section 1.5 gives an example which uses the layout rule.

B.4 Context-Free Syntax

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

impdecls → $impdecl_1$ `;` ... `;` $impdecl_n$ ($n \geq 1$)

exports → `(` $export_1$ `,` ... `,` $export_n$ [`,`] `)`   ($n \geq 0$)

export → *qvar*
  | *qtycon* [`(..)` | `(` $qcname_1$ `,` ... `,` $qcname_n$ `)`]   ($n \geq 1$)
  | *qtycls* [`(..)` | `(` $qvar_1$ `,` ... `,` $qvar_n$ `)`]   ($n \geq 0$)
  | `module` *modid*
qcname → *qvar* | *qcon*

impdecl → `import` [`qualified`] *modid* [`as` *modid*] [*impspec*]
impspec → `(` $import_1$ `,` ... `,` $import_n$ [`,`] `)`   ($n \geq 0$)
  | `hiding` `(` $import_1$ `,` ... `,` $import_n$ [`,`] `)`   ($n \geq 0$)

import → *var*
  | *tycon* [`(..)` | `(` $cname_1$ `,` ... `,` $cname_n$ `)`]   ($n \geq 1$)
  | *tycls* [`(..)` | `(` var_1 `,` ... `,` var_n `)`]   ($n \geq 0$)
cname → *var* | *con*

fixdecls → fix_1 `;` ... `;` fix_n ($n \geq 1$)
fix → `infixl` [*digit*] *ops*
  | `infixr` [*digit*] *ops*
  | `infix` [*digit*] *ops*
ops → op_1 `,` ... `,` op_n ($n \geq 1$)

topdecls → $topdecl_1$ `;` ... `;` $topdecl_n$ ($n \geq 0$)
topdecl → `type` *simpletype* `=` *type*
  | `data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]
  | `newtype` [*context* `=>`] *simpletype* `=` *con* *atype* [*deriving*]
  | `class` [*context* `=>`] *simpleclass* [`where` `{` *cbody* [`;`] `}`]
  | `instance` [*context* `=>`] *qtycls* *inst* [`where` `{` *valdefs* [`;`] `}`]
  | `default` `(`$type_1$ `,` ... `,` $type_n$`)`   ($n \geq 0$)
  | *decl*

decls	$\rightarrow$	$decl_1$; ... ; $decl_n$	$(n \geq 0)$
decl	$\rightarrow$	*signdecl*	
	$\mid$	*valdef*	
decllist	$\rightarrow$	`{` *decls* [;] `}`	
signdecl	$\rightarrow$	*vars* :: [*context* `=>`] *type*	
vars	$\rightarrow$	var_1 , ..., var_n	$(n \geq 1)$
type	$\rightarrow$	*btype* [`->` *type*]	(function type)
btype	$\rightarrow$	[*btype*] *atype*	(type application)
atype	$\rightarrow$	*gtycon*	
	$\mid$	*tyvar*	
	$\mid$	`(` $type_1$, ... , $type_k$ `)`	(tuple type, $k \geq 2$)
	$\mid$	`[` *type* `]`	(list type)
	$\mid$	`(` *type* `)`	(parenthesised constructor)
gtycon	$\rightarrow$	*qtycon*	
	$\mid$	`()`	(unit type)
	$\mid$	`[]`	(list constructor)
	$\mid$	`(->)`	(function constructor)
	$\mid$	`(,{,})`	(tupling constructors)
context	$\rightarrow$	*class*	
	$\mid$	`(` $class_1$, ... , $class_n$ `)`	$(n \geq 1)$
class	$\rightarrow$	*qtycls tyvar*	
simpletype	$\rightarrow$	*tycon* $tyvar_1$ ... $tyvar_k$	
constrs	$\rightarrow$	$constr_1$ `|` ... `|` $constr_n$	$(n \geq 1)$
constrs	$\rightarrow$	$constr_1$ `|` ... `|` $constr_n$	$(n \geq 1)$
constr	$\rightarrow$	*con* [`!`] $atype_1$ ... [`!`] $atype_k$	(arity *con* $= k$, $k \geq 0$)
	$\mid$	(*btype* | `!` *atype*) *conop* (*btype* | `!` *atype*)	(infix *conop*)
	$\mid$	*con* `{` $fielddecl_1$, ... , $fielddecl_n$ `}`	$(n \geq 1)$
fielddecl	$\rightarrow$	*vars* :: (*type* | `!` *atype*)	
deriving	$\rightarrow$	`deriving` (*dclass* | `(`$dclass_1$, ... , $dclass_n$`)`)	$(n \geq 0)$
dclass	$\rightarrow$	*qtycls*	
simpleclass	$\rightarrow$	*tycls tyvar*	
cbody	$\rightarrow$	[*cmethods* [; *cdefaults*]]	
cmethods	$\rightarrow$	$signdecl_1$; ... ; $signdecl_n$	$(n \geq 1)$
cdefaults	$\rightarrow$	$valdef_1$; ... ; $valdef_n$	$(n \geq 1)$

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
valdefs → $valdef_1$ `;` … `;` $valdef_n$ — ($n \geq 0$)

valdef → *lhs* `=` *exp* [`where` *decllist*]
| *lhs* *gdrhs* [`where` *decllist*]

lhs → pat^0
| *funlhs*

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} $varop^{(\mathrm{r},i)}$ $rpat^i$

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}]
| $lexp^i$
| $rexp^i$
$lexp^i$ → ($lexp^i \mid exp^{i+1}$) $qop^{(\mathrm{l},i)}$ exp^{i+1}
$lexp^6$ → `-` exp^7
$rexp^i$ → exp^{i+1} $qop^{(\mathrm{r},i)}$ ($rexp^i \mid exp^{i+1}$)
exp^{10} → `\` $apat_1$ … $apat_n$ `->` *exp* — (lambda abstraction, $n \geq 1$)
| `let` *decllist* `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* `)` — (parenthesised expression)

	$\mid$	`(` exp_1 `,` ... `,` exp_k `)`	(tuple, $k \geq 2$)
	$\mid$	`[` exp_1 `,` ... `,` exp_k `]`	(list, $k \geq 1$)
	$\mid$	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	$\mid$	`[` exp `|` $qual_1$ `,` ... `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	$\mid$	`(` exp^{i+1} $qop^{(a,i)}$ `)`	(left section)
	$\mid$	`(` $qop^{(a,i)}$ exp^{i+1} `)`	(right section)
	$\mid$	$qcon$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
	$\mid$	$aexp_{\{qcon\}}$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled update, $n \geq 1$)
$qual$	$\rightarrow$	pat `<-` exp	
	$\mid$	`let` $decllist$	
	$\mid$	exp	
$alts$	$\rightarrow$	alt_1 `;` ... `;` alt_n	($n \geq 1$)
alt	$\rightarrow$	pat `->` exp [`where` $decllist$]	
	$\mid$	pat $gdpat$ [`where` $decllist$]	
$gdpat$	$\rightarrow$	gd `->` exp [$gdpat$]	
$stmts$	$\rightarrow$	exp [`;` $stmts$]	
	$\mid$	pat `<-` exp `;` $stmts$	
	$\mid$	`let` $decllist$ `;` $stmts$	
$fbinds$	$\rightarrow$	`{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	($n \geq 0$)
$fbind$	$\rightarrow$	$var \mid var$ `=` exp	
pat	$\rightarrow$	var `+` $integer$	(successor pattern)
	$\mid$	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}	
$lpat^6$	$\rightarrow$	`-` $(integer \mid float)$	(negative literal)
$rpat^i$	$\rightarrow$	pat^{i+1} $qconop^{(\mathrm{r},i)}$ $(rpat^i \mid pat^{i+1})$	
pat^{10}	$\rightarrow$	$apat$	
	$\mid$	$gcon$ $apat_1$ ... $apat_k$	(arity $gcon = k$, $k \geq 1$)
$apat$	$\rightarrow$	var [`@` $apat$]	(as pattern)
	$\mid$	$gcon$	(arity $gcon = 0$)
	$\mid$	$qcon$ `{` $fpat_1$ `,` ... `,` $fpat_k$ `}`	(labeled pattern, $k \geq 0$)
	$\mid$	$literal$	

	$\vert$	`_`	(wildcard)
	$\vert$	`(` *pat* `)`	(parenthesised pattern)
	$\vert$	`(` pat_1 `,` ... `,` pat_k `)`	(tuple pattern, $k \geq 2$)
	$\vert$	`[` pat_1 `,` ... `,` pat_k `]`	(list pattern, $k \geq 1$)
	$\vert$	`~` *apat*	(irrefutable pattern)
fpat	$\rightarrow$	*var* `=` *pat*	
	$\vert$	*var*	
gcon	$\rightarrow$	`()`	
	$\vert$	`[]`	
	$\vert$	`(,{,})`	
	$\vert$	*qcon*	
var	$\rightarrow$	*varid* $\vert$ `(` *varsym* `)`	(variable)
qvar	$\rightarrow$	*qvarid* $\vert$ `(` *qvarsym* `)`	(qualified variable)
con	$\rightarrow$	*conid* $\vert$ `(` *consym* `)`	(constructor)
qcon	$\rightarrow$	*qconid* $\vert$ `(` *qconsym* `)`	(qualified constructor)
varop	$\rightarrow$	*varsym* $\vert$ `` ` `` *varid* `` ` ``	(variable operator)
qvarop	$\rightarrow$	*qvarsym* $\vert$ `` ` `` *qvarid* `` ` ``	(qualified variable operator)
conop	$\rightarrow$	*consym* $\vert$ `` ` `` *conid* `` ` ``	(constructor operator)
qconop	$\rightarrow$	*qconsym* $\vert$ `` ` `` *qconid* `` ` ``	(qualified constructor operator)
op	$\rightarrow$	*varop* $\vert$ *conop*	(operator)
qop	$\rightarrow$	*qvarop* $\vert$ *qconop*	(qualified operator)

C 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. Within the program part, the usual "`--`" and "`{- -}`" comment conventions may still be used. 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 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 which are entirely enclosed between `\begin{code}`...`\end{code}` delimiters are treated as program text; all other lines are comment. 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.

D Specification of Derived Instances

A *derived instance* is an instance declaration which is generated automatically in conjunction with a `data` or `newtype` declaration. The body of a derived instance declaration is derived syntacticly from the definition of the associated type. Derived instances are possible only for classes known to the compiler: those defined in either the Prelude or a standard library. In this appendix, we describe the derivation of classes defined by the Prelude.

If T is an algebraic datatype declared by:

$$\texttt{data}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \ \texttt{=}\ \ K_1\ t_{11}\ \ldots\ t_{1k_1}\ \texttt{|}\ \cdots\ \texttt{|}\ K_n\ t_{n1}\ \ldots\ t_{nk_n}$$
$$\texttt{deriving}\ (C_1,\ \ldots,\ C_m)$$

(where $m \geq 0$ and the parentheses may be omitted if $m = 1$) then a derived instance declaration is possible for a class C if these conditions hold:

1. C is one of `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, or `Read`.
2. There is a context c' such that $c' \Rightarrow C\ t_{ij}$ holds for each of the constituent types t_{ij}.
3. If C is `Bounded`, the type must be either an enumeration (all constructors must by nullary) or have only one constructor.
4. If C is `Enum`, the type must be an enumeration.
5. There must be no explicit instance declaration elsewhere in the program which makes $T\ u_1\ \ldots\ u_k$ an instance of C.

For the purposes of derived instances, a `newtype` declaration is treated as a `data` declaration with a single constructor.

If the `deriving` form is present, an instance declaration is automatically generated for $T\ u_1\ \ldots\ u_k$ over each class C_i. If the derived instance declaration is impossible for any of the C_i then a static error results. If no derived instances are required, the `deriving` form may be omitted or the form `deriving ()` may be used.

Each derived instance declaration will have the form:

$$\texttt{instance}\ (c,\ C'_1\ u'_1,\ \ldots,\ C'_j\ u'_j\ )\ \texttt{=>}\ C_i\ (T\ u_1\ \ldots\ u_k)\ \texttt{where \{}\ d\ \texttt{\}}$$

where d is derived automatically depending on C_i and the data type declaration for T (as will be described in the remainder of this section), and u'_1 through u'_j form a subset of u_1 through u_k. When inferring the context for the derived instances, type synonyms must be expanded out first. Free names in the declarations d are all defined in the Prelude; the qualifier '`Prelude.`' is implicit here. The remaining details of the derived instances for each of the derivable Prelude classes are now given.

Derived instances of `Eq` and `Ord`. The class methods automatically introduced by derived instances of `Eq` and `Ord` are `(==)`, `(/=)`, `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min`. The latter seven operators are defined so as to compare their arguments lexicographically with respect to the constructor set given, with earlier constructors in the datatype declaration counting as smaller than later ones. For example, for the `Bool` datatype, we have that `(True > False) == True`.

Derived comparisons always traverse constructors from left to right. These examples illustrate this property:

```
(1,undefined) == (2,undefined) ⇒    False
(undefined,1) == (undefined,2) ⇒    ⊥
```

Derived instances of `Enum` Derived instance declarations for the class `Enum` are only possible for enumerations. The nullary constructors are assumed to be numbered left-to-right with the indices 0 through $n-1$. `Enum` introduces the class methods `toEnum`, `fromEnum`, `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo`, which are used to define arithmetic sequences as described in Section 3.10.

The `toEnum` and `fromEnum` operators map enumerated values to and from the `Int` type. `enumFrom n` returns a list corresponding to the complete enumeration of `n`'s type starting at the value `n`. Similarly, `enumFromThen n n'` is the enumeration starting at `n`, but with second element `n'`, and with subsequent elements generated at a spacing equal to the difference between `n` and `n'`. `enumFromTo` and `enumFromThenTo` are as defined by the default class methods for `Enum` (see Figure 5, page 66). For example, given the datatype:

```
data  Color = Red | Orange | Yellow | Green  deriving (Enum)
```

we would have:

```
[Orange..]          ==  [Orange, Yellow, Green]
fromEnum Yellow     ==  2
```

Derived instances of `Bounded`. The `Bounded` class introduces the class methods `minBound` and `maxBound`, which define the minimal and maximal elements of the type. For an enumeration, the first and last constructors listed in the `data` declaration are the bounds. For a type with a single constructor, the constructor is applied to the bounds for the constituent types. For example, the following datatype:

```
data  Pair a b = Pair a b deriving Bounded
```

would generate the following `Bounded` instance:

```
instance (Bounded a,Bounded b) => Bounded (Pair a b) where
  minBound = Pair minBound minBound
  maxBound = Pair maxBound maxBound
```

Derived instances of `Read` and `Show`. The class methods automatically introduced by derived instances of `Read` and `Show` are `showsPrec`, `readsPrec`, `showList`, and `readList`. They are used to coerce values into strings and parse strings into values.

The function `showsPrec d x r` accepts a precedence level `d` (a number from `0` to `10`), a value `x`, and a string `r`. It returns a string representing `x` concatenated to `r`. `showsPrec` satisfies the law:

```
showsPrec d x r ++ s  ==  showsPrec d x (r ++ s)
```

The representation will be enclosed in parentheses if the precedence of the top-level constructor operator in `x` is less than `d`. Thus, if `d` is `0` then the result is never surrounded in parentheses; if `d` is `10` it is always surrounded in parentheses, unless it is an atomic expression. The extra parameter `r` is essential if tree-like structures are to be printed in linear time rather than time quadratic in the size of the tree.

The function `readsPrec d s` accepts a precedence level `d` (a number from `0` to `10`) and a string `s`, and returns a list of pairs `(x,r)` such that `showsPrec d x r == s`. `readsPrec` is a parse function, returning a list of (parsed value, remaining string) pairs. If there is no successful parse, the returned list is empty.

`showList` and `readList` allow lists of objects to be represented using non-standard denotations. This is especially useful for strings (lists of `Char`).

`readsPrec` will parse any valid representation of the standard types apart from lists, for which only the bracketed form `[...]` is accepted. See Appendix A for full details.

A precise definition of the derived `Read` and `Show` instances for general types is beyond the scope of this report. However, the derived `Read` and `Show` instances have the following properties:

- The result of `show` is a syntactically correct Haskell expression containing only constants given the fixity declarations in force at the point where the type is declared.
- The result of `show` is readable by `read` if all component types are readable. (This is true for all instances defined in the Prelude but may not be true for user-defined instances.)
- The instance generated by `Read` allows arbitrary whitespace between tokens on the input string. Extra parenthesis are also allowed.
- The result of `show` contains only the constructor names defined in the data type, parenthesis, and spaces. When labeled constructor fields are used, braces, commas, field names, and equal signs are also used. No leading or trailing spaces are generated. Parenthesis are only added where needed. No line breaks are added.
- If a constructor is defined using labeled field syntax then the derived `show` for that constructor will this same syntax; the fields will be in the order declared in the `data` declaration. The derived `Read` instance will require this same syntax: all fields must be present and the declared order must be maintained.

- If a constructor is defined in the infix style, the derived `Show` instance will also use infix style. The derived `Read` instance will require that the constructor be infix.

The derived `Read` and `Show` instances may be unsuitable for some uses. Some problems include:

- Circular structures cannot be printed or read by these instances.
- The printer loses shared substructure; the printed representation of an object may be much larger that necessary.
- The parsing techniques used by the reader are very inefficient; reading a large structure may be quite slow.
- There is no user control over the printing of types defined in the Prelude. For example, there is no way to change the formatting of floating point numbers.

D.1 An example

As a complete example, consider a tree datatype:

```
data Tree a = Leaf a | Tree a :^: Tree a
     deriving (Eq, Ord, Read, Show)
```

Automatic derivation of instance declarations for `Bounded` and `Enum` are not possible, as `Tree` is not an enumeration or single-constructor datatype. The complete instance declarations for `Tree` are shown in Figure 8, Note the implicit use of default class method definitions—for example, only `<=` is defined for `Ord`, with the other class methods (`<`, `>`, `>=`, `max`, and `min`) being defined by the defaults given in the class declaration shown in Figure 5 (page 66).

```
infix 4 :^:
data Tree a =  Leaf a  |  Tree a :^: Tree a

instance (Eq a) => Eq (Tree a) where
        Leaf m == Leaf n  =  m==n
        u:^:v  == x:^:y   =  u==x && v==y
             _ == _       =  False

instance (Ord a) => Ord (Tree a) where
        Leaf m <= Leaf n  =  m<=n
        Leaf m <= x:^:y   =  True
        u:^:v  <= Leaf n  =  False
        u:^:v  <= x:^:y   =  u<x || u==x && v<=y

instance (Show a) => Show (Tree a) where

        showsPrec d (Leaf m) = showParen (d >= 10) showStr
          where
             showStr = showString "Leaf " . showsPrec 10 m

        showsPrec d (u :^: v) = showParen (d > 4) showStr
          where
             showStr = showsPrec 5 u .
                       showString " :^: " .
                       showsPrec 5 v

instance (Read a) => Read (Tree a) where

        readsPrec d r =  readParen (d > 4)
                         (\r -> [(u:^:v,w) |
                                 (u,s) <- readsPrec 5 r,
                                 (":^:",t) <- lex s,
                                 (v,w) <- readsPrec 5 t]) r

                      ++ readParen (d > 9)
                         (\r -> [(Leaf m,t) |
                                 ("Leaf",s) <- lex r,
                                 (m,t) <- readsPrec 10 s]) r
```

Figure 8: Example of Derived Instances

E Compiler Pragmas

Some compiler implementations support compiler *pragmas*, which are used to give additional instructions or hints to the compiler, but which do not form part of the Haskell language proper and do not change a program's semantics. This section summarizes this existing practice. An implementation is not required to respect any pragma, but the pragma should be ignored if an implementation is not prepared to handle it. Lexically, pragmas appear as comments, except that the enclosing syntax is `{-# #-}`.

E.1 Inlining

decl	$\rightarrow$	`{-# inline` [*digit*] *qvars* `#-}`
decl	$\rightarrow$	`{-# notInline` *qvars* `#-}`

The optional digit represents the level of optimization at which the inlining is to occur. If omitted, it is assumed to be 0. A compiler may use a numeric optimization level setting in which increasing level numbers indicate increasing amounts of optimization. Trivial inlinings which have no impact on compilation time or code size should have an optimization level of 0; more complex inlinings which may lead to slow compilation or large executables should be associated with higher optimization levels.

Compilers will often automatically inline simple expressions. This may be prevented by the `notInline` pragma.

E.2 Specialization

decl	$\rightarrow$	`{-# specialize` $spec_1$, ... , $spec_k$ `#-}`	$(k \geq 1)$
spec	$\rightarrow$	*vars* :: *type*	

Specialization is used to avoid inefficiencies involved in dispatching overloaded functions. For example, in

```
factorial :: Num a => a -> a
factorial 0 = 0
factorial n = n * factorial (n-1)
{-# specialize factorial :: Int -> Int,
               factorial :: Integer -> Integer #-}
```

calls to `factorial` in which the compiler can detect that the parameter is either `Int` or `Integer` will use specialized versions of `factorial` which do not involved overloaded numeric operations.

E.3 Optimization

decl	$\rightarrow$	*optdecl*	
exp^0	$\rightarrow$	*optdecl exp*^{0}	
optdecl	$\rightarrow$	`{-#` `optimize` *optd*$_1$ `,` … `,` *optd*$_k$ `#-}`	$(k \geq 1)$
optd	$\rightarrow$	*digit*	
	$\mid$	`speed` *digit*	
	$\mid$	`space` *digit*	
	$\mid$	`compilationSpeed` *digit*	
	$\mid$	`debug` *digit*	

The `optimize` pragma provides explicit control over the optimization levels of the compiler. If used as a declaration, this applies to all values defined in the declaration group (and recursively to any nested values). Used as an expression, it applies only to the prefixed expression. If no attribute is named, the `speed` attribute is assumed.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[2] H.B. Curry and R. Feys. *Combinatory Logic.* North-Holland Pub. Co., Amsterdam, 1958.

[3] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, N.M., January 1982.

[4] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[5] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical Report YALEU/DCS/RR-901, Yale University, May 1996.

[6] ISO. *A Character Set for Western European Languages.* ISO Standard 8859-1, 1989.

[7] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

[8] P. Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings*, pages 248–256, San Francisco, September 1981.

[9] J. Peterson (editor). The Haskell Library Report. Technical Report YALEU/DCS/RR-1105, Yale University, May 1996.

[10] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

[11] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.

Index

Index entries that refer to nonterminals in the Haskell syntax are shown in an *italic* font. Code entities defined in the standard prelude (Appendix A) or in the Haskell Library Report[9] are shown in `typewriter` font. Ordinary index entries are shown in a roman font.