Haskell 2010
Language Report

Simon Marlow
(editor)

Contents

Preface

> "*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* [3], 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 (and subsequent) 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.

Haskell 2010: language and libraries

The committee intended that Haskell would serve as a basis for future research in language design, and hoped that extensions or variants of the language would appear, incorporating experimental features.

Haskell has indeed evolved continuously since its original publication. By the middle of 1997, there had been five versions of the language design (from Haskell 1.0 – 1.4). At the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed; this became "Haskell 98" and was published in February 1999. The fixing of minor bugs led to the *Revised* Haskell 98 Report in 2002.

At the 2005 Haskell Workshop, the consensus was that so many extensions to the official language were widely used (and supported by multiple implementations), that it was worthwhile to define another iteration of the language standard, essentially to codify (and legitimise) the status quo.

The Haskell Prime effort was thus conceived as a relatively conservative extension of Haskell 98, taking on board new features only where they were well understood and widely agreed upon. It too was intended to be a "stable" language, yet reflecting the considerable progress in research on language design in recent years.

After several years exploring the design space, it was decided that a single monolithic revision of the language was too large a task, and the best way to make progress was to evolve the language in small incremental steps, each revision integrating only a small number of well-understood extensions and changes. Haskell 2010 is the first revision to be created in this way, and new revisions are expected once per year.

Extensions to Haskell 98

The most significant language changes in Haskell 2010 relative to Haskell 98 are listed here.

New language features:

- A Foreign Function Interface (FFI).
- Hierarchical module names, e.g. `Data.Bool`.
- Pattern guards.

Removed language features:

- The $(n + k)$ pattern syntax.

Haskell Resources

The Haskell web site `http://haskell.org` gives access to many useful resources, including:

- Online versions of the language and library definitions.

- Tutorial material on Haskell.
- Details of the Haskell mailing list.
- Implementations of Haskell.
- Contributed Haskell tools and libraries.
- Applications of Haskell.
- User-contributed wiki pages.
- News and events in the Haskell community.

You are welcome to comment on, suggest improvements to, and criticise the language or its presentation in the report, via the Haskell mailing list, or the Haskell wiki.

Building the language

Haskell was created, and continues to be sustained, by an active community of researchers and application programmers. Those who served on the Language and Library committees, in particular, devoted a huge amount of time and energy to the language. Here they are, with their affiliation(s) for the relevant period:

Arvind (MIT)
Lennart Augustsson (Chalmers University)
Dave Barton (Mitre Corp)
Brian Boutel (Victoria University of Wellington)
Warren Burton (Simon Fraser University)
Manuel M T Chakravarty (University of New South Wales)
Duncan Coutts (Well-Typed LLP)
Jon Fairbairn (University of Cambridge)
Joseph Fasel (Los Alamos National Laboratory)
John Goerzen
Andy Gordon (University of Cambridge)
Maria Guzman (Yale University)
Kevin Hammond [editor] (University of Glasgow)
Bastiaan Heeren (Utrecht University)
Ralf Hinze (University of Bonn)
Paul Hudak [editor] (Yale University)
John Hughes [editor] (University of Glasgow; Chalmers University)
Thomas Johnsson (Chalmers University)
Isaac Jones (Galois, inc.)
Mark Jones (Yale University, University of Nottingham, Oregon Graduate Institute)
Dick Kieburtz (Oregon Graduate Institute)
John Launchbury (University of Glasgow; Oregon Graduate Institute; Galois, inc.)
Andres Löh (Utrecht University)
Ian Lynagh (Well-Typed LLP)
Simon Marlow [editor] (Microsoft Research)
John Meacham
Erik Meijer (Utrecht University)

Ravi Nanavati (Bluespec, inc.)
Rishiyur Nikhil (MIT)
Henrik Nilsson (University of Nottingham)
Ross Paterson (City University, London)
John Peterson [editor] (Yale University)
Simon Peyton Jones [editor] (University of Glasgow; Microsoft Research Ltd)
Mike Reeve (Imperial College)
Alastair Reid (University of Glasgow)
Colin Runciman (University of York)
Don Stewart (Galois, inc.)
Martin Sulzmann (Informatik Consulting Systems AG)
Audrey Tang
Simon Thompson (University of Kent)
Philip Wadler [editor] (University of Glasgow)
Malcolm Wallace (University of York)
Stephanie Weirich (University of Pennsylvania)
David Wise (Indiana University)
Jonathan Young (Yale University)

Those marked [editor] served as the co-ordinating editor for one or more revisions of the language.

In addition, dozens of other people made helpful contributions, some small but many substantial. They are as follows: Hans Aberg, Kris Aerts, Sten Anderson, Richard Bird, Tom Blenko, Stephen Blott, Duke Briscoe, Paul Callaghan, Magnus Carlsson, Mark Carroll, Franklin Chen, Olaf Chitil, Chris Clack, Guy Cousineau, Tony Davie, Craig Dickson, Chris Dornan, Laura Dutton, Chris Fasel, Pat Fasel, Sigbjorn Finne, Michael Fryers, Peter Gammie, Andy Gill, Mike Gunter, Cordy Hall, Mark Hall, Thomas Hallgren, Matt Harden, Klemens Hemm, Fergus Henderson, Dean Herington, Bob Hiromoto, Nic Holt, Ian Holyer, Randy Hudson, Alexander Jacobson, Patrik Jansson, Robert Jeschofnik, Orjan Johansen, Simon B. Jones, Stef Joosten, Mike Joy, Wolfram Kahl, Stefan Kahrs, Antti-Juhani Kaijanaho, Jerzy Karczmarczuk, Kent Karlsson, Martin D. Kealey, Richard Kelsey, Siau-Cheng Khoo, Amir Kishon, Feliks Kluzniak, Jan Kort, Marcin Kowalczyk, Jose Labra, Jeff Lewis, Mark Lillibridge, Bjorn Lisper, Sandra Loosemore, Pablo Lopez, Olaf Lubeck, Christian Maeder, Ketil Malde, Michael Marte, Jim Mattson, John Meacham, Sergey Mechveliani, Gary Memovich, Randy Michelsen, Rick Mohr, Andy Moran, Graeme Moss, Arthur Norman, Nick North, Chris Okasaki, Bjarte M. Østvold, Paul Otto, Sven Panne, Dave Parrott, Larne Pekowsky, Rinus Plasmeijer, Ian Poole, Stephen Price, John Robson, Andreas Rossberg, George Russell, Patrick Sansom, Michael Schneider, Felix Schroeter, Julian Seward, Nimish Shah, Christian Sievers, Libor Skarvada, Jan Skibinski, Lauren Smith, Raman Sundaresh, Josef Svenningsson, Ken Takusagawa, Wolfgang Thaller, Satish Thatte, Tom Thomson, Tommy Thorn, Dylan Thurston, Mike Thyer, Mark Tullsen, David Tweed, Pradeep Varma, Keith Wansbrough, Tony Warnock, Michael Webber, Carl Witty, Stuart Wray, and Bonnie Yantis.

Finally, aside from the important foundational work laid by Church, Rosser, Curry, and others on the lambda calculus, it is right to acknowledge the influence of many noteworthy programming languages developed over the years. Although it is difficult to pinpoint the origin of many ideas, the following languages were particularly influential: 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.

Simon Marlow
Cambridge, April 2010

Part I

The Haskell 2010 Language

Chapter 1

Introduction

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

This 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 Chapter 5. Modules provide a way to control namespaces and to re-use software in large programs.
2. The top level of a module consists of a collection of *declarations*, of which there are several kinds, all described in Chapter 4. Declarations define things such as ordinary values, datatypes, type classes, and fixity information.
3. At the next lower level are *expressions*, described in Chapter 3. An expression denotes a *value* and has a *static type*; expressions are at the heart of Haskell programming "in the small."
4. At the bottom level is Haskell's *lexical structure*, defined in Chapter 2. The lexical structure captures the concrete representation of Haskell programs in text files.

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

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

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

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

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

1.2 The Haskell Kernel

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

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

1.3 Values and Types

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

Errors in Haskell are semantically equivalent to $\perp$ ("bottom"). Technically, they are indistinguishable from nontermination, so the language includes no mechanism for detecting or acting upon errors. However, implementations will probably try to provide useful information about errors. See Section 3.1.

1.4 Namespaces

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

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

2. An identifier must not be used as the name of a type constructor and a class in the same scope.

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

Chapter 2

Lexical Structure

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

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

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

Haskell uses the Unicode [2] character set. However, source programs are currently biased toward the ASCII character set used in earlier versions of Haskell.

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

2.2 Lexical Program Structure

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

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

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

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

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

ascSymbol → `!` | `#` | `$` | `%` | `&` | `⋆` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@`
  | `\` | `^` | `|` | `-` | `~` | `:`
uniSymbol → any Unicode symbol or punctuation
digit → *ascDigit* | *uniDigit*
ascDigit → `0` | `1` | ... | `9`
uniDigit → any Unicode decimal digit
octit → `0` | `1` | ... | `7`
hexit → *digit* | `A` | ... | `F` | `a` | ... | `f`

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

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

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

2.3 Comments

Comments are valid whitespace.

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

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

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

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

Nested comments are also used for compiler pragmas, as explained in Chapter 12.

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

2.4 Identifiers and Operators

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

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

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

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

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

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

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

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

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

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

Variables and type variables are represented by identifiers beginning with small letters, and the others by identifiers beginning with capitals; also, variables and constructors have infix forms, the other four do not. Module names are a dot-separated sequence of *conids*. Namespaces are also discussed in Section 1.4.

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

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

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

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

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

2.5 Numeric Literals

$$\begin{array}{lcl}
\mathit{decimal} & \rightarrow & \mathit{digit}\{\mathit{digit}\} \\
\mathit{octal} & \rightarrow & \mathit{octit}\{\mathit{octit}\} \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\{\mathit{hexit}\}
\end{array}$$

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

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

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

2.6 Character and String Literals

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

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

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

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

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

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

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

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

2.7 Layout

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

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

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

Section 10.3 gives a more precise definition of the layout rules.

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

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

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

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

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

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

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

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

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

Figure 2.1: A sample program

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

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

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

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

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

Figure 2.2: Sample program with layout expanded

Chapter 3

Expressions

In this chapter, 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. Free variables and constructors used in these translations always refer to entities defined by the `Prelude`. For example, "`concatMap`" used in the translation of list comprehensions (Section 3.11) means the `concatMap` defined by the `Prelude`, regardless of whether or not the identifier "`concatMap`" is in scope where the list comprehension is used, and (if it is in scope) what it is bound to.

exp	→	*infixexp* `::` [*context* `=>`] *type*	(expression type signature)
	|	*infixexp*	
infixexp	→	*lexp qop infixexp*	(infix operator application)
	|	`-` *infixexp*	(prefix negation)
	|	*lexp*	
lexp	→	`\` $apat_1$ … $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)
	|	`let` *decls* `in` *exp*	(let expression)
	|	`if` *exp* [`;`] `then` *exp* [`;`] `else` *exp*	(conditional)
	|	`case` *exp* `of` `{` *alts* `}`	(case expression)
	|	`do` `{` *stmts* `}`	(do expression)
	|	*fexp*	
fexp	→	[*fexp*] *aexp*	(function application)
aexp	→	*qvar*	(variable)
	|	*gcon*	(general constructor)
	|	*literal*	
	|	`(` *exp* `)`	(parenthesized expression)
	|	`(` exp_1 `,` … `,` exp_k `)`	(tuple, $k \geq 2$)
	|	`[` exp_1 `,` … `,` exp_k `]`	(list, $k \geq 1$)
	|	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	|	`[` *exp* `|` $qual_1$ `,` … `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	|	`(` *infixexp qop* `)`	(left section)
	|	`(` $qop_{\langle - \rangle}$ *infixexp* `)`	(right section)

$|$ $qcon$ { $fbind_1$, … , $fbind_n$ } (labeled construction, $n \geq 0$)
$|$ $aexp_{\langle qcon \rangle}$ { $fbind_1$, … , $fbind_n$ } (labeled update, $n \geq 1$)

Expressions involving infix operators are disambiguated by the operator's fixity (see Section 4.4.2). Consecutive unparenthesized operators with the same precedence must both be either left or right associative to avoid a syntax error. Given an unparenthesized expression "$x\ qop^{(a,i)}\ y\ qop^{(b,j)}\ z$" (where $qop^{(a,i)}$ means an operator with associativity a and precedence i), 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.

An example algorithm for resolving expressions involving infix operators is given in Section 10.6.

Negation is the only prefix operator in Haskell; it has the same precedence as the infix - operator defined in the Prelude (see Section 4.4.2, Figure 4.1).

The grammar is ambiguous regarding the extent of lambda abstractions, let expressions, and conditionals. The ambiguity is resolved by the meta-rule that each of these constructs extends as far to the right as possible.

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 will assume that expressions involving infix operators have been resolved according to the fixities of the operators.

3.1 Errors

Errors during expression evaluation, denoted by $\perp$ ("bottom"), are indistinguishable by a Haskell program from non-termination. Since Haskell is a non-strict 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, Operators, and Literals

aexp	→	*qvar*	(variable)
	$\mid$	*gcon*	(general constructor)
	$\mid$	*literal*	

gcon	→	`()`	
	$\mid$	`[]`	
	$\mid$	`(,{,})`	
	$\mid$	*qcon*	

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

Haskell provides special syntax to support infix notation. An *operator* is a function that can be applied using infix syntax (Section 3.4), or partially applied using a *section* (Section 3.5).

An *operator* is either an *operator symbol*, such as `+` or `$$`, or is an ordinary identifier enclosed in grave accents (backquotes), such as `` `op` ``. For example, instead of writing the prefix application `op x y`, one can write the infix application ``x `op` y``. If no fixity declaration is given for `` `op` `` then it defaults to highest precedence and left associativity (see Section 4.4.2).

Dually, an operator symbol can be converted to an ordinary identifier by enclosing it in parentheses. For example, `(+) x y` is equivalent to `x + y`, and `foldr (*) 1 xs` is equivalent to `foldr (\x y -> x*y) 1 xs`.

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 6.4.1).
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	→	[*fexp*] *aexp*	(function application)
lexp	→	`\` $apat_1$ ... $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)

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; it may legally be written as `\(x:xs)->x`.

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

Translation: The following identity holds:

$$\backslash\ p_1\ \ldots\ p_n\ \texttt{->}\ e \quad = \quad \backslash\ x_1\ \ldots\ x_n\ \texttt{->}\ \texttt{case}\ \texttt{(}x_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{)}\ \texttt{of}\ \texttt{(}p_1\texttt{,}\ \ldots\texttt{,}\ p_n\texttt{)}\ \texttt{->}\ e$$

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 $\bot$.

3.4 Operator Applications

infixexp	→	*lexp qop infixexp*	
	|	`-` *infixexp*	(prefix negation)
	|	*lexp*	
qop	→	*qvarop* | *qconop*	(qualified operator)

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 Table 4.1). 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: The following identities hold:

$$\begin{array}{lcl} e_1\ op\ e_2 & = & \texttt{(}op\texttt{)}\ e_1\ e_2 \\ \texttt{-}e & = & \texttt{negate}\ (e) \end{array}$$

3.5 Sections

aexp	$\rightarrow$	`(` *infixexp qop* `)`	(left section)
	$\vert$	`(` $qop_{\langle - \rangle}$ *infixexp* `)`	(right section)

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.

Syntactic precedence rules apply to sections as follows. `(`*op e*`)` is legal if and only if `(x` *op e*`)` parses in the same way as `(x` *op* `(`*e*`))`; and similarly for `(`*e op*`)`. For example, `(*a+b)` is syntactically invalid, but `(+a*b)` and `(*(a+b))` are valid. Because `(+)` is left associative, `(a+b+)` is syntactically correct, but `(+a+b)` is not; the latter may legally be written as `(+(a+b))`. As another example, the expression

```
(let n = 10 in n +)
```

is invalid because, by the let/lambda meta-rule (Section 3), the expression

```
(let n = 10 in n + x)
```

parses as

```
(let n = 10 in (n + x))
```

rather than

```
((let n = 10 in n) + x)
```

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: The following identities hold:

$$\begin{array}{lcl} (op\ e) & = & \backslash\ x \texttt{ -> } x\ op\ e \\ (e\ op) & = & \backslash\ x \texttt{ -> } e\ op\ x \end{array}$$

where *op* is a binary operator, e is an expression, and x is a variable that does not occur free in e.

3.6 Conditionals

lexp	$\rightarrow$	`if` *exp* [`;`] `then` *exp* [`;`] `else` *exp*

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 $\bot$ otherwise.

Translation: The following identity holds:

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \quad = \quad \texttt{case}\ e_1\ \texttt{of \{ True ->}\ e_2\ \texttt{; False ->}\ e_3\ \texttt{\}}$$

where `True` and `False` are the two nullary constructors from the type `Bool`, as defined in the Prelude. The type of e_1 must be `Bool`; e_2 and e_3 must have the same type, which is also the type of the entire conditional expression.

3.7 Lists

$infixexp \rightarrow exp_1\ qop\ exp_2$
$aexp \rightarrow \texttt{[}\ exp_1\ \texttt{,}\ \ldots\ \texttt{,}\ exp_k\ \texttt{]} \qquad (k \geq 1)$
$\quad\ \mid\ gcon$
$gcon \rightarrow \texttt{[]}$
$\quad\ \mid\ qcon$
$qcon \rightarrow \texttt{(}\ gconsym\ \texttt{)}$
$qop \rightarrow qconop$
$qconop \rightarrow gconsym$
$gconsym \rightarrow \texttt{:}$

Lists are written $\texttt{[}e_1\texttt{, \ldots, }e_k\texttt{]}$, where $k \geq 1$. The list constructor is `:`, and the empty list is denoted `[]`. Standard operations on lists are given in the Prelude (see Section 6.1.3, and Chapter 9 notably Section 9.1).

Translation: The following identity holds:

$$\texttt{[}e_1\texttt{, \ldots, }e_k\texttt{]} \quad = \quad e_1\ \texttt{: (}e_2\ \texttt{: (\ldots (}e_k\ \texttt{: []))))}$$

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 $\texttt{[}t\texttt{]}$ (see Section 4.1.2).

The constructor "`:`" is reserved solely for list construction; like `[]`, it is considered part of the language syntax, and cannot be hidden or redefined. It is a right-associative operator, with precedence level 5 (Section 4.4.2).

3.8 Tuples

$aexp \rightarrow \texttt{(}\ exp_1\ \texttt{,}\ \ldots\ \texttt{,}\ exp_k\ \texttt{)} \qquad (k \geq 2)$
$\quad\ \mid\ qcon$
$qcon \rightarrow \texttt{(,}\{\texttt{,}\}\texttt{)}$

Tuples are written $\texttt{(}e_1\texttt{, \ldots, }e_k\texttt{)}$, and may be of arbitrary length $k \geq 2$. The constructor for an n-tuple is denoted by `(,...,)`, where there are $n - 1$ commas. Thus `(a,b,c)` and `(,,) a b c` denote the same value. Standard operations on tuples are given in the Prelude (see Section 6.1.4 and Chapter 9).

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 4.1.2).

3.9 Unit Expressions and Parenthesized Expressions

aexp → *gcon*
| `(` *exp* `)`
gcon → `()`

The form `(`e`)` is simply a *parenthesized expression*, and is equivalent to e. The *unit expression* `()` has type `()` (see Section 4.1.2). It is the only member of that type apart from $\bot$, and 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 *arithmetic sequence* `[`e_1`,` e_2 `..` e_3`]` denotes a list of values of type t, where each of the e_i has type t, and t is an instance of class `Enum`.

Translation: Arithmetic sequences satisfy these identities:

$$\begin{array}{lcl}
[\; e_1 .. \;] & = & \texttt{enumFrom}\ e_1 \\
[\; e_1 , e_2 .. \;] & = & \texttt{enumFromThen}\ e_1\ e_2 \\
[\; e_1 .. e_3 \;] & = & \texttt{enumFromTo}\ e_1\ e_3 \\
[\; e_1 , e_2 .. e_3 \;] & = & \texttt{enumFromThenTo}\ e_1\ e_2\ e_3
\end{array}$$

where `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo` are class methods in the class `Enum` as defined in the Prelude (see Figure 6.1).

The semantics of arithmetic sequences therefore depends entirely on the instance declaration for the type t. See Section 6.3.4 for more details of which `Prelude` types are in `Enum` and their semantics.

3.11 List Comprehensions

aexp → `[` *exp* `|` $qual_1$ `,` … `,` $qual_n$ `]` (list comprehension, $n \geq 1$)
qual → *pat* `<-` *exp* (generator)
| `let` *decls* (local declaration)
| *exp* (boolean guard)

A *list comprehension* has the form `[` e `|` q_1`,` $\ldots$`,` 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`]`
- *local bindings* that provide new definitions for use in the generated expression e or subsequent boolean guards and generators
- *boolean guards*, which are arbitrary expressions of type `Bool`.

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 boolean 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 | True ]         =  [e]
[ e | q ]            =  [ e | q, True ]
[ e | b, Q ]         =  if b then [ e | Q ] else []
[ e | p <- l, Q ]    =  let ok p = [ e | Q ]
                            ok _ = []
                        in concatMap ok l
[ e | let decls, Q ] =  let decls in [ e | Q ]
```

where e ranges over expressions, p over patterns, l over list-valued expressions, b over boolean expressions, *decls* over declaration lists, q over qualifiers, and Q over sequences of qualifiers. `ok` is a fresh variable. The function `concatMap`, and boolean value `True`, are 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 4.5.4).

3.12 Let Expressions

lexp → `let` *decls* `in` *exp*

Let expressions have the general form `let {` d_1 `;` $\ldots$ `;` 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 Chapter 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` is 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 4.4.3. Once done, these identities hold, which may be used as a translation into the kernel:

```
let {p1=e1; ... ; pn=en} in e0   =   let (~p1, ... ,~pn) = (e1, ... ,en) in e0
let p = e1  in  e0               =   case e1 of ~p -> e0
                                     where no variable in p appears free in e1
let p = e1  in  e0               =   let p = fix ( \ ~p -> e1) in e0
```

where `fix` is the least fixpoint operator. Note the use of the irrefutable patterns ~p. 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 4.4.3.

3.13 Case Expressions

lexp	→	`case` *exp* `of` `{` *alts* `}`	
alts	→	*alt*$_1$ `;` ... `;` *alt*$_n$	$(n \geq 1)$
alt	→	*pat* `->` *exp* [`where` *decls*]	
	|	*pat* *gdpat* [`where` *decls*]	
	|		(*empty alternative*)
gdpat	→	*guards* `->` *exp* [*gdpat*]	
guards	→	`|` *guard*$_1$, ..., *guard*$_n$	$(n \geq 1)$
guard	→	*pat* `<-` *infixexp*	(pattern guard)
	|	`let` *decls*	(local declaration)
	|	*infixexp*	(boolean guard)

A *case expression* has the general form

```
case e of { p1 match1 ; ... ; pn matchn }
```

where each $match_i$ is of the general form

```
| gsi1    -> ei1
...
| gsimi   -> eimi
where declsi
```

(Notice that in the syntax rule for *guards*, the "`|`" is a terminal symbol, not the syntactic metasymbol for alternation.) Each alternative p_i $match_i$ consists of a pattern p_i and its matches, $match_i$. Each match in turn consists of a sequence of pairs of guards gs_{ij} and bodies e_{ij} (expressions), followed by optional bindings ($decls_i$) that scope over all of the guards and expressions of the alternative.

A *guard* has one of the following forms:

- *pattern guards* are of the form p `<-` e, where p is a pattern (see Section 3.17) of type t and e is an expression type t[1]. They succeed if the expression e matches the pattern p, and introduce the bindings of the pattern to the environment.
- *local bindings* are of the form `let` *decls*. They always succeed, and they introduce the names defined in *decls* to the environment.
- *boolean guards* are arbitrary expressions of type `Bool`. They succeed if the expression evaluates to `True`, and they do not introduce new names to the environment. A boolean guard, g, is semantically equivalent to the pattern guard `True <-` g.

An alternative of the form

pat `->` *exp* `where` *decls*

is treated as shorthand for:

pat `| True -> ` *exp*
`where` *decls*

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 alternatives are tried sequentially, from top to bottom. If e matches the pattern of an alternative, then the guarded expressions for that alternative are tried sequentially from top to bottom in the environment of the case expression extended first by the bindings created during the matching of the pattern, and then by the $decls_i$ in the `where` clause associated with that alternative.

For each guarded expression, the comma-separated guards are tried sequentially from left to right. If all of them succeed, then the corresponding expression is evaluated in the environment extended with the bindings introduced by the guards. That is, the bindings that are introduced by a guard (either by using a let clause or a pattern guard) are in scope in the following guards and the corresponding expression. If any of the guards fail, then this guarded expression fails and the next guarded expression is tried.

If none of the guarded expressions for a given alternative succeed, then matching continues with the next alternative. If no alternative succeeds, then the result is $\perp$. Pattern matching is described in Section 3.17, with the formal semantics of case expressions in Section 3.17.3.

A note about parsing. The expression

```
case x of { (a,_) | let b = not a in b :: Bool -> a }
```

is tricky to parse correctly. It has a single unambiguous parse, namely

```
case x of { (a,_) | (let b = not a in b :: Bool) -> a }
```

However, the phrase `Bool -> a` is syntactically valid as a type, and parsers with limited lookahead may incorrectly commit to this choice, and hence reject the program. Programmers are advised, therefore, to avoid guards that end with a type signature — indeed that is why a *guard* contains an *infixexp* not an *exp*.

---

[1] Note that the syntax of a pattern guard is the same as that of a generator in a list comprehension. The contextual difference is that, in a list comprehension, a pattern of type t goes with an expression of type `[`t`]`.

3.14 Do Expressions

lexp → `do {` *stmts* `}` (do expression)
stmts → $stmt_1$ … $stmt_n$ *exp* [`;`] ($n \geq 0$)
stmt → *exp* `;`
| *pat* `<-` *exp* `;`
| `let` *decls* `;`
| `;` (*empty statement*)

A *do expression* provides a more conventional syntax for monadic programming. It allows an expression such as

```
putStr "x: "     >>
getLine          >>= \l ->
return (words l)
```

to be written in a more traditional way as:

```
do putStr "x: "
   l <- getLine
   return (words l)
```

Translation: Do expressions satisfy these identities, which may be used as a translation into the kernel, after eliminating empty *stmts*:

```
do {e}                  =  e
do {e;stmts}            =  e >> do {stmts}
do {p <- e; stmts}      =  let ok p = do {stmts}
                                  ok _ = fail "..."
                                in e >>= ok
do {let decls; stmts}   =  let decls in do {stmts}
```

The ellipsis `"..."` stands for a compiler-generated error message, passed to `fail`, preferably giving some indication of the location of the pattern-match failure; the functions `>>`, `>>=`, and `fail` are operations in the class `Monad`, as defined in the Prelude; and `ok` is a fresh identifier.

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 define field labels (see Section 4.2.1). 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 label can be used in more than one constructor provided the field has the same typing in all constructors. To illustrate the last point, consider:

```
data S = S1 { x :: Int } | S2 { x :: Int }    -- OK
data T = T1 { y :: Int } | T2 { y :: Bool }   -- BAD
```

Here `S` is legal but `T` is not, because `y` is given inconsistent typings in the latter.

3.15.1 Field Selection

$aexp \rightarrow qvar$

Field labels are used as selector functions. When used as a variable, a field label serves as a function that 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 record construction (Section 3.15.2) and update (Section 3.15.3), field labels cannot be confused with ordinary variables.

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

$$f\ \texttt{x}\ \ \texttt{=}\ \ \texttt{case x of \{}\ C_1\ p_{11}\ \ldots\ p_{1k}\ \texttt{->}\ e_1\ \texttt{;}\ \ldots\ \texttt{;}\ C_n\ p_{n1}\ \ldots\ p_{nk}\ \texttt{->}\ e_n\ \texttt{\}}$$

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

$aexp \rightarrow qcon\ \texttt{\{}\ fbind_1\ \texttt{,}\ \ldots\ \texttt{,}\ fbind_n\ \texttt{\}}$ (labeled construction, $n \geq 0$)
$fbind \rightarrow qvar = exp$

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 labels is subject to the following constraints:

- Only field labels declared with the specified constructor may be mentioned.
- A field label may not be mentioned more than once.
- Fields not mentioned are initialized to $\perp$.
- 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.

The expression `F {}`, where `F` is a data constructor, is legal *whether or not* `F` *was declared with record syntax* (provided `F` has no strict fields — see the fourth bullet above); it denotes `F` $\perp_1 \ldots \perp_n$, where n is the arity of `F`.

Translation: In the binding $f = v$, the field f labels v.

$$C\ \{\ bs\ \}\quad =\quad C\ (pick^{C}_{1}\ bs\ \texttt{undefined})\ \ldots\ (pick^{C}_{k}\ bs\ \texttt{undefined})$$

where 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 label 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

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

Values belonging to a datatype with field labels 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.

Translation: Using the prior definition of $pick$,

$$\begin{array}{lll}
e\ \{\ bs\ \} & = & \texttt{case}\ e\ \texttt{of} \\
 & & \quad C_1\ v_1\ \ldots\ v_{k_1}\ \texttt{->}\ C_1\ (pick^{C_1}_{1}\ bs\ v_1)\ \ldots\ (pick^{C_1}_{k_1}\ bs\ v_{k_1}) \\
 & & \quad \ldots \\
 & & \quad C_j\ v_1\ \ldots\ v_{k_j}\ \texttt{->}\ C_j\ (pick^{C_j}_{1}\ bs\ v_1)\ \ldots\ (pick^{C_j}_{k_j}\ bs\ v_{k_j}) \\
 & & \quad \texttt{\_ -> error "Update error"}
\end{array}$$

where $\{C_1, \ldots, C_j\}$ is the set of constructors containing all labels in bs, 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. This example translates expressions using constructors in field-label notation into equivalent expressions using the same constructors without field labels. A compile-time error will result if no single constructor defines the set of field labels 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 4.1.2); 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.

Translation:

e `::` t = `let {` v `::` t`;` v `=` e `} in` v

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	→	*lpat qconop pat*	(infix constructor)
	|	*lpat*	
lpat	→	*apat*	
	|	`-` (*integer* | *float*)	(negative literal)
	|	*gcon* $apat_1$ … $apat_k$	(arity *gcon* $= k$, $k \geq 1$)
apat	→	*var* [`@` *apat*]	(as pattern)
	|	*gcon*	(arity *gcon* $= 0$)
	|	*qcon* `{` $fpat_1$ `,` … `,` $fpat_k$ `}`	(labeled pattern, $k \geq 0$)
	|	*literal*	
	|	`_`	(wildcard)
	|	`(` *pat* `)`	(parenthesized pattern)
	|	`(` pat_1 `,` … `,` pat_k `)`	(tuple pattern, $k \geq 2$)
	|	`[` pat_1 `,` … `,` pat_k `]`	(list pattern, $k \geq 1$)
	|	`~` *apat*	(irrefutable pattern)
fpat	→	*qvar* `=` *pat*	

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. For example, this definition is illegal:

```
f (x,x) = x     -- ILLEGAL; x used twice in pattern
```

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 }
```

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 the following rules:

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

   Operationally, this means that no matching is done on a ˜ $apat$ pattern until one of the variables in $apat$ 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.
3. Matching the wildcard pattern _ against any value always succeeds, and no binding is done.
4. Matching the pattern $con\ pat$ against a value, where con is a constructor defined by `newtype`, depends on the value:
   - If the value is of the form $con\ v$, then pat is matched against v.
   - If the value is $\perp$, then pat is matched against $\perp$.

   That is, constructors associated with `newtype` serve only to change the type of a value.
5. Matching the pattern $con\ pat_1\ \ldots\ pat_n$ against a value, where con is a constructor defined by `data`, depends on the value:

  - If the value is of the form $con\ v_1\ \ldots\ v_n$, sub-patterns are matched 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.
  - If the value is of the form $con'\ v_1\ \ldots\ v_m$, where con is a different constructor to con', the match fails.
  - If the value is $\bot$, the match diverges.

6. 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 _).
7. Matching a numeric, character, or string literal pattern k against a value v succeeds if $v\ ==\ k$, where == is overloaded based on the type of the pattern. The match diverges if this test diverges.

   The interpretation of numeric literals is exactly as described in Section 3.2; that is, the overloaded function `fromInteger` or `fromRational` is applied to an `Integer` or `Rational` literal (resp) to convert it to the appropriate type.
8. Matching an as-pattern var`@`$apat$ against a value v is the result of matching $apat$ against v, augmented with the binding of var to v. If the match of $apat$ against v 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), the following static class constraints hold:

- An integer literal pattern can only be matched against a value in the class `Num`.
- A floating literal pattern can only be matched against a value in the class `Fractional`.

It is sometimes helpful to distinguish two kinds of patterns. Matching an *irrefutable pattern* is non-strict: the pattern matches even if the value to be matched is $\bot$. Matching a *refutable* pattern is strict: if the value to be matched is $\bot$ the match diverges. The irrefutable patterns are as follows: 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*.

Here are some examples:

1. If the pattern `['a','b']` is matched against `['x',`$\bot$`]`, then `'a'` *fails* to match against `'x'`, and the result is a failed match. But if `['a','b']` is matched against `[`$\bot$`,'x']`, then attempting to match `'a'` against $\bot$ 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) ⊥   ⇒   ⊥:⊥:⊥
```

3. Consider the following declarations:

   ```
   newtype N = N Bool
   data    D = D !Bool
   ```

   These examples illustrate the difference in pattern matching between types defined by `data` and `newtype`:

   ```
   (\  (N True) -> True) ⊥     ⇒    ⊥
   (\  (D True) -> True) ⊥     ⇒    ⊥
   (\ ~(D True) -> True) ⊥     ⇒    True
   ```

   Additional examples may be found in Section 4.2.3.

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*. See Section 3.13 for the syntax and semantics of guards.

The guard semantics have an 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 :: (Int,Int,Int) -> [Int] -> Int
f ~(x,y,z) [a] | (a == y) = 1
```

both `a` and `y` will be evaluated by == in the guard.

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 3.1–3.3. 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 3.1–3.3: e, e' and e_i are expressions; g_i and gs_i are guards and sequences of guards respecively; p and p_i are patterns; v, x, and x_i are variables; K and K' are algebraic datatype (`data`) constructors (including tuple constructors); and N is a `newtype` constructor.

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 $gs_{i,j}$ in the $match_i$ forms. Subsequent identities manipulate the resulting `case` expression into simpler and simpler forms.

Rule (h) in Figure 3.2 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), (j), and (q) use a lambda rather than a `let`; this indicates that variables bound by `case` are monomorphically typed (Section 4.1.4).

```
(a)  case e of { alts } = (\v -> case v of { alts }) e
     where v is a new variable
(b)  case v of { p_1 match_1;  ... ; p_n match_n }
     =  case v of { p_1 match_1 ;
                    _  -> ... case v of {
                                 p_n match_n ;
                                 _  -> error "No match" }...}
      where each match_i has the form:
       | gs_{i,1}  -> e_{i,1} ; ... ; | gs_{i,m_i} -> e_{i,m_i} where { decls_i }
(c)  case v of { p | gs_1 -> e_1 ; ...
                   | gs_n -> e_n where { decls }
                 _      -> e' }
     = case e' of { y ->
        case v of {
          p -> let { decls } in
                case () of {
                  () | gs_1 -> e_1;
                  _ -> ... case () of {
                               () | gs_n -> e_n;
                               _  -> y } ... }
          _ -> y }}
     where y is a new variable
(d)  case v of { ~p -> e; _ -> e' }
     = (\x_1 ... x_n -> e ) (case v of { p-> x_1 }) ... (case v of { p -> x_n})
     where x_1, ..., x_n are all the variables in p
(e)  case v of { x@p -> e; _ -> e' }
     =  case v of { p -> ( \ x -> e ) v ; _ -> e' }
(f)  case v of { _ -> e; _ -> e' } = e
```

Figure 3.1: Semantics of Case Expressions, Part 1

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

(h) $\mathtt{case}\ v\ \mathtt{of}\ \{\ k\ \mathtt{->}\ e\mathtt{;}\ \_\ \mathtt{->}\ e'\ \}\ =\ \mathtt{if}\ (v\mathtt{==}k)\ \mathtt{then}\ e\ \mathtt{else}\ e'$
where k is a numeric, character, or string literal

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

(j) $\mathtt{case}\ v\ \mathtt{of}\ \{\ x\ \mathtt{->}\ e\ \}\ =\ (\ \backslash\ x\ \mathtt{->}\ e\ )\ v$

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

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

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

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

(o) $\mathtt{case}\ v\ \mathtt{of}\ \{\ K\ \{\}\ \mathtt{->}\ e\ \mathtt{;}\ \_\ \mathtt{->}\ e'\ \}$
$=\ \mathtt{case}\ v\ \mathtt{of}\ \{$
$\quad K\,\_\ldots\_\ \mathtt{->}\ e\ \mathtt{;}\ \_\ \mathtt{->}\ e'\ \}$

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

(q) $\mathtt{case}\ (K\ e_1\ \ldots\ e_n)\ \mathtt{of}\ \{\ K\ x_1\ \ldots\ x_n\ \mathtt{->}\ e\mathtt{;}\ \_\ \mathtt{->}\ e'\ \}$
$=\ (\backslash x_1\ \ldots\ x_n\ \mathtt{->}\ e)\ e_1\ \ldots\ e_n$
where K is a `data` constructor of arity n

(r) $\mathtt{case}\,\bot\,\mathtt{of}\ \{\ K\ x_1\ \ldots\ x_n\ \mathtt{->}\ e\mathtt{;}\ \_\ \mathtt{->}\ e'\ \}\ =\ \bot$
where K is a `data` constructor of arity n

Figure 3.2: Semantics of Case Expressions, Part 2

```
(s)  case () of { () | $g_1$, ..., $g_n$ -> $e$; _ -> $e'$ }
     = case () of {
          () | $g_1$ -> ... case () of {
                                () | $g_n$ -> $e$;
                               _ -> $e'$ } ...
         _ -> $e'$ }
     where $y$ is a new variable

(t)  case () of { () | $p$ <- $e_0$ -> $e$; _ -> $e'$ }
     = case $e_0$ of { $p$ -> $e$; _ -> $e'$ }
(u)  case () of { () | let $decls$ -> $e$; _ -> $e'$ }
     = let $decls$ in $e$

(v)  case () of { () | $e_0$ -> $e$; _ -> $e'$ }
     = if $e_0$ then $e$ else $e'$
```

Figure 3.3: Semantics of Case Expressions, Part 3

Chapter 4

Declarations and Bindings

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

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

topdecls → $topdecl_1$ `;` … `;` $topdecl_n$ $(n \geq 1)$
topdecl → `type` *simpletype* = *type*
| `data` [*context* =>] *simpletype* [= *constrs*] [*deriving*]
| `newtype` [*context* =>] *simpletype* = *newconstr* [*deriving*]
| `class` [*scontext* =>] *tycls* *tyvar* [`where` *cdecls*]
| `instance` [*scontext* =>] *qtycls* *inst* [`where` *idecls*]
| `default` `(`$type_1$ `,` … `,` $type_n$`)` $(n \geq 0)$
| `foreign` *fdecl*
| *decl*

decls → `{` $decl_1$ `;` … `;` $decl_n$ `}` $(n \geq 0)$
decl → *gendecl*
| (*funlhs* | *pat*) *rhs*

cdecls → `{` $cdecl_1$ `;` … `;` $cdecl_n$ `}` $(n \geq 0)$
cdecl → *gendecl*
| (*funlhs* | *var*) *rhs*

idecls → `{` $idecl_1$ `;` … `;` $idecl_n$ `}` $(n \geq 0)$
idecl → (*funlhs* | *var*) *rhs*
| (empty)

gendecl → *vars* `::` [*context* =>] *type* (type signature)
| *fixity* [*integer*] *ops* (fixity declaration)
| (empty declaration)

ops → op_1 `,` … `,` op_n $(n \geq 1)$
vars → var_1 `,` … `,` var_n $(n \geq 1)$
fixity → `infixl` | `infixr` | `infix`

The declarations in the syntactic category *topdecls* are only allowed at the top level of a Haskell module (see Chapter 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, type signatures, and fixity declarations (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 [4, 6], but the type system has been extended with *type 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     -- (Num is defined in the Prelude)
  negate :: a -> a
```

This declaration may be read "a type `a` is an instance of the class `Num` if there are class methods `(+)` and `negate`, of the given 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 classes can be found in the papers by Jones [8] or Wadler and Blott [13]. 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 Kinds

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.

Kind inference checks the validity of type expressions in a similar way that type inference checks the validity of value expressions. However, unlike types, kinds are entirely implicit and are not a visible part of the language. Kind inference is discussed in Section 4.6.

4.1.2 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. Unlike data constructors, infix type constructors are not allowed (other than `(->)`).

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 an identifier 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 $* \rightarrow *$.
   - The declarations `data T ...` or `newtype T ...` add the type constructor `T` to the type vocabulary. The kind of `T` is determined by kind inference.

   Special syntax is provided for certain built-in 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 *parenthesized 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. For example, `Int -> Int -> Float` means `Int -> (Int -> Float)`.
2. A *tuple type* has the form `(`t_1`, ... ,` t_k`)` where $k \geq 2$, which is equivalent to the type `(,...,)` $t_1 \ldots t_k$ 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).

These special syntactic forms always denote the built-in type constructors for functions, tuples, and lists, regardless of what is in scope. In a similar way, the prefix type constructors `(->)`, `[]`, `()`, `(,)`, and so on, always denote the built-in type constructors; they cannot be qualified, nor mentioned in import or export lists (Chapter 5). (Hence the special production, "gtycon", above.)

Although the list and tuple types have special syntax, their semantics is the same as the equivalent user-defined algebraic data types.

Notice that 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 (that of the distinguished type variable in a class declaration (Section 4.3.1)), the type variables in a Haskell type expression are all assumed to be universally quantified; there is no explicit syntax for universal quantification [4]. 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. When we write an explicitly quantified type, the scope of the $\forall$ extends as far to the right as possible; for example, $\forall\, a.\ a \rightarrow a$ means $\forall\, a.\ (a \rightarrow a)$.

4.1.3 Syntax of Class Assertions and Contexts

$$
\begin{array}{lcll}
\mathit{context} & \rightarrow & \mathit{class} & \\
 & | & \texttt{(}\ \mathit{class}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{class}_n\ \texttt{)} & (n \geq 0) \\
\mathit{class} & \rightarrow & \mathit{qtycls}\ \mathit{tyvar} & \\
 & | & \mathit{qtycls}\ \texttt{(}\ \mathit{tyvar}\ \mathit{atype}_1\ \ldots\ \mathit{atype}_n\ \texttt{)} & (n \geq 1) \\
\mathit{qtycls} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{tycls} & \\
\mathit{tycls} & \rightarrow & \mathit{conid} & \\
\mathit{tyvar} & \rightarrow & \mathit{varid} &
\end{array}
$$

A *class assertion* has form $\mathit{qtycls}\ \mathit{tyvar}$, and indicates the membership of the type tyvar in the class qtycls. A class identifier begins with an uppercase letter. A *context* consists of zero or more class assertions, and has the general form

$$\texttt{(}\ C_1\ u_1,\ \ldots,\ C_n\ u_n\ \texttt{)}$$

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

4.1.4 Semantics of Types and Classes

In this section, we provide informal details of the type system. (Wadler and Blott [13] and Jones [8] 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 $\forall\, \overline{u}.\ cx \Rightarrow t$, 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 that are free in cx must also be free in t. Furthermore, the context cx must be of the form given above in Section 4.1.3. For example, here are some valid types:

```
Eq a => a -> a
(Eq a, Show a, Eq b) => [a] -> [b] -> String
(Eq (f a), Functor f) => (a -> b) -> f a -> f b -> Bool
```

In the third type, the constraint `Eq (f a)` cannot be made simpler because `f` is universally quantified.

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 preorder (specified below); the most general type, up to the equivalence induced by the generalization preorder, 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 $\forall\,\overline{u}.\ cx_1 \ \Rightarrow\ t_1$ is *more general than* the type $\forall\,\overline{w}.\ cx_2 \ \Rightarrow\ t_2$ 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 cx_2 holds in the class environment, $S(cx_1)$ also holds.

A value of type $\forall\,\overline{u}.\ cx \ \Rightarrow\ t$, may be instantiated at types $\overline{s}$ if and only if the context $cx[\overline{s}/\overline{u}]$ holds. 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 `Num Int` holds, because `Int` is an instance of the class `Num`. However, `double` may not normally be applied to values of type `Char`, because `Char` is not normally an instance of class `Num`. The user may choose to declare such an instance, in which case `double` may indeed be applied to a `Char`.

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* $tyvar_1 \ \ldots \ tyvar_k$ $(k \geq 0)$

constrs → $constr_1$ `|` … `|` $constr_n$ $(n \geq 1)$
constr → *con* [`!`] $atype_1$ … [`!`] $atype_k$ (arity $con \ = \ k, \ k \geq 0$)
| (*btype* | `!` *atype*) *conop* (*btype* | `!` *atype*) (infix *conop*)
| *con* `{` $fielddecl_1$ `,` … `,` $fielddecl_n$ `}` $(n \geq 0)$
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 has the form:

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

where cx is a context. This declaration introduces a new *type constructor* T with zero or more constituent *data constructors* $K_1, \ldots, K_n$. In this Report, the unqualified term "constructor" always means "data constructor".

The types of the data constructors are given by:

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

where cx_i is the largest subset of cx 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 cx and the t_{ij}; it is a static error for any other type variable to appear in cx 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

$$\begin{array}{ll}\texttt{NilSet} & ::\ \forall\, a.\ \texttt{Set}\ a \\ \texttt{ConsSet} & ::\ \forall\, a.\ \texttt{Eq}\ a\ \Rightarrow\ a\ \rightarrow\ \texttt{Set}\ a\ \rightarrow\ \texttt{Set}\ a\end{array}$$

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`. Pattern matching against `ConsSet` also gives rise to an `Eq a` constraint. For example:

```
f (ConsSet a s) = a
```

the function `f` has inferred type `Eq a => Set a -> a`. 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 5.8.

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

Labelled 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 may assign labels to the fields of the constructor, using the record syntax (`C { ... }`). 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.

The pattern `F {}` matches any value built with constructor `F`, *whether or not* `F` *was declared with record syntax.*

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, denoted by an exclamation point, “`!`”. Lexically, “`!`” is an ordinary varsym not a *reservedop*; it has special significance only in the context of the argument types of a data declaration.

Translation: A declaration of the form

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

where each s_i is either of the form `!` t_i or t_i, replaces every occurrence of K in an expression by

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

where op_i is the non-strict apply function `$` if s_i is of the form t_i, and op_i is the strict apply function `$!` (see Section 6.2) if s_i is of the form `!` t_i. Pattern matching on K is not affected by strictness flags.

4.2.2 Type Synonym Declarations

topdecl	→	`type` *simpletype* = *type*	
simpletype	→	*tycon tyvar*$_1$ … *tyvar*$_k$	$(k \geq 0)$

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

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

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]         -- invalid
```

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

Type synonyms are a convenient, but strictly syntactic, mechanism to make type signatures more readable. A synonym and its definition are completely interchangeable, except in the instance type of an `instance` declaration (Section 4.3.2).

4.2.3 Datatype Renamings

topdecl	→	`newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]	
newconstr	→	*con atype*	
	|	*con* `{` *var* `::` *type* `}`	
simpletype	→	*tycon* $tyvar_1$ … $tyvar_k$	$(k \geq 0)$

A declaration of the form

$$\texttt{newtype}\ cx \Rightarrow T\ u_1\ \ldots\ u_k = N\ t$$

introduces a new type whose representation is the same as an existing type. The type $(T\ u_1 \ldots u_k)$ renames the datatype t. It differs from a type synonym in that it creates a distinct type that must be explicitly coerced to or from the original type. Also, unlike type synonyms, `newtype` may be used to define recursive types. 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 may be implemented without 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 may make 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 ⊥)`, `(d2 ⊥)` and `(d2 (D2 ⊥))` are all equivalent to $\bot$, whereas `(n ⊥)`, `(n (N ⊥))`, `(d1 (D1 ⊥))` and `(s ⊥)` are all equivalent to `42`. In particular, `(N ⊥)` is equivalent to $\bot$ while `(D1 ⊥)` 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.

A `newtype` declaration may use field-naming syntax, though of course there may only be one field. Thus:

```
newtype Age = Age { unAge :: Int }
```

brings into scope both a constructor and a de-constructor:

```
Age   :: Int -> Age
unAge :: Age -> Int
```

4.3 Type Classes and Overloading

4.3.1 Class Declarations

topdecl	$\rightarrow$	`class` [*scontext* `=>`] *tycls tyvar* [`where` *cdecls*]	
scontext	$\rightarrow$	*simpleclass*	
	$\vert$	`(` $simpleclass_1$ `,` … `,` $simpleclass_n$ `)`	$(n \geq 0)$
simpleclass	$\rightarrow$	*qtycls tyvar*	
cdecls	$\rightarrow$	`{` $cdecl_1$ `;` … `;` $cdecl_n$ `}`	$(n \geq 0)$
cdecl	$\rightarrow$	*gendecl*	
	$\vert$	(*funlhs* $\vert$ *var*) *rhs*	

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

`class` *cx* `=>` C u `where` *cdecls*

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 cx specifies the superclasses of C, as described below; the only type variable that may be referred to in cx is u.

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

The *cdecls* part of a `class` declaration contains three kinds of declarations:

- The class declaration introduces new *class methods* v_i, whose scope extends outside the `class` declaration. The class methods of a class declaration are precisely the v_i for which there is an explicit type signature

  $$v_i \texttt{ :: } cx_i \texttt{ => } t_i$$

  in *cdecls*. 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.

  The type of the top-level class method v_i is:

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

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

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

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

- The *cdecls* may also contain a *fixity declaration* for any of the class methods (but for no other values). However, since class methods declare top-level values, the fixity declaration for a class method may alternatively appear at top level, outside the class declaration.

- Lastly, the *cdecls* may contain a *default class method* for any of the v_i. 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). The default method declaration is a normal value definition, except that the left hand side may only be a variable or function definition. For example:

  ```
  class Foo a where
    op1, op2 :: a -> a
    (op1, op2) = ...
  ```

  is not permitted, because the left hand side of the default declaration is a pattern.

Other than these cases, no other declarations are permitted in *cdecls*.

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.

4.3.2 Instance Declarations

topdecl	→	`instance` [*scontext* `=>`] *qtycls inst* [`where` *idecls*]	
inst	→	*gtycon*	
	|	`(` *gtycon* $tyvar_1$ … $tyvar_k$ `)`	($k \geq 0$, *tyvars* distinct)
	|	`(` $tyvar_1$ `,` … `,` $tyvar_k$ `)`	($k \geq 2$, *tyvars* distinct)
	|	`[` *tyvar* `]`	

	$\vert$	($tyvar_1$ `->` $tyvar_2$)	($tyvar_1$ and $tyvar_2$ distinct)
idecls	$\rightarrow$	`{` $idecl_1$ `;` $\ldots$ `;` $idecl_n$ `}`	($n \geq 0$)
idecl	$\rightarrow$	($funlhs$ $\vert$ var) rhs	
	$\vert$		($empty$)

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

$$\texttt{class}\ cx\ \texttt{=>}\ C\ u\ \texttt{where \{}\ cbody\ \texttt{\}}$$

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

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

where $k \geq 0$. The type $(T\ u_1\ \ldots\ u_k)$ must take the form of a type constructor T applied to simple type variables $u_1, \ldots u_k$; furthermore, T must not be a type synonym, and the u_i must all be distinct.

This prohibits instance declarations such as:

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

The declarations d may contain bindings only for the class methods of C. It is illegal to give a binding for a class method that is not in scope, but the name under which it is in scope is immaterial; in particular, it may be a qualified name. (This rule is identical to that used for subordinate names in export lists — Section 5.2.) For example, this is legal, even though `range` is in scope only with the qualified name `Data.Ix.range`.

```
module A where
  import qualified Data.Ix

  instance Data.Ix.Ix T where
    range = ...
```

The declarations may not contain any type signatures or fixity declarations, since these have already been given in the `class` declaration. As in the case of default class methods (Section 4.3.1), the method declarations must take the form of a variable or function definition.

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; this can be determined using kind inference as described in Section 4.6.
- Assume that the type variables in the instance type $(T\ u_1\ \ldots\ u_k)$ satisfy the constraints in the instance context cx'. Under this assumption, the following two conditions must also be satisfied:

1. The constraints expressed by the superclass context $cx[(T\ u1\ \ldots\ uk)/u]$ 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 cx'.
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 cx' satisfying the above two constraints, but it is nevertheless mandatory to write an explicit instance context.

The following example 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 example is valid Haskell. 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 Chapter 9.

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 mentioned in Figure 6.1. The precise details of how the derived instances are generated for each of these classes are provided in Chapter 11, including a specification of when such derived instances are possible. 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 are derived for that datatype; that is, omitting a `deriving` form is equivalent to including an empty deriving form: `deriving ()`.

4.3.4 Ambiguous Types, and Defaults for Overloaded Numeric Operations

$$\textit{topdecl} \quad \rightarrow \quad \texttt{default (}\textit{type}_1 \texttt{ , ... , } \textit{type}_n\texttt{)} \qquad (n \geq 0)$$

A problem inherent with Haskell-style overloading is the possibility of an *ambiguous type*. For example, using the `read` and `show` functions defined in Chapter 11, 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`,

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

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` has an *ambiguous type* if, in its type $\forall\, \overline{u}.\ cx \Rightarrow t$, there is a type variable u in $\overline{u}$ that occurs in cx but not in t. Such types are invalid.

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

Ambiguous types 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` (Chapter 9): 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 6.4.6 for a description of `encodeFloat` and `exponent`.)

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 type for which `Num` t_i holds. In situations where an ambiguous type is discovered, an ambiguous type variable, v, is defaultable if:

- v appears only in constraints of the form $C\ v$, where C is a class, and
- at least one of these classes is a numeric class, (that is, `Num` or a subclass of `Num`), and
- all of these classes are defined in the Prelude or a standard library (Figures 6.2–6.3 show the numeric classes, and Figure 6.1 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 (Integer, Double)
```

The empty default declaration, `default ()`, turns 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

$$\begin{array}{lll} \mathit{gendecl} & \rightarrow & \mathit{vars}\ \texttt{::}\ [\mathit{context}\ \texttt{=>}]\ \mathit{type} \\ \mathit{vars} & \rightarrow & \mathit{var}_1\ \texttt{,}\ \dots\texttt{,}\ \mathit{var}_n \qquad\qquad (n \geq 1) \end{array}$$

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

$$v_1,\ \dots,\ v_n\ \texttt{::}\ cx\ \texttt{=>}\ t$$

which is equivalent to asserting $v_i\ \texttt{::}\ cx\ \texttt{=>}\ t$ 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, even if the signatures are identical.

As mentioned in Section 4.1.2, 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 dependent type; this is explained in Section 4.5.4.)

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 Fixity Declarations

gendecl	$\rightarrow$	*fixity* [*integer*] *ops*	
fixity	$\rightarrow$	`infixl` | `infixr` | `infix`	
ops	$\rightarrow$	op_1 `,` … `,` op_n	$(n \geq 1)$
op	$\rightarrow$	*varop* | *conop*	

A fixity declaration gives the fixity and binding precedence of one or more operators. The *integer* in a fixity declaration must be in the range 0 to 9. A fixity declaration may appear anywhere that a type signature appears and, like a type signature, declares a property of a particular operator. Also like a type signature, a fixity declaration can only occur in the same sequence of declarations as the declaration of the operator itself, and at most one fixity declaration may be given for any operator. (Class methods are a minor exception; their fixity declarations can occur either in the class declaration itself or at top level.)

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

Precedence	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 4.1: Precedences and fixities of prelude operators

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 4.1 lists the fixities and precedences of the operators defined in the Prelude.

Fixity is a property of a particular entity (constructor or variable), just like its type; fixity is not a property of that entity's *name*. For example:

```
module Bar( op ) where
  infixr 7 `op`
  op = ...

module Foo where
  import qualified Bar
  infix 3 `op`

  a `op` b = (a `Bar.op` b) + 1

  f x = let
           p `op` q = (p `Foo.op` q) * 2
        in ...
```

Here, `` `Bar.op` `` is `infixr 7`, `` `Foo.op` `` is `infix 3`, and the nested definition of `op` in `f`'s right-hand side has the default fixity of `infixl 9`. (It would also be possible to give a fixity to the nested definition of `` `op` `` with a nested fixity declaration.)

4.4.3 Function and Pattern Bindings

decl → (*funlhs* | *pat*) *rhs*

funlhs → *var apat* { *apat* }
| *pat varop pat*

	|	(*funlhs*) *apat* { *apat* }	
rhs	→	= *exp* [`where` *decls*]	
	|	*gdrhs* [`where` *decls*]	
gdrhs	→	*guards* = *exp* [*gdrhs*]	
guards	→	| $guard_1$, . . . , $guard_n$	$(n \geq 1)$
guard	→	*pat* <− *infixexp*	(pattern guard)
	|	`let` *decls*	(local declaration)
	|	*infixexp*	(boolean guard)

We distinguish two cases within this syntax: a *pattern binding* occurs when the left hand side is a *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.

4.4.3.1 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:

$$= e_i \ \texttt{where} \ \{ \ decls_i \ \}$$

or

$$\begin{array}{ll} \mid \ gs_{i1} & = e_{i1} \\ \ldots \\ \mid \ gs_{im_i} & = e_{im_i} \\ & \texttt{where} \ \{ \ decls_i \ \} \end{array}$$

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:

$$\mid \ \texttt{True} \ = e_i \ \texttt{where} \ \{ \ decls_i \ \}$$

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 three function definitions are all equivalent:

```
plus x y z = x+y+z
x ‘plus‘ y = \ z -> x+y+z
(x ‘plus‘ y) z = x+y+z
```

Note that fixity resolution applies to the infix variants of the function binding in the same way as for expressions (Section 10.6). Applying fixity resolution to the left side of the equals in a function binding must leave the *varop* being defined at the top level. For example, if we are defining a new operator `##` with precedence 6, then this definition would be illegal:

```
a ## b : xs = exp
```

because `:` has precedence 5, so the left hand side resolves to `(a ## x) : xs`, and this cannot be a pattern binding because `(a ## x)` is not a valid pattern.

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

$$\begin{array}{ll} x = \backslash\ x_1 \ \ldots\ x_k \texttt{ -> case (}x_1\texttt{, ..., }x_k\texttt{) of} & \texttt{(}p_{11}\texttt{, ..., }p_{1k}\texttt{)}\ match_1 \\ & \ldots \\ & \texttt{(}p_{n1}\texttt{, ..., }p_{nk}\texttt{)}\ match_n \end{array}$$

where the x_i are new identifiers.

4.4.3.2 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:

$$\begin{array}{lll} p & \texttt{|}\ gs_1 & = e_1 \\ & \texttt{|}\ gs_2 & = e_2 \\ & \ \ldots & \\ & \texttt{|}\ gs_m & = e_m \\ & \multicolumn{2}{l}{\texttt{where \{}\ decls\ \texttt{\}}} \end{array}$$

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

$$\begin{array}{lll} p & = & \texttt{let}\ decls\ \texttt{in} \\ & & \texttt{case () of} \\ & & \quad \texttt{() |}\ gs_1\ \texttt{->}\ e_1 \\ & & \quad \quad\ \ \texttt{|}\ gs_2\ \texttt{->}\ e_2 \\ & & \quad \quad\ \ \ \ \ldots \\ & & \quad \quad\ \ \texttt{|}\ gs_m\ \texttt{->}\ e_m \\ & & \quad \texttt{\_ -> error "Unmatched pattern"} \end{array}$$

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 applying the normal Hindley-Milner inference rules. In order to increase polymorphism, these rules are applied to groups of bindings identified by a *dependency analysis*.

A binding $b1$ *depends* on a binding $b2$ in the same list of declarations if either

1. $b1$ contains a free identifier that has no type signature and is bound by $b2$, or
2. $b1$ depends on a binding that depends on $b2$.

A *declaration group* is a minimal set of mutually dependent bindings. Hindley-Milner type inference is applied to each declaration group in dependency order. The order of declarations in `where`/`let` constructs is irrelevant.

4.5.2 Generalization

The Hindley-Milner type system assigns types to a let-expression in two stages:

1. The declaration groups are considered in dependency order. For each group, a type with no universal quantification is inferred for each variable bound in the group. Then, all type variables that occur in these types are universally quantified unless they are associated with bound variables in the type environment; this is called generalization.
2. 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 $a \rightarrow a \rightarrow \texttt{String}$, and the accumulated constraints are `Ord` a (arising from the use of `>`), and `Show` a (arising from the use of `show`). 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.

4.5.3 Context Reduction Errors

As mentioned in Section 4.1.4, the context of a type may constrain only a type variable, or the application of a type variable to one or more types. Hence, types produced by generalization must be expressed in a form in which all context constraints have be reduced to this "head normal form". 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.

Here is an example that shows the need for a constraint of the form $C\ (m\ t)$ where m is one of the type variables being generalized; that is, where the class C applies to a type expression that is not a type variable or a type constructor. Consider:

```
f :: (Monad m, Eq (m a)) => a -> m a -> Bool
f x y = return x == y
```

The type of `return` is `Monad m => a -> m a`; the type of `(==)` is `Eq a => a -> a -> Bool`. The type of `f` should be therefore `(Monad m, Eq (m a)) => a -> m a -> Bool`, and the context cannot be simplified further.

The instance declaration derived from a data type `deriving` clause (see Section 4.3.3) must, like any instance declaration, have a *simple* context; that is, all the constraints must be of the form $C\ a$, where a is a type variable. For example, in the type

```
data Apply a b = App (a b)  deriving Show
```

the derived Show instance will produce a context `Show (a b)`, which cannot be reduced and is not simple; thus a static error results.

4.5.4 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 $a \ \rightarrow \ b \ \rightarrow \ \texttt{(}\texttt{[}a\texttt{]}\texttt{,}b\texttt{)}$. The generalization step attributes to `g` the type $\forall \ b. \ a \ \rightarrow \ b \ \rightarrow \ \texttt{(}\texttt{[}a\texttt{]}\texttt{,}b\texttt{)}$; 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 $\forall \ \overline{u}. \ cx \ \Rightarrow \ t$ is said to be *monomorphic* in the type variable a if a is free in $\forall \ \overline{u}. \ cx \ \Rightarrow \ t$.

It is worth noting that the explicit type signatures provided by Haskell are not powerful enough to express types that 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`.

4.5.5 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 4.4.3).

The following two rules define the monomorphism restriction:

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 (Section 4.4.3.2), *and*
- **(b):** an explicit type signature is given for every variable in the group that is bound by simple pattern binding.

The usual Hindley-Milner restriction on polymorphism is that only type variables that do not occur 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. Any monomorphic type variables that remain when type inference for an entire module is complete, are considered *ambiguous*, and are resolved to particular types using the defaulting rules (Section 4.3.4).

Motivation Rule 1 is required for two reasons, both of which are fairly subtle.

- *Rule 1 prevents computations from being unexpectedly repeated.* For example, `genericLength` is a standard function (in library `Data.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)
  ```

- *Rule 1 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 $\forall\, a.\, \texttt{Read}\ a \Rightarrow a$ and `s` the type $\forall\, a.\, \texttt{Read}\ a \Rightarrow \texttt{String}$. The latter is an invalid type, because it is inherently ambiguous. It is not possible to determine at what overloading to use `s`, nor can this be solved by adding a type signature for `s`. Hence, when *non-simple* pattern bindings are used (Section 4.4.3.2), the types inferred are always monomorphic in their constrained type variables, irrespective of whether a type signature is provided. In this case, both `n` and `s` are monomorphic in a.

  The same constraint applies to pattern-bound functions. For example, in

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

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

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. Rule 2 states that the exact types of all the variables bound in a module must be determined by that module alone, and not by any modules that import it.

```
module M1(len1) where
  default( Int, Double )
  len1 = genericLength "Hello"

module M2 where
  import M1(len1)
  len2 = (2*len1) :: Rational
```

When type inference on module `M1` is complete, `len1` has the monomorphic type `Num a => a` (by Rule 1). Rule 2 now states that the monomorphic type variable `a` is ambiguous, and must be resolved using the defaulting rules of Section 4.3.4. Hence, `len1` gets type `Int`, and its use in `len2` is type-incorrect. (If the above code is actually what is wanted, a type signature on `len1` would solve the problem.)

This issue does not arise for nested bindings, because their entire scope is visible to the compiler.

Consequences 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
```

Rule 1 applies to both top-level and nested definitions. Consider

```
module M where
  len1 = genericLength "Hello"
  len2 = (2*len1) :: Rational
```

Here, type inference finds that `len1` has the monomorphic type (`Num a => a`); and the type variable `a` is resolved to `Rational` when performing type inference on `len2`.

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 [8]. 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 following definition to those above does 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.

Chapter 5

Modules

A module defines a collection of values, datatypes, type synonyms, classes, etc. (see Chapter 4), in an environment created by a set of *imports* (resources brought into scope from other modules). It *exports* some of these resources, making them available to other modules. We use the term *entity* to refer to a value, type, or class 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`, which must be a computation of type `IO` τ for some type τ (see Chapter 7). When the program is executed, the computation `main` is performed, and its result (of type τ) is discarded.

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.

Modules are used for name-space control, and are not first class values. A multi-module Haskell program can be converted into a single-module program by giving each entity a unique name, changing all occurrences to refer to the appropriate unique name, and then concatenating all the module bodies[1]. For example, here is a three-module program:

```
module Main where
  import A
  import B
  main = A.f >> B.f

module A where
  f = ...

module B where
  f = ...
```

It is equivalent to the following single-module program:

```
module Main where
  main = af >> bf
```

---

[1] There are two minor exceptions to this statement. First, `default` declarations scope over a single module (Section 4.3.4). Second, Rule 2 of the monomorphism restriction (Section 4.5.5) is affected by module boundaries.

```
    af = ...

    bf = ...
```

Because they are allowed to be mutually recursive, modules allow a program to be partitioned freely without regard to dependencies.

A module name (lexeme *modid*) is a sequence of one or more identifiers beginning with capital letters, separated by dots, with no intervening spaces. For example, `Data.Bool`, `Main` and `Foreign.Marshal.Alloc` are all valid module names.

modid	→	{*conid* `.`} *conid*	(modules)

Module names can be thought of as being arranged in a hierarchy in which appending a new component creates a child of the original module name. For example, the module `Control.Monad.ST` is a child of the `Control.Monad` sub-hierarchy. This is purely a convention, however, and not part of the language definition; in this report a *modid* is treated as a single identifier occupying a flat namespace.

There is one distinguished module, `Prelude`, which is imported into all modules by default (see Section 5.6), plus a set of standard library modules that may be imported as required (see Part II).

5.1 Module Structure

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

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

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 a possibly-empty list of `import` declarations (*impdecls*, Section 5.3) that specify modules to be imported, optionally restricting the imported bindings. This is followed by a possibly-empty list of top-level declarations (*topdecls*, Chapter 4).

An abbreviated form of module, consisting only of the module body, is permitted. 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.

5.2 Export Lists

exports	$\rightarrow$	`(` *export*$_1$ `,` … `,` *export*$_n$ [`,`] `)`	$(n \geq 0)$
export	$\rightarrow$	*qvar*	
	$\vert$	*qtycon* [`(..)` $\vert$ `(` *cname*$_1$ `,` … `,` *cname*$_n$ `)`]	$(n \geq 0)$
	$\vert$	*qtycls* [`(..)` $\vert$ `(` *var*$_1$ `,` … `,` *var*$_n$ `)`]	$(n \geq 0)$
	$\vert$	`module` *modid*	
cname	$\rightarrow$	*var* $\vert$ *con*	

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. A value, field name, or class method, whether declared in the module body or imported, may be named by giving the name of the value as a *qvarid*, which must be in scope. 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 5.8).
   - The form T `(`c_1`,` … `,`c_n`)`, names the type and some or all of its constructors and field names.
   - The abbreviated form T `(..)` names the type and all its constructors and field names that are currently in scope (whether qualified or not).

   In all cases, the (possibly-qualified) type constructor T must be in scope. The constructor and field names c_i in the second form are unqualified; one of these subordinate names is legal if and only if (a) it names a constructor or field of T, and (b) the constructor or field is in scope in the module body *regardless of whether it is in scope under a qualified or unqualified name.* For example, the following is legal

   ```
   module A( Mb.Maybe( Nothing, Just ) ) where
     import qualified Data.Maybe as Mb
   ```

   Data constructors cannot be named in export lists except as subordinate names, because they cannot otherwise be distinguished from type constructors.

3. A type synonym T declared by a `type` declaration may be named by the form T, where T is in scope.

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`)`, names the class and some or all of its methods.
   - The abbreviated form C `(..)` names the class and all its methods that are in scope (whether qualified or not).

In all cases, C must be in scope. In the second form, one of the (unqualified) subordinate names f_i is legal if and only if (a) it names a class method of C, and (b) the class method is in scope in the module body regardless of whether it is in scope under a qualified or unqualified name.

5. The form "`module M`" names the set of all entities that are in scope with both an unqualified name "`e`" and a qualified name "`M.e`". This set may be empty. 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`.

A module can name its own local definitions in its export list using its own name in the "`module M`" syntax, because a local declaration brings into scope both a qualified and unqualified name (Section 5.5.1). For example:

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

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

It is an error to use `module M` in an export list unless `M` is the module bearing the export list, or `M` is imported by at least one import declaration (qualified or unqualified).

Exports lists are cumulative: the set of entities exported by an export list is the union of the entities exported by the individual items of the list.

It makes no difference to an importing module how an entity was exported. For example, a field name `f` from data type `T` may be exported individually (`f`, item (1) above); or as an explicitly-named member of its data type (`T(f)`, item (2)); or as an implicitly-named member (`T(..)`, item(2)); or by exporting an entire module (`module M`, item (5)).

The *unqualified* names of the entities exported by a module must all be distinct (within their respective namespace). For example

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

There are no name clashes within module `A` itself, but there are name clashes in the export list between `C.g` and `g` (assuming `C.g` and `g` are different entities – remember, modules can import each other recursively), and between `module B` and `C.f` (assuming `B.f` and `C.f` are different entities).

5.3 Import Declarations

impdecl	$\rightarrow$	`import` [`qualified`] *modid* [`as` *modid*] [*impspec*]	
	$\vert$		(*empty declaration*)
impspec	$\rightarrow$	`(` *import*$_1$ `,` … `,` *import*$_n$ [`,`] `)`	($n \geq 0$)
	$\vert$	`hiding` `(` *import*$_1$ `,` … `,` *import*$_n$ [`,`] `)`	($n \geq 0$)

$$\begin{array}{lcll}
\mathit{import} & \rightarrow & \mathit{var} & \\
 & | & \mathit{tycon}\ [\ \texttt{(..)}\ |\ \texttt{(}\ \mathit{cname}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{cname}_n\ \texttt{)}\,] & (n \geq 0) \\
 & | & \mathit{tycls}\ [\texttt{(..)}\ |\ \texttt{(}\ \mathit{var}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{var}_n\ \texttt{)}\,] & (n \geq 0) \\
\mathit{cname} & \rightarrow & \mathit{var}\ |\ \mathit{con} &
\end{array}$$

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 strictly cumulative: an entity is in scope if it is imported by any of the `import` declarations in a module. The ordering of import declarations is irrelevant.

Lexically, the terminal symbols "`as`", "`qualified`" and "`hiding`" are each a *varid* rather than a *reservedid*. They have special significance only in the context of an `import` declaration; they may also be used as variables.

5.3.1 What is imported

Exactly which entities are to be imported can be specified in one of the following 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. When the `(..)` form of import is used for a type or class, the `(..)` refers to all of the constructors, methods, or field names exported from the module.

   The list must name only entities exported by the imported module. The list may be empty, in which case nothing except the instances is 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. Data constructors may be named directly in hiding lists without being prefixed by the associated type. Thus, in

   ```
   import M hiding (C)
   ```

   any constructor, class, or type named `C` is excluded. In contrast, using `C` in an import list names only a class or type.

   It is an error to hide an entity that is not, in fact, exported by the imported module.

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

5.3.2 Qualified import

For each entity imported under the rules of Section 5.3.1, the top-level environment is extended. If the import declaration used the `qualified` keyword, only the *qualified name* of the entity is brought into scope. If the `qualified` keyword is omitted, then *both* the qualified *and* unqualified name of the entity is brought into scope. Section 5.5.1 describes qualified names in more detail.

The qualifier on the imported name is either the name of the imported module, or the local alias given in the `as` clause (Section 5.3.3) on the `import` statement. Hence, *the qualifier is not necessarily the name of the module in which the entity was originally declared.*

The ability to exclude the unqualified names 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
import Data.List( nub )

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

succ = (Prelude.+ 1)
```

5.3.3 Local aliases

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

```
import qualified VeryLongModuleName as C
```

entities must be referenced using '`C.`' as a qualifier instead of '`VeryLongModuleName.`'. This also allows a different module to be substituted for `VeryLongModuleName` without changing the qualifiers used for the imported module. It is legal for more than one module in scope to use the same qualifier, provided that all names can still be resolved unambiguously. For example:

```
module M where
  import qualified Foo as A
  import qualified Baz as A
  x = A.f
```

This module is legal provided only that `Foo` and `Baz` do not both export `f`.

An `as` clause may also be used on an un-`qualified` `import` statement:

```
import Foo as A(f)
```

This declaration brings into scope `f` and `A.f`.

5.3.4 Examples

To clarify the above import rules, suppose the module `A` exports `x` and `y`. Then this table shows what names are brought into scope by the specified import statement:

Import declaration	Names brought into scope
`import A`	`x`, `y`, `A.x`, `A.y`
`import A()`	(nothing)
`import A(x)`	`x`, `A.x`
`import qualified A`	`A.x`, `A.y`
`import qualified A()`	(nothing)
`import qualified A(x)`	`A.x`
`import A hiding ()`	`x`, `y`, `A.x`, `A.y`
`import A hiding (x)`	`y`, `A.y`
`import qualified A hiding ()`	`A.x`, `A.y`
`import qualified A hiding (x)`	`A.y`
`import A as B`	`x`, `y`, `B.x`, `B.y`
`import A as B(x)`	`x`, `B.x`
`import qualified A as B`	`B.x`, `B.y`

In all cases, all instance declarations in scope in module `A` are imported (Section 5.4).

5.4 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()` does not bring any new names in scope from module `M`, but does bring in any instances visible in `M`. A module whose only purpose is to provide instance declarations can have an empty export list. For example

```
module MyInstances() where
  instance Show (a -> b) where
    show fn = "<<function>>"
  instance Show (IO a) where
    show io = "<<IO action>>"
```

5.5 Name Clashes and Closure

5.5.1 Qualified names

A *qualified name* is written as $modid$ **.** $name$ (Section 2.4). A qualified name is brought into scope:

- *By a top level declaration.* A top-level declaration brings into scope both the unqualified *and* the qualified name of the entity being defined. Thus:

  ```
  module M where
    f x = ...
    g x = M.f x x
  ```

is legal. The *defining* occurrence must mention the *unqualified* name; therefore, it is illegal to write

```
module M where
  M.f x = ...                 -- ILLEGAL
  g x = let M.y = x+1 in ...  -- ILLEGAL
```

- *By an* `import` *declaration.* An `import` declaration, whether `qualified` or not, always brings into scope the qualified name of the imported entity (Section 5.3). This allows a qualified import to be replaced with an unqualified one without forcing changes in the references to the imported names.

5.5.2 Name clashes

If a module contains a bound occurrence of a name, such as `f` or `A.f`, it must be possible unambiguously to resolve which entity is thereby referred to; that is, there must be only one binding for `f` or `A.f` respectively.

It is *not* an error for there to exist names that cannot be so resolved, provided that the program does not mention those names. For example:

```
module A where
  import B
  import C
  tup = (b, c, d, x)

module B( d, b, x, y ) where
  import D
  x = ...
  y = ...
  b = ...

module C( d, c, x, y ) where
  import D
  x = ...
  y = ...
  c = ...

module D( d ) where
  d = ...
```

Consider the definition of `tup`.

- The references to `b` and `c` can be unambiguously resolved to `b` declared in `B`, and `c` declared in `C` respectively.
- The reference to `d` is unambiguously resolved to `d` declared in `D`. In this case the same entity is brought into scope by two routes (the import of `B` and the import of `C`), and can be referred to in `A` by the names `d`, `B.d`, and `C.d`.
- The reference to `x` is ambiguous: it could mean `x` declared in `B`, or `x` declared in `C`. The ambiguity could be fixed by replacing the reference to `x` by `B.x` or `C.x`.
- There is no reference to `y`, so it is not erroneous that distinct entities called `y` are exported by both `B` and `C`. An error is only reported if `y` is actually mentioned.

The name occurring in a type signature or fixity declarations is always unqualified, and unambiguously refers to another declaration in the same declaration list (except that the fixity declaration for a class method can occur at top level — Section 4.4.2). For example, the following module is legal:

```
module F where

  sin :: Float -> Float
  sin x = (x::Float)

  f x = Prelude.sin (F.sin x)
```

The local declaration for `sin` is legal, even though the Prelude function `sin` is implicitly in scope. The references to `Prelude.sin` and `F.sin` must both be qualified to make it unambiguous which `sin` is meant. However, the unqualified name `sin` in the type signature in the first line of `F` unambiguously refers to the local declaration for `sin`.

5.5.3 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. However, entities that 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 that 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 that 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 finds the definition of `T` if needed whether or not it is exported.

5.6 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 module `Prelude`. There are also many predefined library modules, which provide less frequently used functions and types. For example, complex numbers, arrays, and most of the input/output are all part of the standard libraries. These are defined in Part II. 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 consulting the source code of the Prelude.

5.6.1 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 entities defined in the Prelude to be selectively imported, just like those from any other module.

The semantics of the entities in `Prelude` is specified by a reference implementation of `Prelude` written in Haskell, given in Chapter 9. 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 Chapter 9. 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.

Chapter 9 defines the module `Prelude` using several other modules: `PreludeList`, `PreludeIO`, and so on. These modules are *not* part of Haskell 98, and they cannot be imported separately. They are simply there to help explain the structure of the `Prelude` module; they should be considered part of its implementation, not part of the language definition.

5.6.2 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( null, nonNull ) where
  import Prelude hiding( null )
  null, nonNull :: Int -> Bool
  null    x = x == 0
  nonNull x = not (null x)
```

Module `A` redefines `null`, and contains an unqualified reference to `null` on the right hand side of `nonNull`. The latter would be ambiguous without the `hiding(null)` on the `import Prelude` statement. Every module that imports `A` unqualified, and then makes an unqualified reference to `null` must also resolve the ambiguous use of `null` 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 Prelude()
  import MyPrelude
  f x = (x,x)
  g x = (,) x x
  h x = [x] ++ []
```

the explicit `import Prelude()` declaration prevents the automatic import of `Prelude`, while the declaration `import MyPrelude` brings the non-standard prelude into scope. The special syntax for tuples (such as `(x,x)` and `(,)`) and lists (such as `[x]` and `[]`) continues to refer to the tuples and lists defined by the standard `Prelude`; there is no way to redefine the meaning of `[x]`, for example, in terms of a different implementation of lists. On the other hand, the use of `++` is not special syntax, so it refers to `++` imported from `MyPrelude`.

It is not possible, however, to hide `instance` declarations in the `Prelude`. For example, one cannot define a new instance for `Show Char`.

5.7 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.

5.8 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. Instead, they must use `push`, `pop`, and `empty` to construct such values.

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 (_:s)) = Stk s
  empty = Stk []
```

Chapter 6

Predefined Types and Classes

The Haskell Prelude contains predefined classes, types, and functions that are implicitly imported into every Haskell program. In this chapter, 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 Chapter 9. Other predefined types such as arrays, complex numbers, and rationals are defined in Part II.

6.1 Standard Haskell Types

These types are defined by the Haskell Prelude. Numeric types are described in Section 6.4. 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 whose values represent Unicode characters [2]. The lexical syntax for characters is defined in Section 2.6; 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 from class `Enum`, map characters to and from the `Int` type.

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 2.6. 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 Section 9.1) 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`, `Functor`, 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. All tuples are instances of `Eq`, `Ord`, `Bounded`, `Read`, and `Show` (provided, of course, that all their component types are).

There is no upper bound on the size of a tuple, but some Haskell implementations may restrict the size of tuples, and limit the instances associated with larger tuples. However, every Haskell implementation must support tuples up to size 15, together with the instances for `Eq`, `Ord`, `Bounded`, `Read`, and `Show`. The Prelude and libraries define tuple functions such as `zip` for tuples up to a size of 7.

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 same holds for tuple type constructors; thus, `(Int,Bool,Int)` and `(,,) Int Bool Int` denote the same type.

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 Function Types

Functions are an abstract type: no constructors directly create functional values. The following simple functions are found in the Prelude: `id`, `const`, `(.)`, `flip`, `($)`, and `until`.

6.1.7 The IO and IOError Types

The `IO` type serves as a tag for operations (actions) that 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 `Functor` classes. Chapter 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 Prelude contains a few I/O functions (defined in Section 9.3), and Part II contains many more.

6.1.8 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`, 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 Strict Evaluation

Function application in Haskell is non-strict; that is, a function argument is evaluated only when required. Sometimes it is desirable to force the evaluation of a value, using the `seq` function:

```
seq :: a -> b -> b
```

The function `seq` is defined by the equations:

$$\begin{aligned} \texttt{seq}\,\bot\;b &= \bot \\ \texttt{seq}\;a\;b &= b,\ \textit{if}\ a \neq \bot \end{aligned}$$

`seq` is usually introduced to improve performance by avoiding unneeded laziness. Strict datatypes (see Section 4.2.1) are defined in terms of the `$!` operator. However, the provision of `seq` has important semantic consequences, because it is available *at every type*. As a consequence, $\bot$ is not the same as `\x ->` $\bot$, since `seq` can be used to distinguish them. For the same reason, the existence of `seq` weakens Haskell's parametricity properties.

The operator `$!` is strict (call-by-value) application, and is defined in terms of `seq`. The Prelude also defines the `$` operator to perform non-strict application.

```
infixr 0 $, $!
($), ($!) :: (a -> b) -> a -> b
f $  x   =          f x
f $! x   =  x `seq` f x
```

The non-strict application operator `$` may appear redundant, since ordinary application `(f x)` means the same as `(f $ x)`. However, `$` has low, right-associative binding precedence, so it sometimes allows parentheses to be omitted; for example:

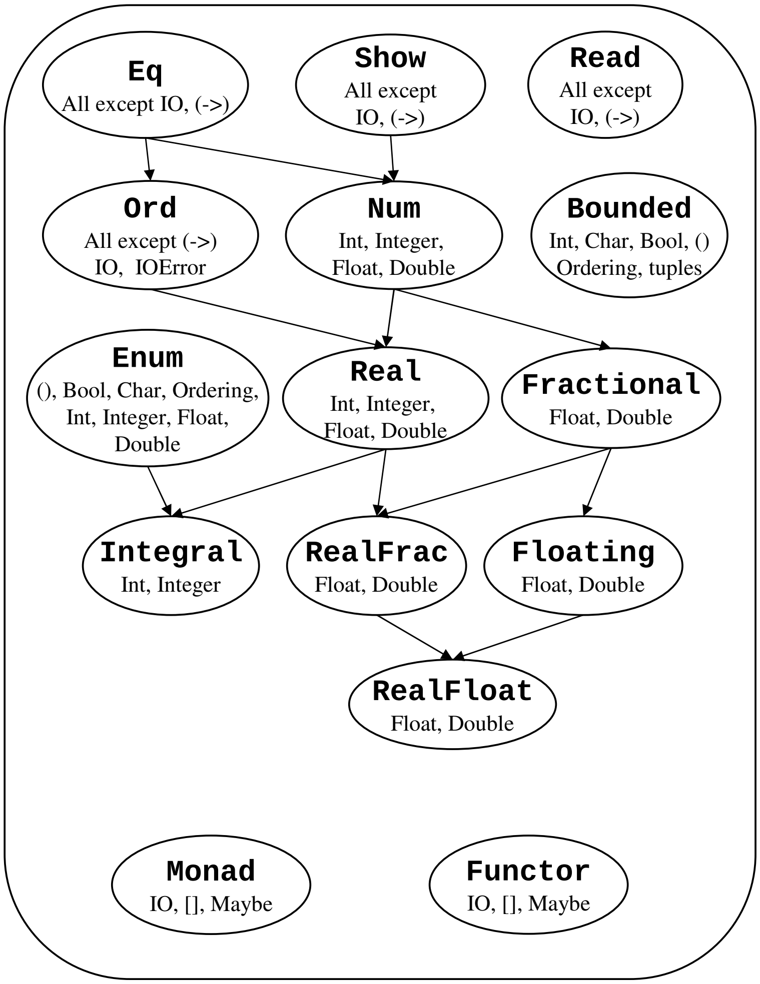

Figure 6.1: Standard Haskell Classes

```
    f $ g $ h x  =  f (g (h x))
```

It is also useful in higher-order situations, such as `map ($ 0) xs`, or `zipWith ($) fs xs`.

6.3 Standard Haskell Classes

Figure 6.1 shows the hierarchy of Haskell classes defined in the Prelude and the Prelude types that are instances of these classes.

Default class method declarations (Section 4.3) are provided for many of the methods in standard classes. A comment with each `class` declaration in Chapter 9 specifies the smallest collection of method definitions that, together with the default declarations, provide a reasonable definition for all the class methods. If there is no such comment, then all class methods must be given to fully specify an instance.

6.3.1 The Eq Class

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

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

The `Eq` class provides equality (==) and inequality (/=) methods. 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`.

This declaration gives default method declarations for both /= and ==, each being defined in terms of the other. If an instance declaration for `Eq` defines neither == nor /=, then both will loop. If one is defined, the default method for the other will make use of the one that is defined. If both are defined, neither default method is used.

6.3.2 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    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

The `Ord` class is used for totally ordered datatypes. All basic datatypes except for functions, `IO`, and `IOError`, 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 default declarations allow a user to create an `Ord` instance either with a type-specific `compare` function or with type-specific == and <= functions.

6.3.3 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]
    -- ... default decl for readList given in Prelude

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

    showsPrec _ x s   = show x ++ s
    show x            = showsPrec 0 x ""
    -- ... default decl for showList given in Prelude
```

The `Read` and `Show` classes are used to convert values to or from strings. The `Int` argument to `showsPrec` and `readsPrec` gives the operator precedence of the enclosing context (see Section 11.4).

`showsPrec` and `showList` return a `String`-to-`String` function, to allow constant-time concatenation of its results using function composition. A specialised variant, `show`, is also provided, which uses precedence context zero, and returns an ordinary `String`. The method `showList` is provided to allow the programmer to give a specialised way of showing lists of values. This is particularly useful for the `Char` type, where values of type `String` should be shown in double quotes, rather than between square brackets.

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`.

All `Prelude` types, except function types and `IO` types, are instances of `Show` and `Read`. (If desired, a programmer can easily make functions and `IO` types into (vacuous) instances of `Show`, by providing an instance declaration.)

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"
```

`shows` and `reads` use a default precedence of 0. The `read` function reads input from a string, which must be completely consumed by the input process.

The function `lex :: ReadS String`, used by `read`, is also part of the Prelude. It reads a single lexeme from the input, discarding initial white space, and returning the characters that constitute the lexeme. If the input string contains only white space, `lex` returns a single successful "lexeme" consisting of the empty string. (Thus `lex "" = [("","")]`.) If there is no legal lexeme at the beginning of the input string, `lex` fails (i.e. returns `[]`).

6.3.4 The Enum Class

```
class  Enum a  where
    succ, pred     :: a -> a
    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]

    -- Default declarations given in Prelude
```

Class `Enum` defines operations on sequentially ordered types. The functions `succ` and `pred` return the successor and predecessor, respectively, of a value. The functions `fromEnum` and `toEnum` map values from a type in `Enum` to and from `Int`. The `enumFrom`... methods are used when translating arithmetic sequences (Section 3.10).

Instances of `Enum` may be derived for any enumeration type (types whose constructors have no fields); see Chapter 11.

For any type that is an instance of class `Bounded` as well as `Enum`, the following should hold:

- The calls `succ maxBound` and `pred minBound` should result in a runtime error.
- `fromEnum` and `toEnum` should give a runtime error if the result value is not representable in the result type. For example, `toEnum 7 :: Bool` is an error.
- `enumFrom` and `enumFromThen` should be defined with an implicit bound, thus:

```
enumFrom     x   = enumFromTo     x maxBound
enumFromThen x y = enumFromThenTo x y bound
  where
    bound | fromEnum y >= fromEnum x = maxBound
          | otherwise                = minBound
```

The following `Prelude` types are instances of `Enum`:

- Enumeration types: `()`, `Bool`, and `Ordering`. The semantics of these instances is given by Chapter 11. For example, `[LT ..]` is the list `[LT,EQ,GT]`.
- `Char`: the instance is given in Chapter 9, based on the primitive functions that convert between a `Char` and an `Int`. For example, `enumFromTo 'a' 'z'` denotes the list of lowercase letters in alphabetical order.
- Numeric types: `Int`, `Integer`, `Float`, `Double`. The semantics of these instances is given next.

For all four numeric types, `succ` adds 1, and `pred` subtracts 1. The conversions `fromEnum` and `toEnum` convert between the type and `Int`. In the case of `Float` and `Double`, the digits after the decimal point may be lost. It is implementation-dependent what `fromEnum` returns when applied to a value that is too large to fit in an `Int`.

For the types `Int` and `Integer`, the enumeration functions have the following meaning:

- The sequence `enumFrom` e_1 is the list `[`e_1`,`$e_1 + 1$`,`$e_1 + 2$`, ...]`.
- The sequence `enumFromThen` e_1 e_2 is the list `[`e_1`,`$e_1 + i$`,`$e_1 + 2i$`, ...]`, where the increment, i, is $e_2 - e_1$. The increment may be zero or negative. If the increment is zero, all the list elements are the same.
- The sequence `enumFromTo` e_1 e_3 is the list `[`e_1`,`$e_1 + 1$`,`$e_1 + 2$`, ...`e_3`]`. The list is empty if $e_1 > e_3$.
- The sequence `enumFromThenTo` e_1 e_2 e_3 is the list `[`e_1`,`$e_1 + i$`,`$e_1 + 2i$`, ...`e_3`]`, where the increment, i, is $e_2 - e_1$. If the increment is positive or zero, the list terminates when the next element would be greater than e_3; the list is empty if $e_1 > e_3$. If the increment is negative, the list terminates when the next element would be less than e_3; the list is empty if $e1 < e_3$.

For `Float` and `Double`, the semantics of the `enumFrom` family is given by the rules for `Int` above, except that the list terminates when the elements become greater than $e_3 + i/2$ for positive increment i, or when they become less than $e_3 + i/2$ for negative i.

For all four of these Prelude numeric types, all of the `enumFrom` family of functions are strict in all their arguments.

6.3.5 The Functor Class

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

The `Functor` class is used for types that can be mapped over. Lists, `IO`, and `Maybe` are in this class.

Instances of `Functor` should satisfy the following laws:

```
fmap id      =  id
fmap (f . g) =  fmap f . fmap g
```

All instances of `Functor` defined in the Prelude satisfy these laws.

6.3.6 The Monad Class

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

    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

The `Monad` class defines the basic operations over a *monad*. See Chapter 7 for more information about monads.

"`do`" expressions provide a convenient syntax for writing monadic expressions (see Section 3.14). The `fail` method is invoked on pattern-match failure in a `do` expression.

In the Prelude, lists, `Maybe`, and `IO` are all instances of `Monad`. The `fail` method for lists returns the empty list `[]`, for `Maybe` returns `Nothing`, and for `IO` raises a user exception in the IO monad (see Section 7.3).

Instances of `Monad` should satisfy the following laws:

```
return a >>= k           =  k a
m >>= return             =  m
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Instances of both `Monad` and `Functor` should additionally satisfy the law:

```
fmap f xs  =  xs >>= return . f
```

All instances of `Monad` defined in the Prelude satisfy these laws.

The Prelude provides the following auxiliary functions:

```
sequence  :: 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 ()
(=<<)     :: Monad m => (a -> m b) -> m a -> m b
```

6.3.7 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`.

6.4 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 6.1. 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 integers of both limited and unlimited range; 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 `Ratio` library.

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 6.1: Standard Numeric Types

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

The standard numeric types are listed in Table 6.1. The finite-precision integer 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 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 6.2–6.3. Figure 6.1 shows the class dependencies and built-in types that are instances of the numeric classes.

6.4.1 Numeric Literals

The syntax of numeric literals is given in Section 2.5. 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.

6.4.2 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 if `y` is non-zero:

```
class  (Eq a, Show 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 6.2: Standard Numeric Classes and Related Operations, Part 1

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

`` `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 (positive) integer that divides both x and y; for example `gcd (-3) 6` = `3`, `gcd (-3) (-6)` = `3`, `gcd 0 4` = `4`. `gcd 0 0` raises a runtime error.

`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
    atan2               :: a -> a -> a

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

fromIntegral :: (Integral a, Num b) => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
```

Figure 6.3: Standard Numeric Classes and Related Operations, Part 2

6.4.3 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 `1` if y is `1`, and `0` otherwise.

6.4.4 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
```

6.4.5 Trigonometric Functions

Class `Floating` provides the circular and hyperbolic sine, cosine, and tangent functions and their inverses. Default implementations of `tan`, `tanh`, `logBase`, `**`, and `sqrt` are provided, but implementors are free to provide more accurate implementations.

Class `RealFloat` provides a version of arctangent taking two real floating-point arguments. For real floating x and y, `atan2` y x computes the angle (from the positive x-axis) of the vector from the origin to the point (x, y). `atan2` y x returns a value in the range `[-pi, pi]`. It follows the Common Lisp semantics for the origin when signed zeroes are supported. `atan2` y `1`, with y in a type that is `RealFloat`, should return the same value as `atan` y. A default definition of `atan2` is provided, but implementors can provide a more accurate implementation.

The precise definition of the above functions is as in Common Lisp, which in turn follows Penfield's proposal for APL [12]. See these references for discussions of branch cuts, discontinuities, and implementation.

6.4.6 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 (n,f) such that $x = n+f$, and: n is an integral number with the same sign as x; and f is a fraction f 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 `properFraction`.

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
realToFrac   :: (Real a, Fractional b) => a -> b
```

Chapter 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* that 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 6.3.6) sequentially compose actions, corresponding to sequencing operators (such as the semicolon) in imperative languages.

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 that are exported from the Prelude, and which are described in this section.

All I/O functions defined here are character oriented. The treatment of the newline character will vary on different systems. For example, two characters of input, return and linefeed, may read as a single newline character. These functions cannot be used portably for binary I/O.

In the following, recall that `String` is a synonym for `[Char]` (Section 6.1.2).

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. Printable types are those that 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
readLn      :: Read a => IO a
```

The `getChar` operation raises an exception (Section 7.3) on end-of-file; a predicate `isEOFError` that identifies this exception is defined in the `IO` library. The `getLine` operation raises an exception under the same circumstances as `hGetLine`, defined the `IO` 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 is passed to this function as its argument, and the resulting string is output on the standard output device.

Typically, the `read` operation from class `Read` is used to convert the string to a value. The `readIO` function is similar to `read` except that it signals parse failure to the I/O monad instead of terminating the program. The `readLn` function combines `getLine` and `readIO`.

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 type constructor `IO` is an instance of the `Monad` class. 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:

```
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 that construct and examine `IOError` values. The only Prelude function that creates an `IOError` value is `userError`. User error values include a string describing the error.

```
userError :: String -> IOError
```

Exceptions are raised and caught using the following functions:

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

The `ioError` function raises an exception; the `catch` function establishes a handler that 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 ioError 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 `fail` method of the `IO` instance of the `Monad` class (Section 6.3.6) raises a `userError`, thus:

```
instance Monad IO where
  ...bindings for return, (>>=), (>>)

  fail s = ioError (userError s)
```

The exceptions raised by the I/O functions in the Prelude are defined in Chapter 42.

Chapter 8

Foreign Function Interface

The Foreign Function Interface (FFI) has two purposes: it enables (1) to describe in Haskell the interface to foreign language functionality and (2) to use from foreign code Haskell routines. More generally, its aim is to support the implementation of programs in a mixture of Haskell and other languages such that the source code is portable across different implementations of Haskell and non-Haskell systems as well as independent of the architecture and operating system.

8.1 Foreign Languages

The Haskell FFI currently only specifies the interaction between Haskell code and foreign code that follows the C calling convention. However, the design of the FFI is such that it enables the modular extension of the present definition to include the calling conventions of other programming languages, such as C++ and Java. A precise definition of the support for those languages is expected to be included in later versions of the language. The second major omission is the definition of the interaction with multithreading in the foreign language and, in particular, the treatment of thread-local state, and so these details are currently implementation-defined.

The core of the present specification is independent of the foreign language that is used in conjunction with Haskell. However, there are two areas where FFI specifications must become language specific: (1) the specification of external names and (2) the marshalling of the basic types of a foreign language. As an example of the former, consider that in C [9] a simple identifier is sufficient to identify an object, while Java [5], in general, requires a qualified name in conjunction with argument and result types to resolve possible overloading. Regarding the second point, consider that many languages do not specify the exact representation of some basic types. For example the type `int` in C may be 16, 32, or 64 bits wide. Similarly, Haskell guarantees only that `Int` covers at least the range $[-2^{29}, 2^{29}-1]$ (Section 6.4). As a consequence, to reliably represent values of C's `int` in Haskell, we have to introduce a new type `CInt`, which is guaranteed to match the representation of `int`.

The specification of external names, dependent on a calling convention, is described in Section 8.5, whereas the marshalling of the basic types in dependence on a foreign language is described in Section 8.6.

8.2 Contexts

For a given Haskell system, we define the *Haskell context* to be the execution context of the abstract machine on which the Haskell system is based. This includes the heap, stacks, and the registers of the abstract machine and their mapping onto a concrete architecture. We call any other execution context an *external context.* Generally, we cannot assume any compatibility between the data formats and calling conventions between the Haskell context and a given external context, except where Haskell explicitly prescribes a specific data format.

The principal goal of a foreign function interface is to provide a programmable interface between the Haskell context and external contexts. As a result Haskell threads can access data in external contexts and invoke functions that are executed in an external context as well as vice versa. In the rest of this definition, external contexts are usually identified by a calling convention.

8.2.1 Cross Language Type Consistency

Given that many external languages support static types, the question arises whether the consistency of Haskell types with the types of the external language can be enforced for foreign functions. Unfortunately, this is, in general, not possible without a significant investment on the part of the implementor of the Haskell system (i.e., without implementing a dedicated type checker). For example, in the case of the C calling convention, the only other approach would be to generate a C prototype from the Haskell type and leave it to the C compiler to match this prototype with the prototype that is specified in a C header file for the imported function. However, the Haskell type is lacking some information that would be required to pursue this route. In particular, the Haskell type does not contain any information as to when `const` modifiers have to be emitted.

As a consequence, this definition does not require the Haskell system to check consistency with foreign types. Nevertheless, Haskell systems are encouraged to provide any cross language consistency checks that can be implemented with reasonable effort.

8.3 Lexical Structure

The FFI reserves a single keyword `foreign`, and a set of special identifiers. The latter have a special meaning only within foreign declarations, but may be used as ordinary identifiers elsewhere.

The special identifiers `ccall`, `cplusplus`, `dotnet`, `jvm`, and `stdcall` are defined to denote calling conventions. However, a concrete implementation of the FFI is free to support additional, system-specific calling conventions whose name is not explicitly listed here.

To refer to objects of an external C context, we introduce the following phrases:

$chname$	$\rightarrow$	$\{chchar\}$ `.h`	(C header filename)
cid	$\rightarrow$	$letter\ \{letter \mid ascDigit\}$	(C identifier)
$chchar$	$\rightarrow$	$letter \mid ascSymbol_{\langle \& \rangle}$	
$letter$	$\rightarrow$	$ascSmall \mid ascLarge \mid \_$	

The range of lexemes that are admissible for $chname$ is a subset of those permitted as arguments to the `#include` directive in C. In particular, a file name $chname$ must end in the suffix `.h`. The lexemes produced by cid coincide with those allowed as C identifiers, as specified in [9].

8.4 Foreign Declarations

The syntax of foreign declarations is as follows:

topdecl → `foreign` *fdecl*
fdecl → `import` *callconv* [*safety*] *impent* *var* `::` *ftype* (define variable)
| `export` *callconv* *expent* *var* `::` *ftype* (expose variable)
callconv → `ccall` | `stdcall` | `cplusplus` (calling convention)
| `jvm` | `dotnet`
| **system-specific calling conventions**
impent → [*string*]
expent → [*string*]
safety → `unsafe` | `safe`

There are two flavours of foreign declarations: import and export declarations. An import declaration makes an *external entity,* i.e., a function or memory location defined in an external context, available in the Haskell context. Conversely, an export declaration defines a function of the Haskell context as an external entity in an external context. Consequently, the two types of declarations differ in that an import declaration defines a new variable, whereas an export declaration uses a variable that is already defined in the Haskell module.

The external context that contains the external entity is determined by the calling convention given in the foreign declaration. Consequently, the exact form of the specification of the external entity is dependent on both the calling convention and on whether it appears in an import declaration (as *impent*) or in an export declaration (as *expent*). To provide syntactic uniformity in the presence of different calling conventions, it is guaranteed that the description of an external entity lexically appears as a Haskell string lexeme. The only exception is where this string would be the empty string (i.e., be of the form `""`); in this case, the string may be omitted in its entirety.

8.4.1 Calling Conventions

The binary interface to an external entity on a given architecture is determined by a calling convention. It often depends on the programming language in which the external entity is implemented, but usually is more dependent on the system for which the external entity has been compiled.

As an example of how the calling convention is dominated by the system rather than the programming language, consider that an entity compiled to byte code for the Java Virtual Machine (JVM) [11] needs to be invoked by the rules of the JVM rather than that of the source language in which it is implemented (the entity might be implemented in Oberon, for example).

Any implementation of the Haskell FFI must at least implement the C calling convention denoted by `ccall`. All other calling conventions are optional. Generally, the set of calling conventions is open, i.e., individual implementations may elect to support additional calling conventions. In addition to `ccall`, Table 8.1 specifies a range of identifiers for common calling conventions. Implementations need not implement all of these conventions, but if any is implemented, it must use the listed name. For any other calling convention, implementations are free to choose a suitable name.

Only the semantics of the calling conventions `ccall` and `stdcall` are defined herein; more calling conventions may be added in future versions of Haskell.

Identifier	Represented calling convention
`ccall`	Calling convention of the standard C compiler on a system
`cplusplus`	Calling convention of the standard C++ compiler on a system
`dotnet`	Calling convention of the .NET platform
`jvm`	Calling convention of the Java Virtual Machine
`stdcall`	Calling convention of the Win32 API (matches Pascal conventions)

Table 8.1: Calling conventions

It should be noted that the code generated by a Haskell system to implement a particular calling convention may vary widely with the target code of that system. For example, the calling convention `jvm` will be trivial to implement for a Haskell compiler generating Java code, whereas for a Haskell compiler generating C code, the Java Native Interface (JNI) [10] has to be targeted.

8.4.2 Foreign Types

The following types constitute the set of *basic foreign types*:

- `Char`, `Int`, `Double`, `Float`, and `Bool` as exported by the Haskell `Prelude` as well as
- `Int8`, `Int16`, `Int32`, `Int64`, `Word8`, `Word16`, `Word32`, `Word64`, `Ptr a`, `FunPtr a`, and `StablePtr a`, for any type `a`, as exported by `Foreign` (Section 24).

A Haskell system that implements the FFI needs to be able to pass these types between the Haskell and the external context as function arguments and results.

Foreign types are produced according to the following grammar:

$$
\begin{array}{lcll}
\mathit{ftype} & \rightarrow & \mathit{frtype} & \\
 & | & \mathit{fatype}\ \texttt{->}\ \mathit{ftype} & \\
\mathit{frtype} & \rightarrow & \mathit{fatype} & \\
 & | & \texttt{()} & \\
\mathit{fatype} & \rightarrow & \mathit{qtycon}\ \mathit{atype}_1\ \ldots\ \mathit{atype}_k & (k\ \geq\ 0)
\end{array}
$$

A foreign type is the Haskell type of an external entity. Only a subset of Haskell's types are permissible as foreign types, as only a restricted set of types can be canonically transferred between the Haskell context and an external context. A foreign type has the form

$$\mathit{at}_1\ \texttt{->}\ \cdots\ \texttt{->}\ \mathit{at}_n\ \texttt{->}\ \mathit{rt}$$

where $n \geq 0$. It implies that the arity of the external entity is n.

External functions are strict in all arguments.

Marshallable foreign types. The argument types at_i produced by fatype must be *marshallable foreign types;* that is, either

- a basic foreign type,
- a type synonym that expands to a marshallable foreign type,
- a type $T\ t'_1\ \ldots\ t'_n$ where T is defined by a `newtype` declaration

  `newtype` $T\ a_1\ \ldots\ a_n$ `=` $N\ t$

  and

  - the constructor N is visible where T is used,
  - $t[t'_1/a_1..t'_n/a_n]$ is a marshallable foreign type

Consequently, in order for a type defined by `newtype` to be used in a `foreign` declaration outside of the module that defines it, the type must not be exported abstractly. The module `Foreign.C.Types` that defines the Haskell equivalents for C types follows this convention; see Chapter 28.

Marshallable foreign result types. The result type *rt* produced by *frtype* must be a *marshallable foreign result type;* that is, either

- the type `()`,
- a type matching `Prelude.IO` t, where t is a marshallable foreign type or `()`,
- a basic foreign type,
- a type synonym that expands to marshallable foreign result type,
- a type $T\ t'_1\ \ldots\ t'_n$ where T is defined by a `newtype` declaration

  `newtype` $T\ a_1\ \ldots\ a_n$ `=` $N\ t$

  and

  - the constructor N is visible where T is used,
  - $t[t'_1/a_1..t'_n/a_n]$ is a marshallable foreign result type

8.4.3 Import Declarations

Generally, an import declaration has the form

`foreign import` $c\ e\ v$ `::` t

which declares the variable v of type t to be defined externally. Moreover, it specifies that v is evaluated by executing the external entity identified by the string e using calling convention c. The precise form of e depends on the calling convention and is detailed in Section 8.5. If a variable v is defined by an import declaration, no other top-level declaration for v is allowed in the same module. For example, the declaration

```
foreign import ccall "string.h strlen"
   cstrlen :: Ptr CChar -> IO CSize
```

introduces the function `cstrlen`, which invokes the external function `strlen` using the standard C calling convention. Some external entities can be imported as pure functions; for example,

```
foreign import ccall "math.h sin"
   sin :: CDouble -> CDouble.
```

Such a declaration asserts that the external entity is a true function; i.e., when applied to the same argument values, it always produces the same result.

Whether a particular form of external entity places a constraint on the Haskell type with which it can be imported is defined in Section 8.5. Although, some forms of external entities restrict the set of Haskell types that are permissible, the system can generally not guarantee the consistency between the Haskell type given in an import declaration and the argument and result types of the external entity. It is the responsibility of the programmer to ensure this consistency.

Optionally, an import declaration can specify, after the calling convention, the safety level that should be used when invoking an external entity. A `safe` call is less efficient, but guarantees to leave the Haskell system in a state that allows callbacks from the external code. In contrast, an `unsafe` call, while carrying less overhead, must not trigger a callback into the Haskell system. If it does, the system behaviour is undefined. The default for an invocation is to be `safe`. Note that a callback into the Haskell system implies that a garbage collection might be triggered after an external entity was called, but before this call returns. Consequently, objects other than stable pointers (cf. Section 36) may be moved or garbage collected by the storage manager.

8.4.4 Export Declarations

The general form of export declarations is

$$\texttt{foreign export } c\ e\ v\ \texttt{::}\ t$$

Such a declaration enables external access to v, which may be a value, field name, or class method that is declared on the top-level of the same module or imported. Moreover, the Haskell system defines the external entity described by the string e, which may be used by external code using the calling convention c; an external invocation of the external entity e is translated into evaluation of v. The type t must be an instance of the type of v. For example, we may have

```
foreign export ccall "addInt"   (+) :: Int   -> Int   -> Int
foreign export ccall "addFloat" (+) :: Float -> Float -> Float
```

If an evaluation triggered by an external invocation of an exported Haskell value returns with an exception, the system behaviour is undefined. Thus, Haskell exceptions have to be caught within Haskell and explicitly marshalled to the foreign code.

8.5 Specification of External Entities

Each foreign declaration has to specify the external entity that is accessed or provided by that declaration. The syntax and semantics of the notation that is required to uniquely determine an external entity depends heavily on the calling convention by which this entity is accessed. For example, for the calling convention `ccall`, a global label is sufficient. However, to uniquely identify a method in the calling convention `jvm`, type information has to be provided. For the latter, there is a choice between the Java source-level syntax of types and the syntax expected by JNI—but, clearly, the syntax of the specification of an external entity depends on the calling convention and may be non-trivial.

Consequently, the FFI does not fix a general syntax for denoting external entities, but requires both *impent* and *expent* to take the form of a Haskell *string* literal. The formation rules for the values of these strings depend on the calling convention and a Haskell system implementing a particular calling convention will have to parse these strings in accordance with the calling convention.

Defining *impent* and *expent* to take the form of a *string* implies that all information that is needed to statically analyse the Haskell program is separated from the information needed to generate the code interacting with the foreign language. This is, in particular, helpful for tools processing Haskell source code. When ignoring the entity information provided by *impent* or *expent*, foreign import and export declarations are still sufficient to infer identifier definition and use information as well as type information.

For more complex calling conventions, there is a choice between the user-level syntax for identifying entities (e.g., Java or C++) and the system-level syntax (e.g., the type syntax of JNI or mangled C++, respectively). If such a choice exists, the user-level syntax is preferred. Not only because it is more user friendly, but also because the system-level syntax may not be entirely independent of the particular implementation of the foreign language.

The following defines the syntax for specifying external entities and their semantics for the calling conventions `ccall` and `stdcall`. Other calling conventions from Table 8.1 are expected to be defined in future versions of Haskell.

8.5.1 Standard C Calls

The following defines the structure of external entities for foreign declarations under the `ccall` calling convention for both import and export declarations separately. Afterwards additional constraints on the type of foreign functions are defined.

The FFI covers only access to C functions and global variables. There are no mechanisms to access other entities of C programs. In particular, there is no support for accessing pre-processor symbols from Haskell, which includes `#define`d constants. Access from Haskell to such entities is the domain of language-specific tools, which provide added convenience over the plain FFI as defined here.

Import Declarations For import declarations, the syntax for the specification of external entities under the `ccall` calling convention is as follows:

impent	→	`"` [`static`] [*chname*] [`&`] [*cid*] `"`	(static function or address)
	|	`"` *dynamic* `"`	(stub factory importing addresses)
	|	`"` *wrapper* `"`	(stub factory exporting thunks)

The first alternative either imports a static function *cid* or, if `&` precedes the identifier, a static address. If *cid* is omitted, it defaults to the name of the imported Haskell variable. The optional filename *chname* specifies a C header file, where the intended meaning is that the header file declares the C entity identified by *cid*. In particular, when the Haskell system compiles Haskell to C code, the directive

```
#include "chname"
```

needs to be placed into any generated C file that refers to the foreign entity before the first occurrence of that entity in the generated C file.

The second and third alternative, identified by the keywords `dynamic` and `wrapper`, respectively, import stub functions that have to be generated by the Haskell system. In the case of `dynamic`, the stub converts C function pointers into Haskell functions; and conversely, in the case of `wrapper`, the stub converts Haskell thunks to C function pointers. If neither of the specifiers `static`, `dynamic`, or `wrapper` is given, `static` is assumed. The specifier `static` is nevertheless needed to import C routines that are named `dynamic` or `wrapper`.

It should be noted that a static foreign declaration that does not import an address (i.e., where `&` is not used in the specification of the external entity) always refers to a C function, even if the Haskell type is non-functional. For example,

```
foreign import ccall foo :: CInt
```

refers to a pure C function `foo` with no arguments that returns an integer value. Similarly, if the type is `IO CInt`, the declaration refers to an impure nullary function. If a Haskell program needs to access a C variable `bar` of integer type,

```
foreign import ccall "&" bar :: Ptr CInt
```

must be used to obtain a pointer referring to the variable. The variable can be read and updated using the routines provided by the module `Foreign.Storable` (cf. Section 37).

Export Declarations External entities in *ccall* export declarations are of the form

$$expent \quad \rightarrow \quad \texttt{"}\ [cid]\ \texttt{"}$$

The optional C identifier *cid* defines the external name by which the exported Haskell variable is accessible in C. If it is omitted, the external name defaults to the name of the exported Haskell variable.

Constraints on Foreign Function Types In the case of import declaration, there are, depending on the kind of import declaration, constraints regarding the admissible Haskell type that the variable defined in the import may have. These constraints are specified in the following.

Static Functions. A static function can be of any foreign type; in particular, the result type may or may not be in the IO monad. If a function that is not pure is not imported in the IO monad, the system behaviour is undefined. Generally, no check for consistency with the C type of the imported label is performed.

As an example, consider

```
foreign import ccall "static stdlib.h"
   system :: Ptr CChar -> IO CInt
```

This declaration imports the `system()` function whose prototype is available from `stdlib.h`.

Static addresses. The type of an imported address is constrained to be of the form `Ptr` *a* or `FunPtr` *a*, where *a* can be any type.

As an example, consider

```
foreign import ccall "errno.h &errno" errno :: Ptr CInt
```

It imports the address of the variable `errno`, which is of the C type `int`.

Dynamic import. The type of a *dynamic* stub has to be of the form `(FunPtr` *ft*`) ->` *ft*, where *ft* may be any foreign type.

As an example, consider

```
foreign import ccall "dynamic"
  mkFun :: FunPtr (CInt -> IO ()) -> (CInt -> IO ())
```

The stub factory `mkFun` converts any pointer to a C function that gets an integer value as its only argument and does not have a return value into a corresponding Haskell function.

Dynamic wrapper. The type of a *wrapper* stub has to be of the form *ft* `-> IO (FunPtr` *ft*`)`, where *ft* may be any foreign type.

As an example, consider

```
foreign import ccall "wrapper"
  mkCallback :: IO () -> IO (FunPtr (IO ()))
```

The stub factory `mkCallback` turns any Haskell computation of type `IO ()` into a C function pointer that can be passed to C routines, which can call back into the Haskell context by invoking the referenced function.

Specification of Header Files A C header specified in an import declaration is always included by `#include "`*chname*`"`. There is no explicit support for `#include <`*chname*`>` style inclusion. The ISO C99 [7] standard guarantees that any search path that would be used for a `#include <`*chname*`>` is also used for `#include "`*chname*`"` and it is guaranteed that these paths are searched after all paths that are unique to `#include "`*chname*`"`. Furthermore, we require that *chname* ends in `.h` to make parsing of the specification of external entities unambiguous.

The specification of include files has been kept to a minimum on purpose. Libraries often require a multitude of include directives, some of which may be system-dependent. Any design that attempts to cover all possible configurations would introduce significant complexity. Moreover, in the current design, a custom include file can be specified that uses the standard C preprocessor features to include all relevant headers.

Header files have no impact on the semantics of a foreign call, and whether an implementation uses the header file or not is implementation-defined. However, as some implementations may require a header file that supplies a correct prototype for external functions in order to generate correct code, portable FFI code must include suitable header files.

C Argument Promotion The argument passing conventions of C are dependent on whether a function prototype for the called functions is in scope at a call site. In particular, if no function prototype is in scope, *default argument promotion* is applied to integral and floating types. In general, it cannot be expected from a Haskell system that it is aware of whether a given C function was compiled with or without a function prototype being in scope. For the sake of portability, we thus require that a Haskell system generally implements calls to C functions as well as C stubs for Haskell functions as if a function prototype for the called function is in scope.

This convention implies that the onus for ensuring the match between C and Haskell code is placed on the FFI user. In particular, when a C function that was compiled without a prototype is called from Haskell, the Haskell signature at the corresponding `foreign import` declaration must use the types *after* argument promotion. For example, consider the following C function definition, which lacks a prototype:

```
void foo (a)
float a;
{
  ...
}
```

The lack of a prototype implies that a C compiler will apply default argument promotion to the parameter `a`, and thus, `foo` will expect to receive a value of type `double`, *not* `float`. Hence, the correct `foreign import` declaration is

```
foreign import ccall foo :: Double -> IO ()
```

In contrast, a C function compiled with the prototype

```
void foo (float a);
```

requires

```
foreign import ccall foo :: Float -> IO ()
```

A similar situation arises in the case of `foreign export` declarations that use types that would be altered under the C default argument promotion rules. When calling such Haskell functions from C, a function prototype matching the signature provided in the `foreign export` declaration must be in scope; otherwise, the C compiler will erroneously apply the promotion rules to all function arguments.

Note that for a C function defined to accept a variable number of arguments, all arguments beyond the explicitly typed arguments suffer argument promotion. However, because C permits the calling convention to be different for such functions, a Haskell system will, in general, not be able to make use of variable argument functions. Hence, their use is deprecated in portable code.

8.5.2 Win32 API Calls

The specification of external entities under the `stdcall` calling convention is identical to that for standard C calls. The two calling conventions only differ in the generated code.

8.6 Marshalling

In addition to the language extension discussed in previous sections, the FFI includes a set of standard libraries, which ease portable use of foreign functions as well as marshalling of compound structures. Generally, the marshalling of Haskell structures into a foreign representation and vice versa can be implemented in either Haskell or the foreign language. At least where the foreign language is at a significantly lower level, e.g. C, there are good reasons for doing the marshalling in Haskell:

- Haskell's lazy evaluation strategy would require any foreign code that attempts to access Haskell structures to force the evaluation of these structures before accessing them. This would lead to complicated code in the foreign language, but does not need any extra consideration when coding the marshalling in Haskell.

C symbol	Haskell symbol	Constraint on concrete C type
`HsChar`	`Char`	integral type
`HsInt`	`Int`	signed integral type, $\geq$ 30 bit
`HsInt8`	`Int8`	signed integral type, 8 bit; `int8_t` if available
`HsInt16`	`Int16`	signed integral type, 16 bit; `int16_t` if available
`HsInt32`	`Int32`	signed integral type, 32 bit; `int32_t` if available
`HsInt64`	`Int64`	signed integral type, 64 bit; `int64_t` if available
`HsWord8`	`Word8`	unsigned integral type, 8 bit; `uint8_t` if available
`HsWord16`	`Word16`	unsigned integral type, 16 bit; `uint16_t` if available
`HsWord32`	`Word32`	unsigned integral type, 32 bit; `uint32_t` if available
`HsWord64`	`Word64`	unsigned integral type, 64 bit; `uint64_t` if available
`HsFloat`	`Float`	floating point type
`HsDouble`	`Double`	floating point type
`HsBool`	`Bool`	`int`
`HsPtr`	`Ptr a`	`(void *)`
`HsFunPtr`	`FunPtr a`	`(void (*)(void))`
`HsStablePtr`	`StablePtr a`	`(void *)`

Table 8.2: C Interface to Basic Haskell Types

- Despite the fact that marshalling code in Haskell tends to look like C in Haskell syntax, the strong type system still catches many errors that would otherwise lead to difficult-to-debug runtime faults.
- Direct access to Haskell heap structures from a language like C—especially, when marshalling from C to Haskell, i.e., when Haskell structures are created—carries the risk of corrupting the heap, which usually leads to faults that are very hard to debug.

Consequently, the Haskell FFI emphasises Haskell-side marshalling.

The interface to the marshalling libraries is provided by the module `Foreign` (Chapter 24) plus a language-dependent module per supported language. In particular, the standard requires the availability of the module `Foreign.C` (Chapter 25), which simplifies portable interfacing with external C code. Language-dependent modules, such as `Foreign.C`, generally provide Haskell types representing the basic types of the foreign language using a representation that is compatible with the foreign types as implemented by the default implementation of the foreign language on the present architecture. This is especially important for languages where the standard leaves some aspects of the implementation of basic types open. For example, in C, the size of the various integral types is not fixed. Thus, to represent C interfaces faithfully in Haskell, for each integral type in C, we need to have an integral type in Haskell that is guaranteed to have the same size as the corresponding C type.

8.7 The External C Interface

Every Haskell system that implements the FFI needs to provide a C header file named `HsFFI.h` that defines the C symbols listed in Tables 8.2 and 8.3. Table 8.2 table lists symbols that represent types together with the Haskell type that they represent and any constraints that are placed on the concrete C types that implement these symbols. When a C type `HsT` represents a Haskell type `T`, the occurrence of `T` in a foreign function declaration should be matched by `HsT` in the corresponding C function prototype. Indeed, where the Haskell

CPP symbol	Haskell value	Description
`HS_CHAR_MIN`	`minBound :: Char`	
`HS_CHAR_MAX`	`maxBound :: Char`	
`HS_INT_MIN`	`minBound :: Int`	
`HS_INT_MAX`	`maxBound :: Int`	
`HS_INT8_MIN`	`minBound :: Int8`	
`HS_INT8_MAX`	`maxBound :: Int8`	
`HS_INT16_MIN`	`minBound :: Int16`	
`HS_INT16_MAX`	`maxBound :: Int16`	
`HS_INT32_MIN`	`minBound :: Int32`	
`HS_INT32_MAX`	`maxBound :: Int32`	
`HS_INT64_MIN`	`minBound :: Int64`	
`HS_INT64_MAX`	`maxBound :: Int64`	
`HS_WORD8_MAX`	`maxBound :: Word8`	
`HS_WORD16_MAX`	`maxBound :: Word16`	
`HS_WORD32_MAX`	`maxBound :: Word32`	
`HS_WORD64_MAX`	`maxBound :: Word64`	
`HS_FLOAT_RADIX`	`floatRadix :: Float`	
`HS_FLOAT_ROUND`	n/a	rounding style as per [7]
`HS_FLOAT_EPSILON`	n/a	difference between 1 and the least value greater than 1 as per [7]
`HS_DOUBLE_EPSILON`	n/a	(as above)
`HS_FLOAT_DIG`	n/a	number of decimal digits as per [7]
`HS_DOUBLE_DIG`	n/a	(as above)
`HS_FLOAT_MANT_DIG`	`floatDigits :: Float`	
`HS_DOUBLE_MANT_DIG`	`floatDigits :: Double`	
`HS_FLOAT_MIN`	n/a	minimum floating point number as per [7]
`HS_DOUBLE_MIN`	n/a	(as above)
`HS_FLOAT_MIN_EXP`	`fst . floatRange :: Float`	
`HS_DOUBLE_MIN_EXP`	`fst . floatRange :: Double`	
`HS_FLOAT_MIN_10_EXP`	n/a	minimum decimal exponent as per [7]
`HS_DOUBLE_MIN_10_EXP`	n/a	(as above)
`HS_FLOAT_MAX`	n/a	maximum floating point number as per [7]
`HS_DOUBLE_MAX`	n/a	(as above)
`HS_FLOAT_MAX_EXP`	`snd . floatRange :: Float`	
`HS_DOUBLE_MAX_EXP`	`snd . floatRange :: Double`	
`HS_FLOAT_MAX_10_EXP`	n/a	maximum decimal exponent as per [7]
`HS_DOUBLE_MAX_10_EXP`	n/a	(as above)
`HS_BOOL_FALSE`	False	
`HS_BOOL_TRUE`	True	

Table 8.3: C Interface to Range and Precision of Basic Types

system translates Haskell to C code that invokes `foreign import`ed C routines, such prototypes need to be provided and included via the header that can be specified in external entity strings for foreign C functions (cf. Section 8.5.1); otherwise, the system behaviour is undefined. It is guaranteed that the Haskell value `nullPtr` is mapped to `(HsPtr) NULL` in C and `nullFunPtr` is mapped to `(HsFunPtr) NULL` and vice versa.

Table 8.3 contains symbols characterising the range and precision of the types from Table 8.2. Where available, the table states the corresponding Haskell values. All C symbols, with the exception of `HS_FLOAT_ROUND` are constants that are suitable for use in `#if` preprocessing directives. Note that there is only one rounding style (`HS_FLOAT_ROUND`) and one radix (`HS_FLOAT_RADIX`), as this is all that is supported by ISO C [7].

Moreover, an implementation that does not support 64 bit integral types on the C side should implement `HsInt64` and `HsWord64` as a structure. In this case, the bounds `HS_INT64_MIN`, `HS_INT64_MAX`, and `HS_WORD64_MAX` are undefined.

In addition, to the symbols from Table 8.2 and 8.3, the header `HsFFI.h` must also contain the following prototypes:

```
void hs_init     (int *argc, char **argv[]);
void hs_exit     (void);
void hs_set_argv (int argc, char *argv[]);

void hs_perform_gc (void);

void hs_free_stable_ptr (HsStablePtr sp);
void hs_free_fun_ptr    (HsFunPtr fp);
```

These routines are useful for mixed language programs, where the main application is implemented in a foreign language that accesses routines implemented in Haskell. The function `hs_init()` initialises the Haskell system and provides it with the available command line arguments. Upon return, the arguments solely intended for the Haskell runtime system are removed (i.e., the values that `argc` and `argv` point to may have changed). This function must be called during program startup before any Haskell function is invoked; otherwise, the system behaviour is undefined. Conversely, the Haskell system is deinitialised by a call to `hs_exit()`. Multiple invocations of `hs_init()` are permitted, provided that they are followed by an equal number of calls to `hs_exit()` and that the first call to `hs_exit()` is after the last call to `hs_init()`. In addition to nested calls to `hs_init()`, the Haskell system may be de-initialised with `hs_exit()` and be re-initialised with `hs_init()` at a later point in time. This ensures that repeated initialisation due to multiple libraries being implemented in Haskell is covered.

The Haskell system will ignore the command line arguments passed to the second and any following calls to `hs_init()`. Moreover, `hs_init()` may be called with `NULL` for both `argc` and `argv`, signalling the absence of command line arguments.

The function `hs_set_argv()` sets the values returned by the functions `getProgName` and `getArgs` of the module `System.Environment` (Section 39). This function may only be invoked after `hs_init()`. Moreover, if `hs_set_argv()` is called at all, this call must precede the first invocation of `getProgName` and `getArgs`. Note that the separation of `hs_init()` and `hs_set_argv()` is essential in cases where in addition to the Haskell system other libraries that process command line arguments during initialisation are used.

The function `hs_perform_gc()` advises the Haskell storage manager to perform a garbage collection, where the storage manager makes an effort to releases all unreachable objects. This function must not be invoked from C functions that are imported `unsafe` into Haskell code nor may it be used from a finalizer.

Finally, `hs_free_stable_ptr()` and `hs_free_fun_ptr()` are the C counterparts of the Haskell functions `freeStablePtr` and `freeHaskellFunPtr`.

Chapter 9

Standard Prelude

In this chapter the entire Haskell Prelude is given. It constitutes a *specification* for the Prelude. Many of the definitions are written with clarity rather than efficiency in mind, and it is not required that the specification be implemented as shown here.

The default method definitions, given with `class` declarations, constitute a specification *only* of the default method. They do not constitute a specification of the meaning of the method in all instances. To take one particular example, the default method for `enumFrom` in class `Enum` will not work properly for types whose range exceeds that of `Int` (because `fromEnum` cannot map all values in the type to distinct `Int` values).

The Prelude shown here is organized into a root module, `Prelude`, and three sub-modules, `PreludeList`, `PreludeText`, and `PreludeIO`. This structure is purely presentational. An implementation is not required to use this organisation for the Prelude, nor are these three modules available for import separately. Only the exports of module `Prelude` are significant.

Some of these modules import Library modules, such as `Data.Char`, `Control.Monad`, `System.IO`, and `Numeric`. These modules are described fully in Part II. These imports are not, of course, part of the specification of the `Prelude`. That is, an implementation is free to import more, or less, of the Library modules, as it pleases.

Primitives that 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 that 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 syntactically invalid. An ellipsis "`...`" is often used in places where the remainder of a definition cannot be given in Haskell.

To reduce the occurrence of unexpected ambiguity errors, and to improve efficiency, a number of commonly-used functions over lists use the `Int` type rather than using a more general numeric type, such as `Integral a` or `Num a`. These functions are: `take`, `drop`, `!!`, `length`, `splitAt`, and `replicate`. The more general versions are given in the `Data.List` library, with the prefix "`generic`"; for example `genericLength`.

```
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, Rational, IO,


--      These built-in types are defined in the Prelude, but
--      are denoted by built-in syntax, and cannot legally
--      appear in an export list.
--  List type: []((:), [])
--  Tuple types: (,)((,)), (,,)((,,)), etc.
--  Trivial type: ()(())
--  Functions: (->)

    Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
    Bounded(minBound, maxBound),
    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, atan2),
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    maybe, either,
    (&&), (||), not, otherwise,
    subtract, even, odd, gcd, lcm, (^), (^^),
    fromIntegral, realToFrac,
    fst, snd, curry, uncurry, id, const, (.), flip, ($), until,
    asTypeOf, error, undefined,
    seq, ($!)
  ) where


import PreludeBuiltin                      -- Contains all ‘prim’ values
import UnicodePrims( primUnicodeMaxChar )  -- Unicode primitives
import PreludeList
import PreludeText
import PreludeIO
import Data.Ratio( Rational )


infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
```

```
-- The (:) operator is built-in syntax, and cannot legally be given
-- a fixity declaration; but its fixity is given by:
--   infixr 5  :

infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`


-- Standard types, classes, instances and related functions

-- Equality and Ordered classes

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

        -- Minimal complete definition:
        --      (==) or (/=)
    x /= y     =  not (x == y)
    x == y     =  not (x /= y)


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

        -- Minimal complete definition:
        --      (<=) or compare
        -- 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


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


-- Enumeration and Bounded classes

class  Enum a  where
    succ, pred       :: a -> a
    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]

        -- Minimal complete definition:
        --      toEnum, fromEnum
        --
        -- NOTE: these default methods only make sense for types
        --       that map injectively into Int using fromEnum
        --       and toEnum.
    succ             =  toEnum . (+1) . fromEnum
    pred             =  toEnum . (subtract 1) . fromEnum
    enumFrom x       =  map toEnum [fromEnum x ..]
    enumFromTo x y   =  map toEnum [fromEnum x .. fromEnum y]
    enumFromThen x y =  map toEnum [fromEnum x, fromEnum y ..]
    enumFromThenTo x y z =
                        map toEnum [fromEnum x, fromEnum y .. fromEnum z]


class  Bounded a  where
    minBound         :: a
    maxBound         :: a


-- Numeric classes

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

        -- Minimal complete definition:
        --      All, except negate or (-)
    x - y            =  x + negate y
    negate x         =  0 - x


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


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

        -- Minimal complete definition:
        --      quotRem, toInteger
    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

        -- Minimal complete definition:
        --      fromRational and (recip or (/))
    recip x          =  1 / x
    x / y            =  x * recip y


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

        -- Minimal complete definition:
        --      pi, exp, log, sin, cos, sinh, cosh
        --      asin, acos, atan
        --      asinh, acosh, atanh
    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

        -- Minimal complete definition:
        --      properFraction
    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
    atan2            :: a -> a -> a

        -- Minimal complete definition:
        --      All except exponent, significand,
        --                 scaleFloat, atan2
    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

    atan2 y x
      | x>0           =  atan (y/x)
      | x==0 && y>0   =  pi/2
      | x<0  && y>0   =  pi + atan (y/x)
      |(x<=0 && y<0)  ||
       (x<0 && isNegativeZero y) ||
       (isNegativeZero x && isNegativeZero y)
                      = -atan2 (-y) x
      | y==0 && (x<0 || isNegativeZero x)
                      =  pi    -- must be after the previous test on zero y
      | x==0 && y==0  =  y     -- must be after the other double zero tests
      | otherwise     =  x + y -- x or y is a NaN, return a NaN (via +)

-- 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

realToFrac     :: (Real a, Fractional b) => a -> b
realToFrac      =  fromRational . toRational

-- Monadic classes

class  Functor f  where
    fmap              :: (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
    fail   :: String -> m a

        -- Minimal complete definition:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s

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

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

-- The xxxM functions take list arguments, but lift the function or
-- list element to a monad type
mapM             :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as        =  sequence (map f as)

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

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

```
-- Trivial type

data  ()  =  ()  deriving (Eq, Ord, Enum, Bounded)
        -- Not legal Haskell; for illustration only


-- Function type

-- 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


seq :: a -> b -> b
seq = ...       -- Primitive


-- right-associating infix application operators
-- (useful in continuation-passing style)
($), ($!) :: (a -> b) -> a -> b
f $  x    =  f x
f $! x    =  x `seq` f x


-- 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' ... -- Unicode 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  =  primUnicodeMaxChar


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
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)


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


-- 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
   fmap f x           =  x >>= (return . f)


instance Monad IO where
   (>>=)  = ...
   return = ...
   fail s = ioError (userError s)


-- 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 .. 0.95].  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
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- may overflow
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo


instance  Enum Double  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- may overflow
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo


numericEnumFrom         :: (Fractional a) => a -> [a]
numericEnumFromThen     :: (Fractional a) => a -> a -> [a]
numericEnumFromTo       :: (Fractional a, Ord a) => a -> a -> [a]
numericEnumFromThenTo   :: (Fractional a, Ord a) => a -> a -> a -> [a]
numericEnumFrom         =  iterate (+1)
numericEnumFromThen n m =  iterate (+(m-n)) n
numericEnumFromTo n m   =  takeWhile (<= m+1/2) (numericEnumFrom n)
numericEnumFromThenTo n n’ m = takeWhile p (numericEnumFromThen n n’)
                             where
                               p | n’ >= n   = (<= m + (n’-n)/2)
                                 | otherwise = (>= m + (n’-n)/2)


-- Lists

data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
        -- Not legal Haskell; for illustration only


instance Functor [] where
    fmap = map
```

```
instance  Monad []  where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []

-- Tuples

data  (a,b)   =  (a,b)    deriving (Eq, Ord, Bounded)
data  (a,b,c) =  (a,b,c)  deriving (Eq, Ord, Bounded)
        -- Not legal Haskell; for illustration only

-- 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)

-- Misc functions

-- 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 that are more appropriate to the context in which undefined
-- appears.

undefined        :: a
undefined        =  error "Prelude.undefined"
```

9.1 Prelude `PreludeList`

```
-- Standard list functions

module PreludeList (
    map, (++), filter, concat, concatMap,
    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,
    zip, zip3, zipWith, zipWith3, unzip, unzip3)
  where

import qualified Data.Char(isSpace)

infixl 9  !!
infixr 5  ++
infix  4  `elem`, `notElem`

-- Map and append
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

filter :: (a -> Bool) -> [a] -> [a]
filter p []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

-- 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 "Prelude.head: empty list"
```

```
tail             :: [a] -> [a]
tail (_:xs)      =  xs
tail []          =  error "Prelude.tail: empty list"

last             :: [a] -> a
last [x]         =  x
last (_:xs)      =  last xs
last []          =  error "Prelude.last: empty list"

init             :: [a] -> [a]
init [x]         =  []
init (x:xs)      =  x : init xs
init []          =  error "Prelude.init: empty list"

null             :: [a] -> Bool
null []          =  True
null (_:_)       =  False

-- length returns the length of a finite list as an Int.
length           :: [a] -> Int
length []        =  0
length (_:l)     =  1 + length l

-- List index (subscript) operator, 0-origin
(!!)                :: [a] -> Int -> a
xs     !! n | n < 0 =  error "Prelude.!!: negative index"
[]     !! _         =  error "Prelude.!!: index too large"
(x:_)  !! 0         =  x
(_:xs) !! n         =  xs !! (n-1)

-- 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 "Prelude.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 _ []      =  []

-- 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 "Prelude.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 []     =  []
scanr1 f [x]    =  [x]
scanr1 f (x:xs) =  f x q : qs
                   where qs@(q:_) = scanr1 f xs

-- 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 []         =  error "Prelude.cycle: empty list"
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 n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

```
drop                   :: Int -> [a] -> [a]
drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs


splitAt                  :: Int -> [a] -> ([a],[a])
splitAt n xs             =  (take n xs, drop n xs)


-- 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:ys,zs)
            | otherwise =  ([],xs)
                           where (ys,zs) = span p 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 (/= 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 "Prelude.maximum: empty list"
maximum xs       =  foldl1 max xs


minimum []       =  error "Prelude.minimum: empty list"
minimum xs       =  foldl1 min xs
```

```
-- 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))
                          ([],[],[])
```

9.2 Prelude `PreludeText`

```
module PreludeText (
    ReadS, ShowS,
    Read(readsPrec, readList),
    Show(showsPrec, show, showList),
    reads, shows, read, lex,
    showChar, showString, readParen, showParen ) where

-- The instances of Read and Show for
--      Bool, Maybe, Either, Ordering
-- are done via "deriving" clauses in Prelude.hs

import Data.Char(isSpace, isAlpha, isDigit, isAlphaNum,
                 showLitChar, readLitChar, lexLitChar)

import Numeric(showSigned, showInt, readSigned, readDec, showFloat,
               readFloat, lexDigits)

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

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

        -- Minimal complete definition:
        --      readsPrec
    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
    show             :: a -> String
    showList         :: [a] -> ShowS

        -- Mimimal complete definition:
        --      show or showsPrec
    showsPrec _ x s   = show x ++ s

    show x            = showsPrec 0 x ""

    showList []       = showString "[]"
    showList (x:xs)   = showChar '[' . shows x . showl xs
                        where showl []     = showChar ']'
                              showl (x:xs) = showChar ',' . 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 "Prelude.read: no parse"
                         _   -> error "Prelude.read: ambiguous parse"

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
--    Octal and hexidecimal numerics are not recognized as a single token
--    Comments are not treated properly

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 ('.':c:cs) | isDigit c
                            = [('.':ds++e,u) | (ds,t) <- lexDigits (c:cs),
                                               (e,u)  <- lexExp t]
              lexFracExp s  = lexExp 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)]

instance  Show Int  where
    showsPrec n = showsPrec n . toInteger
        -- Converting to Integer avoids
        -- possible difficulty with minInt


instance  Read Int  where
  readsPrec p r = [(fromInteger i, t) | (i,t) <- readsPrec p r]
        -- Reading at the Integer type avoids
        -- possible difficulty with minInt


instance  Show Integer  where
    showsPrec           = showSigned showInt

instance  Read Integer  where
    readsPrec p         = readSigned readDec

instance  Show Float  where
    showsPrec p         = showFloat

instance  Read Float  where
    readsPrec p         = readSigned readFloat

instance  Show Double  where
    showsPrec p         = showFloat

instance  Read Double  where
    readsPrec p         = readSigned readFloat

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
    showsPrec p ’\’’ = showString "’\\’’"
    showsPrec p c    = showChar ’\’’ . showLitChar c . showChar ’\’’

    showList cs = showChar ’"’ . showl cs
                 where showl ""       = showChar ’"’
                       showl (’"’:cs) = showString "\\\"" . showl cs
                       showl (c:cs)   = showLitChar c . showl cs


instance  Read Char  where
    readsPrec p      = readParen False
                            (\r -> [(c,t) | (’\’’:s,t)<- lex r,
                                            (c,"\’")  <- readLitChar s])

    readList = readParen False (\r -> [(l,t) | (’"’:s, t) <- lex r,
                                               (l,_)      <- readl s ])
        where readl (’"’:s)      = [("",s)]
              readl (’\\’:’&’:s) = readl s
              readl s            = [(c:cs,u) | (c ,t) <- readLitChar s,
                                               (cs,u) <- readl t       ]


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 ] )


-- Other tuples have similar Read and Show instances
```

9.3 Prelude `PreludeIO`

```
module PreludeIO (
    FilePath, IOError, ioError, 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 dependent

instance  Show IOError  where ...
instance  Eq IOError  where ...

ioError    ::  IOError -> IO a
ioError    =   primIOError

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_ putChar s

putStrLn   :: String -> IO ()
putStrLn s =  do putStr s
                 putStr "\n"

print      :: Show a => 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 ()
-- The hSetBuffering ensures the expected interactive behaviour
interact f  =  do hSetBuffering stdin  NoBuffering
                  hSetBuffering stdout NoBuffering
                  s <- getContents
                  putStr (f s)


readFile   :: FilePath -> IO String
readFile   =  primReadFile


writeFile  :: FilePath -> String -> IO ()
writeFile  =  primWriteFile


appendFile :: FilePath -> String -> IO ()
appendFile =  primAppendFile

  -- raises an exception instead of an error
readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")


readLn :: Read a => IO a
readLn =  do l <- getLine
             r <- readIO l
             return r
```

Chapter 10

Syntax Reference

10.1 Notational Conventions

These notational conventions are used for presenting syntax:

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

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

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

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

10.2 Lexical Syntax

$$\begin{array}{lcl}
program & \rightarrow & \{\ lexeme \mid whitespace\ \} \\
lexeme & \rightarrow & qvarid \mid qconid \mid qvarsym \mid qconsym \\
 & \mid & literal \mid special \mid reservedop \mid reservedid
\end{array}$$

literal → *integer* | *float* | *char* | *string*
special → (|) | , | ; | [|] | ` | { | }

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

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

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

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

ascSymbol → ! | # | $ | % | & | ⋆ | + | . | / | < | = | > | ? | @
| \ | ^ | | | - | ~ | :
uniSymbol → any Unicode symbol or punctuation
digit → *ascDigit* | *uniDigit*
ascDigit → 0 | 1 | ... | 9
uniDigit → any Unicode decimal digit
octit → 0 | 1 | ... | 7
hexit → *digit* | A | ... | F | a | ... | f

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

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

$$\begin{array}{lcll}
\mathit{varid} & & & \text{(variables)} \\
\mathit{conid} & & & \text{(constructors)} \\
\mathit{tyvar} & \rightarrow & \mathit{varid} & \text{(type variables)} \\
\mathit{tycon} & \rightarrow & \mathit{conid} & \text{(type constructors)} \\
\mathit{tycls} & \rightarrow & \mathit{conid} & \text{(type classes)} \\
\mathit{modid} & \rightarrow & \{\mathit{conid}\ \texttt{.}\}\ \mathit{conid} & \text{(modules)}
\end{array}$$

$$\begin{array}{lcl}
\mathit{qvarid} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{varid} \\
\mathit{qconid} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{conid} \\
\mathit{qtycon} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{tycon} \\
\mathit{qtycls} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{tycls} \\
\mathit{qvarsym} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{varsym} \\
\mathit{qconsym} & \rightarrow & [\ \mathit{modid}\ \texttt{.}\ ]\ \mathit{consym}
\end{array}$$

$$\begin{array}{lcl}
\mathit{decimal} & \rightarrow & \mathit{digit}\{\mathit{digit}\} \\
\mathit{octal} & \rightarrow & \mathit{octit}\{\mathit{octit}\} \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\{\mathit{hexit}\}
\end{array}$$

$$\begin{array}{lcl}
\mathit{integer} & \rightarrow & \mathit{decimal} \\
 & \mid & \texttt{0o}\ \mathit{octal} \mid \texttt{0O}\ \mathit{octal} \\
 & \mid & \texttt{0x}\ \mathit{hexadecimal} \mid \texttt{0X}\ \mathit{hexadecimal} \\
\mathit{float} & \rightarrow & \mathit{decimal}\ \texttt{.}\ \mathit{decimal}\ [\mathit{exponent}] \\
 & \mid & \mathit{decimal}\ \mathit{exponent} \\
\mathit{exponent} & \rightarrow & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{decimal}
\end{array}$$

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

10.3 Layout

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

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

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

- A stream of lexemes as specified by the lexical syntax in the Haskell report, with the following additional tokens:
  - If a `let`, `where`, `do`, or `of` keyword is not followed by the lexeme `{`, the token $\{n\}$ is inserted after the keyword, where n is the indentation of the next lexeme if there is one, or *0* if the end of file has been reached.
  - If the first lexeme of a module is not `{` or `module`, then it is preceded by $\{n\}$ where n is the indentation of the lexeme.
  - Where the start of a lexeme is preceded only by white space on the same line, this lexeme is preceded by $< n >$ where n is the indentation of the lexeme, provided that it is not, as a consequence of the first two rules, preceded by $\{n\}$. (NB: a string literal may span multiple lines – Section 2.6. So in the fragment

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

    There is no $< n >$ inserted before the `\Bill`, because it is not the beginning of a complete lexeme; nor before the `,`, because it is not preceded only by white space.)
- A stack of "layout contexts", in which each element is either:
  - Zero, indicating that the enclosing context is explicit (i.e. the programmer supplied the opening brace). If the innermost context is 0, then no layout tokens will be inserted until either the enclosing context ends or a new context is pushed.
  - A positive integer, which is the indentation column of the enclosing layout context.

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

- The characters $newline$, $return$, $linefeed$, and $formfeed$, all start a new line.
- The first column is designated column 1, not 0.
- Tab stops are 8 characters apart.
- A tab character causes the insertion of enough spaces to align the current position with the next tab stop.

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

The application

$$L\ tokens\ []$$

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

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

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

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

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

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

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

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

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

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

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

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

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

```
let x = e; y = x in e′
```

is valid, because it translates to

```
let { x = e; y = x } in e′
```

The close brace is inserted due to the parse error rule above.

10.4 Literate comments

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

The program text is recovered by taking only those lines beginning with “>”, and replacing the leading “>” with a space. Layout and comments apply exactly as described in Chapter 10 in the resulting text.

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

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

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

> main :: IO ()

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

  This is the factorial function.

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

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

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

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

```
\documentstyle{article}

\begin{document}

\chapter{Introduction}

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

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

\end{document}
```

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

10.5 Context-Free Syntax

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

impdecls → $\mathit{impdecl}_1$ `;` … `;` $\mathit{impdecl}_n$ ($n \geq 1$)

exports → `(` export_1 `,` … `,` export_n [`,`] `)` ($n \geq 0$)

export → *qvar*
| *qtycon* [`(..)` | `(` cname_1 `,` … `,` cname_n `)`] ($n \geq 0$)
| *qtycls* [`(..)` | `(` qvar_1 `,` … `,` qvar_n `)`] ($n \geq 0$)
| `module` *modid*

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

impspec → `(` import_1 `,` … `,` import_n [`,`] `)` ($n \geq 0$)
| `hiding` `(` import_1 `,` … `,` import_n [`,`] `)` ($n \geq 0$)

import → *var*
| *tycon* [`(..)` | `(` cname_1 `,` … `,` cname_n `)`] ($n \geq 0$)
| *tycls* [`(..)` | `(` var_1 `,` … `,` var_n `)`] ($n \geq 0$)
cname → *var* | *con*

topdecls → $\mathit{topdecl}_1$ `;` … `;` $\mathit{topdecl}_n$ ($n \geq 0$)
topdecl → `type` *simpletype* `=` *type*
| `data` [*context* `=>`] *simpletype* [`=` *constrs*] [*deriving*]
| `newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]
| `class` [*scontext* `=>`] *tycls* *tyvar* [`where` *cdecls*]
| `instance` [*scontext* `=>`] *qtycls* *inst* [`where` *idecls*]
| `default` `(`type_1 `,` … `,` type_n`)` ($n \geq 0$)
| `foreign` *fdecl*
| *decl*

decls → `{` decl_1 `;` … `;` decl_n `}` ($n \geq 0$)
decl → *gendecl*
| (*funlhs* | *pat*) *rhs*

cdecls → `{` cdecl_1 `;` … `;` cdecl_n `}` ($n \geq 0$)
cdecl → *gendecl*
| (*funlhs* | *var*) *rhs*

idecls → `{` idecl_1 `;` … `;` idecl_n `}` ($n \geq 0$)
idecl → (*funlhs* | *var*) *rhs*
| (empty)

gendecl	→	*vars* `::` [*context* `=>`] *type*	(type signature)
	$\mid$	*fixity* [*integer*] *ops*	(fixity declaration)
	$\mid$		(empty declaration)

ops	→	op_1 `,` … `,` op_n	($n \geq 1$)
vars	→	var_1 `,` …`,` var_n	($n \geq 1$)
fixity	→	`infixl` $\mid$ `infixr` $\mid$ `infix`	

type	→	*btype* [`->` *type*]	(function type)

btype	→	[*btype*] *atype*	(type application)

atype	→	*gtycon*	
	$\mid$	*tyvar*	
	$\mid$	`(` $type_1$ `,` … `,` $type_k$ `)`	(tuple type, $k \geq 2$)
	$\mid$	`[` *type* `]`	(list type)
	$\mid$	`(` *type* `)`	(parenthesized constructor)

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

context	→	*class*	
	$\mid$	`(` $class_1$ `,` … `,` $class_n$ `)`	($n \geq 0$)
class	→	*qtycls tyvar*	
	$\mid$	*qtycls* `(` *tyvar* $atype_1$ … $atype_n$ `)`	($n \geq 1$)
scontext	→	*simpleclass*	
	$\mid$	`(` $simpleclass_1$ `,` … `,` $simpleclass_n$ `)`	($n \geq 0$)
simpleclass	→	*qtycls tyvar*	

simpletype	→	*tycon* $tyvar_1$ … $tyvar_k$	($k \geq 0$)
constrs	→	$constr_1$ `|` … `|` $constr_n$	($n \geq 1$)
constr	→	*con* [`!`] $atype_1$ … [`!`] $atype_k$	(arity *con* $= k$, $k \geq 0$)
	$\mid$	(*btype* $\mid$ `!` *atype*) *conop* (*btype* $\mid$ `!` *atype*)	(infix *conop*)
	$\mid$	*con* `{` $fielddecl_1$ `,` … `,` $fielddecl_n$ `}`	($n \geq 0$)
newconstr	→	*con atype*	
	$\mid$	*con* `{` *var* `::` *type* `}`	
fielddecl	→	*vars* `::` (*type* $\mid$ `!` *atype*)	
deriving	→	`deriving` (*dclass* $\mid$ `(`$dclass_1$`,` … `,` $dclass_n$`)`)	($n \geq 0$)
dclass	→	*qtycls*	

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

fdecl	→	`import` *callconv* [*safety*] *impent var* `::` *ftype*	(define variable)
	$\mid$	`export` *callconv expent var* `::` *ftype*	(expose variable)

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

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

varop → *varsym* | \` *varid* \` (variable operator)
qvarop → *qvarsym* | \` *qvarid* \` (qualified variable operator)
conop → *consym* | \` *conid* \` (constructor operator)
qconop → *gconsym* | \` *qconid* \` (qualified constructor operator)
op → *varop* | *conop* (operator)
qop → *qvarop* | *qconop* (qualified operator)
gconsym → **:** | *qconsym*

10.6 Fixity Resolution

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

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

```
import Control.Monad

type Prec   = Int
type Var    = String

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

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

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

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

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

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

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

       -- case (3): op1 and op2 should associate to the right
```

```
            | otherwise
            = do (r,rest') <- parseNeg op2 rest
                 parse op1 (OpApp e1 op2 r) rest'
            where
              Op _ prec1 fix1 = op1
              Op _ prec2 fix2 = op2
```

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

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

which means that we are looking at an expression like

```
E0 `op1` E1 `op2` ...     (1)
```

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

There are three cases to consider:

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

   ```
   E0 `op1` (E1 `op2` R) `op3` ...
   ```

   where `op3` is the next operator in the input. This is an instance of (1) above, so to continue we call parse, with the new ``E1 == (E1 `op2` R)``.

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

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

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

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

Chapter 11

Specification of Derived Instances

A *derived instance* is an instance declaration that is generated automatically in conjunction with a `data` or `newtype` declaration. The body of a derived instance declaration is derived syntactically 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 chapter, we describe the derivation of classes defined by the Prelude.

If T is an algebraic datatype declared by:

$$\begin{array}{lcl} \texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \ldots\ u_k & = & K_1\ t_{11}\ \ldots\ t_{1k_1}\ \texttt{|}\ \cdots\ \texttt{|}\ K_n\ t_{n1}\ \ldots\ t_{nk_n} \\ & & \texttt{deriving (}C_1\texttt{,}\ \ldots\texttt{,}\ C_m\texttt{)} \end{array}$$

(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 cx' such that $cx' \Rightarrow C\ t_{ij}$ holds for each of the constituent types t_{ij}.
3. If C is `Bounded`, the type must be either an enumeration (all constructors must be 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 that makes $T\ u_1\ \ldots\ u_k$ an instance of C.
6. If the data declaration has no constructors (i.e. when $n = 0$), then no classes are derivable (i.e. $m = 0$)

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:

`instance` $($`cx`$,\ cx')$ `=>` $C_i\ (T\ u_1\ \ldots\ u_k)$ `where {` d `}`

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).

The context cx' is the smallest context satisfying point (2) above. For mutually recusive data types, the compiler may need to perform a fixpoint calculation to compute it.

The remaining details of the derived instances for each of the derivable Prelude classes are now given. Free variables and constructors used in these translations always refer to entities defined by the `Prelude`.

11.1 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) ⇒    ⊥
```

All derived operations of class `Eq` and `Ord` are strict in both arguments. For example, `False <=` $\perp$ is $\perp$, even though `False` is the first constructor of the `Bool` type.

11.2 Derived instances of `Enum`

Derived instance declarations for the class `Enum` are only possible for enumerations (data types with only nullary constructors).

The nullary constructors are assumed to be numbered left-to-right with the indices 0 through $n-1$. The `succ` and `pred` operators give the successor and predecessor respectively of a value, under this numbering scheme. It is an error to apply `succ` to the maximum element, or `pred` to the minimum element.

The `toEnum` and `fromEnum` operators map enumerated values to and from the `Int` type; `toEnum` raises a runtime error if the `Int` argument is not the index of one of the constructors.

The definitions of the remaining methods are

```
enumFrom x           = enumFromTo x lastCon
enumFromThen x y     = enumFromThenTo x y bound
                       where
                         bound | fromEnum y >= fromEnum x = lastCon
                               | otherwise                = firstCon
enumFromTo x y       = map toEnum [fromEnum x .. fromEnum y]
enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

where `firstCon` and `lastCon` are respectively the first and last constructors listed in the `data` declaration. For example, given the datatype:

```
data  Color = Red | Orange | Yellow | Green  deriving (Enum)
```

we would have:

```
[Orange ..]         ==  [Orange, Yellow, Green]
fromEnum Yellow     ==  2
```

11.3 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
```

11.4 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 `11`), 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 in `x` is less than `d`. Thus, if `d` is `0` then the result is never surrounded in parentheses; if `d` is `11` it is always surrounded in parentheses, unless it is an atomic expression (recall that function application has precedence `10`). 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 attempts to parse a value from the front of the string, returning a list of (parsed value, remaining string) pairs. If there is no successful parse, the returned list is empty. Parsing of an un-parenthesised infix operator application succeeds only if the precedence of the operator is greater than or equal to `d`.

It should be the case that

`(x,"")` is an element of `(readsPrec d (showsPrec d x ""))`

That is, `readsPrec` should be able to parse the string produced by `showsPrec`, and should deliver the value that `showsPrec` started with.

`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 strings, for which only quoted strings are accepted, and other lists, for which only the bracketed form `[...]` is accepted. See Chapter 9 for full details.

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. It contains only the constructor names defined in the data type, parentheses, and spaces. When labelled constructor fields are used, braces, commas, field names, and equal signs are also used. Parentheses are only added where needed, *ignoring associativity*. No line breaks are added. 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.)

Derived instances of `Read` make the following assumptions, which derived instances of `Show` obey:

- If the constructor is defined to be an infix operator, then the derived `Read` instance will parse only infix applications of the constructor (not the prefix form).
- Associativity is not used to reduce the occurrence of parentheses, although precedence may be. For example, given

  ```
  infixr 4 :$
  data T = Int :$ T  |  NT
  ```

  then:

  - `show (1 :$ 2 :$ NT)` produces the string `"1 :$ (2 :$ NT)"`.
  - `read "1 :$ (2 :$ NT)"` succeeds, with the obvious result.
  - `read "1 :$ 2 :$ NT"` fails.
- If the constructor is defined using record syntax, the derived `Read` will parse only the record-syntax form, and furthermore, the fields must be given in the same order as the original declaration.
- The derived `Read` instance allows arbitrary Haskell whitespace between tokens of the input string. Extra parentheses are also allowed.

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 than 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.

11.5 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 11.1, 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 6.1.

```
infixr 5 :^:
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 > app_prec) showStr
          where
             showStr = showString "Leaf " . showsPrec (app_prec+1) m

        showsPrec d (u :^: v) = showParen (d > up_prec) showStr
          where
             showStr = showsPrec (up_prec+1) u .
                       showString " :^: "      .
                       showsPrec (up_prec+1) v
                -- Note: right-associativity of :^: ignored

instance (Read a) => Read (Tree a) where

        readsPrec d r =  readParen (d > up_prec)
                         (\r -> [(u:^:v,w) |
                                 (u,s) <- readsPrec (up_prec+1) r,
                                 (":^:",t) <- lex s,
                                 (v,w) <- readsPrec (up_prec+1) t]) r

                      ++ readParen (d > app_prec)
                         (\r -> [(Leaf m,t) |
                                 ("Leaf",s) <- lex r,
                                 (m,t) <- readsPrec (app_prec+1) s]) r

up_prec  = 5    -- Precedence of :^:
app_prec = 10   -- Application has precedence one more than
                -- the most tightly-binding operator
```

Figure 11.1: Example of Derived Instances

Chapter 12

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 chapter summarizes this existing practice. An implementation is not required to respect any pragma, although pragmas that are not recognised by the implementation should be ignored. Implementations are strongly encouraged to support the LANGUAGE pragma described below as there are many language extensions being used in practice.

Lexically, pragmas appear as comments, except that the enclosing syntax is `{-# #-}`.

12.1 Inlining

decl → `{-# INLINE` *qvars* `#-}`
decl → `{-# NOINLINE` *qvars* `#-}`

The `INLINE` pragma instructs the compiler to inline the specified variables at their use sites. Compilers will often automatically inline simple expressions. This may be prevented by the `NOINLINE` pragma.

12.2 Specialization

decl → `{-# SPECIALIZE` $spec_1$ `,` ... `,` $spec_k$ `#-}` $(k \geq 1)$
spec → *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 involve overloaded numeric operations.

12.3 Language extensions

The `LANGUAGE` pragma is a file-header pragma. A file-header pragma must precede the module keyword in a source file. There can be as many file-header pragmas as you please, and they can be preceded or followed by comments. An individual language pragma begins with the keyword `LANGUAGE` and is followed by a comma-separated list of named language features.

For example, to enable scoped type variables and preprocessing with CPP, if your Haskell implementation supports these extensions:

```
{-# LANGUAGE ScopedTypeVariables, CPP #-}
```

If a Haskell implementation does not recognize or support a particular language feature that a source file requests (or cannot support the combination of language features requested), any attempt to compile or otherwise use that file with that Haskell implementation must fail with an error.

In the interests of portability, multiple attempts to enable the same, supported language features (e.g. via command-line arguments, implementation-specific features dependencies or non-standard pragmas) are specifically permitted. Haskell 2010 implementations that support the `LANGUAGE` pragma are required to support

```
{-# LANGUAGE Haskell2010 #-}
```

Those implementations are also encouraged to support the following named language features:

```
PatternGuards, NoNPlusKPatterns, RelaxedPolyRec,
EmptyDataDecls, ForeignFunctionInterface
```

These are the named language extensions supported by some pre-Haskell 2010 implementations, that have been integrated into this report.

Part II

The Haskell 2010 Libraries

Chapter 13

`Control.Monad`

```
module Control.Monad (
    Functor(fmap),  Monad((>>=), (>>), return, fail),  MonadPlus(mzero, mplus),
    mapM,  mapM_,  forM,  forM_,  sequence,  sequence_,  (=<<),  (>=>),  (<=<),
    forever,  void,  join,  msum,  filterM,  mapAndUnzipM,  zipWithM,
    zipWithM_,  foldM,  foldM_,  replicateM,  replicateM_,  guard,  when,
    unless,  liftM,  liftM2,  liftM3,  liftM4,  liftM5,  ap
  ) where
```

The `Control.Monad` module provides the `Functor`, `Monad` and `MonadPlus` classes, together with some useful operations on monads.

13.1 Functor and monad classes

`class Functor f where`

The `Functor` class is used for types that can be mapped over. Instances of `Functor` should satisfy the following laws:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

The instances of `Functor` for lists, `Data.Maybe.Maybe` and `System.IO.IO` satisfy these laws.

Methods

`fmap :: (a -> b) -> f a -> f b`

`instance Functor []`
`instance Functor IO`
`instance Functor Maybe`
`instance Ix i => Functor (Array i)`

`class Monad m where`

The `Monad` class defines the basic operations over a *monad*, a concept 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* of actions. Haskell's `do` expressions provide a convenient syntax for writing monadic expressions.

Minimal complete definition: `>>=` and `return`.

Instances of `Monad` should satisfy the following laws:

```
return a >>= k  ==  k a
m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Instances of both `Monad` and `Functor` should additionally satisfy the law:

```
fmap f xs  ==  xs >>= return . f
```

The instances of `Monad` for lists, `Data.Maybe.Maybe` and `System.IO.IO` defined in the `Prelude` satisfy these laws.

Methods

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

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

`(>>) :: m a -> m b -> m b`

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

`return :: a -> m a`

Inject a value into the monadic type.

`fail :: String -> m a`

Fail with a message. This operation is not part of the mathematical definition of a monad, but is invoked on pattern-match failure in a `do` expression.

```
instance Monad []
instance Monad IO
instance Monad Maybe
```

`class Monad m => MonadPlus m where`

Monads that also support choice and failure.

Methods

`mzero :: m a`

the identity of `mplus`. It should also satisfy the equations

```
mzero >>= f  =  mzero
v >> mzero   =  mzero
```

`mplus :: m a -> m a -> m a`

an associative operation

`instance MonadPlus []`
`instance MonadPlus Maybe`

13.2 Functions

13.2.1 Naming conventions

The functions in this library use the following naming conventions:

- A postfix '`M`' always stands for a function in the Kleisli category: The monad type constructor `m` is added to function results (modulo currying) and nowhere else. So, for example,

```
filter  ::              (a ->   Bool) -> [a] ->   [a]
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

- A postfix '_' changes the result type from `(m a)` to `(m ())`. Thus, for example:

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- A prefix '`m`' generalizes an existing function to a monadic form. Thus, for example:

```
sum  :: Num a       => [a]   -> a
msum :: MonadPlus m => [m a] -> m a
```

13.2.2 Basic `Monad` functions

`mapM :: Monad m => (a -> m b) -> [a] -> m [b]`

`mapM f` is equivalent to `sequence . map f`.

`mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`

`mapM_ f` is equivalent to `sequence_ . map f`.

`forM :: Monad m => [a] -> (a -> m b) -> m [b]`

`forM` is `mapM` with its arguments flipped

`forM_ :: Monad m => [a] -> (a -> m b) -> m ()`

`forM_` is `mapM_` with its arguments flipped

`sequence :: Monad m => [m a] -> m [a]`

Evaluate each action in the sequence from left to right, and collect the results.

`sequence_ :: Monad m => [m a] -> m ()`

Evaluate each action in the sequence from left to right, and ignore the results.

`(=<<) :: Monad m => (a -> m b) -> m a -> m b`

Same as `>>=`, but with the arguments interchanged.

`(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`

Left-to-right Kleisli composition of monads.

`(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c`

Right-to-left Kleisli composition of monads. `(>=>)`, with the arguments flipped

`forever :: Monad m => m a -> m b`

`forever act` repeats the action infinitely.

`void :: Functor f => f a -> f ()`

`void value` discards or ignores the result of evaluation, such as the return value of an `IO` action.

13.2.3 Generalisations of list functions

`join :: Monad m => m (m a) -> m a`

The `join` function is the conventional monad join operator. It is used to remove one level of monadic structure, projecting its bound argument into the outer level.

`msum :: MonadPlus m => [m a] -> m a`

This generalizes the list-based `concat` function.

`filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]`

This generalizes the list-based `filter` function.

`mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])`

The `mapAndUnzipM` function maps its first argument over a list, returning the result as a pair of lists. This function is mainly used with complicated data structures or a state-transforming monad.

`zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]`

The `zipWithM` function generalizes `zipWith` to arbitrary monads.

`zipWithM_ :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()`

`zipWithM_` is the extension of `zipWithM` which ignores the final result.

`foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a`

The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the 'folded function' are not commutative.

```
foldM f a1 [x1, x2, ..., xm]
```

==

```
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

If right-to-left evaluation is required, the input list should be reversed.

`foldM_ :: Monad m => (a -> b -> m a) -> a -> [b] -> m ()`

Like `foldM`, but discards the result.

`replicateM :: Monad m => Int -> m a -> m [a]`

`replicateM n act` performs the action `n` times, gathering the results.

`replicateM_ :: Monad m => Int -> m a -> m ()`

Like `replicateM`, but discards the result.

13.2.4 Conditional execution of monadic expressions

`guard :: MonadPlus m => Bool -> m ()`

`guard b` is `return ()` if `b` is `True`, and `mzero` if `b` is `False`.

`when :: Monad m => Bool -> m () -> m ()`

Conditional execution of monadic expressions. For example,

```
when debug (putStr "Debugging\n")
```

will output the string `Debugging\n` if the Boolean value `debug` is `True`, and otherwise do nothing.

`unless :: Monad m => Bool -> m () -> m ()`

The reverse of `when`.

13.2.5 Monadic lifting operators

`liftM :: Monad m => (a1 -> r) -> m a1 -> m r`

Promote a function to a monad.

`liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r`

Promote a function to a monad, scanning the monadic arguments from left to right. For example,

```
liftM2 (+) [0,1] [0,2] = [0,2,1,3]
liftM2 (+) (Just 1) Nothing = Nothing
```

```
liftM3 :: Monad m => (a1 -> a2 -> a3 -> r)
                  -> m a1 -> m a2 -> m a3 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right (cf. `liftM2`).

```
liftM4 :: Monad m => (a1 -> a2 -> a3 -> a4 -> r)
                  -> m a1 -> m a2 -> m a3 -> m a4 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right (cf. `liftM2`).

```
liftM5 :: Monad m => (a1 -> a2 -> a3 -> a4 -> a5 -> r)
                  -> m a1 -> m a2 -> m a3 -> m a4 -> m a5 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right (cf. `liftM2`).

`ap :: Monad m => m (a -> b) -> m a -> m b`

In many situations, the `liftM` operations can be replaced by uses of `ap`, which promotes function application.

```
return f `ap` x1 `ap` ... `ap` xn
```

is equivalent to

```
liftMn f x1 x2 ... xn
```

Chapter 14

Data.Array

```
module Data.Array (
    module Data.Ix,  Array,  array,  listArray,  accumArray,  (!),  bounds,
    indices,  elems,  assocs,  (//),  accum,  ixmap
  ) where
```

14.1 Immutable non-strict arrays

Haskell provides indexable *arrays*, which may be thought of as functions whose domains are isomorphic to contiguous subsets of the integers. Functions restricted in this way can be implemented efficiently; in particular, a programmer may reasonably expect rapid access to the components. To ensure the possibility of such an implementation, arrays are treated as data, not as general functions.

Since most array functions involve the class `Ix`, the contents of the module `Data.Ix` are re-exported from `Data.Array` for convenience:

`module Data.Ix`

`data Ix i => Array i e`

The type of immutable non-strict (boxed) arrays with indices in `i` and elements in `e`.

```
instance Ix i => Functor (Array i)
instance (Ix i, Eq e) => Eq (Array i e)
instance (Ix i, Ord e) => Ord (Array i e)
instance (Ix a, Read a, Read b) => Read (Array a b)
instance (Ix a, Show a, Show b) => Show (Array a b)
```

14.2 Array construction

`array`

`::`	`Ix i`	
`=>`	`(i, i)`	a pair of *bounds*, each of the index type of the array. These bounds are the lowest and highest indices in the array, in that order. For example, a one-origin vector of length '10' has bounds '(1,10)', and a one-origin '10' by '10' matrix has bounds '((1,1),(10,10))'.
`->`	`[(i, e)]`	a list of *associations* of the form (*index*, *value*). Typically, this list will be expressed as a comprehension. An association '(i, x)' defines the value of the array at index `i` to be `x`.
`->`	`Array i e`	

Construct an array with the specified bounds and containing values for given indices within these bounds.

The array is undefined (i.e. bottom) if any index in the list is out of bounds. If any two associations in the list have the same index, the value at that index is undefined (i.e. bottom).

Because the indices must be checked for these errors, `array` is strict in the bounds argument and in the indices of the association list, but non-strict in the values. Thus, recurrences such as the following are possible:

```
a = array (1,100) ((1,1) : [(i, i * a!(i-1)) | i <- [2..100]])
```

Not every index within the bounds of the array need appear in the association list, but the values associated with indices that do not appear will be undefined (i.e. bottom).

If, in any dimension, the lower bound is greater than the upper bound, then the array is legal, but empty. Indexing an empty array always gives an array-bounds error, but `bounds` still yields the bounds with which the array was constructed.

`listArray :: Ix i => (i, i) -> [e] -> Array i e`

Construct an array from a pair of bounds and a list of values in index order.

`accumArray`

`::`	`Ix i`	
`=>`	`(e -> a -> e)`	accumulating function
`->`	`e`	initial value
`->`	`(i, i)`	bounds of the array
`->`	`[(i, a)]`	association list
`->`	`Array i e`	

The `accumArray` function deals with repeated indices in the association list using an *accumulating function* which combines the values of associations with the same index. For example, given a list of values of some index type, `hist` produces a histogram of the number of occurrences of each index within a specified range:

```
hist :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | i<-is, inRange bnds i]
```

If the accumulating function is strict, then `accumArray` is strict in the values, as well as the indices, in the association list. Thus, unlike ordinary arrays built with `array`, accumulated arrays should not in general be recursive.

14.3 Accessing arrays

`(!) :: Ix i => Array i e -> i -> e`

The value at the given index in an array.

`bounds :: Ix i => Array i e -> (i, i)`

The bounds with which an array was constructed.

`indices :: Ix i => Array i e -> [i]`

The list of indices of an array in ascending order.

`elems :: Ix i => Array i e -> [e]`

The list of elements of an array in index order.

`assocs :: Ix i => Array i e -> [(i, e)]`

The list of associations of an array in index order.

14.4 Incremental array updates

`(//) :: Ix i => Array i e -> [(i, e)] -> Array i e`

Constructs an array identical to the first argument except that it has been updated by the associations in the right argument. For example, if `m` is a 1-origin, `n` by `n` matrix, then

```
m//[((i,i), 0) | i <- [1..n]]
```

is the same matrix, except with the diagonal zeroed.

Repeated indices in the association list are handled as for `array`: the resulting array is undefined (i.e. bottom),

```
accum :: Ix i => (e -> a -> e)
              -> Array i e -> [(i, a)] -> Array i e
```

`accum f` takes an array and an association list and accumulates pairs from the list into the array with the accumulating function `f`. Thus `accumArray` can be defined using `accum`:

```
accumArray f z b = accum f (array b [(i, z) | i <- range b])
```

14.5 Derived arrays

```
ixmap :: (Ix i, Ix j) => (i, i)
                      -> (i -> j) -> Array j e -> Array i e
```

`ixmap` allows for transformations on array indices. It may be thought of as providing function composition on the right with the mapping that the original array embodies.

A similar transformation of array values may be achieved using `fmap` from the `Array` instance of the `Functor` class.

14.6 Specification

```
module  Array (
    module Data.Ix,  -- export all of Data.Ix
    Array, array, listArray, (!), bounds, indices, elems, assocs,
    accumArray, (//), accum, ixmap ) where

import Data.Ix
import Data.List( (\\) )

infixl 9  !, //

data (Ix a) => Array a b = MkArray (a,a) (a -> b) deriving ()

array       :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
array b ivs
  | any (not . inRange b. fst) ivs
     = error "Data.Array.array: out-of-range array association"
  | otherwise
     = MkArray b arr
  where
    arr j = case [ v | (i,v) <- ivs, i == j ] of
              [v]   -> v
              []    -> error "Data.Array.!: undefined array element"
              _     -> error "Data.Array.!: multiply defined array element"

listArray             :: (Ix a) => (a,a) -> [b] -> Array a b
listArray b vs        =  array b (zipWith (\ a b -> (a,b)) (range b) vs)

(!)                   :: (Ix a) => Array a b -> a -> b
(!) (MkArray _ f)     =  f

bounds                :: (Ix a) => Array a b -> (a,a)
bounds (MkArray b _)  =  b

indices               :: (Ix a) => Array a b -> [a]
indices               =  range . bounds

elems                 :: (Ix a) => Array a b -> [b]
elems a               =  [a!i | i <- indices a]

assocs                :: (Ix a) => Array a b -> [(a,b)]
assocs a              =  [(i, a!i) | i <- indices a]

(//)                  :: (Ix a) => Array a b -> [(a,b)] -> Array a b
a // new_ivs          = array (bounds a) (old_ivs ++ new_ivs)
                      where
                        old_ivs = [(i,a!i) | i <- indices a,
                                             i `notElem` new_is]
                        new_is  = [i | (i,_) <- new_ivs]

accum                 :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                                   -> Array a b
accum f               =  foldl (\a (i,v) -> a // [(i,f (a!i) v)])
```

```
accumArray            :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                                   -> Array a b
accumArray f z b      =  accum f (array b [(i,z) | i <- range b])

ixmap                 :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                                         -> Array a c
ixmap b f a           = array b [(i, a ! f i) | i <- range b]

instance  (Ix a)          => Functor (Array a) where
    fmap fn (MkArray b f) =  MkArray b (fn . f)

instance  (Ix a, Eq b)  => Eq (Array a b)  where
    a == a' =  assocs a == assocs a'

instance  (Ix a, Ord b) => Ord (Array a b)  where
    a <= a' =  assocs a <= assocs a'

instance  (Ix a, Show a, Show b) => Show (Array a b)  where
    showsPrec p a = showParen (p > arrPrec) (
                    showString "array " .
                    showsPrec (arrPrec+1) (bounds a) . showChar ' ' .
                    showsPrec (arrPrec+1) (assocs a)                  )

instance  (Ix a, Read a, Read b) => Read (Array a b)  where
    readsPrec p = readParen (p > arrPrec)
           (\r -> [ (array b as, u)
                  | ("array",s) <- lex r,
                    (b,t)       <- readsPrec (arrPrec+1) s,
                    (as,u)      <- readsPrec (arrPrec+1) t ])

-- Precedence of the 'array' function is that of application itself
arrPrec = 10
```

Chapter 15

Data.Bits

```
module Data.Bits (
    Bits((.&.),
         (.|.),
         xor,
         complement,
         shift,
         rotate,
         bit,
         setBit,
         clearBit,
         complementBit,
         testBit,
         bitSize,
         isSigned,
         shiftL,
         shiftR,
         rotateL,
         rotateR)
  ) where
```

This module defines bitwise operations for signed and unsigned integers.

`class Num a => Bits a where`

The `Bits` class defines bitwise operations over integral types.

- Bits are numbered from 0 with bit 0 being the least significant bit.

Minimal complete definition: `.&.`, `.|.`, `xor`, `complement`, (`shift` or (`shiftL` and `shiftR`)), (`rotate` or (`rotateL` and `rotateR`)), `bitSize` and `isSigned`.

Methods

`(.&.) :: a -> a -> a`

Bitwise ”and”

`(.|.) :: a -> a -> a`

Bitwise ”or”

`xor :: a -> a -> a`

Bitwise ”xor”

`complement :: a -> a`

Reverse all the bits in the argument

`shift :: a -> Int -> a`

`shift x i` shifts `x` left by `i` bits if `i` is positive, or right by `-i` bits otherwise. Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the `x` is negative and with 0 otherwise.

An instance can define either this unified `shift` or `shiftL` and `shiftR`, depending on which is more convenient for the type in question.

`rotate :: a -> Int -> a`

`rotate x i` rotates `x` left by `i` bits if `i` is positive, or right by `-i` bits otherwise.

For unbounded types like `Integer`, `rotate` is equivalent to `shift`.

An instance can define either this unified `rotate` or `rotateL` and `rotateR`, depending on which is more convenient for the type in question.

`bit :: Int -> a`

`bit i` is a value with the `i`th bit set and all other bits clear

`setBit :: a -> Int -> a`

`` x `setBit` i `` is the same as `x .|. bit i`

`clearBit :: a -> Int -> a`

`` x `clearBit` i `` is the same as `x .&. complement (bit i)`

`complementBit :: a -> Int -> a`

`` x `complementBit` i `` is the same as `` x `xor` bit i ``

`testBit :: a -> Int -> Bool`

Return `True` if the `n`th bit of the argument is 1

`bitSize :: a -> Int`

Return the number of bits in the type of the argument. The actual value of the argument is ignored. The function `bitSize` is undefined for types that do not have a fixed bitsize, like `Integer`.

`isSigned :: a -> Bool`

Return `True` if the argument is a signed type. The actual value of the argument is ignored

`shiftL :: a -> Int -> a`

Shift the argument left by the specified number of bits (which must be non-negative).

An instance can define either this and `shiftR` or the unified `shift`, depending on which is more convenient for the type in question.

`shiftR :: a -> Int -> a`

Shift the first argument right by the specified number of bits (which must be non-negative). Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the `x` is negative and with 0 otherwise.

An instance can define either this and `shiftL` or the unified `shift`, depending on which is more convenient for the type in question.

`rotateL :: a -> Int -> a`

Rotate the argument left by the specified number of bits (which must be non-negative).

An instance can define either this and `rotateR` or the unified `rotate`, depending on which is more convenient for the type in question.

`rotateR :: a -> Int -> a`

Rotate the argument right by the specified number of bits (which must be non-negative).

An instance can define either this and `rotateL` or the unified `rotate`, depending on which is more convenient for the type in question.

```
instance Bits Int
instance Bits Int8
instance Bits Int16
instance Bits Int32
instance Bits Int64
instance Bits Integer
instance Bits Word
instance Bits Word8
instance Bits Word16
instance Bits Word32
instance Bits Word64
instance Bits WordPtr
instance Bits IntPtr
instance Bits CChar
instance Bits CSChar
instance Bits CUChar
instance Bits CShort
instance Bits CUShort
instance Bits CInt
instance Bits CUInt
instance Bits CLong
instance Bits CULong
instance Bits CLLong
instance Bits CULLong
instance Bits CPtrdiff
instance Bits CSize
instance Bits CWchar
instance Bits CSigAtomic
instance Bits CIntPtr
instance Bits CUIntPtr
instance Bits CIntMax
instance Bits CUIntMax
```

Chapter 16

`Data.Char`

```
module Data.Char (
    Char,  String,  isControl,  isSpace,  isLower,  isUpper,  isAlpha,
    isAlphaNum,  isPrint,  isDigit,  isOctDigit,  isHexDigit,  isLetter,
    isMark,  isNumber,  isPunctuation,  isSymbol,  isSeparator,  isAscii,
    isLatin1,  isAsciiUpper,  isAsciiLower,
    GeneralCategory(UppercaseLetter,
                    LowercaseLetter,
                    TitlecaseLetter,
                    ModifierLetter,
                    OtherLetter,
                    NonSpacingMark,
                    SpacingCombiningMark,
                    EnclosingMark,
                    DecimalNumber,
                    LetterNumber,
                    OtherNumber,
                    ConnectorPunctuation,
                    DashPunctuation,
                    OpenPunctuation,
                    ClosePunctuation,
                    InitialQuote,
                    FinalQuote,
                    OtherPunctuation,
                    MathSymbol,
                    CurrencySymbol,
                    ModifierSymbol,
                    OtherSymbol,
                    Space,
                    LineSeparator,
                    ParagraphSeparator,
                    Control,
                    Format,
                    Surrogate,
                    PrivateUse,
                    NotAssigned),
```

```
    generalCategory, toUpper, toLower, toTitle, digitToInt, intToDigit,
    ord, chr, showLitChar, lexLitChar, readLitChar
  ) where
```

16.1 Characters and strings

`data Char`

The character type `Char` is an enumeration whose values represent Unicode (or equivalently ISO/IEC 10646) characters (see `http://www.unicode.org/` for details). This set extends the ISO 8859-1 (Latin-1) character set (the first 256 charachers), which is itself an extension of the ASCII character set (the first 128 characters). A character literal in Haskell has type `Char`.

To convert a `Char` to or from the corresponding `Int` value defined by Unicode, use `Prelude.toEnum` and `Prelude.fromEnum` from the `Prelude.Enum` class respectively (or equivalently `ord` and `chr`).

`instance Bounded Char`
`instance Enum Char`
`instance Eq Char`
`instance Ord Char`
`instance Read Char`
`instance Show Char`
`instance Ix Char`
`instance Storable Char`

`type String = [Char]`

A `String` is a list of characters. String constants in Haskell are values of type `String`.

16.2 Character classification

Unicode characters are divided into letters, numbers, marks, punctuation, symbols, separators (including spaces) and others (including control characters).

`isControl :: Char -> Bool`

Selects control characters, which are the non-printing characters of the Latin-1 subset of Unicode.

`isSpace :: Char -> Bool`

Returns `True` for any Unicode space character, and the control characters `\t`, `\n`, `\r`, `\f`, `\v`.

`isLower :: Char -> Bool`

Selects lower-case alphabetic Unicode characters (letters).

`isUpper :: Char -> Bool`

Selects upper-case or title-case alphabetic Unicode characters (letters). Title case is used by a small number of letter ligatures like the single-character form of *Lj*.

`isAlpha :: Char -> Bool`

Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters). This function is equivalent to `Data.Char.isLetter`.

`isAlphaNum :: Char -> Bool`

Selects alphabetic or numeric digit Unicode characters.

Note that numeric digits outside the ASCII range are selected by this function but not by `isDigit`. Such digits may be part of identifiers but are not used by the printer and reader to represent numbers.

`isPrint :: Char -> Bool`

Selects printable Unicode characters (letters, numbers, marks, punctuation, symbols and spaces).

`isDigit :: Char -> Bool`

Selects ASCII digits, i.e. `'0'`..`'9'`.

`isOctDigit :: Char -> Bool`

Selects ASCII octal digits, i.e. `'0'`..`'7'`.

`isHexDigit :: Char -> Bool`

Selects ASCII hexadecimal digits, i.e. `'0'`..`'9'`, `'a'`..`'f'`, `'A'`..`'F'`.

`isLetter :: Char -> Bool`

Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters). This function is equivalent to `Data.Char.isAlpha`.

`isMark :: Char -> Bool`

Selects Unicode mark characters, e.g. accents and the like, which combine with preceding letters.

`isNumber :: Char -> Bool`

Selects Unicode numeric characters, including digits from various scripts, Roman numerals, etc.

`isPunctuation :: Char -> Bool`

Selects Unicode punctuation characters, including various kinds of connectors, brackets and quotes.

`isSymbol :: Char -> Bool`

Selects Unicode symbol characters, including mathematical and currency symbols.

`isSeparator :: Char -> Bool`

Selects Unicode space and separator characters.

16.2.1 Subranges

`isAscii :: Char -> Bool`

Selects the first 128 characters of the Unicode character set, corresponding to the ASCII character set.

`isLatin1 :: Char -> Bool`

Selects the first 256 characters of the Unicode character set, corresponding to the ISO 8859-1 (Latin-1) character set.

`isAsciiUpper :: Char -> Bool`

Selects ASCII upper-case letters, i.e. characters satisfying both `isAscii` and `isUpper`.

`isAsciiLower :: Char -> Bool`

Selects ASCII lower-case letters, i.e. characters satisfying both `isAscii` and `isLower`.

16.2.2 Unicode general categories

```
data GeneralCategory
    =  UppercaseLetter        Lu: Letter, Uppercase
    |  LowercaseLetter        Ll: Letter, Lowercase
    |  TitlecaseLetter        Lt: Letter, Titlecase
    |  ModifierLetter         Lm: Letter, Modifier
    |  OtherLetter            Lo: Letter, Other
    |  NonSpacingMark         Mn: Mark, Non-Spacing
    |  SpacingCombiningMark   Mc: Mark, Spacing Combining
    |  EnclosingMark          Me: Mark, Enclosing
    |  DecimalNumber          Nd: Number, Decimal
    |  LetterNumber           Nl: Number, Letter
    |  OtherNumber            No: Number, Other
    |  ConnectorPunctuation   Pc: Punctuation, Connector
    |  DashPunctuation        Pd: Punctuation, Dash
    |  OpenPunctuation        Ps: Punctuation, Open
    |  ClosePunctuation       Pe: Punctuation, Close
    |  InitialQuote           Pi: Punctuation, Initial quote
    |  FinalQuote             Pf: Punctuation, Final quote
    |  OtherPunctuation       Po: Punctuation, Other
    |  MathSymbol             Sm: Symbol, Math
    |  CurrencySymbol         Sc: Symbol, Currency
    |  ModifierSymbol         Sk: Symbol, Modifier
    |  OtherSymbol            So: Symbol, Other
    |  Space                  Zs: Separator, Space
    |  LineSeparator          Zl: Separator, Line
    |  ParagraphSeparator     Zp: Separator, Paragraph
    |  Control                Cc: Other, Control
    |  Format                 Cf: Other, Format
    |  Surrogate              Cs: Other, Surrogate
    |  PrivateUse             Co: Other, Private Use
    |  NotAssigned            Cn: Other, Not Assigned
```

Unicode General Categories (column 2 of the UnicodeData table) in the order they are listed in the Unicode standard.

```
instance Bounded GeneralCategory
instance Enum GeneralCategory
instance Eq GeneralCategory
instance Ord GeneralCategory
instance Read GeneralCategory
instance Show GeneralCategory
instance Ix GeneralCategory
```

`generalCategory :: Char -> GeneralCategory`

The Unicode general category of the character.

16.3 Case conversion

`toUpper :: Char -> Char`

Convert a letter to the corresponding upper-case letter, if any. Any other character is returned unchanged.

`toLower :: Char -> Char`

Convert a letter to the corresponding lower-case letter, if any. Any other character is returned unchanged.

`toTitle :: Char -> Char`

Convert a letter to the corresponding title-case or upper-case letter, if any. (Title case differs from upper case only for a small number of ligature letters.) Any other character is returned unchanged.

16.4 Single digit characters

`digitToInt :: Char -> Int`

Convert a single digit `Char` to the corresponding `Int`. This function fails unless its argument satisfies `isHexDigit`, but recognises both upper and lower-case hexadecimal digits (i.e. `'0'`..`'9'`, `'a'`..`'f'`, `'A'`..`'F'`).

`intToDigit :: Int -> Char`

Convert an `Int` in the range `0..15` to the corresponding single digit `Char`. This function fails on other inputs, and generates lower-case hexadecimal digits.

16.5 Numeric representations

`ord :: Char -> Int`

The `Prelude.fromEnum` method restricted to the type `Data.Char.Char`.

`chr :: Int -> Char`

The `Prelude.toEnum` method restricted to the type `Data.Char.Char`.

16.6 String representations

`showLitChar :: Char -> ShowS`

Convert a character to a string using only printable characters, using Haskell source-language escape conventions. For example:

```
showLitChar '\n' s  =  "\\n" ++ s
```

`lexLitChar :: ReadS String`

Read a string representation of a character, using Haskell source-language escape conventions. For example:

```
lexLitChar  "\\nHello"  =  [("\\n", "Hello")]
```

`readLitChar :: ReadS Char`

Read a string representation of a character, using Haskell source-language escape conventions, and convert it to the character that it encodes. For example:

```
readLitChar "\\nHello"  =  [('\n', "Hello")]
```

Chapter 17

`Data.Complex`

```
module Data.Complex (
    Complex(:+),  realPart,  imagPart,  mkPolar,  cis,  polar,  magnitude,
    phase,  conjugate
  ) where
```

17.1 Rectangular form

`data RealFloat a => Complex a`

`= !a :+ !a` forms a complex number from its real and imaginary rectangular components.

Complex numbers are an algebraic type.

For a complex number `z`, `abs z` is a number with the magnitude of `z`, but oriented in the positive real direction, whereas `signum z` has the phase of `z`, but unit magnitude.

`instance RealFloat a => Eq (Complex a)`
`instance RealFloat a => Floating (Complex a)`
`instance RealFloat a => Fractional (Complex a)`
`instance RealFloat a => Num (Complex a)`
`instance (Read a, RealFloat a) => Read (Complex a)`
`instance RealFloat a => Show (Complex a)`

`realPart :: RealFloat a => Complex a -> a`

Extracts the real part of a complex number.

`imagPart :: RealFloat a => Complex a -> a`

Extracts the imaginary part of a complex number.

17.2 Polar form

`mkPolar :: RealFloat a => a -> a -> Complex a`

Form a complex number from polar components of magnitude and phase.

`cis :: RealFloat a => a -> Complex a`

`cis t` is a complex value with magnitude `1` and phase `t` (modulo `2*pi`).

`polar :: RealFloat a => Complex a -> (a, a)`

The function `polar` takes a complex number and returns a (magnitude, phase) pair in canonical form: the magnitude is nonnegative, and the phase in the range `(-pi, pi]`; if the magnitude is zero, then so is the phase.

`magnitude :: RealFloat a => Complex a -> a`

The nonnegative magnitude of a complex number.

`phase :: RealFloat a => Complex a -> a`

The phase of a complex number, in the range `(-pi, pi]`. If the magnitude is zero, then so is the phase.

17.3 Conjugate

`conjugate :: RealFloat a => Complex a -> Complex a`

The conjugate of a complex number.

17.4 Specification

```
module Data.Complex(Complex((:+)), realPart, imagPart, conjugate, mkPolar,
                    cis, polar, magnitude, phase)  where

infix  6  :+

data  (RealFloat a)     => Complex a = !a :+ !a  deriving (Eq,Read,Show)


realPart, imagPart :: (RealFloat a) => Complex a -> a
realPart (x:+y)       =  x
imagPart (x:+y)       =  y

conjugate      :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y) =  x :+ (-y)

mkPolar                :: (RealFloat a) => a -> a -> Complex a
mkPolar r theta        =  r * cos theta :+ r * sin theta
```

```
cis            :: (RealFloat a) => a -> Complex a
cis theta      =  cos theta :+ sin theta

polar          :: (RealFloat a) => Complex a -> (a,a)
polar z                =  (magnitude z, phase z)

magnitude :: (RealFloat a) => Complex a -> a
magnitude (x:+y) =  scaleFloat k
                   (sqrt ((scaleFloat mk x)^2 + (scaleFloat mk y)^2))
                  where k  = max (exponent x) (exponent y)
                        mk = - k

phase :: (RealFloat a) => Complex a -> a
phase (0 :+ 0) = 0
phase (x :+ y) = atan2 y x


instance  (RealFloat a) => Num (Complex a)  where
    (x:+y) + (x':+y') =  (x+x') :+ (y+y')
    (x:+y) - (x':+y') =  (x-x') :+ (y-y')
    (x:+y) * (x':+y') =  (x*x'-y*y') :+ (x*y'+y*x')
    negate (x:+y)     =  negate x :+ negate y
    abs z             =  magnitude z :+ 0
    signum 0          =  0
    signum z@(x:+y)   =  x/r :+ y/r  where r = magnitude z
    fromInteger n     =  fromInteger n :+ 0

instance  (RealFloat a) => Fractional (Complex a)  where
    (x:+y) / (x':+y') =  (x*x''+y*y'') / d :+ (y*x''-x*y'') / d
                         where x'' = scaleFloat k x'
                               y'' = scaleFloat k y'
                               k   = - max (exponent x') (exponent y')
                               d   = x'*x'' + y'*y''

    fromRational a    =  fromRational a :+ 0

instance  (RealFloat a) => Floating (Complex a)       where
    pi             =  pi :+ 0
    exp (x:+y)     =  expx * cos y :+ expx * sin y
                      where expx = exp x
    log z          =  log (magnitude z) :+ phase z

    sqrt 0         =  0
    sqrt z@(x:+y)  =  u :+ (if y < 0 then -v else v)
                      where (u,v) = if x < 0 then (v',u') else (u',v')
                            v'    = abs y / (u'*2)
                            u'    = sqrt ((magnitude z + abs x) / 2)

    sin (x:+y)     =  sin x * cosh y :+ cos x * sinh y
    cos (x:+y)     =  cos x * cosh y :+ (- sin x * sinh y)
    tan (x:+y)     =  (sinx*coshy:+cosx*sinhy)/(cosx*coshy:+(-sinx*sinhy))
                      where sinx  = sin x
                            cosx  = cos x
                            sinhy = sinh y
```

```
                        coshy = cosh y

sinh (x:+y)    =  cos y * sinh x :+ sin  y * cosh x
cosh (x:+y)    =  cos y * cosh x :+ sin y * sinh x
tanh (x:+y)    =  (cosy*sinhx:+siny*coshx)/(cosy*coshx:+siny*sinhx)
                  where siny  = sin y
                        cosy  = cos y
                        sinhx = sinh x
                        coshx = cosh x

asin z@(x:+y)  =  y':+(-x')
                  where  (x':+y') = log (((-y):+x) + sqrt (1 - z*z))
acos z@(x:+y)  =  y'':+(-x'')
                  where (x'':+y'') = log (z + ((-y'):+x'))
                        (x':+y')   = sqrt (1 - z*z)
atan z@(x:+y)  =  y':+(-x')
                  where (x':+y') = log (((1-y):+x) / sqrt (1+z*z))

asinh z        =  log (z + sqrt (1+z*z))
acosh z        =  log (z + (z+1) * sqrt ((z-1)/(z+1)))
atanh z        =  log ((1+z) / sqrt (1-z*z))
```

Chapter 18

`Data.Int`

```
module Data.Int (
    Int,  Int8,  Int16,  Int32,  Int64
  ) where
```

18.1 Signed integer types

This module provides signed integer types of unspecified width (`Int`) and fixed widths (`Int8`, `Int16`, `Int32` and `Int64`). All arithmetic is performed modulo 2^n, where `n` is the number of bits in the type.

For coercing between any two integer types, use `fromIntegral`. Coercing word types (see `Data.Word`) to and from integer types preserves representation, not sign.

The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell language report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Int` types defined here.

Right and left shifts by amounts greater than or equal to the width of the type result in either zero or -1, depending on the sign of the value being shifted. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example `1 << 32 == 1` in some C implementations.

`data Int`

A fixed-precision integer type with at least the range `[-2^29 .. 2^29-1]`. The exact range for a given implementation can be determined by using `Prelude.minBound` and `Prelude.maxBound` from the `Prelude.Bounded` class.

```
instance Bounded Int
instance Enum Int
instance Eq Int
instance Integral Int
instance Num Int
instance Ord Int
instance Read Int
instance Real Int
instance Show Int
instance Ix Int
instance Storable Int
instance Bits Int
```

data Int8

8-bit signed integer type

```
instance Bounded Int8
instance Enum Int8
instance Eq Int8
instance Integral Int8
instance Num Int8
instance Ord Int8
instance Read Int8
instance Real Int8
instance Show Int8
instance Ix Int8
instance Storable Int8
instance Bits Int8
```

data Int16

16-bit signed integer type

```
instance Bounded Int16
instance Enum Int16
instance Eq Int16
instance Integral Int16
instance Num Int16
instance Ord Int16
instance Read Int16
instance Real Int16
instance Show Int16
instance Ix Int16
instance Storable Int16
instance Bits Int16
```

data Int32

32-bit signed integer type

```
instance Bounded Int32
instance Enum Int32
instance Eq Int32
instance Integral Int32
instance Num Int32
instance Ord Int32
instance Read Int32
instance Real Int32
instance Show Int32
instance Ix Int32
instance Storable Int32
instance Bits Int32
```

`data Int64`

64-bit signed integer type

```
instance Bounded Int64
instance Enum Int64
instance Eq Int64
instance Integral Int64
instance Num Int64
instance Ord Int64
instance Read Int64
instance Real Int64
instance Show Int64
instance Ix Int64
instance Storable Int64
instance Bits Int64
```

Chapter 19

`Data.Ix`

```
module Data.Ix (
    Ix(range, index, inRange, rangeSize)
  ) where
```

19.1 The `Ix` class

`class Ord a => Ix a where`

The `Ix` class is used to map a contiguous subrange of values in a type onto integers. It is used primarily for array indexing (see the array package).

The first argument `(l,u)` of each of these operations is a pair specifying the lower and upper bounds of a contiguous subrange of values.

An implementation is entitled to assume the following laws about these operations:

- `inRange (l,u) i == elem i (range (l,u))`
- `range (l,u) !! index (l,u) i == i`, when `inRange (l,u) i`
- `map (index (l,u)) (range (l,u))) == [0..rangeSize (l,u)-1]`
- `rangeSize (l,u) == length (range (l,u))`

Minimal complete instance: `range`, `index` and `inRange`.

Methods

`range :: (a, a) -> [a]`

The list of values in the subrange defined by a bounding pair.

`index :: (a, a) -> a -> Int`

The position of a subscript in the subrange.

`inRange :: (a, a) -> a -> Bool`

Returns `True` the given subscript lies in the range defined the bounding pair.

`rangeSize :: (a, a) -> Int`

The size of the subrange defined by a bounding pair.

```
instance Ix Bool
instance Ix Char
instance Ix Int
instance Ix Int8
instance Ix Int16
instance Ix Int32
instance Ix Int64
instance Ix Integer
instance Ix Ordering
instance Ix Word
instance Ix Word8
instance Ix Word16
instance Ix Word32
instance Ix Word64
instance Ix ()
instance Ix GeneralCategory
instance Ix SeekMode
instance Ix IOMode
instance (Ix a, Ix b) => Ix (a, b)
instance (Ix a1, Ix a2, Ix a3) => Ix (a1, a2, a3)
instance (Ix a1, Ix a2, Ix a3, Ix a4) => Ix (a1, a2, a3, a4)
instance (Ix a1, Ix a2, Ix a3, Ix a4, Ix a5) => Ix (a1, a2, a3, a4, a5)
```

19.2 Deriving Instances of `Ix`

It is possible to derive an instance of `Ix` automatically, using a `deriving` clause on a `data` declaration. Such derived instance declarations for the class `Ix` are only possible for enumerations (i.e. datatypes having only nullary constructors) and single-constructor datatypes, whose constituent types are instances of `Ix`. A Haskell implementation must provide `Ix` instances for tuples up to at least size 15.

For an *enumeration*, the nullary constructors are assumed to be numbered left-to-right with the indices being 0 to n-1 inclusive. This is the same numbering defined by the `Enum` class. For example, given the datatype:

```
data Colour = Red | Orange | Yellow | Green | Blue | Indigo | Violet
```

we would have:

```
range   (Yellow,Blue)        ==  [Yellow,Green,Blue]
index   (Yellow,Blue) Green  ==  1
inRange (Yellow,Blue) Red    ==  False
```

For *single-constructor datatypes*, the derived instance declarations are as shown for tuples:

```
instance  (Ix a, Ix b)  => Ix (a,b) where
        range ((l,l'),(u,u'))
                = [(i,i') | i <- range (l,u), i' <- range (l',u')]
        index ((l,l'),(u,u')) (i,i')
                =  index (l,u) i * rangeSize (l',u') + index (l',u') i'
        inRange ((l,l'),(u,u')) (i,i')
                = inRange (l,u) i && inRange (l',u') i'

-- Instances for other tuples are obtained from this scheme:
--
--  instance  (Ix a1, Ix a2, ... , Ix ak) => Ix (a1,a2,...,ak)  where
--      range ((l1,l2,...,lk),(u1,u2,...,uk)) =
--          [(i1,i2,...,ik) | i1 <- range (l1,u1),
--                            i2 <- range (l2,u2),
--                            ...
--                            ik <- range (lk,uk)]
--
--      index ((l1,l2,...,lk),(u1,u2,...,uk)) (i1,i2,...,ik) =
--        index (lk,uk) ik + rangeSize (lk,uk) * (
--         index (lk-1,uk-1) ik-1 + rangeSize (lk-1,uk-1) * (
--          ...
--           index (l1,u1)))
--
--      inRange ((l1,l2,...lk),(u1,u2,...,uk)) (i1,i2,...,ik) =
--          inRange (l1,u1) i1 && inRange (l2,u2) i2 &&
--              ... && inRange (lk,uk) ik
```

Chapter 20

Data.List

```
module Data.List (
    (++), head, last, tail, init, null, length, map, reverse,
    intersperse, intercalate, transpose, subsequences, permutations,
    foldl, foldl', foldl1, foldl1', foldr, foldr1, concat, concatMap,
    and, or, any, all, sum, product, maximum, minimum, scanl, scanl1,
    scanr, scanr1, mapAccumL, mapAccumR, iterate, repeat, replicate,
    cycle, unfoldr, take, drop, splitAt, takeWhile, dropWhile, span,
    break, stripPrefix, group, inits, tails, isPrefixOf, isSuffixOf,
    isInfixOf, elem, notElem, lookup, find, filter, partition, (!!),
    elemIndex, elemIndices, findIndex, findIndices, zip, zip3, zip4,
    zip5, zip6, zip7, zipWith, zipWith3, zipWith4, zipWith5, zipWith6,
    zipWith7, unzip, unzip3, unzip4, unzip5, unzip6, unzip7, lines,
    words, unlines, unwords, nub, delete, (\\), union, intersect, sort,
    insert, nubBy, deleteBy, deleteFirstsBy, unionBy, intersectBy,
    groupBy, sortBy, insertBy, maximumBy, minimumBy, genericLength,
    genericTake, genericDrop, genericSplitAt, genericIndex, genericReplicate
  ) where
```

20.1 Basic functions

`(++) :: [a] -> [a] -> [a]`

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
```

If the first list is not finite, the result is the first list.

`head :: [a] -> a`

Extract the first element of a list, which must be non-empty.

`last :: [a] -> a`

Extract the last element of a list, which must be finite and non-empty.

`tail :: [a] -> [a]`

Extract the elements after the head of a list, which must be non-empty.

`init :: [a] -> [a]`

Return all the elements of a list except the last one. The list must be non-empty.

`null :: [a] -> Bool`

Test whether a list is empty.

`length :: [a] -> Int`

O(n). `length` returns the length of a finite list as an `Int`. It is an instance of the more general `Data.List.genericLength`, the result type of which may be any kind of number.

20.2 List transformations

`map :: (a -> b) -> [a] -> [b]`

`map f xs` is the list obtained by applying `f` to each element of `xs`, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

`reverse :: [a] -> [a]`

`reverse xs` returns the elements of `xs` in reverse order. `xs` must be finite.

`intersperse :: a -> [a] -> [a]`

The `intersperse` function takes an element and a list and 'intersperses' that element between the elements of the list. For example,

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

`intercalate :: [a] -> [[a]] -> [a]`

`intercalate xs xss` is equivalent to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

`transpose :: [[a]] -> [[a]]`

The `transpose` function transposes the rows and columns of its argument. For example,

```
transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]
```

`subsequences :: [a] -> [[a]]`

The `subsequences` function returns the list of all subsequences of the argument.

```
subsequences "abc" == ["","a","b","ab","c","ac","bc","abc"]
```

`permutations :: [a] -> [[a]]`

The `permutations` function returns the list of all permutations of the argument.

```
permutations "abc" == ["abc","bac","cba","bca","cab","acb"]
```

20.3 Reducing lists (folds)

`foldl :: (a -> b -> a) -> a -> [b] -> a`

`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
```

The list must be finite.

`foldl' :: (a -> b -> a) -> a -> [b] -> a`

A strict version of `foldl`.

`foldl1 :: (a -> a -> a) -> [a] -> a`

`foldl1` is a variant of `foldl` that has no starting value argument, and thus must be applied to non-empty lists.

`foldl1' :: (a -> a -> a) -> [a] -> a`

A strict version of `foldl1`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr`, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

`foldr1 :: (a -> a -> a) -> [a] -> a`

`foldr1` is a variant of `foldr` that has no starting value argument, and thus must be applied to non-empty lists.

20.3.1 Special folds

`concat :: [[a]] -> [a]`

Concatenate a list of lists.

`concatMap :: (a -> [b]) -> [a] -> [b]`

Map a function over a list and concatenate the results.

`and :: [Bool] -> Bool`

`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 :: [Bool] -> Bool`

`or` returns the disjunction of a Boolean list. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value at a finite index of a finite or infinite list.

`any :: (a -> Bool) -> [a] -> Bool`

Applied to a predicate and a list, `any` determines if any element of the list satisfies the predicate. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value for the predicate applied to an element at a finite index of a finite or infinite list.

`all :: (a -> Bool) -> [a] -> Bool`

Applied to a predicate and a list, `all` determines if all elements of the list satisfy the predicate. For the result to be `True`, the list must be finite; `False`, however, results from a `False` value for the predicate applied to an element at a finite index of a finite or infinite list.

`sum :: Num a => [a] -> a`

The `sum` function computes the sum of a finite list of numbers.

`product :: Num a => [a] -> a`

The `product` function computes the product of a finite list of numbers.

`maximum :: Ord a => [a] -> a`

`maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type. It is a special case of `maximumBy`, which allows the programmer to supply their own comparison function.

`minimum :: Ord a => [a] -> a`

`minimum` returns the minimum value from a list, which must be non-empty, finite, and of an ordered type. It is a special case of `minimumBy`, which allows the programmer to supply their own comparison function.

20.4 Building lists

20.4.1 Scans

`scanl :: (a -> b -> a) -> a -> [b] -> [a]`

`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 :: (a -> a -> a) -> [a] -> [a]`

`scanl1` is a variant of `scanl` that has no starting value argument:

```
scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
```

`scanr :: (a -> b -> b) -> b -> [a] -> [b]`

`scanr` is the right-to-left dual of `scanl`. Note that

```
head (scanr f z xs) == foldr f z xs.
```

`scanr1 :: (a -> a -> a) -> [a] -> [a]`

`scanr1` is a variant of `scanr` that has no starting value argument.

20.4.2 Accumulating maps

`mapAccumL :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])`

The `mapAccumL` function behaves like a combination of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

`mapAccumR :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])`

The `mapAccumR` function behaves like a combination of `map` and `foldr`; it applies a function to each element of a list, passing an accumulating parameter from right to left, and returning a final value of this accumulator together with the new list.

20.4.3 Infinite lists

`iterate :: (a -> a) -> a -> [a]`

`iterate f x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

`repeat :: a -> [a]`

`repeat x` is an infinite list, with `x` the value of every element.

`replicate :: Int -> a -> [a]`

`replicate n x` is a list of length `n` with `x` the value of every element. It is an instance of the more general `Data.List.genericReplicate`, in which `n` may be of any integral type.

`cycle :: [a] -> [a]`

`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.

20.4.4 Unfolding

`unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`

The `unfoldr` function is a 'dual' to `foldr`: while `foldr` reduces a list to a summary value, `unfoldr` builds a list from a seed value. The function takes the element and returns `Nothing` if it is done producing the list or returns `Just (a,b)`, in which case, `a` is a prepended to the list and `b` is used as the next element in a recursive call. For example,

```
iterate f == unfoldr (\x -> Just (x, f x))
```

In some cases, `unfoldr` can undo a `foldr` operation:

```
unfoldr f' (foldr f z xs) == xs
```

if the following holds:

```
f' (f x y) = Just (x,y)
f' z       = Nothing
```

A simple use of unfoldr:

```
unfoldr (\b -> if b == 0 then Nothing else Just (b, b-1)) 10
 [10,9,8,7,6,5,4,3,2,1]
```

20.5 Sublists

20.5.1 Extracting sublists

`take :: Int -> [a] -> [a]`

`take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`:

```
take 5 "Hello World!" == "Hello"
take 3 [1,2,3,4,5] == [1,2,3]
take 3 [1,2] == [1,2]
take 3 [] == []
take (-1) [1,2] == []
take 0 [1,2] == []
```

It is an instance of the more general `Data.List.genericTake`, in which `n` may be of any integral type.

`drop :: Int -> [a] -> [a]`

`drop n xs` returns the suffix of `xs` after the first `n` elements, or `[]` if `n > length xs`:

```
drop 6 "Hello World!" == "World!"
drop 3 [1,2,3,4,5] == [4,5]
drop 3 [1,2] == []
drop 3 [] == []
drop (-1) [1,2] == [1,2]
drop 0 [1,2] == [1,2]
```

It is an instance of the more general `Data.List.genericDrop`, in which `n` may be of any integral type.

`splitAt :: Int -> [a] -> ([a], [a])`

`splitAt n xs` returns a tuple where first element is `xs` prefix of length `n` and second element is the remainder of the list:

```
splitAt 6 "Hello World!" == ("Hello ","World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])
splitAt 1 [1,2,3] == ([1],[2,3])
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])
```

It is equivalent to `(take n xs, drop n xs)`. `splitAt` is an instance of the more general `Data.List.genericSplitAt`, in which `n` may be of any integral type.

`takeWhile :: (a -> Bool) -> [a] -> [a]`

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`:

```
takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
takeWhile (< 9) [1,2,3] == [1,2,3]
takeWhile (< 0) [1,2,3] == []
```

`dropWhile :: (a -> Bool) -> [a] -> [a]`

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`:

```
dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]
dropWhile (< 9) [1,2,3] == []
dropWhile (< 0) [1,2,3] == [1,2,3]
```

`span :: (a -> Bool) -> [a] -> ([a], [a])`

`span`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that satisfy `p` and second element is the remainder of the list:

```
span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
span (< 9) [1,2,3] == ([1,2,3],[])
span (< 0) [1,2,3] == ([],[1,2,3])
```

`span p xs` is equivalent to `(takeWhile p xs, dropWhile p xs)`

`break :: (a -> Bool) -> [a] -> ([a], [a])`

`break`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that *do not satisfy* `p` and second element is the remainder of the list:

```
break (> 3) [1,2,3,4,1,2,3,4] == ([1,2,3],[4,1,2,3,4])
break (< 9) [1,2,3] == ([],[1,2,3])
break (> 9) [1,2,3] == ([1,2,3],[])
```

`break p` is equivalent to `span (not . p)`.

`stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]`

The `stripPrefix` function drops the given prefix from a list. It returns `Nothing` if the list did not start with the prefix given, or `Just` the list after the prefix, if it does.

```
stripPrefix "foo" "foobar" == Just "bar"
stripPrefix "foo" "foo" == Just ""
stripPrefix "foo" "barfoo" == Nothing
stripPrefix "foo" "barfoobaz" == Nothing
```

`group :: Eq a => [a] -> [[a]]`

The `group` function takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements. For example,

```
group "Mississippi" = ["M","i","ss","i","ss","i","pp","i"]
```

It is a special case of `groupBy`, which allows the programmer to supply their own equality test.

`inits :: [a] -> [[a]]`

The `inits` function returns all initial segments of the argument, shortest first. For example,

```
inits "abc" == ["","a","ab","abc"]
```

`tails :: [a] -> [[a]]`

The `tails` function returns all final segments of the argument, longest first. For example,

```
tails "abc" == ["abc", "bc", "c",""]
```

20.5.2 Predicates

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

The `isPrefixOf` function takes two lists and returns `True` iff the first list is a prefix of the second.

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

The `isSuffixOf` function takes two lists and returns `True` iff the first list is a suffix of the second. Both lists must be finite.

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

The `isInfixOf` function takes two lists and returns `True` iff the first list is contained, wholly and intact, anywhere within the second.

Example:

```
isInfixOf "Haskell" "I really like Haskell." == True
isInfixOf "Ial" "I really like Haskell." == False
```

20.6 Searching lists

20.6.1 Searching by equality

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

`elem` is the list membership predicate, usually written in infix form, e.g., `` x `elem` xs ``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

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

`notElem` is the negation of `elem`.

`lookup :: Eq a => a -> [(a, b)] -> Maybe b`

`lookup key assocs` looks up a key in an association list.

20.6.2 Searching with a predicate

`find :: (a -> Bool) -> [a] -> Maybe a`

The `find` function takes a predicate and a list and returns the first element in the list matching the predicate, or `Nothing` if there is no such element.

`filter :: (a -> Bool) -> [a] -> [a]`

`filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

```
filter p xs = [ x | x <- xs, p x]
```

`partition :: (a -> Bool) -> [a] -> ([a], [a])`

The `partition` function takes a predicate a list and returns the pair of lists of elements which do and do not satisfy the predicate, respectively; i.e.,

```
partition p xs == (filter p xs, filter (not . p) xs)
```

20.7 Indexing lists

These functions treat a list `xs` as a indexed collection, with indices ranging from 0 to `length xs - 1`.

`(!!) :: [a] -> Int -> a`

List index (subscript) operator, starting from 0. It is an instance of the more general `Data.List.genericIndex`, which takes an index of any integral type.

`elemIndex :: Eq a => a -> [a] -> Maybe Int`

The `elemIndex` function returns the index of the first element in the given list which is equal (by `==`) to the query element, or `Nothing` if there is no such element.

`elemIndices :: Eq a => a -> [a] -> [Int]`

The `elemIndices` function extends `elemIndex`, by returning the indices of all elements equal to the query element, in ascending order.

`findIndex :: (a -> Bool) -> [a] -> Maybe Int`

The `findIndex` function takes a predicate and a list and returns the index of the first element in the list satisfying the predicate, or `Nothing` if there is no such element.

`findIndices :: (a -> Bool) -> [a] -> [Int]`

The `findIndices` function extends `findIndex`, by returning the indices of all elements satisfying the predicate, in ascending order.

20.8 Zipping and unzipping lists

`zip :: [a] -> [b] -> [(a, b)]`

`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 :: [a] -> [b] -> [c] -> [(a, b, c)]`

`zip3` takes three lists and returns a list of triples, analogous to `zip`.

```
zip4 :: [a] -> [b] -> [c] -> [d] -> [(a, b, c, d)]
```

The `zip4` function takes four lists and returns a list of quadruples, analogous to `zip`.

```
zip5 :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a, b, c, d, e)]
```

The `zip5` function takes five lists and returns a list of five-tuples, analogous to `zip`.

```
zip6 :: [a]
        -> [b] -> [c] -> [d] -> [e] -> [f] -> [(a, b, c, d, e, f)]
```

The `zip6` function takes six lists and returns a list of six-tuples, analogous to `zip`.

```
zip7 :: [a]
        -> [b]
          -> [c] -> [d] -> [e] -> [f] -> [g] -> [(a, b, c, d, e, f, g)]
```

The `zip7` function takes seven lists and returns a list of seven-tuples, analogous to `zip`.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

`zipWith` generalises `zip` 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.

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

The `zipWith3` function takes a function which combines three elements, as well as three lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith4 :: (a -> b -> c -> d -> e)
            -> [a] -> [b] -> [c] -> [d] -> [e]
```

The `zipWith4` function takes a function which combines four elements, as well as four lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith5 :: (a -> b -> c -> d -> e -> f)
            -> [a] -> [b] -> [c] -> [d] -> [e] -> [f]
```

The `zipWith5` function takes a function which combines five elements, as well as five lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith6 :: (a -> b -> c -> d -> e -> f -> g)
            -> [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
```

The `zipWith6` function takes a function which combines six elements, as well as six lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith7 :: (a -> b -> c -> d -> e -> f -> g -> h)
            -> [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [h]
```

The `zipWith7` function takes a function which combines seven elements, as well as seven lists and returns a list of their point-wise combination, analogous to `zipWith`.

`unzip :: [(a, b)] -> ([a], [b])`

`unzip` transforms a list of pairs into a list of first components and a list of second components.

`unzip3 :: [(a, b, c)] -> ([a], [b], [c])`

The `unzip3` function takes a list of triples and returns three lists, analogous to `unzip`.

`unzip4 :: [(a, b, c, d)] -> ([a], [b], [c], [d])`

The `unzip4` function takes a list of quadruples and returns four lists, analogous to `unzip`.

`unzip5 :: [(a, b, c, d, e)] -> ([a], [b], [c], [d], [e])`

The `unzip5` function takes a list of five-tuples and returns five lists, analogous to `unzip`.

`unzip6 :: [(a, b, c, d, e, f)] -> ([a], [b], [c], [d], [e], [f])`

The `unzip6` function takes a list of six-tuples and returns six lists, analogous to `unzip`.

```
unzip7 :: [(a, b, c, d, e, f, g)]
        -> ([a], [b], [c], [d], [e], [f], [g])
```

The `unzip7` function takes a list of seven-tuples and returns seven lists, analogous to `unzip`.

20.9 Special lists

20.9.1 Functions on strings

`lines :: String -> [String]`

`lines` breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

`words :: String -> [String]`

`words` breaks a string up into a list of words, which were delimited by white space.

`unlines :: [String] -> String`

`unlines` is an inverse operation to `lines`. It joins lines, after appending a terminating newline to each.

`unwords :: [String] -> String`

`unwords` is an inverse operation to `words`. It joins words with separating spaces.

20.9.2 ”Set” operations

`nub :: Eq a => [a] -> [a]`

> *O(n^2)*. The `nub` function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. (The name `nub` means ‘essence’.) It is a special case of `nubBy`, which allows the programmer to supply their own equality test.

`delete :: Eq a => a -> [a] -> [a]`

> `delete x` removes the first occurrence of `x` from its list argument. For example,
>
> ```
> delete 'a' "banana" == "bnana"
> ```
>
> It is a special case of `deleteBy`, which allows the programmer to supply their own equality test.

`(\\) :: Eq a => [a] -> [a] -> [a]`

> The `\\` function is list difference ((non-associative). In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus
>
> ```
> (xs ++ ys) \\ xs == ys.
> ```
>
> It is a special case of `deleteFirstsBy`, which allows the programmer to supply their own equality test.

`union :: Eq a => [a] -> [a] -> [a]`

> The `union` function returns the list union of the two lists. For example,
>
> ```
> "dog" `union` "cow" == "dogcw"
> ```
>
> Duplicates, and elements of the first list, are removed from the the second list, but if the first list contains duplicates, so will the result. It is a special case of `unionBy`, which allows the programmer to supply their own equality test.

`intersect :: Eq a => [a] -> [a] -> [a]`

> The `intersect` function takes the list intersection of two lists. For example,
>
> ```
> [1,2,3,4] `intersect` [2,4,6,8] == [2,4]
> ```
>
> If the first list contains duplicates, so will the result.
>
> ```
> [1,2,2,3,4] `intersect` [6,4,4,2] == [2,2,4]
> ```
>
> It is a special case of `intersectBy`, which allows the programmer to supply their own equality test.

20.9.3 Ordered lists

`sort :: Ord a => [a] -> [a]`

The `sort` function implements a stable sorting algorithm. It is a special case of `sortBy`, which allows the programmer to supply their own comparison function.

`insert :: Ord a => a -> [a] -> [a]`

The `insert` function takes an element and a list and inserts the element into the list at the last position where it is still less than or equal to the next element. In particular, if the list is sorted before the call, the result will also be sorted. It is a special case of `insertBy`, which allows the programmer to supply their own comparison function.

20.10 Generalized functions

20.10.1 The ”`By`” operations

By convention, overloaded functions have a non-overloaded counterpart whose name is suffixed with ‘`By`’.

20.10.1.1 User-supplied equality (replacing an `Eq` context)

The predicate is assumed to define an equivalence.

`nubBy :: (a -> a -> Bool) -> [a] -> [a]`

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded == function.

`deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]`

The `deleteBy` function behaves like `delete`, but takes a user-supplied equality predicate.

`deleteFirstsBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

The `deleteFirstsBy` function takes a predicate and two lists and returns the first list with the first occurrence of each element of the second list removed.

`unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

The `unionBy` function is the non-overloaded version of `union`.

`intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

The `intersectBy` function is the non-overloaded version of `intersect`.

`groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`

The `groupBy` function is the non-overloaded version of `group`.

20.10.1.2 User-supplied comparison (replacing an `Ord` context)

The function is assumed to define a total ordering.

`sortBy :: (a -> a -> Ordering) -> [a] -> [a]`

The `sortBy` function is the non-overloaded version of `sort`.

`insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]`

The non-overloaded version of `insert`.

`maximumBy :: (a -> a -> Ordering) -> [a] -> a`

The `maximumBy` function takes a comparison function and a list and returns the greatest element of the list by the comparison function. The list must be finite and non-empty.

`minimumBy :: (a -> a -> Ordering) -> [a] -> a`

The `minimumBy` function takes a comparison function and a list and returns the least element of the list by the comparison function. The list must be finite and non-empty.

20.10.2 The "`generic`" operations

The prefix '`generic`' indicates an overloaded function that is a generalized version of a `Prelude` function.

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

The `genericLength` function is an overloaded version of `length`. In particular, instead of returning an `Int`, it returns any type which is an instance of `Num`. It is, however, less efficient than `length`.

`genericTake :: Integral i => i -> [a] -> [a]`

The `genericTake` function is an overloaded version of `take`, which accepts any `Integral` value as the number of elements to take.

`genericDrop :: Integral i => i -> [a] -> [a]`

The `genericDrop` function is an overloaded version of `drop`, which accepts any `Integral` value as the number of elements to drop.

`genericSplitAt :: Integral i => i -> [b] -> ([b], [b])`

The `genericSplitAt` function is an overloaded version of `splitAt`, which accepts any `Integral` value as the position at which to split.

`genericIndex :: Integral a => [b] -> a -> b`

The `genericIndex` function is an overloaded version of `!!`, which accepts any `Integral` value as the index.

`genericReplicate :: Integral i => i -> a -> [a]`

The `genericReplicate` function is an overloaded version of `replicate`, which accepts any `Integral` value as the number of repetitions to make.

Chapter 21

`Data.Maybe`

```
module Data.Maybe (
    Maybe(Nothing, Just),  maybe,  isJust,  isNothing,  fromJust,  fromMaybe,
    listToMaybe,  maybeToList,  catMaybes,  mapMaybe
  ) where
```

21.1 The `Maybe` type and operations

`data Maybe a`

```
= Nothing
| Just a
```

The `Maybe` type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented as `Just a`), or it is empty (represented as `Nothing`). Using `Maybe` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as `error`.

The `Maybe` type is also a monad. It is a simple kind of error monad, where all errors are represented by `Nothing`. A richer error monad can be built using the `Data.Either.Either` type.

```
instance Monad Maybe
instance Functor Maybe
instance MonadPlus Maybe
instance Eq a => Eq (Maybe a)
instance Ord a => Ord (Maybe a)
instance Read a => Read (Maybe a)
instance Show a => Show (Maybe a)
```

`maybe :: b -> (a -> b) -> Maybe a -> b`

The `maybe` function takes a default value, a function, and a `Maybe` value. If the `Maybe` value is `Nothing`, the function returns the default value. Otherwise, it applies the function to the value inside the `Just` and returns the result.

`isJust :: Maybe a -> Bool`

The `isJust` function returns `True` iff its argument is of the form `Just _`.

`isNothing :: Maybe a -> Bool`

The `isNothing` function returns `True` iff its argument is `Nothing`.

`fromJust :: Maybe a -> a`

The `fromJust` function extracts the element out of a `Just` and throws an error if its argument is `Nothing`.

`fromMaybe :: a -> Maybe a -> a`

The `fromMaybe` function takes a default value and and `Maybe` value. If the `Maybe` is `Nothing`, it returns the default values; otherwise, it returns the value contained in the `Maybe`.

`listToMaybe :: [a] -> Maybe a`

The `listToMaybe` function returns `Nothing` on an empty list or `Just a` where `a` is the first element of the list.

`maybeToList :: Maybe a -> [a]`

The `maybeToList` function returns an empty list when given `Nothing` or a singleton list when not given `Nothing`.

`catMaybes :: [Maybe a] -> [a]`

The `catMaybes` function takes a list of `Maybe`s and returns a list of all the `Just` values.

`mapMaybe :: (a -> Maybe b) -> [a] -> [b]`

The `mapMaybe` function is a version of `map` which can throw out elements. In particular, the functional argument returns something of type `Maybe b`. If this is `Nothing`, no element is added on to the result list. If it just `Just b`, then `b` is included in the result list.

21.2 Specification

```
module Data.Maybe(
    Maybe(Nothing, Just),
    isJust, isNothing,
    fromJust, fromMaybe, listToMaybe, maybeToList,
    catMaybes, mapMaybe,
    maybe
  ) where

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

```
isJust                 :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False

isNothing              :: Maybe a -> Bool
isNothing              =  not . isJust

fromJust               :: Maybe a -> a
fromJust (Just a)      =  a
fromJust Nothing       =  error "Maybe.fromJust: Nothing"

fromMaybe              :: a -> Maybe a -> a
fromMaybe d Nothing    =  d
fromMaybe d (Just a)   =  a

maybeToList            :: Maybe a -> [a]
maybeToList Nothing    =  []
maybeToList (Just a)   =  [a]

listToMaybe            :: [a] -> Maybe a
listToMaybe []         =  Nothing
listToMaybe (a:_)      =  Just a

catMaybes              :: [Maybe a] -> [a]
catMaybes ms           =  [ m | Just m <- ms ]

mapMaybe               :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f             =  catMaybes . map f
```

Chapter 22

`Data.Ratio`

```
module Data.Ratio (
    Ratio,  Rational,  (%),  numerator,  denominator,  approxRational
  ) where
```

`data Integral a => Ratio a`

Rational numbers, with numerator and denominator of some `Integral` type.

```
instance Integral a => Enum (Ratio a)
instance Integral a => Eq (Ratio a)
instance Integral a => Fractional (Ratio a)
instance Integral a => Num (Ratio a)
instance Integral a => Ord (Ratio a)
instance (Integral a, Read a) => Read (Ratio a)
instance Integral a => Real (Ratio a)
instance Integral a => RealFrac (Ratio a)
instance Integral a => Show (Ratio a)
```

`type Rational = Ratio Integer`

Arbitrary-precision rational numbers, represented as a ratio of two `Integer` values. A rational number may be constructed using the `%` operator.

`(%) :: Integral a => a -> a -> Ratio a`

Forms the ratio of two integral numbers.

`numerator :: Integral a => Ratio a -> a`

Extract the numerator of the ratio in reduced form: the numerator and denominator have no common factor and the denominator is positive.

`denominator :: Integral a => Ratio a -> a`

Extract the denominator of the ratio in reduced form: the numerator and denominator have no common factor and the denominator is positive.

`approxRational :: RealFrac a => a -> a -> Rational`

`approxRational`, applied to two real fractional numbers `x` and `epsilon`, returns the simplest rational number within `epsilon` of `x`. A rational number `y` is said to be *simpler* than another `y'` if

- `abs (numerator y) <= abs (numerator y')`, and
- `denominator y <= denominator y'`.

Any real interval contains a unique simplest rational; in particular, note that `0/1` is the simplest rational of all.

22.1 Specification

```
module  Data.Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %

ratPrec = 7 :: Int

data  (Integral a)      => Ratio a = !a :% !a  deriving (Eq)
type  Rational          =  Ratio Integer

(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational


-- "reduce" is a subsidiary function used only in this module.
-- It normalises a ratio by dividing both numerator
-- and denominator by their greatest common divisor.
--
-- E.g., 12 `reduce` 8    ==  3 :%   2
--       12 `reduce` (-8) ==  3 :% (-2)

reduce _ 0              =  error "Data.Ratio.% : zero denominator"
reduce x y              =  (x `quot` d) :% (y `quot` d)
                           where d = gcd x y

x % y                   =  reduce (x * signum y) (abs y)

numerator (x :% _)      =  x

denominator (_ :% y)    =  y


instance  (Integral a)  => Ord (Ratio a)  where
    (x:%y) <= (x':%y')  =  x * y' <= x' * y
```

```
    (x:%y) <  (x’:%y’)  =  x * y’ <  x’ * y

instance  (Integral a)  => Num (Ratio a)  where
    (x:%y) + (x’:%y’)   =  reduce (x*y’ + x’*y) (y*y’)
    (x:%y) * (x’:%y’)   =  reduce (x * x’) (y * y’)
    negate (x:%y)       =  (-x) :% y
    abs (x:%y)          =  abs x :% y
    signum (x:%y)       =  signum x :% 1
    fromInteger x       =  fromInteger x :% 1

instance  (Integral a)  => Real (Ratio a)  where
    toRational (x:%y)   =  toInteger x :% toInteger y

instance  (Integral a)  => Fractional (Ratio a)  where
    (x:%y) / (x’:%y’)   =  (x*y’) % (y*x’)
    recip (x:%y)        =  y % x
    fromRational (x:%y) =  fromInteger x :% fromInteger y

instance  (Integral a)  => RealFrac (Ratio a)  where
    properFraction (x:%y) = (fromIntegral q, r:%y)
                            where (q,r) = quotRem x y

instance  (Integral a)  => Enum (Ratio a)  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate        -- May overflow
    enumFrom         =  numericEnumFrom               -- These numericEnumXXX functions
    enumFromThen     =  numericEnumFromThen   -- are as defined in Prelude.hs
    enumFromTo       =  numericEnumFromTo     -- but not exported from it!
    enumFromThenTo   =  numericEnumFromThenTo

instance  (Read a, Integral a)  => Read (Ratio a)  where
    readsPrec p  =  readParen (p > ratPrec)
                              (\r -> [(x%y,u) | (x,s)   <- readsPrec (ratPrec+1) r,
                                                ("%",t) <- lex s,
                                                (y,u)   <- readsPrec (ratPrec+1) t ])

instance  (Integral a)  => Show (Ratio a)  where
    showsPrec p (x:%y)  =  showParen (p > ratPrec)
                              showsPrec (ratPrec+1) x .
                              showString " % " .
                              showsPrec (ratPrec+1) y)



approxRational x eps    =  simplest (x-eps) (x+eps)
        where simplest x y | y < x      =  simplest y x
                           | x == y     =  xr
                           | x > 0      =  simplest’ n d n’ d’
                           | y < 0      =  - simplest’ (-n’) d’ (-n) d
                           | otherwise  =  0 :% 1
                                        where xr@(n:%d) = toRational x
                                              (n’:%d’)  = toRational y
```

```
simplest' n d n' d'       -- assumes 0 < n%d < n'%d'
          | r == 0     =  q :% 1
          | q /= q'    =  (q+1) :% 1
          | otherwise  =  (q*n''+d'') :% n''
                       where (q,r)      =  quotRem n d
                             (q',r')    =  quotRem n' d'
                             (n'':%d'') =  simplest' d' r' d r
```

Chapter 23

Data.Word

```
module Data.Word (
    Word,  Word8,  Word16,  Word32,  Word64
  ) where
```

23.1 Unsigned integral types

This module provides unsigned integer types of unspecified width (`Word`) and fixed widths (`Word8`, `Word16`, `Word32` and `Word64`). All arithmetic is performed modulo 2^n, where `n` is the number of bits in the type.

For coercing between any two integer types, use `fromIntegral`. Coercing word types to and from integer types preserves representation, not sign.

The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell language report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Word` types defined here.

Right and left shifts by amounts greater than or equal to the width of the type result in a zero result. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example `1 << 32 == 1` in some C implementations.

`data Word`

A `Word` is an unsigned integral type, with the same size as `Int`.

instance Bounded Word
instance Enum Word
instance Eq Word
instance Integral Word
instance Num Word
instance Ord Word
instance Read Word
instance Real Word
instance Show Word
instance Ix Word
instance Storable Word
instance Bits Word

data Word8

8-bit unsigned integer type

instance Bounded Word8
instance Enum Word8
instance Eq Word8
instance Integral Word8
instance Num Word8
instance Ord Word8
instance Read Word8
instance Real Word8
instance Show Word8
instance Ix Word8
instance Storable Word8
instance Bits Word8

data Word16

16-bit unsigned integer type

instance Bounded Word16
instance Enum Word16
instance Eq Word16
instance Integral Word16
instance Num Word16
instance Ord Word16
instance Read Word16
instance Real Word16
instance Show Word16
instance Ix Word16
instance Storable Word16
instance Bits Word16

data Word32

32-bit unsigned integer type

```
instance Bounded Word32
instance Enum Word32
instance Eq Word32
instance Integral Word32
instance Num Word32
instance Ord Word32
instance Read Word32
instance Real Word32
instance Show Word32
instance Ix Word32
instance Storable Word32
instance Bits Word32
```

```
data Word64
```

64-bit unsigned integer type

```
instance Bounded Word64
instance Enum Word64
instance Eq Word64
instance Integral Word64
instance Num Word64
instance Ord Word64
instance Read Word64
instance Real Word64
instance Show Word64
instance Ix Word64
instance Storable Word64
instance Bits Word64
```

Chapter 24

Foreign

```
module Foreign (
    module Data.Bits,  module Data.Int,  module Data.Word,  module Foreign.Ptr,
    module Foreign.ForeignPtr,  module Foreign.StablePtr,
    module Foreign.Storable,  module Foreign.Marshal
  ) where
```

The module `Foreign` combines the interfaces of all modules providing language-independent marshalling support, namely

module Data.Bits
module Data.Int
module Data.Word
module Foreign.Ptr
module Foreign.ForeignPtr
module Foreign.StablePtr
module Foreign.Storable
module Foreign.Marshal

Chapter 25

Foreign.C

```
module Foreign.C (
    module Foreign.C.Types,  module Foreign.C.String,  module Foreign.C.Error
  ) where
```

The module `Foreign.C` combines the interfaces of all modules providing C-specific marshalling support, namely

`module Foreign.C.Types`
`module Foreign.C.String`
`module Foreign.C.Error`

Chapter 26

Foreign.C.Error

```
module Foreign.C.Error (
    Errno(Errno),  eOK,  e2BIG,  eACCES,  eADDRINUSE,  eADDRNOTAVAIL,  eADV,
    eAFNOSUPPORT,  eAGAIN,  eALREADY,  eBADF,  eBADMSG,  eBADRPC,  eBUSY,
    eCHILD,  eCOMM,  eCONNABORTED,  eCONNREFUSED,  eCONNRESET,  eDEADLK,
    eDESTADDRREQ,  eDIRTY,  eDOM,  eDQUOT,  eEXIST,  eFAULT,  eFBIG,  eFTYPE,
    eHOSTDOWN,  eHOSTUNREACH,  eIDRM,  eILSEQ,  eINPROGRESS,  eINTR,  eINVAL,
    eIO,  eISCONN,  eISDIR,  eLOOP,  eMFILE,  eMLINK,  eMSGSIZE,  eMULTIHOP,
    eNAMETOOLONG,  eNETDOWN,  eNETRESET,  eNETUNREACH,  eNFILE,  eNOBUFS,
    eNODATA,  eNODEV,  eNOENT,  eNOEXEC,  eNOLCK,  eNOLINK,  eNOMEM,  eNOMSG,
    eNONET,  eNOPROTOOPT,  eNOSPC,  eNOSR,  eNOSTR,  eNOSYS,  eNOTBLK,
    eNOTCONN,  eNOTDIR,  eNOTEMPTY,  eNOTSOCK,  eNOTTY,  eNXIO,  eOPNOTSUPP,
    ePERM,  ePFNOSUPPORT,  ePIPE,  ePROCLIM,  ePROCUNAVAIL,  ePROGMISMATCH,
    ePROGUNAVAIL,  ePROTO,  ePROTONOSUPPORT,  ePROTOTYPE,  eRANGE,  eREMCHG,
    eREMOTE,  eROFS,  eRPCMISMATCH,  eRREMOTE,  eSHUTDOWN,  eSOCKTNOSUPPORT,
    eSPIPE,  eSRCH,  eSRMNT,  eSTALE,  eTIME,  eTIMEDOUT,  eTOOMANYREFS,
    eTXTBSY,  eUSERS,  eWOULDBLOCK,  eXDEV,  isValidErrno,  getErrno,
    resetErrno,  errnoToIOError,  throwErrno,  throwErrnoIf,  throwErrnoIf_,
    throwErrnoIfRetry,  throwErrnoIfRetry_,  throwErrnoIfMinus1,
    throwErrnoIfMinus1_,  throwErrnoIfMinus1Retry,  throwErrnoIfMinus1Retry_,
    throwErrnoIfNull,  throwErrnoIfNullRetry,  throwErrnoIfRetryMayBlock,
    throwErrnoIfRetryMayBlock_,  throwErrnoIfMinus1RetryMayBlock,
    throwErrnoIfMinus1RetryMayBlock_,  throwErrnoIfNullRetryMayBlock,
    throwErrnoPath,  throwErrnoPathIf,  throwErrnoPathIf_,
    throwErrnoPathIfNull,  throwErrnoPathIfMinus1,  throwErrnoPathIfMinus1_
  ) where
```

The module `Foreign.C.Error` facilitates C-specific error handling of `errno`.

26.1 Haskell representations of `errno` values

`newtype Errno`

= Errno CInt

Haskell representation for `errno` values. The implementation is deliberately exposed, to allow users to add their own definitions of `Errno` values.

instance Eq Errno

26.1.1 Common `errno` symbols

Different operating systems and/or C libraries often support different values of `errno`. This module defines the common values, but due to the open definition of `Errno` users may add definitions which are not predefined.

eOK :: Errno

e2BIG :: Errno

eACCES :: Errno

eADDRINUSE :: Errno

eADDRNOTAVAIL :: Errno

eADV :: Errno

eAFNOSUPPORT :: Errno

eAGAIN :: Errno

eALREADY :: Errno

eBADF :: Errno

eBADMSG :: Errno

eBADRPC :: Errno

eBUSY :: Errno

eCHILD :: Errno

eCOMM :: Errno

eCONNABORTED :: Errno

eCONNREFUSED :: Errno

eCONNRESET :: Errno

eDEADLK :: Errno

eDESTADDRREQ :: Errno

eDIRTY :: Errno

eDOM :: Errno

`eDQUOT :: Errno`

`eEXIST :: Errno`

`eFAULT :: Errno`

`eFBIG :: Errno`

`eFTYPE :: Errno`

`eHOSTDOWN :: Errno`

`eHOSTUNREACH :: Errno`

`eIDRM :: Errno`

`eILSEQ :: Errno`

`eINPROGRESS :: Errno`

`eINTR :: Errno`

`eINVAL :: Errno`

`eIO :: Errno`

`eISCONN :: Errno`

`eISDIR :: Errno`

`eLOOP :: Errno`

`eMFILE :: Errno`

`eMLINK :: Errno`

`eMSGSIZE :: Errno`

`eMULTIHOP :: Errno`

`eNAMETOOLONG :: Errno`

`eNETDOWN :: Errno`

`eNETRESET :: Errno`

`eNETUNREACH :: Errno`

`eNFILE :: Errno`

`eNOBUFS :: Errno`

`eNODATA :: Errno`

`eNODEV :: Errno`

`eNOENT :: Errno`

`eNOEXEC :: Errno`

`eNOLCK :: Errno`

eNOLINK :: Errno

eNOMEM :: Errno

eNOMSG :: Errno

eNONET :: Errno

eNOPROTOOPT :: Errno

eNOSPC :: Errno

eNOSR :: Errno

eNOSTR :: Errno

eNOSYS :: Errno

eNOTBLK :: Errno

eNOTCONN :: Errno

eNOTDIR :: Errno

eNOTEMPTY :: Errno

eNOTSOCK :: Errno

eNOTTY :: Errno

eNXIO :: Errno

eOPNOTSUPP :: Errno

ePERM :: Errno

ePFNOSUPPORT :: Errno

ePIPE :: Errno

ePROCLIM :: Errno

ePROCUNAVAIL :: Errno

ePROGMISMATCH :: Errno

ePROGUNAVAIL :: Errno

ePROTO :: Errno

ePROTONOSUPPORT :: Errno

ePROTOTYPE :: Errno

eRANGE :: Errno

eREMCHG :: Errno

eREMOTE :: Errno

eROFS :: Errno

`eRPCMISMATCH :: Errno`

`eRREMOTE :: Errno`

`eSHUTDOWN :: Errno`

`eSOCKTNOSUPPORT :: Errno`

`eSPIPE :: Errno`

`eSRCH :: Errno`

`eSRMNT :: Errno`

`eSTALE :: Errno`

`eTIME :: Errno`

`eTIMEDOUT :: Errno`

`eTOOMANYREFS :: Errno`

`eTXTBSY :: Errno`

`eUSERS :: Errno`

`eWOULDBLOCK :: Errno`

`eXDEV :: Errno`

26.1.2 `Errno` functions

`isValidErrno :: Errno -> Bool`

Yield `True` if the given `Errno` value is valid on the system. This implies that the `Eq` instance of `Errno` is also system dependent as it is only defined for valid values of `Errno`.

`getErrno :: IO Errno`

Get the current value of `errno` in the current thread.

`resetErrno :: IO ()`

Reset the current thread's `errno` value to `eOK`.

`errnoToIOError`

`::`	**`String`**	the location where the error occurred
`->`	**`Errno`**	the error number
`->`	**`Maybe Handle`**	optional handle associated with the error
`->`	**`Maybe String`**	optional filename associated with the error
`->`	**`IOError`**	

Construct an `IOError` based on the given `Errno` value. The optional information can be used to improve the accuracy of error messages.

`throwErrno`

`::`	**`String`**	textual description of the error location
`->`	**`IO a`**	

Throw an `IOError` corresponding to the current value of `getErrno`.

26.1.3 Guards for IO operations that may fail

`throwErrnoIf`

`::`	**`(a -> Bool)`**	predicate to apply to the result value of the `IO` operation
`->`	**`String`**	textual description of the location
`->`	**`IO a`**	the `IO` operation to be executed
`->`	**`IO a`**	

Throw an `IOError` corresponding to the current value of `getErrno` if the result value of the `IO` action meets the given predicate.

`throwErrnoIf_ :: (a -> Bool) -> String -> IO a -> IO ()`

as `throwErrnoIf`, but discards the result of the `IO` action after error handling.

`throwErrnoIfRetry :: (a -> Bool) -> String -> IO a -> IO a`

as `throwErrnoIf`, but retry the `IO` action when it yields the error code `eINTR` - this amounts to the standard retry loop for interrupted POSIX system calls.

`throwErrnoIfRetry_ :: (a -> Bool) -> String -> IO a -> IO ()`

as `throwErrnoIfRetry`, but discards the result.

`throwErrnoIfMinus1 :: Num a => String -> IO a -> IO a`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns a result of `-1`.

`throwErrnoIfMinus1_ :: Num a => String -> IO a -> IO ()`

as `throwErrnoIfMinus1`, but discards the result.

`throwErrnoIfMinus1Retry :: Num a => String -> IO a -> IO a`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns a result of `-1`, but retries in case of an interrupted operation.

`throwErrnoIfMinus1Retry_ :: Num a => String -> IO a -> IO ()`

as `throwErrnoIfMinus1`, but discards the result.

`throwErrnoIfNull :: String -> IO (Ptr a) -> IO (Ptr a)`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns `nullPtr`.

```
throwErrnoIfNullRetry :: String -> IO (Ptr a) -> IO (Ptr a)
```

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns `nullPtr`, but retry in case of an interrupted operation.

```
throwErrnoIfRetryMayBlock
```

`::`	`(a -> Bool)`	predicate to apply to the result value of the `IO` operation
`->`	`String`	textual description of the location
`->`	`IO a`	the `IO` operation to be executed
`->`	`IO b`	action to execute before retrying if an immediate retry would block
`->`	`IO a`	

as `throwErrnoIfRetry`, but additionally if the operation yields the error code `eAGAIN` or `eWOULDBLOCK`, an alternative action is executed before retrying.

```
throwErrnoIfRetryMayBlock_ :: (a -> Bool)
                           -> String -> IO a -> IO b -> IO ()
```

as `throwErrnoIfRetryMayBlock`, but discards the result.

```
throwErrnoIfMinus1RetryMayBlock :: Num a => String
                                         -> IO a -> IO b -> IO a
```

as `throwErrnoIfMinus1Retry`, but checks for operations that would block.

```
throwErrnoIfMinus1RetryMayBlock_ :: Num a => String
                                          -> IO a -> IO b -> IO ()
```

as `throwErrnoIfMinus1RetryMayBlock`, but discards the result.

```
throwErrnoIfNullRetryMayBlock :: String
                              -> IO (Ptr a) -> IO b -> IO (Ptr a)
```

as `throwErrnoIfNullRetry`, but checks for operations that would block.

```
throwErrnoPath :: String -> FilePath -> IO a
```

as `throwErrno`, but exceptions include the given path when appropriate.

```
throwErrnoPathIf :: (a -> Bool)
                 -> String -> FilePath -> IO a -> IO a
```

as `throwErrnoIf`, but exceptions include the given path when appropriate.

```
throwErrnoPathIf_ :: (a -> Bool)
                  -> String -> FilePath -> IO a -> IO ()
```

as `throwErrnoIf_`, but exceptions include the given path when appropriate.

```
throwErrnoPathIfNull :: String
                     -> FilePath -> IO (Ptr a) -> IO (Ptr a)
```

as `throwErrnoIfNull`, but exceptions include the given path when appropriate.

```
throwErrnoPathIfMinus1 :: Num a => String
                                 -> FilePath -> IO a -> IO a
```

as `throwErrnoIfMinus1`, but exceptions include the given path when appropriate.

```
throwErrnoPathIfMinus1_ :: Num a => String
                                  -> FilePath -> IO a -> IO ()
```

as `throwErrnoIfMinus1_`, but exceptions include the given path when appropriate.

Chapter 27

Foreign.C.String

```
module Foreign.C.String (
    CString,  CStringLen,  peekCString,  peekCStringLen,  newCString,
    newCStringLen,  withCString,  withCStringLen,  charIsRepresentable,
    castCharToCChar,  castCCharToChar,  castCharToCUChar,  castCUCharToChar,
    castCharToCSChar,  castCSCharToChar,  peekCAString,  peekCAStringLen,
    newCAString,  newCAStringLen,  withCAString,  withCAStringLen,  CWString,
    CWStringLen,  peekCWString,  peekCWStringLen,  newCWString,
    newCWStringLen,  withCWString,  withCWStringLen
  ) where
```

Utilities for primitive marshalling of C strings.

The marshalling converts each Haskell character, representing a Unicode code point, to one or more bytes in a manner that, by default, is determined by the current locale. As a consequence, no guarantees can be made about the relative length of a Haskell string and its corresponding C string, and therefore all the marshalling routines include memory allocation. The translation between Unicode and the encoding of the current locale may be lossy.

27.1 C strings

`type CString = Ptr CChar`

A C string is a reference to an array of C characters terminated by NUL.

`type CStringLen = (Ptr CChar, Int)`

A string with explicit length information in bytes instead of a terminating NUL (allowing NUL characters in the middle of the string).

27.1.1 Using a locale-dependent encoding

Currently these functions are identical to their `CAString` counterparts; eventually they will use an encoding determined by the current locale.

`peekCString :: CString -> IO String`

Marshal a NUL terminated C string into a Haskell string.

`peekCStringLen :: CStringLen -> IO String`

Marshal a C string with explicit length into a Haskell string.

`newCString :: String -> IO CString`

Marshal a Haskell string into a NUL terminated C string.

- the Haskell string may *not* contain any NUL characters
- new storage is allocated for the C string and must be explicitly freed using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree`.

`newCStringLen :: String -> IO CStringLen`

Marshal a Haskell string into a C string (ie, character array) with explicit length information.

- new storage is allocated for the C string and must be explicitly freed using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree`.

`withCString :: String -> (CString -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`withCStringLen :: String -> (CStringLen -> IO a) -> IO a`

Marshal a Haskell string into a C string (ie, character array) in temporary storage, with explicit length information.

- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`charIsRepresentable :: Char -> IO Bool`

Determines whether a character can be accurately encoded in a `CString`. Unrepresentable characters are converted to `'?'`.

Currently only Latin-1 characters are representable.

27.1.2 Using 8-bit characters

These variants of the above functions are for use with C libraries that are ignorant of Unicode. These functions should be used with care, as a loss of information can occur.

`castCharToCChar :: Char -> CChar`

Convert a Haskell character to a C character. This function is only safe on the first 256 characters.

`castCCharToChar :: CChar -> Char`

Convert a C byte, representing a Latin-1 character, to the corresponding Haskell character.

`castCharToCUChar :: Char -> CUChar`

Convert a Haskell character to a C `unsigned char`. This function is only safe on the first 256 characters.

`castCUCharToChar :: CUChar -> Char`

Convert a C `unsigned char`, representing a Latin-1 character, to the corresponding Haskell character.

`castCharToCSChar :: Char -> CSChar`

Convert a Haskell character to a C `signed char`. This function is only safe on the first 256 characters.

`castCSCharToChar :: CSChar -> Char`

Convert a C `signed char`, representing a Latin-1 character, to the corresponding Haskell character.

`peekCAString :: CString -> IO String`

Marshal a NUL terminated C string into a Haskell string.

`peekCAStringLen :: CStringLen -> IO String`

Marshal a C string with explicit length into a Haskell string.

`newCAString :: String -> IO CString`

Marshal a Haskell string into a NUL terminated C string.

- the Haskell string may *not* contain any NUL characters
- new storage is allocated for the C string and must be explicitly freed using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree`.

`newCAStringLen :: String -> IO CStringLen`

Marshal a Haskell string into a C string (ie, character array) with explicit length information.

- new storage is allocated for the C string and must be explicitly freed using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree`.

`withCAString :: String -> (CString -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`withCAStringLen :: String -> (CStringLen -> IO a) -> IO a`

Marshal a Haskell string into a C string (ie, character array) in temporary storage, with explicit length information.

- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

27.2 C wide strings

These variants of the above functions are for use with C libraries that encode Unicode using the C `wchar_t` type in a system-dependent way. The only encodings supported are

- UTF-32 (the C compiler defines `__STDC_ISO_10646__`), or
- UTF-16 (as used on Windows systems).

`type CWString = Ptr CWchar`

A C wide string is a reference to an array of C wide characters terminated by NUL.

`type CWStringLen = (Ptr CWchar, Int)`

A wide character string with explicit length information in `CWchar`s instead of a terminating NUL (allowing NUL characters in the middle of the string).

`peekCWString :: CWString -> IO String`

Marshal a NUL terminated C wide string into a Haskell string.

`peekCWStringLen :: CWStringLen -> IO String`

Marshal a C wide string with explicit length into a Haskell string.

`newCWString :: String -> IO CWString`

Marshal a Haskell string into a NUL terminated C wide string.

- the Haskell string may *not* contain any NUL characters
- new storage is allocated for the C wide string and must be explicitly freed using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree`.

`newCWStringLen :: String -> IO CWStringLen`

Marshal a Haskell string into a C wide string (ie, wide character array) with explicit length information.

- new storage is allocated for the C wide string and must be explicitly freed using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree`.

`withCWString :: String -> (CWString -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C wide string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`withCWStringLen :: String -> (CWStringLen -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C wide string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

Chapter 28

Foreign.C.Types

```
module Foreign.C.Types (
    CChar,  CSChar,  CUChar,  CShort,  CUShort,  CInt,  CUInt,  CLong,  CULong,
    CPtrdiff,  CSize,  CWchar,  CSigAtomic,  CLLong,  CULLong,  CIntPtr,
    CUIntPtr,  CIntMax,  CUIntMax,  CClock,  CTime,  CFloat,  CDouble,  CFile,
    CFpos,  CJmpBuf
  ) where
```

28.1 Representations of C types

These types are needed to accurately represent C function prototypes, in order to access C library interfaces in Haskell. The Haskell system is not required to represent those types exactly as C does, but the following guarantees are provided concerning a Haskell type `CT` representing a C type `t`:

- If a C function prototype has `t` as an argument or result type, the use of `CT` in the corresponding position in a foreign declaration permits the Haskell program to access the full range of values encoded by the C type; and conversely, any Haskell value for `CT` has a valid representation in C.
- `sizeOf (undefined :: CT)` will yield the same value as `sizeof (t)` in C.
- `alignment (undefined :: CT)` matches the alignment constraint enforced by the C implementation for `t`.
- The members `peek` and `poke` of the `Storable` class map all values of `CT` to the corresponding value of `t` and vice versa.
- When an instance of `Bounded` is defined for `CT`, the values of `minBound` and `maxBound` coincide with `t_MIN` and `t_MAX` in C.

- When an instance of `Eq` or `Ord` is defined for `CT`, the predicates defined by the type class implement the same relation as the corresponding predicate in C on `t`.
- When an instance of `Num`, `Read`, `Integral`, `Fractional`, `Floating`, `RealFrac`, or `RealFloat` is defined for `CT`, the arithmetic operations defined by the type class implement the same function as the corresponding arithmetic operations (if available) in C on `t`.
- When an instance of `Bits` is defined for `CT`, the bitwise operation defined by the type class implement the same function as the corresponding bitwise operation in C on `t`.

28.1.1 Integral types

These types are are represented as `newtype`s of types in `Data.Int` and `Data.Word`, and are instances of `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Storable`, `Bounded`, `Real`, `Integral` and `Bits`.

`data CChar`

Haskell type representing the C `char` type.

```
instance Bounded CChar
instance Enum CChar
instance Eq CChar
instance Integral CChar
instance Num CChar
instance Ord CChar
instance Read CChar
instance Real CChar
instance Show CChar
instance Storable CChar
instance Bits CChar
```

`data CSChar`

Haskell type representing the C `signed char` type.

```
instance Bounded CSChar
instance Enum CSChar
instance Eq CSChar
instance Integral CSChar
instance Num CSChar
instance Ord CSChar
instance Read CSChar
instance Real CSChar
instance Show CSChar
instance Storable CSChar
instance Bits CSChar
```

`data CUChar`

Haskell type representing the C `unsigned char` type.

```
instance Bounded CUChar
instance Enum CUChar
instance Eq CUChar
instance Integral CUChar
instance Num CUChar
instance Ord CUChar
instance Read CUChar
instance Real CUChar
instance Show CUChar
instance Storable CUChar
instance Bits CUChar
```

`data CShort`

Haskell type representing the C `short` type.

```
instance Bounded CShort
instance Enum CShort
instance Eq CShort
instance Integral CShort
instance Num CShort
instance Ord CShort
instance Read CShort
instance Real CShort
instance Show CShort
instance Storable CShort
instance Bits CShort
```

`data CUShort`

Haskell type representing the C `unsigned short` type.

```
instance Bounded CUShort
instance Enum CUShort
instance Eq CUShort
instance Integral CUShort
instance Num CUShort
instance Ord CUShort
instance Read CUShort
instance Real CUShort
instance Show CUShort
instance Storable CUShort
instance Bits CUShort
```

`data CInt`

Haskell type representing the C `int` type.

instance Bounded CInt
instance Enum CInt
instance Eq CInt
instance Integral CInt
instance Num CInt
instance Ord CInt
instance Read CInt
instance Real CInt
instance Show CInt
instance Storable CInt
instance Bits CInt

data CUInt

Haskell type representing the C `unsigned int` type.

instance Bounded CUInt
instance Enum CUInt
instance Eq CUInt
instance Integral CUInt
instance Num CUInt
instance Ord CUInt
instance Read CUInt
instance Real CUInt
instance Show CUInt
instance Storable CUInt
instance Bits CUInt

data CLong

Haskell type representing the C `long` type.

instance Bounded CLong
instance Enum CLong
instance Eq CLong
instance Integral CLong
instance Num CLong
instance Ord CLong
instance Read CLong
instance Real CLong
instance Show CLong
instance Storable CLong
instance Bits CLong

data CULong

Haskell type representing the C `unsigned long` type.

```
instance Bounded CULong
instance Enum CULong
instance Eq CULong
instance Integral CULong
instance Num CULong
instance Ord CULong
instance Read CULong
instance Real CULong
instance Show CULong
instance Storable CULong
instance Bits CULong
```

`data CPtrdiff`

Haskell type representing the C `ptrdiff_t` type.

```
instance Bounded CPtrdiff
instance Enum CPtrdiff
instance Eq CPtrdiff
instance Integral CPtrdiff
instance Num CPtrdiff
instance Ord CPtrdiff
instance Read CPtrdiff
instance Real CPtrdiff
instance Show CPtrdiff
instance Storable CPtrdiff
instance Bits CPtrdiff
```

`data CSize`

Haskell type representing the C `size_t` type.

```
instance Bounded CSize
instance Enum CSize
instance Eq CSize
instance Integral CSize
instance Num CSize
instance Ord CSize
instance Read CSize
instance Real CSize
instance Show CSize
instance Storable CSize
instance Bits CSize
```

`data CWchar`

Haskell type representing the C `wchar_t` type.

instance Bounded CWchar
instance Enum CWchar
instance Eq CWchar
instance Integral CWchar
instance Num CWchar
instance Ord CWchar
instance Read CWchar
instance Real CWchar
instance Show CWchar
instance Storable CWchar
instance Bits CWchar

data CSigAtomic

Haskell type representing the C `sig_atomic_t` type.

instance Bounded CSigAtomic
instance Enum CSigAtomic
instance Eq CSigAtomic
instance Integral CSigAtomic
instance Num CSigAtomic
instance Ord CSigAtomic
instance Read CSigAtomic
instance Real CSigAtomic
instance Show CSigAtomic
instance Storable CSigAtomic
instance Bits CSigAtomic

data CLLong

Haskell type representing the C `long long` type.

instance Bounded CLLong
instance Enum CLLong
instance Eq CLLong
instance Integral CLLong
instance Num CLLong
instance Ord CLLong
instance Read CLLong
instance Real CLLong
instance Show CLLong
instance Storable CLLong
instance Bits CLLong

data CULLong

Haskell type representing the C `unsigned long long` type.

```
instance Bounded CULLong
instance Enum CULLong
instance Eq CULLong
instance Integral CULLong
instance Num CULLong
instance Ord CULLong
instance Read CULLong
instance Real CULLong
instance Show CULLong
instance Storable CULLong
instance Bits CULLong
```

```
data CIntPtr
```

```
instance Bounded CIntPtr
instance Enum CIntPtr
instance Eq CIntPtr
instance Integral CIntPtr
instance Num CIntPtr
instance Ord CIntPtr
instance Read CIntPtr
instance Real CIntPtr
instance Show CIntPtr
instance Storable CIntPtr
instance Bits CIntPtr
```

```
data CUIntPtr
```

```
instance Bounded CUIntPtr
instance Enum CUIntPtr
instance Eq CUIntPtr
instance Integral CUIntPtr
instance Num CUIntPtr
instance Ord CUIntPtr
instance Read CUIntPtr
instance Real CUIntPtr
instance Show CUIntPtr
instance Storable CUIntPtr
instance Bits CUIntPtr
```

```
data CIntMax
```

```
instance Bounded CIntMax
instance Enum CIntMax
instance Eq CIntMax
instance Integral CIntMax
instance Num CIntMax
instance Ord CIntMax
instance Read CIntMax
instance Real CIntMax
instance Show CIntMax
instance Storable CIntMax
instance Bits CIntMax
```

data CUIntMax

instance Bounded CUIntMax
instance Enum CUIntMax
instance Eq CUIntMax
instance Integral CUIntMax
instance Num CUIntMax
instance Ord CUIntMax
instance Read CUIntMax
instance Real CUIntMax
instance Show CUIntMax
instance Storable CUIntMax
instance Bits CUIntMax

28.1.2 Numeric types

These types are are represented as `newtype`s of basic foreign types, and are instances of `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum` and `Storable`.

data CClock

Haskell type representing the C `clock_t` type.

instance Enum CClock
instance Eq CClock
instance Num CClock
instance Ord CClock
instance Read CClock
instance Real CClock
instance Show CClock
instance Storable CClock

data CTime

Haskell type representing the C `time_t` type.

instance Enum CTime
instance Eq CTime
instance Num CTime
instance Ord CTime
instance Read CTime
instance Real CTime
instance Show CTime
instance Storable CTime

28.1.3 Floating types

These types are are represented as `newtype`s of `Float` and `Double`, and are instances of `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Storable`, `Real`, `Fractional`, `Floating`, `RealFrac` and `RealFloat`.

data CFloat

Haskell type representing the C `float` type.

```
instance Enum CFloat
instance Eq CFloat
instance Floating CFloat
instance Fractional CFloat
instance Num CFloat
instance Ord CFloat
instance Read CFloat
instance Real CFloat
instance RealFloat CFloat
instance RealFrac CFloat
instance Show CFloat
instance Storable CFloat
```

data CDouble

Haskell type representing the C `double` type.

```
instance Enum CDouble
instance Eq CDouble
instance Floating CDouble
instance Fractional CDouble
instance Num CDouble
instance Ord CDouble
instance Read CDouble
instance Real CDouble
instance RealFloat CDouble
instance RealFrac CDouble
instance Show CDouble
instance Storable CDouble
```

28.1.4 Other types

data CFile

Haskell type representing the C `FILE` type.

data CFpos

Haskell type representing the C `fpos_t` type.

data CJmpBuf

Haskell type representing the C `jmp_buf` type.

Chapter 29

Foreign.ForeignPtr

```
module Foreign.ForeignPtr (
    ForeignPtr,  FinalizerPtr,  FinalizerEnvPtr,  newForeignPtr,
    newForeignPtr_,  addForeignPtrFinalizer,  newForeignPtrEnv,
    addForeignPtrFinalizerEnv,  withForeignPtr,  finalizeForeignPtr,
    unsafeForeignPtrToPtr,  touchForeignPtr,  castForeignPtr,
    mallocForeignPtr,  mallocForeignPtrBytes,  mallocForeignPtrArray,
    mallocForeignPtrArray0
  ) where
```

29.1 Finalised data pointers

`data ForeignPtr a`

The type `ForeignPtr` represents references to objects that are maintained in a foreign language, i.e., that are not part of the data structures usually managed by the Haskell storage manager. The essential difference between `ForeignPtr`s and vanilla memory references of type `Ptr a` is that the former may be associated with *finalizers*. A finalizer is a routine that is invoked when the Haskell storage manager detects that - within the Haskell heap and stack - there are no more references left that are pointing to the `ForeignPtr`. Typically, the finalizer will, then, invoke routines in the foreign language that free the resources bound by the foreign object.

The `ForeignPtr` is parameterised in the same way as `Ptr`. The type argument of `ForeignPtr` should normally be an instance of class `Storable`.

`instance Eq (ForeignPtr a)`
`instance Ord (ForeignPtr a)`
`instance Show (ForeignPtr a)`

`type FinalizerPtr a = FunPtr (Ptr a -> IO ())`

A finalizer is represented as a pointer to a foreign function that, at finalisation time, gets as an argument a plain pointer variant of the foreign pointer that the finalizer is associated with.

`type FinalizerEnvPtr env a = FunPtr (Ptr env -> Ptr a -> IO ())`

29.1.1 Basic operations

`newForeignPtr :: FinalizerPtr a -> Ptr a -> IO (ForeignPtr a)`

Turns a plain memory reference into a foreign pointer, and associates a finalizer with the reference. The finalizer will be executed after the last reference to the foreign object is dropped. There is no guarantee of promptness, however the finalizer will be executed before the program exits.

`newForeignPtr_ :: Ptr a -> IO (ForeignPtr a)`

Turns a plain memory reference into a foreign pointer that may be associated with finalizers by using `addForeignPtrFinalizer`.

`addForeignPtrFinalizer :: FinalizerPtr a -> ForeignPtr a -> IO ()`

This function adds a finalizer to the given foreign object. The finalizer will run *before* all other finalizers for the same object which have already been registered.

```
newForeignPtrEnv :: FinalizerEnvPtr env a
                 -> Ptr env -> Ptr a -> IO (ForeignPtr a)
```

This variant of `newForeignPtr` adds a finalizer that expects an environment in addition to the finalized pointer. The environment that will be passed to the finalizer is fixed by the second argument to `newForeignPtrEnv`.

```
addForeignPtrFinalizerEnv :: FinalizerEnvPtr env a
                          -> Ptr env -> ForeignPtr a -> IO ()
```

Like `addForeignPtrFinalizerEnv` but allows the finalizer to be passed an additional environment parameter to be passed to the finalizer. The environment passed to the finalizer is fixed by the second argument to `addForeignPtrFinalizerEnv`

`withForeignPtr :: ForeignPtr a -> (Ptr a -> IO b) -> IO b`

This is a way to look at the pointer living inside a foreign object. This function takes a function which is applied to that pointer. The resulting `IO` action is then executed. The foreign object is kept alive at least during the whole action, even if it is not used directly inside. Note that it is not safe to return the pointer from the action and use it after the action completes. All uses of the pointer should be inside the `withForeignPtr` bracket. The reason for this unsafeness is the same as for `unsafeForeignPtrToPtr` below: the finalizer may run earlier than expected, because the compiler can only track usage of the `ForeignPtr` object, not a `Ptr` object made from it.

This function is normally used for marshalling data to or from the object pointed to by the `ForeignPtr`, using the operations from the `Storable` class.

`finalizeForeignPtr :: ForeignPtr a -> IO ()`

Causes the finalizers associated with a foreign pointer to be run immediately.

29.1.2 Low-level operations

`unsafeForeignPtrToPtr :: ForeignPtr a -> Ptr a`

This function extracts the pointer component of a foreign pointer. This is a potentially dangerous operations, as if the argument to `unsafeForeignPtrToPtr` is the last usage occurrence of the given foreign pointer, then its finalizer(s) will be run, which potentially invalidates the plain pointer just obtained. Hence, `touchForeignPtr` must be used wherever it has to be guaranteed that the pointer lives on - i.e., has another usage occurrence.

To avoid subtle coding errors, hand written marshalling code should preferably use `Foreign.ForeignPtr.withForeignPtr` rather than combinations of `unsafeForeignPtrToPtr` and `touchForeignPtr`. However, the latter routines are occasionally preferred in tool generated marshalling code.

`touchForeignPtr :: ForeignPtr a -> IO ()`

This function ensures that the foreign object in question is alive at the given place in the sequence of IO actions. In particular `withForeignPtr` does a `touchForeignPtr` after it executes the user action.

Note that this function should not be used to express dependencies between finalizers on `ForeignPtr`s. For example, if the finalizer for a `ForeignPtr F1` calls `touchForeignPtr` on a second `ForeignPtr F2`, then the only guarantee is that the finalizer for `F2` is never started before the finalizer for `F1`. They might be started together if for example both `F1` and `F2` are otherwise unreachable.

In general, it is not recommended to use finalizers on separate objects with ordering constraints between them. To express the ordering robustly requires explicit synchronisation between finalizers.

`castForeignPtr :: ForeignPtr a -> ForeignPtr b`

This function casts a `ForeignPtr` parameterised by one type into another type.

29.1.3 Allocating managed memory

`mallocForeignPtr :: Storable a => IO (ForeignPtr a)`

Allocate some memory and return a `ForeignPtr` to it. The memory will be released automatically when the `ForeignPtr` is discarded.

`mallocForeignPtr` is equivalent to

```
do { p <- malloc; newForeignPtr finalizerFree p }
```

although it may be implemented differently internally: you may not assume that the memory returned by `mallocForeignPtr` has been allocated with `Foreign.Marshal.Alloc.malloc`.

`mallocForeignPtrBytes :: Int -> IO (ForeignPtr a)`

This function is similar to `mallocForeignPtr`, except that the size of the memory required is given explicitly as a number of bytes.

`mallocForeignPtrArray :: Storable a => Int -> IO (ForeignPtr a)`

This function is similar to `Foreign.Marshal.Array.mallocArray`, but yields a memory area that has a finalizer attached that releases the memory area. As with `mallocForeignPtr`, it is not guaranteed that the block of memory was allocated by `Foreign.Marshal.Alloc.malloc`.

`mallocForeignPtrArray0 :: Storable a => Int -> IO (ForeignPtr a)`

This function is similar to `Foreign.Marshal.Array.mallocArray0`, but yields a memory area that has a finalizer attached that releases the memory area. As with `mallocForeignPtr`, it is not guaranteed that the block of memory was allocated by `Foreign.Marshal.Alloc.malloc`.

Chapter 30

`Foreign.Marshal`

```
module Foreign.Marshal (
    module Foreign.Marshal.Alloc,  module Foreign.Marshal.Array,
    module Foreign.Marshal.Error,  module Foreign.Marshal.Utils,
    unsafeLocalState
  ) where
```

The module `Foreign.Marshal` re-exports the other modules in the `Foreign.Marshal` hierarchy:

`module Foreign.Marshal.Alloc`
`module Foreign.Marshal.Array`
`module Foreign.Marshal.Error`
`module Foreign.Marshal.Utils`

and provides one function:

`unsafeLocalState :: IO a -> a`

> Sometimes an external entity is a pure function, except that it passes arguments and/or results via pointers. The function `unsafeLocalState` permits the packaging of such entities as pure functions.
>
> The only IO operations allowed in the IO action passed to `unsafeLocalState` are (a) local allocation (`alloca`, `allocaBytes` and derived operations such as `withArray` and `withCString`), and (b) pointer operations (`Foreign.Storable` and `Foreign.Ptr`) on the pointers to local storage, and (c) foreign functions whose only observable effect is to read and/or write the locally allocated memory. Passing an IO operation that does not obey these rules results in undefined behaviour.
>
> It is expected that this operation will be replaced in a future revision of Haskell.

Chapter 31

`Foreign.Marshal.Alloc`

```
module Foreign.Marshal.Alloc (
    alloca,  allocaBytes,  malloc,  mallocBytes,  realloc,  reallocBytes,
    free,  finalizerFree
  ) where
```

The module `Foreign.Marshal.Alloc` provides operations to allocate and deallocate blocks of raw memory (i.e., unstructured chunks of memory outside of the area maintained by the Haskell storage manager). These memory blocks are commonly used to pass compound data structures to foreign functions or to provide space in which compound result values are obtained from foreign functions.

If any of the allocation functions fails, a value of `nullPtr` is produced. If `free` or `reallocBytes` is applied to a memory area that has been allocated with `alloca` or `allocaBytes`, the behaviour is undefined. Any further access to memory areas allocated with `alloca` or `allocaBytes`, after the computation that was passed to the allocation function has terminated, leads to undefined behaviour. Any further access to the memory area referenced by a pointer passed to `realloc`, `reallocBytes`, or `free` entails undefined behaviour.

All storage allocated by functions that allocate based on a *size in bytes* must be sufficiently aligned for any of the basic foreign types that fits into the newly allocated storage. All storage allocated by functions that allocate based on a specific type must be sufficiently aligned for that type. Array allocation routines need to obey the same alignment constraints for each array element.

31.1 Memory allocation

31.1.1 Local allocation

`alloca :: Storable a => (Ptr a -> IO b) -> IO b`

`alloca f` executes the computation `f`, passing as argument a pointer to a temporarily allocated block of memory sufficient to hold values of type `a`.

The memory is freed when `f` terminates (either normally or via an exception), so the pointer passed to `f` must *not* be used after this.

`allocaBytes :: Int -> (Ptr a -> IO b) -> IO b`

`allocaBytes n f` executes the computation `f`, passing as argument a pointer to a temporarily allocated block of memory of `n` bytes. The block of memory is sufficiently aligned for any of the basic foreign types that fits into a memory block of the allocated size.

The memory is freed when `f` terminates (either normally or via an exception), so the pointer passed to `f` must *not* be used after this.

31.1.2 Dynamic allocation

`malloc :: Storable a => IO (Ptr a)`

Allocate a block of memory that is sufficient to hold values of type `a`. The size of the area allocated is determined by the `sizeOf` method from the instance of `Storable` for the appropriate type.

The memory may be deallocated using `free` or `finalizerFree` when no longer required.

`mallocBytes :: Int -> IO (Ptr a)`

Allocate a block of memory of the given number of bytes. The block of memory is sufficiently aligned for any of the basic foreign types that fits into a memory block of the allocated size.

The memory may be deallocated using `free` or `finalizerFree` when no longer required.

`realloc :: Storable b => Ptr a -> IO (Ptr b)`

Resize a memory area that was allocated with `malloc` or `mallocBytes` to the size needed to store values of type `b`. The returned pointer may refer to an entirely different memory area, but will be suitably aligned to hold values of type `b`. The contents of the referenced memory area will be the same as of the original pointer up to the minimum of the original size and the size of values of type `b`.

If the argument to `realloc` is `nullPtr`, `realloc` behaves like `malloc`.

`reallocBytes :: Ptr a -> Int -> IO (Ptr a)`

Resize a memory area that was allocated with `malloc` or `mallocBytes` to the given size. The returned pointer may refer to an entirely different memory area, but will be sufficiently aligned for any of the basic foreign types that fits into a memory block of the given size. The contents of the referenced memory area will be the same as of the original pointer up to the minimum of the original size and the given size.

If the pointer argument to `reallocBytes` is `nullPtr`, `reallocBytes` behaves like `malloc`. If the requested size is 0, `reallocBytes` behaves like `free`.

`free :: Ptr a -> IO ()`

Free a block of memory that was allocated with `malloc`, `mallocBytes`, `realloc`, `reallocBytes`, `Foreign.Marshal.Utils.new` or any of the `new`*X* functions in `Foreign.Marshal.Array` or `Foreign.C.String`.

`finalizerFree :: FinalizerPtr a`

A pointer to a foreign function equivalent to `free`, which may be used as a finalizer (cf `Foreign.ForeignPtr.ForeignPtr`) for storage allocated with `malloc`, `mallocBytes`, `realloc` or `reallocBytes`.

Chapter 32

Foreign.Marshal.Array

```
module Foreign.Marshal.Array (
    mallocArray,  mallocArray0,  allocaArray,  allocaArray0,  reallocArray,
    reallocArray0,  peekArray,  peekArray0,  pokeArray,  pokeArray0,  newArray,
    newArray0,  withArray,  withArray0,  withArrayLen,  withArrayLen0,
    copyArray,  moveArray,  lengthArray0,  advancePtr
  ) where
```

The module `Foreign.Marshal.Array` provides operations for marshalling Haskell lists into monolithic arrays and vice versa. Most functions come in two flavours: one for arrays terminated by a special termination element and one where an explicit length parameter is used to determine the extent of an array. The typical example for the former case are C's NUL terminated strings. However, please note that C strings should usually be marshalled using the functions provided by `Foreign.C.String` as the Unicode encoding has to be taken into account. All functions specifically operating on arrays that are terminated by a special termination element have a name ending on `0`—e.g., `mallocArray` allocates space for an array of the given size, whereas `mallocArray0` allocates space for one more element to ensure that there is room for the terminator.

32.1 Marshalling arrays

32.1.1 Allocation

`mallocArray :: Storable a => Int -> IO (Ptr a)`

Allocate storage for the given number of elements of a storable type (like `Foreign.Marshal.Alloc.malloc`, but for multiple elements).

`mallocArray0 :: Storable a => Int -> IO (Ptr a)`

Like `mallocArray`, but add an extra position to hold a special termination element.

`allocaArray :: Storable a => Int -> (Ptr a -> IO b) -> IO b`

Temporarily allocate space for the given number of elements (like `Foreign.Marshal.Alloc.alloca`, but for multiple elements).

`allocaArray0 :: Storable a => Int -> (Ptr a -> IO b) -> IO b`

Like `allocaArray`, but add an extra position to hold a special termination element.

`reallocArray :: Storable a => Ptr a -> Int -> IO (Ptr a)`

Adjust the size of an array

`reallocArray0 :: Storable a => Ptr a -> Int -> IO (Ptr a)`

Adjust the size of an array including an extra position for the end marker.

32.1.2 Marshalling

`peekArray :: Storable a => Int -> Ptr a -> IO [a]`

Convert an array of given length into a Haskell list.

`peekArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO [a]`

Convert an array terminated by the given end marker into a Haskell list

`pokeArray :: Storable a => Ptr a -> [a] -> IO ()`

Write the list elements consecutive into memory

`pokeArray0 :: Storable a => a -> Ptr a -> [a] -> IO ()`

Write the list elements consecutive into memory and terminate them with the given marker element

32.1.3 Combined allocation and marshalling

`newArray :: Storable a => [a] -> IO (Ptr a)`

Write a list of storable elements into a newly allocated, consecutive sequence of storable values (like `Foreign.Marshal.Utils.new`, but for multiple elements).

`newArray0 :: Storable a => a -> [a] -> IO (Ptr a)`

Write a list of storable elements into a newly allocated, consecutive sequence of storable values, where the end is fixed by the given end marker

`withArray :: Storable a => [a] -> (Ptr a -> IO b) -> IO b`

Temporarily store a list of storable values in memory (like `Foreign.Marshal.Utils.with`, but for multiple elements).

`withArray0 :: Storable a => a -> [a] -> (Ptr a -> IO b) -> IO b`

Like `withArray`, but a terminator indicates where the array ends

`withArrayLen :: Storable a => [a] -> (Int -> Ptr a -> IO b) -> IO b`

Like `withArray`, but the action gets the number of values as an additional parameter

```
withArrayLen0 :: Storable a => a
                              -> [a] -> (Int -> Ptr a -> IO b) -> IO b
```

Like `withArrayLen`, but a terminator indicates where the array ends

32.1.4 Copying

(argument order: destination, source)

`copyArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()`

Copy the given number of elements from the second array (source) into the first array (destination); the copied areas may *not* overlap

`moveArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()`

Copy the given number of elements from the second array (source) into the first array (destination); the copied areas *may* overlap

32.1.5 Finding the length

`lengthArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO Int`

Return the number of elements in an array, excluding the terminator

32.1.6 Indexing

`advancePtr :: Storable a => Ptr a -> Int -> Ptr a`

Advance a pointer into an array by the given number of elements

Chapter 33

`Foreign.Marshal.Error`

```
module Foreign.Marshal.Error (
    throwIf,  throwIf_,  throwIfNeg,  throwIfNeg_,  throwIfNull,  void
  ) where
```

`throwIf`

`::`	**`(a -> Bool)`**	error condition on the result of the `IO` action
`->`	**`(a -> String)`**	computes an error message from erroneous results of the `IO` action
`->`	**`IO a`**	the `IO` action to be executed
`->`	**`IO a`**	

Execute an `IO` action, throwing a `userError` if the predicate yields `True` when applied to the result returned by the `IO` action. If no exception is raised, return the result of the computation.

`throwIf_ :: (a -> Bool) -> (a -> String) -> IO a -> IO ()`

Like `throwIf`, but discarding the result

`throwIfNeg :: (Ord a, Num a) => (a -> String) -> IO a -> IO a`

Guards against negative result values

`throwIfNeg_ :: (Ord a, Num a) => (a -> String) -> IO a -> IO ()`

Like `throwIfNeg`, but discarding the result

`throwIfNull :: String -> IO (Ptr a) -> IO (Ptr a)`

Guards against null pointers

`void :: IO a -> IO ()`

Discard the return value of an `IO` action

Chapter 34

`Foreign.Marshal.Utils`

```
module Foreign.Marshal.Utils (
    with,  new,  fromBool,  toBool,  maybeNew,  maybeWith,  maybePeek,
    withMany,  copyBytes,  moveBytes
  ) where
```

34.1 General marshalling utilities

34.1.1 Combined allocation and marshalling

`with :: Storable a => a -> (Ptr a -> IO b) -> IO b`

`with val f` executes the computation `f`, passing as argument a pointer to a temporarily allocated block of memory into which `val` has been marshalled (the combination of `alloca` and `poke`).

The memory is freed when `f` terminates (either normally or via an exception), so the pointer passed to `f` must *not* be used after this.

`new :: Storable a => a -> IO (Ptr a)`

Allocate a block of memory and marshal a value into it (the combination of `malloc` and `poke`). The size of the area allocated is determined by the `Foreign.Storable.sizeOf` method from the instance of `Storable` for the appropriate type.

The memory may be deallocated using `Foreign.Marshal.Alloc.free` or `Foreign.Marshal.Alloc.finalizerFree` when no longer required.

34.1.2 Marshalling of Boolean values (non-zero corresponds to `True`)

`fromBool :: Num a => Bool -> a`

Convert a Haskell `Bool` to its numeric representation

`toBool :: Num a => a -> Bool`

Convert a Boolean in numeric representation to a Haskell value

34.1.3 Marshalling of Maybe values

`maybeNew :: (a -> IO (Ptr a)) -> Maybe a -> IO (Ptr a)`

Allocate storage and marshal a storable value wrapped into a `Maybe`

- the `nullPtr` is used to represent `Nothing`

```
maybeWith :: (a -> (Ptr b -> IO c) -> IO c)
             -> Maybe a -> (Ptr b -> IO c) -> IO c
```

Converts a `withXXX` combinator into one marshalling a value wrapped into a `Maybe`, using `nullPtr` to represent `Nothing`.

`maybePeek :: (Ptr a -> IO b) -> Ptr a -> IO (Maybe b)`

Convert a peek combinator into a one returning `Nothing` if applied to a `nullPtr`

34.1.4 Marshalling lists of storable objects

`withMany :: (a -> (b -> res) -> res) -> [a] -> ([b] -> res) -> res`

Replicates a `withXXX` combinator over a list of objects, yielding a list of marshalled objects

34.1.5 Haskellish interface to memcpy and memmove

(argument order: destination, source)

`copyBytes :: Ptr a -> Ptr a -> Int -> IO ()`

Copies the given number of bytes from the second area (source) into the first (destination); the copied areas may *not* overlap

`moveBytes :: Ptr a -> Ptr a -> Int -> IO ()`

Copies the given number of bytes from the second area (source) into the first (destination); the copied areas *may* overlap

Chapter 35

`Foreign.Ptr`

```
module Foreign.Ptr (
    Ptr,  nullPtr,  castPtr,  plusPtr,  alignPtr,  minusPtr,  FunPtr,
    nullFunPtr,  castFunPtr,  castFunPtrToPtr,  castPtrToFunPtr,
    freeHaskellFunPtr,  IntPtr,  ptrToIntPtr,  intPtrToPtr,  WordPtr,
    ptrToWordPtr,  wordPtrToPtr
  ) where
```

The module `Foreign.Ptr` provides typed pointers to foreign entities. We distinguish two kinds of pointers: pointers to data and pointers to functions. It is understood that these two kinds of pointers may be represented differently as they may be references to data and text segments, respectively.

35.1 Data pointers

`data Ptr a`

> A value of type `Ptr a` represents a pointer to an object, or an array of objects, which may be marshalled to or from Haskell values of type `a`.
>
> The type `a` will often be an instance of class `Foreign.Storable.Storable` which provides the marshalling operations. However this is not essential, and you can provide your own operations to access the pointer. For example you might write small foreign functions to get or set the fields of a C `struct`.

`instance Eq (Ptr a)`
`instance Ord (Ptr a)`
`instance Show (Ptr a)`
`instance Storable (Ptr a)`

`nullPtr :: Ptr a`

The constant `nullPtr` contains a distinguished value of `Ptr` that is not associated with a valid memory location.

`castPtr :: Ptr a -> Ptr b`

The `castPtr` function casts a pointer from one type to another.

`plusPtr :: Ptr a -> Int -> Ptr b`

Advances the given address by the given offset in bytes.

`alignPtr :: Ptr a -> Int -> Ptr a`

Given an arbitrary address and an alignment constraint, `alignPtr` yields the next higher address that fulfills the alignment constraint. An alignment constraint `x` is fulfilled by any address divisible by `x`. This operation is idempotent.

`minusPtr :: Ptr a -> Ptr b -> Int`

Computes the offset required to get from the second to the first argument. We have

```
p2 == p1 `plusPtr` (p2 `minusPtr` p1)
```

35.2 Function pointers

`data FunPtr a`

A value of type `FunPtr a` is a pointer to a function callable from foreign code. The type `a` will normally be a *foreign type*, a function type with zero or more arguments where

- the argument types are *marshallable foreign types*, i.e. `Char`, `Int`, `Double`, `Float`, `Bool`, `Data.Int.Int8`, `Data.Int.Int16`, `Data.Int.Int32`, `Data.Int.Int64`, `Data.Word.Word8`, `Data.Word.Word16`, `Data.Word.Word32`, `Data.Word.Word64`, `Ptr a`, `FunPtr a`, `Foreign.StablePtr.StablePtr a` or a renaming of any of these using `newtype`.
- the return type is either a marshallable foreign type or has the form `IO t` where `t` is a marshallable foreign type or `()`.

A value of type `FunPtr a` may be a pointer to a foreign function, either returned by another foreign function or imported with a a static address import like

```
foreign import ccall "stdlib.h &free"
  p_free :: FunPtr (Ptr a -> IO ())
```

or a pointer to a Haskell function created using a *wrapper* stub declared to produce a `FunPtr` of the correct type. For example:

```
type Compare = Int -> Int -> Bool
foreign import ccall "wrapper"
  mkCompare :: Compare -> IO (FunPtr Compare)
```

Calls to wrapper stubs like `mkCompare` allocate storage, which should be released with `Foreign.Ptr.freeHaskellFunPtr` when no longer required.

To convert `FunPtr` values to corresponding Haskell functions, one can define a *dynamic* stub for the specific foreign type, e.g.

```
type IntFunction = CInt -> IO ()
foreign import ccall "dynamic"
  mkFun :: FunPtr IntFunction -> IntFunction
```

`instance Eq (FunPtr a)`
`instance Ord (FunPtr a)`
`instance Show (FunPtr a)`
`instance Storable (FunPtr a)`

`nullFunPtr :: FunPtr a`

The constant `nullFunPtr` contains a distinguished value of `FunPtr` that is not associated with a valid memory location.

`castFunPtr :: FunPtr a -> FunPtr b`

Casts a `FunPtr` to a `FunPtr` of a different type.

`castFunPtrToPtr :: FunPtr a -> Ptr b`

Casts a `FunPtr` to a `Ptr`.

Note: this is valid only on architectures where data and function pointers range over the same set of addresses, and should only be used for bindings to external libraries whose interface already relies on this assumption.

`castPtrToFunPtr :: Ptr a -> FunPtr b`

Casts a `Ptr` to a `FunPtr`.

Note: this is valid only on architectures where data and function pointers range over the same set of addresses, and should only be used for bindings to external libraries whose interface already relies on this assumption.

`freeHaskellFunPtr :: FunPtr a -> IO ()`

Release the storage associated with the given `FunPtr`, which must have been obtained from a wrapper stub. This should be called whenever the return value from a foreign import wrapper function is no longer required; otherwise, the storage it uses will leak.

35.3 Integral types with lossless conversion to and from pointers

`data IntPtr`

A signed integral type that can be losslessly converted to and from `Ptr`. This type is also compatible with the C99 type `intptr_t`, and can be marshalled to and from that type safely.

instance Bounded IntPtr
instance Enum IntPtr
instance Eq IntPtr
instance Integral IntPtr
instance Num IntPtr
instance Ord IntPtr
instance Read IntPtr
instance Real IntPtr
instance Show IntPtr
instance Storable IntPtr
instance Bits IntPtr

ptrToIntPtr :: Ptr a -> IntPtr

casts a `Ptr` to an `IntPtr`

intPtrToPtr :: IntPtr -> Ptr a

casts an `IntPtr` to a `Ptr`

data WordPtr

An unsigned integral type that can be losslessly converted to and from `Ptr`. This type is also compatible with the C99 type `uintptr_t`, and can be marshalled to and from that type safely.

instance Bounded WordPtr
instance Enum WordPtr
instance Eq WordPtr
instance Integral WordPtr
instance Num WordPtr
instance Ord WordPtr
instance Read WordPtr
instance Real WordPtr
instance Show WordPtr
instance Storable WordPtr
instance Bits WordPtr

ptrToWordPtr :: Ptr a -> WordPtr

casts a `Ptr` to a `WordPtr`

wordPtrToPtr :: WordPtr -> Ptr a

casts a `WordPtr` to a `Ptr`

Chapter 36

`Foreign.StablePtr`

```
module Foreign.StablePtr (
    StablePtr,  newStablePtr,  deRefStablePtr,  freeStablePtr,
    castStablePtrToPtr,  castPtrToStablePtr
  ) where
```

36.1 Stable references to Haskell values

`data StablePtr a`

A *stable pointer* is a reference to a Haskell expression that is guaranteed not to be affected by garbage collection, i.e., it will neither be deallocated nor will the value of the stable pointer itself change during garbage collection (ordinary references may be relocated during garbage collection). Consequently, stable pointers can be passed to foreign code, which can treat it as an opaque reference to a Haskell value.

A value of type `StablePtr a` is a stable pointer to a Haskell expression of type `a`.

`instance Eq (StablePtr a)`
`instance Storable (StablePtr a)`

`newStablePtr :: a -> IO (StablePtr a)`

Create a stable pointer referring to the given Haskell value.

`deRefStablePtr :: StablePtr a -> IO a`

Obtain the Haskell value referenced by a stable pointer, i.e., the same value that was passed to the corresponding call to `makeStablePtr`. If the argument to `deRefStablePtr` has already been freed using `freeStablePtr`, the behaviour of `deRefStablePtr` is undefined.

`freeStablePtr :: StablePtr a -> IO ()`

Dissolve the association between the stable pointer and the Haskell value. Afterwards, if the stable pointer is passed to `deRefStablePtr` or `freeStablePtr`, the behaviour is undefined. However, the stable pointer may still be passed to `castStablePtrToPtr`, but the `Foreign.Ptr.Ptr ()` value returned by `castStablePtrToPtr`, in this case, is undefined (in particular, it may be `Foreign.Ptr.nullPtr`). Nevertheless, the call to `castStablePtrToPtr` is guaranteed not to diverge.

`castStablePtrToPtr :: StablePtr a -> Ptr ()`

Coerce a stable pointer to an address. No guarantees are made about the resulting value, except that the original stable pointer can be recovered by `castPtrToStablePtr`. In particular, the address may not refer to an accessible memory location and any attempt to pass it to the member functions of the class `Foreign.Storable.Storable` leads to undefined behaviour.

`castPtrToStablePtr :: Ptr () -> StablePtr a`

The inverse of `castStablePtrToPtr`, i.e., we have the identity

```
sp == castPtrToStablePtr (castStablePtrToPtr sp)
```

for any stable pointer `sp` on which `freeStablePtr` has not been executed yet. Moreover, `castPtrToStablePtr` may only be applied to pointers that have been produced by `castStablePtrToPtr`.

36.1.1 The C-side interface

The following definition is available to C programs inter-operating with Haskell code when including the header `HsFFI.h`.

```
typedef void *HsStablePtr;
```

Note that no assumptions may be made about the values representing stable pointers. In fact, they need not even be valid memory addresses. The only guarantee provided is that if they are passed back to Haskell land, the function `deRefStablePtr` will be able to reconstruct the Haskell value referred to by the stable pointer.

Chapter 37

Foreign.Storable

```
module Foreign.Storable (
    Storable(sizeOf,
             alignment,
             peekElemOff,
             pokeElemOff,
             peekByteOff,
             pokeByteOff,
             peek,
             poke)
  ) where
```

`class Storable a where`

The member functions of this class facilitate writing values of primitive types to raw memory (which may have been allocated with the above mentioned routines) and reading values from blocks of raw memory. The class, furthermore, includes support for computing the storage requirements and alignment restrictions of storable types.

Memory addresses are represented as values of type `Ptr a`, for some `a` which is an instance of class `Storable`. The type argument to `Ptr` helps provide some valuable type safety in FFI code (you can't mix pointers of different types without an explicit cast), while helping the Haskell type system figure out which marshalling method is needed for a given pointer.

All marshalling between Haskell and a foreign language ultimately boils down to translating Haskell data structures into the binary representation of a corresponding data structure of the foreign language and vice versa. To code this marshalling in Haskell, it is necessary to manipulate primitive data types stored in unstructured memory blocks. The class `Storable` facilitates this manipulation on all types for which it is instantiated, which are the standard basic types of Haskell, the fixed size `Int` types (`Int8`, `Int16`, `Int32`, `Int64`), the fixed size `Word` types (`Word8`, `Word16`, `Word32`, `Word64`), `StablePtr`, all types from `Foreign.C.Types`, as well as `Ptr`.

Minimal complete definition: `sizeOf`, `alignment`, one of `peek`, `peekElemOff` and `peekByteOff`, and one of `poke`, `pokeElemOff` and `pokeByteOff`.

Methods

`sizeOf :: a -> Int`

Computes the storage requirements (in bytes) of the argument. The value of the argument is not used.

`alignment :: a -> Int`

Computes the alignment constraint of the argument. An alignment constraint `x` is fulfilled by any address divisible by `x`. The value of the argument is not used.

`peekElemOff :: Ptr a -> Int -> IO a`

Read a value from a memory area regarded as an array of values of the same kind. The first argument specifies the start address of the array and the second the index into the array (the first element of the array has index `0`). The following equality holds,

```
peekElemOff addr idx = IOExts.fixIO $ \result ->
  peek (addr `plusPtr` (idx * sizeOf result))
```

Note that this is only a specification, not necessarily the concrete implementation of the function.

`pokeElemOff :: Ptr a -> Int -> a -> IO ()`

Write a value to a memory area regarded as an array of values of the same kind. The following equality holds:

```
pokeElemOff addr idx x =
  poke (addr `plusPtr` (idx * sizeOf x)) x
```

`peekByteOff :: Ptr b -> Int -> IO a`

Read a value from a memory location given by a base address and offset. The following equality holds:

```
peekByteOff addr off = peek (addr `plusPtr` off)
```

`pokeByteOff :: Ptr b -> Int -> a -> IO ()`

Write a value to a memory location given by a base address and offset. The following equality holds:

```
pokeByteOff addr off x = poke (addr `plusPtr` off) x
```

`peek :: Ptr a -> IO a`

Read a value from the given memory location.

Note that the peek and poke functions might require properly aligned addresses to function correctly. This is architecture dependent; thus, portable code should ensure that when peeking or poking values of some type `a`, the alignment constraint for `a`, as given by the function `alignment` is fulfilled.

`poke :: Ptr a -> a -> IO ()`

Write the given value to the given memory location. Alignment restrictions might apply; see `peek`.

```
instance Storable Bool
instance Storable Char
instance Storable Double
instance Storable Float
instance Storable Int
instance Storable Int8
instance Storable Int16
instance Storable Int32
instance Storable Int64
instance Storable Word
instance Storable Word8
instance Storable Word16
instance Storable Word32
instance Storable Word64
instance Storable WordPtr
instance Storable IntPtr
instance Storable CChar
instance Storable CSChar
instance Storable CUChar
instance Storable CShort
instance Storable CUShort
instance Storable CInt
instance Storable CUInt
instance Storable CLong
instance Storable CULong
instance Storable CLLong
instance Storable CULLong
instance Storable CFloat
instance Storable CDouble
instance Storable CPtrdiff
instance Storable CSize
instance Storable CWchar
instance Storable CSigAtomic
instance Storable CClock
instance Storable CTime
instance Storable CIntPtr
instance Storable CUIntPtr
instance Storable CIntMax
instance Storable CUIntMax
instance Storable (StablePtr a)
instance Storable (Ptr a)
instance Storable (FunPtr a)
```

Chapter 38

Numeric

```
module Numeric (
    showSigned,  showIntAtBase,  showInt,  showHex,  showOct,  showEFloat,
    showFFloat,  showGFloat,  showFloat,  floatToDigits,  readSigned,  readInt,
    readDec,  readOct,  readHex,  readFloat,  lexDigits,  fromRat
  ) where
```

38.1 Showing

showSigned

::	**Real a**	
=>	**(a -> ShowS)**	a function that can show unsigned values
->	**Int**	the precedence of the enclosing context
->	**a**	the value to show
->	**ShowS**	

Converts a possibly-negative `Real` value to a string.

showIntAtBase :: Integral a => a -> (Int -> Char) -> a -> ShowS

Shows a *non-negative* `Integral` number using the base specified by the first argument, and the character representation specified by the second.

showInt :: Integral a => a -> ShowS

Show *non-negative* `Integral` numbers in base 10.

showHex :: Integral a => a -> ShowS

Show *non-negative* `Integral` numbers in base 16.

showOct :: Integral a => a -> ShowS

Show *non-negative* `Integral` numbers in base 8.

showEFloat :: RealFloat a => Maybe Int -> a -> ShowS

Show a signed `RealFloat` value using scientific (exponential) notation (e.g. `2.45e2`, `1.5e-3`).

In the call `showEFloat digs val`, if `digs` is `Nothing`, the value is shown to full precision; if `digs` is `Just d`, then at most `d` digits after the decimal point are shown.

showFFloat :: RealFloat a => Maybe Int -> a -> ShowS

Show a signed `RealFloat` value using standard decimal notation (e.g. `245000`, `0.0015`).

In the call `showFFloat digs val`, if `digs` is `Nothing`, the value is shown to full precision; if `digs` is `Just d`, then at most `d` digits after the decimal point are shown.

showGFloat :: RealFloat a => Maybe Int -> a -> ShowS

Show a signed `RealFloat` value using standard decimal notation for arguments whose absolute value lies between `0.1` and `9,999,999`, and scientific notation otherwise.

In the call `showGFloat digs val`, if `digs` is `Nothing`, the value is shown to full precision; if `digs` is `Just d`, then at most `d` digits after the decimal point are shown.

showFloat :: RealFloat a => a -> ShowS

Show a signed `RealFloat` value to full precision using standard decimal notation for arguments whose absolute value lies between `0.1` and `9,999,999`, and scientific notation otherwise.

floatToDigits :: RealFloat a => Integer -> a -> ([Int], Int)

`floatToDigits` takes a base and a non-negative `RealFloat` number, and returns a list of digits and an exponent. In particular, if `x>=0`, and

```
floatToDigits base x = ([d1,d2,...,dn], e)
```

then

1. `n >= 1`
2. `x = 0.d1d2...dn * (base**e)`
3. `0 <= di <= base-1`

38.2 Reading

NB: `readInt` is the ’dual’ of `showIntAtBase`, and `readDec` is the ‘dual’ of `showInt`. The inconsistent naming is a historical accident.

readSigned :: Real a => ReadS a -> ReadS a

Reads a *signed* `Real` value, given a reader for an unsigned value.

`readInt`

`::`	`Num a`	
`=>`	`a`	the base
`->`	`(Char -> Bool)`	a predicate distinguishing valid digits in this base
`->`	`(Char -> Int)`	a function converting a valid digit character to an `Int`
`->`	`ReadS a`	

Reads an *unsigned* `Integral` value in an arbitrary base.

`readDec :: Num a => ReadS a`

Read an unsigned number in decimal notation.

`readOct :: Num a => ReadS a`

Read an unsigned number in octal notation.

`readHex :: Num a => ReadS a`

Read an unsigned number in hexadecimal notation. Both upper or lower case letters are allowed.

`readFloat :: RealFrac a => ReadS a`

Reads an *unsigned* `RealFrac` value, expressed in decimal scientific notation.

`lexDigits :: ReadS String`

Reads a non-empty string of decimal digits.

38.3 Miscellaneous

`fromRat :: RealFloat a => Rational -> a`

Converts a `Rational` value into any type in class `RealFloat`.

Chapter 39

System.Environment

```
module System.Environment (
    getArgs,  getProgName,  getEnv
  ) where
```

---

getArgs :: IO [String]

Computation `getArgs` returns a list of the program's command line arguments (not including the program name).

getProgName :: IO String

Computation `getProgName` returns the name of the program as it was invoked.

However, this is hard-to-impossible to implement on some non-Unix OSes, so instead, for maximum portability, we just return the leafname of the program as invoked. Even then there are some differences between platforms: on Windows, for example, a program invoked as foo is probably really `FOO.EXE`, and that is what `getProgName` will return.

getEnv :: String -> IO String

Computation `getEnv` `var` returns the value of the environment variable `var`.

This computation may fail with:

- `System.IO.Error.isDoesNotExistError` if the environment variable does not exist.

Chapter 40

`System.Exit`

```
module System.Exit (
    ExitCode(ExitSuccess, ExitFailure),  exitWith,  exitFailure,  exitSuccess
  ) where
```

`data ExitCode`

`=`	**`ExitSuccess`**	indicates successful termination;
`|`	**`ExitFailure Int`**	indicates program failure with an exit code. The exact interpretation of the code is operating-system dependent. In particular, some values may be prohibited (e.g. 0 on a POSIX-compliant system).

Defines the exit codes that a program can return.

`instance Eq ExitCode`
`instance Ord ExitCode`
`instance Read ExitCode`
`instance Show ExitCode`

`exitWith :: ExitCode -> IO a`

Computation `exitWith code` terminates the program, returning `code` to the program's caller. The caller may interpret the return code as it wishes, but the program should return `ExitSuccess` to mean normal completion, and `ExitFailure n` to mean that the program encountered a problem from which it could not recover. The value `exitFailure` is equal to `exitWith (ExitFailure exitfail)`, where `exitfail` is implementation-dependent. `exitWith` bypasses the error handling in the I/O monad and cannot be intercepted by `catch` from the `Prelude`.

`exitFailure :: IO a`

The computation `exitFailure` is equivalent to `exitWith (ExitFailure` *exitfail*`)`, where *exitfail* is implementation-dependent.

`exitSuccess :: IO a`

The computation `exitSuccess` is equivalent to `exitWith ExitSuccess`, It terminates the program successfully.

Chapter 41

`System.IO`

```
module System.IO (
    IO,  fixIO,  FilePath,  Handle,  stdin,  stdout,  stderr,  withFile,
    openFile,  IOMode(ReadMode, WriteMode, AppendMode, ReadWriteMode),  hClose,
    readFile,  writeFile,  appendFile,  hFileSize,  hSetFileSize,  hIsEOF,
    isEOF,  BufferMode(NoBuffering, LineBuffering, BlockBuffering),
    hSetBuffering,  hGetBuffering,  hFlush,  hGetPosn,  hSetPosn,  HandlePosn,
    hSeek,  SeekMode(AbsoluteSeek, RelativeSeek, SeekFromEnd),  hTell,
    hIsOpen,  hIsClosed,  hIsReadable,  hIsWritable,  hIsSeekable,
    hIsTerminalDevice,  hSetEcho,  hGetEcho,  hShow,  hWaitForInput,  hReady,
    hGetChar,  hGetLine,  hLookAhead,  hGetContents,  hPutChar,  hPutStr,
    hPutStrLn,  hPrint,  interact,  putChar,  putStr,  putStrLn,  print,
    getChar,  getLine,  getContents,  readIO,  readLn
  ) where
```

41.1 The IO monad

`data IO a`

A value of type `IO a` is a computation which, when performed, does some I/O before returning a value of type `a`.

There is really only one way to ”perform” an I/O action: bind it to `Main.main` in your program. When your program is run, the I/O will be performed. It isn’t possible to perform I/O from an arbitrary function, unless that function is itself in the `IO` monad and called at some point, directly or indirectly, from `Main.main`.

`IO` is a monad, so `IO` actions can be combined using either the do-notation or the >> and >>= operations from the `Monad` class.

`instance Monad IO`
`instance Functor IO`

`fixIO :: (a -> IO a) -> IO a`

41.2 Files and handles

`type FilePath = String`

File and directory names are values of type `String`, whose precise meaning is operating system dependent. Files can be opened, yielding a handle which can then be used to operate on the contents of that file.

`data Handle`

Haskell defines operations to read and write characters from and to files, represented by values of type `Handle`. Each value of this type is a *handle*: a record used by the Haskell run-time system to *manage* I/O with file system objects. A handle has at least the following properties:

- whether it manages input or output or both;
- whether it is *open*, *closed* or *semi-closed*;
- whether the object is seekable;
- whether buffering is disabled, or enabled on a line or block basis;
- a buffer (whose length may be zero).

Most handles will also have a current I/O position indicating where the next input or output operation will occur. A handle is *readable* if it manages only input or both input and output; likewise, it is *writable* if it manages only output or both input and output. A handle is *open* when first allocated. Once it is closed it can no longer be used for either input or output, though an implementation cannot re-use its storage while references remain to it. Handles are in the `Show` and `Eq` classes. The string produced by showing a handle is system dependent; it should include enough information to identify the handle for debugging. A handle is equal according to == only to itself; no attempt is made to compare the internal state of different handles for equality.

`instance Eq Handle`
`instance Show Handle`

41.2.1 Standard handles

Three handles are allocated during program initialisation, and are initially open.

`stdin :: Handle`

A handle managing input from the Haskell program's standard input channel.

`stdout :: Handle`

A handle managing output to the Haskell program's standard output channel.

`stderr :: Handle`

A handle managing output to the Haskell program's standard error channel.

41.3 Opening and closing files

41.3.1 Opening files

`withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r`

`withFile name mode act` opens a file using `openFile` and passes the resulting handle to the computation `act`. The handle will be closed on exit from `withFile`, whether by normal termination or by raising an exception. If closing the handle raises an exception, then this exception will be raised by `withFile` rather than any exception raised by `act`.

`openFile :: FilePath -> IOMode -> IO Handle`

Computation `openFile file mode` allocates and returns a new, open handle to manage the file `file`. It manages input if `mode` is `ReadMode`, output if `mode` is `WriteMode` or `AppendMode`, and both input and output if mode is `ReadWriteMode`.

If the file does not exist and it is opened for output, it should be created as a new file. If `mode` is `WriteMode` and the file already exists, then it should be truncated to zero length. Some operating systems delete empty files, so there is no guarantee that the file will exist following an `openFile` with `mode` `WriteMode` unless it is subsequently written to successfully. The handle is positioned at the end of the file if `mode` is `AppendMode`, and otherwise at the beginning (in which case its internal position is 0). The initial buffer mode is implementation-dependent.

This operation may fail with:

- `isAlreadyInUseError` if the file is already open and cannot be reopened;
- `isDoesNotExistError` if the file does not exist; or
- `isPermissionError` if the user does not have permission to open the file.

`data IOMode`

```
= ReadMode
| WriteMode
| AppendMode
| ReadWriteMode
```

See `System.IO.openFile`

`instance Enum IOMode`
`instance Eq IOMode`
`instance Ord IOMode`
`instance Read IOMode`
`instance Show IOMode`
`instance Ix IOMode`

41.3.2 Closing files

`hClose :: Handle -> IO ()`

Computation `hClose hdl` makes handle `hdl` closed. Before the computation finishes, if `hdl` is writable its buffer is flushed as for `hFlush`. Performing `hClose` on a handle that has already been closed has no effect; doing so is not an error. All other operations on a closed handle will fail. If `hClose` fails for any reason, any further operations (apart from `hClose`) on the handle will still fail as if `hdl` had been successfully closed.

41.3.3 Special cases

These functions are also exported by the `Prelude`.

`readFile :: FilePath -> IO String`

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`.

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

The computation `writeFile file str` function writes the string `str`, to the file `file`.

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

The computation `appendFile file str` function appends the string `str`, to the file `file`.

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]])
```

41.3.4 File locking

Implementations should enforce as far as possible, at least locally to the Haskell process, multiple-reader single-writer locking on files. That is, *there may either be many handles on the same file which manage input, or just one handle on the file which manages output*. If any open or semi-closed handle is managing a file for output, no new handle can be allocated for that file. If any open or semi-closed handle is managing a file for input, new handles can only be allocated if they do not manage output. Whether two files are the same is implementation-dependent, but they should normally be the same if they have the same absolute path name and neither has been renamed, for example.

Warning: the `readFile` operation holds a semi-closed handle on the file until the entire contents of the file have been consumed. It follows that an attempt to write to a file (using `writeFile`, for example) that was earlier opened by `readFile` will usually result in failure with `System.IO.Error.isAlreadyInUseError`.

41.4 Operations on handles

41.4.1 Determining and changing the size of a file

`hFileSize :: Handle -> IO Integer`

For a handle `hdl` which attached to a physical file, `hFileSize hdl` returns the size of that file in 8-bit bytes.

`hSetFileSize :: Handle -> Integer -> IO ()`

`hSetFileSize hdl size` truncates the physical file with handle `hdl` to `size` bytes.

41.4.2 Detecting the end of input

`hIsEOF :: Handle -> IO Bool`

For a readable handle `hdl`, `hIsEOF hdl` returns `True` if no further input can be taken from `hdl` or for a physical file, if the current I/O position is equal to the length of the file. Otherwise, it returns `False`.

NOTE: `hIsEOF` may block, because it has to attempt to read from the stream to determine whether there is any more data to be read.

`isEOF :: IO Bool`

The computation `isEOF` is identical to `hIsEOF`, except that it works only on `stdin`.

41.4.3 Buffering operations

`data BufferMode`

=	**`NoBuffering`**	buffering is disabled if possible.
|	**`LineBuffering`**	line-buffering should be enabled if possible.
|	**`BlockBuffering (Maybe Int)`**	block-buffering should be enabled if possible. The size of the buffer is `n` items if the argument is `Just n` and is otherwise implementation-dependent.

Three kinds of buffering are supported: line-buffering, block-buffering or no-buffering. These modes have the following effects. For output, items are written out, or *flushed*, from the internal buffer according to the buffer mode:

- *line-buffering*: the entire output buffer is flushed whenever a newline is output, the buffer overflows, a `System.IO.hFlush` is issued, or the handle is closed.
- *block-buffering*: the entire buffer is written out whenever it overflows, a `System.IO.hFlush` is issued, or the handle is closed.
- *no-buffering*: output is written immediately, and never stored in the buffer.

An implementation is free to flush the buffer more frequently, but not less frequently, than specified above. The output buffer is emptied as soon as it has been written out.

Similarly, input occurs according to the buffer mode for the handle:

- *line-buffering*: when the buffer for the handle is not empty, the next item is obtained from the buffer; otherwise, when the buffer is empty, characters up to and including the next newline character are read into the buffer. No characters are available until the newline character is available or the buffer is full.
- *block-buffering*: when the buffer for the handle becomes empty, the next block of data is read into the buffer.
- *no-buffering*: the next input item is read and returned. The `System.IO.hLookAhead` operation implies that even a no-buffered handle may require a one-character buffer.

The default buffering mode when a handle is opened is implementation-dependent and may depend on the file system object which is attached to that handle. For most implementations, physical files will normally be block-buffered and terminals will normally be line-buffered.

`instance Eq BufferMode`
`instance Ord BufferMode`
`instance Read BufferMode`
`instance Show BufferMode`

`hSetBuffering :: Handle -> BufferMode -> IO ()`

Computation `hSetBuffering hdl mode` sets the mode of buffering for handle `hdl` on subsequent reads and writes.

If the buffer mode is changed from `BlockBuffering` or `LineBuffering` to `NoBuffering`, then

- if `hdl` is writable, the buffer is flushed as for `hFlush`;
- if `hdl` is not writable, the contents of the buffer is discarded.

This operation may fail with:

- `isPermissionError` if the handle has already been used for reading or writing and the implementation does not allow the buffering mode to be changed.

`hGetBuffering :: Handle -> IO BufferMode`

Computation `hGetBuffering hdl` returns the current buffering mode for `hdl`.

`hFlush :: Handle -> IO ()`

The action `hFlush hdl` causes any items buffered for output in handle `hdl` to be sent immediately to the operating system.

This operation may fail with:

- `isFullError` if the device is full;
- `isPermissionError` if a system resource limit would be exceeded. It is unspecified whether the characters in the buffer are discarded or retained under these circumstances.

41.4.4 Repositioning handles

`hGetPosn :: Handle -> IO HandlePosn`

Computation `hGetPosn hdl` returns the current I/O position of `hdl` as a value of the abstract type `HandlePosn`.

`hSetPosn :: HandlePosn -> IO ()`

If a call to `hGetPosn hdl` returns a position `p`, then computation `hSetPosn p` sets the position of `hdl` to the position it held at the time of the call to `hGetPosn`.

This operation may fail with:

- `isPermissionError` if a system resource limit would be exceeded.

`data HandlePosn`

```
instance Eq HandlePosn
instance Show HandlePosn
```

`hSeek :: Handle -> SeekMode -> Integer -> IO ()`

Computation `hSeek hdl mode i` sets the position of handle `hdl` depending on `mode`. The offset `i` is given in terms of 8-bit bytes.

If `hdl` is block- or line-buffered, then seeking to a position which is not in the current buffer will first cause any items in the output buffer to be written to the device, and then cause the input buffer to be discarded. Some handles may not be seekable (see `hIsSeekable`), or only support a subset of the possible positioning operations (for instance, it may only be possible to seek to the end of a tape, or to a positive offset from the beginning or current position). It is not possible to set a negative I/O position, or for a physical file, an I/O position beyond the current end-of-file.

This operation may fail with:

- `isIllegalOperationError` if the Handle is not seekable, or does not support the requested seek mode.
- `isPermissionError` if a system resource limit would be exceeded.

`data SeekMode`

=	**AbsoluteSeek**	the position of `hdl` is set to `i`.
|	**RelativeSeek**	the position of `hdl` is set to offset `i` from the current position.
|	**SeekFromEnd**	the position of `hdl` is set to offset `i` from the end of the file.

A mode that determines the effect of `hSeek hdl mode i`.

```
instance Enum SeekMode
instance Eq SeekMode
instance Ord SeekMode
instance Read SeekMode
instance Show SeekMode
instance Ix SeekMode
```

`hTell :: Handle -> IO Integer`

Computation `hTell hdl` returns the current position of the handle `hdl`, as the number of bytes from the beginning of the file. The value returned may be subsequently passed to `hSeek` to reposition the handle to the current position.

This operation may fail with:

- `isIllegalOperationError` if the Handle is not seekable.

41.4.5 Handle properties

Each of these operations returns `True` if the handle has the the specified property, or `False` otherwise.

`hIsOpen :: Handle -> IO Bool`

`hIsClosed :: Handle -> IO Bool`

`hIsReadable :: Handle -> IO Bool`

`hIsWritable :: Handle -> IO Bool`

`hIsSeekable :: Handle -> IO Bool`

41.4.6 Terminal operations

`hIsTerminalDevice :: Handle -> IO Bool`

Is the handle connected to a terminal?

`hSetEcho :: Handle -> Bool -> IO ()`

Set the echoing status of a handle connected to a terminal.

`hGetEcho :: Handle -> IO Bool`

Get the echoing status of a handle connected to a terminal.

41.4.7 Showing handle state

`hShow :: Handle -> IO String`

`hShow` is in the `IO` monad, and gives more comprehensive output than the (pure) instance of `Show` for `Handle`.

41.5 Text input and output

41.5.1 Text input

`hWaitForInput :: Handle -> Int -> IO Bool`

Computation `hWaitForInput hdl t` waits until input is available on handle `hdl`. It returns `True` as soon as input is available on `hdl`, or `False` if no input is available within `t` milliseconds. Note that `hWaitForInput` waits until one or more full *characters* are available, which means that it needs to do decoding, and hence may fail with a decoding error.

If `t` is less than zero, then `hWaitForInput` waits indefinitely.

This operation may fail with:

- `isEOFError` if the end of file has been reached.
- a decoding error, if the input begins with an invalid byte sequence in this Handle's encoding.

`hReady :: Handle -> IO Bool`

Computation `hReady hdl` indicates whether at least one item is available for input from handle `hdl`.

This operation may fail with:

- `System.IO.Error.isEOFError` if the end of file has been reached.

`hGetChar :: Handle -> IO Char`

Computation `hGetChar hdl` reads a character from the file or channel managed by `hdl`, blocking until a character is available.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

`hGetLine :: Handle -> IO String`

Computation `hGetLine hdl` reads a line from the file or channel managed by `hdl`.

This operation may fail with:

- `isEOFError` if the end of file is encountered when reading the *first* character of the line.

If `hGetLine` encounters end-of-file at any other point while reading in a line, it is treated as a line terminator and the (partial) line is returned.

`hLookAhead :: Handle -> IO Char`

Computation `hLookAhead` returns the next character from the handle without removing it from the input buffer, blocking until a character is available.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

`hGetContents :: Handle -> IO String`

Computation `hGetContents hdl` returns the list of characters corresponding to the unread portion of the channel or file managed by `hdl`, which is put into an intermediate state, *semi-closed*. In this state, `hdl` is effectively closed, but items are read from `hdl` on demand and accumulated in a special list returned by `hGetContents hdl`.

Any operation that fails because a handle is closed, also fails if a handle is semi-closed. The only exception is `hClose`. A semi-closed handle becomes closed:

- if `hClose` is applied to it;
- if an I/O error occurs when reading an item from the handle;
- or once the entire contents of the handle has been read.

Once a semi-closed handle becomes closed, the contents of the associated list becomes fixed. The contents of this final list is only partially specified: it will contain at least all the items of the stream that were evaluated prior to the handle becoming closed.

Any I/O errors encountered while a handle is semi-closed are simply discarded.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

41.5.2 Text output

`hPutChar :: Handle -> Char -> IO ()`

Computation `hPutChar hdl ch` writes the character `ch` to the file or channel managed by `hdl`. Characters may be buffered if buffering is enabled for `hdl`.

This operation may fail with:

- `isFullError` if the device is full; or
- `isPermissionError` if another system resource limit would be exceeded.

`hPutStr :: Handle -> String -> IO ()`

Computation `hPutStr hdl s` writes the string `s` to the file or channel managed by `hdl`.

This operation may fail with:

- `isFullError` if the device is full; or
- `isPermissionError` if another system resource limit would be exceeded.

`hPutStrLn :: Handle -> String -> IO ()`

The same as `hPutStr`, but adds a newline character.

`hPrint :: Show a => Handle -> a -> IO ()`

Computation `hPrint hdl t` writes the string representation of `t` given by the `shows` function to the file or channel managed by `hdl` and appends a newline.

This operation may fail with:

- `System.IO.Error.isFullError` if the device is full; or
- `System.IO.Error.isPermissionError` if another system resource limit would be exceeded.

41.5.3 Special cases for standard input and output

These functions are also exported by the `Prelude`.

`interact :: (String -> String) -> IO ()`

The `interact` function takes a function of type `String->String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.

`putChar :: Char -> IO ()`

Write a character to the standard output device (same as `hPutChar stdout`).

`putStr :: String -> IO ()`

Write a string to the standard output device (same as `hPutStr stdout`).

putStrLn :: String -> IO ()

The same as `putStr`, but adds a newline character.

print :: Show a => a -> IO ()

The `print` function outputs a value of any printable type to the standard output device. Printable types are those that 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]])
```

getChar :: IO Char

Read a character from the standard input device (same as `hGetChar stdin`).

getLine :: IO String

Read a line from the standard input device (same as `hGetLine stdin`).

getContents :: IO String

The `getContents` operation returns all user input as a single string, which is read lazily as it is needed (same as `hGetContents stdin`).

readIO :: Read a => String -> IO a

The `readIO` function is similar to `read` except that it signals parse failure to the `IO` monad instead of terminating the program.

readLn :: Read a => IO a

The `readLn` function combines `getLine` and `readIO`.

Chapter 42

System.IO.Error

```
module System.IO.Error (
    IOError,  userError,  mkIOError,  annotateIOError,  isAlreadyExistsError,
    isDoesNotExistError,  isAlreadyInUseError,  isFullError,  isEOFError,
    isIllegalOperation,  isPermissionError,  isUserError,  ioeGetErrorString,
    ioeGetHandle,  ioeGetFileName,  IOErrorType,  alreadyExistsErrorType,
    doesNotExistErrorType,  alreadyInUseErrorType,  fullErrorType,
    eofErrorType,  illegalOperationErrorType,  permissionErrorType,
    userErrorType,  ioError,  catch,  try
  ) where
```

42.1 I/O errors

`type IOError = IOError`

Errors of type `IOError` are used by the `IO` monad. This is an abstract type; the module `System.IO.Error` provides functions to interrogate and construct values of type `IOError`.

`userError :: String -> IOError`

Construct an `IOError` value with a string describing the error. The `fail` method of the `IO` instance of the `Monad` class raises a `userError`, thus:

```
instance Monad IO where
  ...
  fail s = ioError (userError s)
```

`mkIOError :: IOErrorType`
`-> String -> Maybe Handle -> Maybe FilePath -> IOError`

Construct an `IOError` of the given type where the second argument describes the error location and the third and fourth argument contain the file handle and file path of the file involved in the error if applicable.

```
annotateIOError :: IOError
                   -> String -> Maybe Handle -> Maybe FilePath -> IOError
```

Adds a location description and maybe a file path and file handle to an `IOError`. If any of the file handle or file path is not given the corresponding value in the `IOError` remains unaltered.

42.1.1 Classifying I/O errors

`isAlreadyExistsError :: IOError -> Bool`

An error indicating that an `IO` operation failed because one of its arguments already exists.

`isDoesNotExistError :: IOError -> Bool`

An error indicating that an `IO` operation failed because one of its arguments does not exist.

`isAlreadyInUseError :: IOError -> Bool`

An error indicating that an `IO` operation failed because one of its arguments is a single-use resource, which is already being used (for example, opening the same file twice for writing might give this error).

`isFullError :: IOError -> Bool`

An error indicating that an `IO` operation failed because the device is full.

`isEOFError :: IOError -> Bool`

An error indicating that an `IO` operation failed because the end of file has been reached.

`isIllegalOperation :: IOError -> Bool`

An error indicating that an `IO` operation failed because the operation was not possible. Any computation which returns an `IO` result may fail with `isIllegalOperation`. In some cases, an implementation will not be able to distinguish between the possible error causes. In this case it should fail with `isIllegalOperation`.

`isPermissionError :: IOError -> Bool`

An error indicating that an `IO` operation failed because the user does not have sufficient operating system privilege to perform that operation.

`isUserError :: IOError -> Bool`

A programmer-defined error value constructed using `userError`.

42.1.2 Attributes of I/O errors

```
ioeGetErrorString :: IOError -> String

ioeGetHandle :: IOError -> Maybe Handle

ioeGetFileName :: IOError -> Maybe FilePath
```

42.2 Types of I/O error

`data IOErrorType`

An abstract type that contains a value for each variant of `IOError`.

`instance Eq IOErrorType`
`instance Show IOErrorType`

`alreadyExistsErrorType :: IOErrorType`

I/O error where the operation failed because one of its arguments already exists.

`doesNotExistErrorType :: IOErrorType`

I/O error where the operation failed because one of its arguments does not exist.

`alreadyInUseErrorType :: IOErrorType`

I/O error where the operation failed because one of its arguments is a single-use resource, which is already being used.

`fullErrorType :: IOErrorType`

I/O error where the operation failed because the device is full.

`eofErrorType :: IOErrorType`

I/O error where the operation failed because the end of file has been reached.

`illegalOperationErrorType :: IOErrorType`

I/O error where the operation is not possible.

`permissionErrorType :: IOErrorType`

I/O error where the operation failed because the user does not have sufficient operating system privilege to perform that operation.

`userErrorType :: IOErrorType`

I/O error that is programmer-defined.

42.3 Throwing and catching I/O errors

`ioError :: IOError -> IO a`

Raise an `IOError` in the `IO` monad.

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

The `catch` function establishes a handler that receives any `IOError` raised in the action protected by `catch`. An `IOError` is caught by the most recent handler established by `catch`. These handlers are not selective: all `IOError`s 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 ioError e)
```

the function `f` returns `[]` when an end-of-file exception (cf. `isEOFError`) occurs in `g`; otherwise, the exception is propagated to the next outer handler.

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

`try :: IO a -> IO (Either IOError a)`

The construct `try comp` exposes IO errors which occur within a computation, and which are not fully handled.

Bibliography

[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] Unicode Consortium. Unicode standard. `http://unicode.org/standard/standard.html`.

[3] H.K. Curry and R. Feys. *Combinatory Logic*. North-Holland Pub. Co., Amsterdam, 1958.

[4] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–12, New York, 1982. ACM Press.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[6] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[7] International Standard ISO/IEC. Programming languages – C. 9899:1999 (E).

[8] MP Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–36, January 1995.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[10] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, 1999.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[12] P. Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings*, pages 248–256, San Francisco, September 1981.

[13] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of 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 are shown in `typewriter` font. Ordinary index entries are shown in a roman font.