Report on the Programming Language

Haskell

A Non-strict, Purely Functional Language

Version 1.1

August 1991

Paul Hudak[1] [editor]
Simon Peyton Jones[2] [editor]
Philip Wadler[2] [editor]
Brian Boutel[3]
Jon Fairbairn[4]
Joseph Fasel[5]
María M. Guzmán[1]
Kevin Hammond[2]
John Hughes[2]
Thomas Johnsson[6]
Dick Kieburtz[7]
Rishiyur Nikhil[8]
Will Partain[2]
John Peterson[1]

Authors' affiliations: (1) Yale University, (2) University of Glasgow, (3) Victoria University of Wellington, (4) Cambridge University, (5) Los Alamos National Laboratory, (6) Chalmers University of Technology, (7) Oregon Graduate Institute of Science and Technology, (8) Massachusetts Institute of Technology.

Contents

Preface to Version 1.1

Following the Version 1.0 Haskell report, several sites have implemented Haskell (or a subset thereof) and people have started to use these implementations. Based on this experience of implementation and use, it became apparent that a modest revision of the language would be desirable, in which some improvements in syntax could be made and certain features generalised. This Version 1.1 report is the result.

This revision was specifically *not* intended to add any substantial new features to the language, but rather to "tidy up" the existing language. Despite this narrow focus, a wide debate ensued, conducted on the Haskell mailing list (see page vi) rather than just among members of the original committee.

In this minor revision, the tricky issues identified in the preface to Version 1.0 remain, so that preface should be read in conjunction with this one.

Summary of changes

The main changes (other than concrete syntax) are as follows.

- Class methods may be polymorphic and overloaded in type variables other than the class variable (Section 4.3.1).
- The "monomorphism restriction" has been made more precise, and relaxed in the case where the programmer supplies a type signature (Section 4.4.2).
- The meaning of contexts in `data` declarations has been clarified (Section 4.2.1), and `type` synonym declarations are no longer permitted to have contexts (Section 4.2.2).
- If the `deriving` clause on a `data` declaration is omitted, no instances are automatically derived (Section 4.3.3).
- A module m may refer to all of its own local definitions in an export list using m`..` (Section 5.2.1).

The main syntactic changes are as follows:

- A new form of expression, a `let`-expression, has been added, which replaces and has the same semantics as a `where` expression. (In particular, the bindings it introduces are mutually recursive; Haskell has no non-recursive `let` construct.) Bindings may also be introduced by a `where` clause, but such `where` clauses are now attached to a group of guarded right-hand sides, and scope over the guards. The previous inability to scope definitions over guards was a significant shortcoming of the language.
- Sections have been introduced for binary operators. For example, the expression `(/ 2)` is the function which divides its argument by 2, and `(2 /)` is the function which divides 2 by its argument.

- The standard prelude has been debugged and revised.

A few other nontrivial changes to the syntax are listed in Appendix B.2.

Implementations

Several groups are working on implementations of Haskell, including those at Chalmers (contact: `hbc@cs.chalmers.se`), Glasgow (`haskell-request@dcs.glasgow.ac.uk`), Syracuse (`polar@top.cis.syr.edu`), and Yale (`haskell-request@cs.yale.edu`). Official announcements about these implementations will appear on the Haskell technical mailing list (see page vi).

Formal Semantics

Work has also been undertaken at Glasgow on a formal static and dynamic semantics for Haskell [6, 15]. These efforts are well advanced but as yet incomplete.

Acknowledgements

Language design is an evolutionary process, and the group of people involved undergoes evolution as well. We wish to thank past members of the Haskell Committee—Arvind, Mike Reeve, David Wise, and Jonathan Young—for their previous contributions and continued support. We also thank those who braved the storm of electronic mail on the Haskell mailing list, and responded with constructive suggestions for the revised language. The following were especially helpful and active: Lennart Augustsson, Cordelia Hall, Kent Karlsson, Mark Jones, Mark Lillibridge, and Satish Thatte.

Numerous others contributed to the debate, and we thank them also.

The Haskell Committee
19 August 1991

Preface to Version 1.0 (revised)

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

Goals

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

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

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

This Report

This report is the official specification of the Haskell language and should be suitable for writing programs and building implementations. It is *not* a tutorial on programming in Haskell, so some familiarity with functional languages is assumed. As this is the first edition of the specification, there may be some errors and inconsistencies; beware.

The Next Stage

Haskell is a large and complex language, designed for a wide spectrum of purposes. It also introduces a major new technical innovation, namely using type classes to handle overloading in a systematic way. This innovation permeates every aspect of the language.

Haskell is bound to contain infelicities and errors of judgement. We welcome your comments, suggestions, and criticisms on the language or its presentation in the report. Together with your input and our own experience of using the language, we plan to meet at some future time to resolve difficulties and further stabilise the design.

A common mailing list for technical discussion of Haskell can be reached at either `haskell@cs.yale.edu` or `haskell@dcs.glasgow.ac.uk`. Errata sheets for this report will be posted there. To subscribe, send a request to `haskell-request@dcs.glasgow.ac.uk` (European residents) or `haskell-request@cs.yale.edu` (residents elsewhere).

We thought it would be helpful to identify the aspects of the language design that seem to be most finely balanced, and hence are the most likely candidates for change when we review the language. The following list summarises these areas. It will only be fully comprehensible after you have read the report.

Mutually recursive modules. Mutual recursion among modules is unrestricted at present, which is obviously desirable from the programmer's point of view, but which poses significant challenges to the compilation system. In particular, it is *not* sufficient to start with trivial interfaces for each module and iterate to a fixpoint, as this example shows:

```
module F( f ) where
        import G
        f [x] = g x

module G( g ) where
        import F
        g = f
```

If a compilation system starts off by giving `F` and `G` interfaces that give the type signatures `f::a` and `g::b` respectively, then compiling the two modules alternately will not reach a fixed point. (This only happens if there is a type error, but it is obviously undesirable behaviour.) In general, a compiler may need to analyse a set of mutually recursive modules as a whole, rather than separately.

Generalising type classes. A number of restrictions are placed on the class system in Haskell. Currently, instances are attached to the top level type of an object and are exported implicitly with classes and types. A number of proposals for generalising the class system have been discussed, among them attaching instances to more complex types, parameterising classes over type constructors, allowing redefinition of instances, and making instances explicit in import and export lists. Some of these proposals have been implemented and are part of the available Haskell systems. As we gain more experience with the class system we hope to improve it in the future.

Default methods. Section 4.3.1 describes how a class declaration may include default methods for some of its operations. We considered extending this so that a class declaration could include default methods *for operations of its superclasses*, which override the superclass's default method. This looks like an attractive idea, which will certainly be considered for a future revision.

Defaults for ambiguous types. Section 4.3.4 describes how ambiguous typings, which arise due to the type-class system, are resolved. Ideally, the choice made should not matter. For example, consider the expression `if round x > 0 then E1 else E2`. It should not matter whether `round` returns `Int` or `Integer`; a bad choice could result in overflow, or a less efficient program, but if a result is produced it will be correct.

Our resolution rules strive only to resolve ambiguous types where the type chosen does not "matter" in this sense, but we have not been entirely successful, for example where floating point is concerned. Further research and practical experience may suggest a better set of rules.

Static semantics of `let` and `where` bindings. The rules at the end of Section 4.4.2 comprise the "monomorphism restriction" in Haskell. The restriction solves two problems, which are summarised below, but at the cost of restricting expressiveness. Only experience will tell how much of a problem this is for the programmer.

These are the two problems. First, the expression `let x = factorial 1000 in (x,x)` looks as though `x` should only be computed once. If `x` were used at different overloadings, however, `factorial 1000` would be computed twice, once at each type. We have found examples where the loss of efficiency is exponential in the size of the program. Modest compiler optimisations can often eliminate the problem, but we have found no simple scheme that can *guarantee* to do so. The restriction solves the problem by ensuring that all uses of `x` are at the same overloading, and hence that its evaluation can be shared as usual.

Second, a rather subtle form of type ambiguity (Section 4.3.4) is eliminated by the restriction to non-overloaded pattern bindings. An example is:

```
readNum s r = (n*r,s') where [(n,s')] = reads s
```

Here `n::(Num a, Text a) => a`, `s'::Text a => String`. If the definition of `[(n,s')]` is polymorphic, the `a`'s may be resolved as different types.

(Note: As of the version 1.1 report, the monomorphism restriction is relaxed, provided that the programmer gives an explicit type signature. See Section 4.5.4 for precise details.)

Overloaded constants. Overloaded constants (e.g. `1`, which has type `Num a => a`) are extraordinarily convenient when programming, but are the source of several serious technical problems, including both of those mentioned in the two preceding items. One could eliminate overloaded constants altogether; we considered this at length, and we are sure to reconsider it when we review the language.

Polymorphism in `case` expressions. The type of a variable bound by a Standard ML case-expression is monomorphic; we have made the same decision in Haskell (Section 3.13.3). The question of whether such types can be made polymorphic interacts with the restrictions on polymorphism for pattern-bound variables, mentioned above. For the present, we have erred on the side of conservatism, but this decision should be reviewed.

Acknowledgements

We heartily thank these people for their useful contributions to this report: Lennart Augustsson, Richard Bird, Stephen Blott, Tom Blenko, Duke Briscoe, Chris Clack, Guy Cousineau, Tony Davie, Chris Fasel, Pat Fasel, Bob Hiromoto, Nic Holt, Simon B. Jones, Stef Joosten, Mike Joy, Richard Kelsey, Siau-Cheng Khoo, Amir Kishon, John Launchbury, Olaf Lubeck, Randy Michelsen, Rick Mohr, Arthur Norman, Paul Otto, Larne Pekowsky, John Peterson, Rinus Plasmeijer, John Robson, Colin Runciman, Lauren Smith, Raman Sundaresh, Tom Thomson, Pradeep Varma, Tony Warnock, Stuart Wray, and Bonnie Yantis. We also thank those who participated in the lively discussions about Haskell on the FP mailing list during an interim period of the design.

Finally, aside from the important foundational work laid by Church, Rosser, Curry, and others on the lambda calculus, we wish to acknowledge the influence of many noteworthy programming languages developed over the years. Although it is difficult to pinpoint the origin of many ideas, we particularly wish to acknowledge the influence of McCarthy's Lisp [11] (and its modern-day incarnation, Scheme [16]); Landin's ISWIM [9]; Backus's FP [1]; Gordon, Milner, and Wadsworth's ML [5]; Burstall, MacQueen, and Sannella's Hope [2]; and Turner's series of languages culminating in Miranda [19].[1] Without these forerunners Haskell would not have been possible.

The Haskell Committee
1 April 1990
(revised) 19 August 1991

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

1 Introduction

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language research, including higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module 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 functional languages—the design has been influenced by languages as old as ISWIM and as new as Miranda.

Although the initial emphasis was on standardisation, Haskell also has several new features that both simplify and generalise the design. For example,

1. Rather than using *ad hoc* techniques for overloading, Haskell provides an explicit overloading facility, integrated with the polymorphic type system, that allows the precise definition of overloading behaviour for any operator or function.

2. The conventional notion of "abstract data type" has been unbundled into two orthogonal components: data abstraction and information hiding.

3. Haskell has a flexible I/O facility that unifies two popular styles of purely functional I/O—the *stream* model and the *continuation* model—and both styles can be mixed within the same program. The system supports most of the standard operations provided by conventional operating systems while retaining referential transparency within a program.

4. Recognising the importance of arrays, Haskell has a family of multidimensional non-strict immutable arrays whose special interaction with list comprehensions provides a convenient "array comprehension" syntax for defining arrays monolithically.

This report defines the syntax for Haskell programs and an informal abstract semantics for the meaning of such programs; the formal abstract semantics is in preparation. 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 batch versus interactive programming environments, and the nature of 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 organisation of the rest of the report.

1. At the topmost level a Haskell program is a set of *modules* (described in Section 5). Modules provide a way to control namespaces and to re-use software in large programs.

2. The top level of a module consists of a collection of *declarations*, of which there are several kinds, all described in Section 4. Declarations define things such as ordinary values, datatypes, type classes, and fixity information.

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

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

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

The sections not mentioned above are Section 6, which describes the standard built-in datatypes in Haskell, and Section 7, which discusses the I/O facility in Haskell (i.e. how Haskell programs communicate with the outside world). Also, there are several appendices describing the standard prelude, the concrete syntax, the semantics of I/O, and the specification of derived instances.

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 italicised names will be mnemonic, such as e for expressions, d for declarations, t for types, etc.

1.2 The Haskell Kernel

Haskell has adopted many of the convenient syntactic structures that have become popular in functional programming. In all cases, their formal semantics can be given via translation into a proper subset of Haskell called the Haskell *kernel*. It is essentially a slightly sugared variant of the lambda calculus with a straightforward denotational semantics. The translation of each syntactic structure into the kernel is given as the syntax is introduced. This modular design facilitates reasoning about Haskell programs and provides useful guidelines for implementors of the language.

1.3 Values and Types

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

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

1.4 Namespaces

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

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

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

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

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

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

1.5 Layout

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

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

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

The layout (or "off-side") rule takes effect whenever the open brace is omitted after the keyword `where`, `let` or `of`. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments). For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the declaration list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the declaration list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted. The layout rule will match only those open braces that it has inserted; an open brace that the user has inserted must be matched by a close brace inserted by the user.

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

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

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

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

As an example, Figure 1 shows a (somewhat contrived) module and Figure 2 shows the result of applying the layout rule to it. Note in particular: (a) the line beginning `}};pop`, where the termination of the previous line invokes three applications of the layout rule, corresponding to the depth (3) of the nested **`where`** clauses, (b) the close braces in the **`where`** clause nested within the tuple and **`case`** expression, inserted because the end of the tuple was detected, and (c) the close brace at the very end, inserted because of the column 0 indentation of the end-of-file token.

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

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

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

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

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

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

Figure 1: A sample program

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

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

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

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

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

Figure 2: Sample program with layout expanded

2 Lexical Structure

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

2.1 Notational Conventions

These notational conventions are used for presenting syntax:

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

Because the syntax in this section describes *lexical* syntax, all whitespace is expressed explicitly; there is no implicit space between juxtaposed symbols. BNF-like syntax is used throughout, with productions having the form:

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

There are some families of nonterminals indexed by precedence levels (written as a superscript). Similarly, the lexeme classes *op*, *varop*, and *conop* have a double index: a letter l, r, or n for left-, right- or nonassociativity and a precedence level. So, for example

$$exp^i \rightarrow exp^{i+1}\ [op^{(\mathrm{n},i)}\ exp^{i+1}] \qquad (0 \leq i \leq 9)$$

actually stands for 10 productions where *op* is non-associative. Refer to Section 5.7 for information on fixity declarations.

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

Haskell source programs are currently biased toward the ASCII character set, although future Haskell standardisation efforts will likely address broader character standards.

2.2 Lexical Program Structure

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

whitespace → *whitestuff* {*whitestuff*}
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *space* | *tab* | *vertab* | *formfeed*
newline → a newline (system dependent)
space → a space
tab → a horizontal tab
vertab → a vertical tab
formfeed → a form feed
comment → `--` {*any*} *newline*
ncomment → `{-` *ANYseq* {*ncomment* *ANYseq*} `-}`
ANYseq → {*ANY*}$_{\{ANY\}\ (\texttt{\{-}\ \mid\ \texttt{-\}})\ \{ANY\}}$
ANY → *any* | *newline* | *vertab* | *formfeed*
any → *graphic* | *space* | *tab*
graphic → *large* | *small* | *digit*
| `!` | `"` | `#` | `$` | `%` | `&` | `´` | `(` | `)` | `*` | `+`
| `,` | `-` | `.` | `/` | `:` | `;` | `<` | `=` | `>` | `?` | `@`
| `[` | `\` | `]` | `^` | `_` | `` ` `` | `{` | `|` | `}` | `~`

small → `a` | `b` | ... | `z`
large → `A` | `B` | ... | `Z`
digit → `0` | `1` | ... | `9`

Characters not in the category *graphic* or *whitestuff* are not valid in Haskell programs and should result in a lexing error.

Comments are valid *whitespace*. An ordinary comment begins with two consecutive dashes (`--`) and extends to the following newline. A nested comment begins with `{-` and ends with `-}`; it can be between any two lexemes. All character sequences not containing `{-` nor `-}` are ignored within a nested comment. Nested comments may be nested to any depth: any occurrence of `{-` within the nested comment starts a new nested comment, terminated by `-}`. Within a nested comment, each `{-` is matched by a corresponding occurrence of `-}`. In an ordinary comment, the character sequences `{-` and `-}` have no special significance, and, in a nested comment, the sequence `--` has no special significance.

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 nesting of the nested comments.

2.3 Identifiers and Operators

avarid → (*small* {*small* | *large* | *digit* | `´` | `_`})$_{\langle reservedid \rangle}$
varid → *avarid* | `(`*avarop*`)`
aconid → *large* {*small* | *large* | *digit* | `´` | `_`}
conid → *aconid* | `(`*aconop*`)`
reservedid → `case` | `class` | `data` | `default` | `deriving` | `else` | `hiding`

| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance` | `interface`
| `let` | `module` | `of` | `renaming` | `then` | `to` | `type` | `where`

An identifier consists of a letter followed by zero or more letters, digits, underscores, and acute accents. Identifiers are lexically distinguished into two classes: those that begin with a small letter (variable identifiers) and those that begin with a capital (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).

avarop → ((*symbol* | *presymbol*) {*symbol* | `:`})$_{\{reservedop\}}$
varop → *avarop* | \`*avarid*\`
aconop → (`:` {*symbol* | `:`})$_{\{reservedop\}}$
conop → *aconop* | \`*aconid*\`
presymbol → `-` | `~`
symbol → `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@` | `\` | `^` | `|`
reservedop → `..` | `::` | `=>` | `=` | `@` | `\` | `|` | `~` | `<-` | `->`

An operator is either symbolic or alphanumeric. Symbolic operators are formed from one or more symbols, as defined above, and are lexically distinguished into two classes: those that start with a colon (constructors) and those that do not (functions).

Alphanumeric operators are formed by enclosing an identifier between grave accents (backquotes). Any variable or constructor may be used as an operator in this way. If *fun* is an identifier (either variable or constructor), then an expression of the form *fun x y* is equivalent to *x* \`*fun*\` *y*. If no fixity declaration is given for \`*fun*\` then it defaults to highest precedence and left associativity (see Section 5.7).

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

No white space is permitted in names such as \`*fun*\` and `(`*op*`)`.

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 3.3). All of the standard infix operators are just predefined symbols and may be rebound.

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

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

var	→	*varid*	(*variables*)
con	→	*conid*	(*constructors*)
tyvar	→	*avarid*	(*type variables*)
tycon	→	*aconid*	(*type constructors*)
tycls	→	*aconid*	(*type classes*)
modid	→	*aconid*	(*modules*)

Variables and type variables are represented by identifiers beginning with small letters, and the other four by identifiers beginning with capitals; also, variables and constructors have infix forms, the other four do not. Namespaces are also discussed in Section 1.4.

2.4 Numeric Literals

integer → *digit*{*digit*}
float → *integer*`.`*integer*[(`e` | `E`)[`-` | `+`]*integer*]

There are two distinct kinds of numeric literals: integer and floating. 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 3.2. The typing of numeric literals is discussed in Section 6.8.2.

2.5 Character and String Literals

char → ´ (*graphic*$_{\{ ´ \mid \backslash \}}$ | *space* | *escape*$_{\{\backslash\&\}}$) ´
string → `"` {*graphic*$_{\{ " \mid \backslash \}}$ | *space* | *escape* | *gap*} `"`
escape → `\` (*charesc* | *ascii* | *integer* | `o` *octit*{*octit*} | `x` *hexit*{*hexit*})
charesc → `a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | ´ | `&`
ascii → `^`*cntrl* | `NUL` | `SOH` | `STX` | `ETX` | `EOT` | `ENQ` | `ACK`
| `BEL` | `BS` | `HT` | `LF` | `VT` | `FF` | `CR` | `SO` | `SI` | `DLE`
| `DC1` | `DC2` | `DC3` | `DC4` | `NAK` | `SYN` | `ETB` | `CAN`
| `EM` | `SUB` | `ESC` | `FS` | `GS` | `RS` | `US` | `SP` | `DEL`
cntrl → *large* | @ | `[` | `\` | `]` | `^` | `_`
gap → `\` *whitechar* {*whitechar*} `\`
hexit → *digit* | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`
octit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

Character literals are written between acute accents, 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 ´ may be used in a string, but must be escaped in a character; similarly, `"` 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 ASCII character set, including control characters such as `\^X`, are also provided. Numeric escapes such as `\137` are used to designate the character with (implementation dependent) decimal representation 137; octal (e.g. `\o137`) and hexadecimal (e.g. `\x137`) representations are also allowed. Numeric escapes that are out-of-range of the ASCII standard are undefined and thus non-portable.

Consistent with the "consume longest lexeme" rule, numeric escape characters in strings consist of all consecutive digits and may be of arbitrary length. Similarly, the one ambiguous ASCII escape code, `"\SOH"`, is parsed as a string of length 1. The escape character `\&` is provided as a "null character" to allow strings such as `"\137\&9"` and `"\SO\&H"` to be constructed (both of length two). Thus `"\&"` is equivalent to `""` and the character `´\&´` is disallowed. Further equivalences of characters are defined in Section 6.2.

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 3.5).

3 Expressions

In this section, we describe the syntax and informal semantics of Haskell *expressions*, including their translations into the Haskell kernel, where appropriate.

exp	$\rightarrow$	`\` $apat_1$ ... $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)
	$\mid$	`let {` *decls* [`;`] `} in` *exp*	(let expression)
	$\mid$	`if` *exp* `then` *exp* `else` *exp*	(conditional)
	$\mid$	`case` *exp* `of {` *alts* [`;`] `}`	(case expression)
	$\mid$	exp^0 `::` [*context* `=>`] *atype*	(expression type signature)
	$\mid$	exp^0	
exp^i	$\rightarrow$	exp^{i+1} [$op^{(\mathrm{n},i)}$ exp^{i+1}]	($0 \leq i \leq 9$)
	$\mid$	$lexp^i$ $op^{(\mathrm{l},i)}$ exp^{i+1}	
	$\mid$	exp^{i+1} $op^{(\mathrm{r},i)}$ $rexp^i$	
$lexp^i$	$\rightarrow$	[$lexp^i$ $op^{(\mathrm{l},i)}$] exp^{i+1}	($0 \leq i \leq 9$)
$lexp^6$	$\rightarrow$	`-` exp^7	
$rexp^i$	$\rightarrow$	exp^{i+1} [$op^{(\mathrm{r},i)}$ $rexp^i$]	($0 \leq i \leq 9$)
exp^{10}	$\rightarrow$	exp^{10} *aexp*	(function application)
	$\mid$	*aexp*	
aexp	$\rightarrow$	*var*	(variable)
	$\mid$	*con*	(constructor)
	$\mid$	*literal*	
	$\mid$	`()`	(unit)
	$\mid$	`(` *exp* `)`	(parenthesised expression)
	$\mid$	`(` exp_1 `,` ... `,` exp_k `)`	(tuple, $k \geq 2$)
	$\mid$	`[` exp_1 `,` ... `,` exp_k `]`	(list, $k \geq 0$)
	$\mid$	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	$\mid$	`[` *exp* `|` $qual_1$ `,` ... `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	$\mid$	`(` exp^{i+1} $op^{(a,i)}$ `)`	($0 \leq i \leq 9$, $a \in \{l, r, n\}$)
	$\mid$	`(` $op^{(a,i)}$ exp^{i+1} `)`	($0 \leq i \leq 9$, $a \in \{l, r, n\}$)
op	$\rightarrow$	*varop* $\mid$ *conop*	

The grammar above embodies the following precedence hierarchy for expressions, using productions with superscripts as described in Section 2.1:

- Function application binds most tightly of all.
- Expressions involving infix operators are disambiguated by the operator's fixity (see Section 5.7). Consecutive unparenthesised operators with the same precedence must both be either left or right associative to avoid a syntax error. Given an unparenthesised expression "x $op^{(a,i)}$ y $op^{(b,j)}$ z", parentheses must be added around either "x $op^{(a,i)}$ y" or "y $op^{(b,j)} z$" when $i = j$ unless $a = b =$ l or $a = b =$ r.

- Negation is the only prefix operator in Haskell; it has the same precedence as the infix `-` operator defined in the standard prelude (see Figure 5).
- Expression type signatures are parsed as if `::` were a left-associative infix operator with precedence lower than any other operator.

Sample parses using this grammar 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)`
`f x y :: Int`	`(f x y) :: Int`
`\ x -> a+b :: Int`	`\ x -> ((a+b) :: Int)`

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

3.1 Curried Applications and Lambda Abstractions

$$
\begin{array}{lcl}
\mathit{exp} & \rightarrow & \mathit{exp}\ \mathit{aexp} \\
\mathit{exp} & \rightarrow & \texttt{\\}\ \mathit{apat}_1\ \ldots\ \mathit{apat}_n\ \texttt{->}\ \mathit{exp}
\end{array}
$$

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

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

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

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

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

where the x_i are new identifiers. Given this translation combined with the semantics of case expressions and pattern-matching described in Section 3.13.3, if the pattern fails to match, then the result is $\perp$.

The type of a variable bound by a lambda abstraction is monomorphic, as is always the case in the Hindley-Milner type system.

3.2 Operator Applications

exp	$\rightarrow$	*exp*$_1$ *op* *exp*$_2$	
	$\vert$	`-` *exp*	(*prefix negation*)

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

The special form `-`e denotes prefix negation, the one and only prefix operator in Haskell, and is simply syntax for `negate` (e), where `negate` is as defined in the standard prelude (see Figure 8, page 57). Because `e1-e2` parses as an infix application of the binary operator `-`, one must write `e1(-e2)` for the alternative parsing. Similarly, `(-)` is syntax for `(\ x y -> x-y)`, as with any infix operator, and does not denote `(\ x -> -x)`—one must use `negate` for that.

> **Translation:** e_1 *op* e_2 is equivalent to `(`*op*`)` e_1 e_2. `-`e is equivalent to `negate` (e) where `negate`, an operator in the class `Num`, is as defined in the standard prelude.

3.3 Sections

aexp	$\rightarrow$	`(` *exp* *op* `)`
	$\vert$	`(` *op* *exp* `)`

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

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

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

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

3.4 Conditionals

exp	$\rightarrow$	`if` *exp*$_1$ `then` *exp*$_2$ `else` *exp*$_3$

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

Translation: `if` e_1 `then` e_2 `else` e_3 is equivalent to:

`case` e_1 `of { True ->` e_2 `; False ->` e_3 `}`

where `True` and `False` are the two nullary constructors from the type `Bool`, as defined in the standard prelude.

3.5 Lists

aexp $\rightarrow$ `[` *exp*$_1$ `,` ... `,` *exp*$_k$ `]` $(k \geq 0)$

Lists are written `[`e_1`,` ... `,` e_k`]`, where $k \geq 0$; the empty list is written `[]`. Standard operations on lists are given in the standard prelude (see Appendix A, notably Section A.5).

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

e_1 `: (`e_2 `: (` ... `(`e_k `: []))))`

where `:` and `[]` are constructors for lists, as defined in the standard prelude (see Section 6.4). The types of e_1 through e_k must all be the same (call it t), and the type of the overall expression is `[`t`]` (see Section 4.1.1).

3.6 Tuples

aexp $\rightarrow$ `(` *exp*$_1$ `,` ... `,` *exp*$_k$ `)` $(k \geq 2)$

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

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

3.7 Unit Expressions and Parenthesised Expressions

aexp $\rightarrow$ `()`
| `(` *exp* `)`

The form `(`e`)` is simply a *parenthesised expression*, and is equivalent to e. The *unit expression* `()` has type `()` (see Section 4.1.1); it is the only member of that type (it can be thought of as the "nullary tuple")—see Section 6.6.

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

3.8 Arithmetic Sequences

$aexp \rightarrow$ `[` exp_1 [`,` exp_2] `..` [exp_3] `]`

The form `[`e_1`,` e_2 `..` e_3`]` denotes an *arithmetic sequence* from e_1 in increments of $e_2 - e_1$ up to e_3 (if the increment is positive) or down to e_3 (if the increment is negative). An infinite list of e_1's results if the increment is zero, and the empty list results if e_3 is less than e_1 and the increment is positive, or if e_3 is greater than e_1 and the increment is negative. If the comma and e_2 are omitted, then the increment is 1; if e_3 is omitted, then the sequence is includes all elements of the enumeration, and is thus infinite for infinite enumerations.

Arithmetic sequences may be defined over any type in class `Enum`, including `Char`, `Int`, and `Integer` (see Figure 4 and Section 4.3.3). For example, `['a'..'z']` denotes the list of lower-case letters in alphabetical order.

Translation: Arithmetic sequences satisfy these identities:

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

where `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo` are operations in the class `Enum` as defined in the standard prelude (see Figure 4).

3.9 List Comprehensions

$aexp \rightarrow$ `[` exp `|` $qual_1$ `,` ... `,` $qual_n$ `]` (list comprehension, $n \geq 1$)
$qual \rightarrow$ pat `<-` exp
$\quad | \;$ exp

A *list comprehension* has the form `[` e `|` q_1 `,` ... `,` q_n `]`, $n \geq 1$, where the q_i qualifiers are either *generators* of the form p `<-` e, where p is a pattern (see Section 3.13) of type t and e is an expression of type `[`t`]`; or *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 3.13), and if a match fails then that element of the list is simply skipped over. Thus:

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

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

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

> **Translation:** List comprehensions satisfy these identities, which may be used as a translation into the kernel:
>
> ```
> [ e | b ]        =  if b then [e] else []
> [ e | q1, q2 ]   =  concat [ [ e | q2 ] | q1 ]
> [ e | p <- l ]   =  let ok p  =  True
>                         ok _  =  False
>                     in
>                      map (\p -> e) (filter ok l)
> ```
>
> where e ranges over expressions, p ranges over patterns, l ranges over list-valued expressions, b ranges over boolean expressions, q_1 and q_2 range over non-empty lists of qualifiers, and `ok` is a new identifier not appearing in e, p, or l. These three equations uniquely define list comprehensions. `True`, `False`, `map`, `concat` and `filter` are all as defined in the standard prelude.

3.10 Let Expressions

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

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

3.11 Case Expressions

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

A *case expression* has the general form

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

where each $match_i$ is of the general form

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

where each clause p_i $matches_i$ consists of a a *pattern* p_i and its $matches_i$, which consists of pairs of optional *guards* g_{ij} and *bodies* e_{ij} (expressions), as well as optional bindings ($decls_i$) that scope over all of the guards and expressions of the clause. An alternative of the form

pat `->` *expr* `where` `{` *decls* `}`

is treated as shorthand for:

pat `|` `True` `->` *expr*
`where` `{` *decls* `}`

A case expression must have at least one clause and each clause 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 clauses. The matches are tried sequentially, from top to bottom. The first successful match causes evaluation of the corresponding clause body, in the environment of the case expression extended by the bindings created during the matching of that clause and by the $decls_i$ associated with that clause. If no match succeeds, the result is $\perp$. Pattern matching is described in Section 3.13, with the formal semantics of case expressions in Section 3.13.3.

3.12 Expression Type-Signatures

exp	$\rightarrow$	*exp* `::` [*context* `=>`] *atype*

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

3.13 Pattern-Matching

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

3.13.1 Patterns

Patterns have this syntax:

pat	$\rightarrow$	pat^{0}	
pat^{i}	$\rightarrow$	$pat_{1}^{i+1}\ [conop^{(\mathrm{n},i)}\ pat_{2}^{i+1}]$	$(0 \leq i \leq 9)$
	$\mid$	$lpat^{i}\ conop^{(\mathrm{l},i)}\ pat^{i+1}$	
	$\mid$	$pat^{i+1}\ conop^{(\mathrm{r},i)}\ rpat^{i}$	
$lpat^{i}$	$\rightarrow$	$[lpat^{i}\ conop^{(\mathrm{l},i)}]\ pat^{i+1}$	$(0 \leq i \leq 9)$
$lpat^{6}$	$\rightarrow$	$lpat^{6}$ `+` $integer$	(successor pattern)
	$\mid$	`-` $\{integer \mid float\}$	(negative literal)
$rpat^{i}$	$\rightarrow$	$pat^{i+1}\ [conop^{(\mathrm{r},i)}\ rpat^{i}]$	$(0 \leq i \leq 9)$
pat^{10}	$\rightarrow$	$apat$	
	$\mid$	$con\ apat_{1}\ \ldots\ apat_{k}$	(arity $con = k \geq 1$)
$apat$	$\rightarrow$	$var\ [$ `@` $apat]$	(as pattern)
	$\mid$	con	(arity $con = 0$)
	$\mid$	$literal$	
	$\mid$	`_`	(wildcard)
	$\mid$	`()`	(unit pattern)
	$\mid$	`(` pat `)`	(parenthesised pattern)
	$\mid$	`(` pat_{1} `,` $\ldots$ `,` pat_{k} `)`	(tuple pattern, $k \geq 2$)
	$\mid$	`[` pat_{1} `,` $\ldots$ `,` pat_{k} `]`	(list pattern, $k \geq 0$)
	$\mid$	`~` $apat$	(irrefutable pattern)

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

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

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

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

is equivalent to:

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

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

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

is equivalent to:

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

In the pattern-matching rules given below we distinguish two kinds of patterns: an *irrefutable pattern* is either of the form `~`*apat*, a variable, or a wildcard; all other patterns are *refutable*.

3.13.2 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 in, according to these rules:

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

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

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

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

   (a) Matching a non-$\perp$ value against a constructed pattern fails if the outermost constructors are different. If the constructors are the same, the result of the match is the result of matching the sub-patterns left-to-right: if all matches succeed, the overall match succeeds; the first to fail or diverge causes the overall match to fail or diverge, respectively.

   Constructed values consist of those created by prefix or infix constructors, tuple or list patterns, and strings (which are lists of characters). Characters and `()` are treated as nullary constructors. Numeric literals are matched using the overloaded `==` function.

(b) Matching a non-$\perp$ value n against a pattern of the form x+k (where x is a variable and k is a positive integer literal) succeeds if $n \geq k$, resulting in the binding of x to $n - k$, and fails if $n < k$. For example, the Fibonacci function may be defined as follows:

```
fib n = case n of {
          0   -> 1 ;
          1   -> 1 ;
          n+2 -> fib n + fib (n+1) }
```

Since `n` must be bound to a positive value, `fib` diverges for a negative argument, and exactly one of the equations matches any non-negative argument.

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

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

Here are some examples:

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

2. These examples demonstrate refutable vs. irrefutable matching:

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

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

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

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

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

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

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

both `a` and `y` will be evaluated.

3.13.3 Formal semantics of pattern-matching

The semantics of all other constructs that use pattern-matching is defined by giving identities that relate those constructs to `case` expressions.

The semantics of `case` expressions are given as a series of identities. Figure 3 shows the identities: e, e' and e_i are arbitrary expressions; g and g_i are boolean-valued expressions; p and p_i are patterns; x and x_i are variables; K and K' are constructors (including tuple constructors); a $match_i$ is a form as shown in rule (a); and k is a character, string, or numeric literal.

For clarity, several rules are expressed using `let` (used only in a non-recursive way); their usual purpose is to prevent name capture (e.g., in rule (b)). The rules may be re-expressed entirely with `case`s by applying this identity:

$$\texttt{let } x \texttt{ = } y \texttt{ in } e \quad = \quad \texttt{case } y \texttt{ of \{ } x \texttt{ -> } e \texttt{ \}}$$

Using all but the last two identities (rules (k) and (l)) in Figure 3 in a left-to-right manner yields a translation into a subset of general `case` expressions called *simple case expressions*. Rule (a) matches a general source-language `case` expression, regardless of whether it actually includes guards—if no guards are written, then `True` is substituted for the guards $g_{i,j}$ in the $match_i$ forms. Subsequent identities manipulate the resulting `case` expression into simpler and simpler forms. The semantics of simple `case` expressions is given by the last two identities ((k) and (l)).

Rules (g) and (h) in Figure 3 involve the overloaded operators `==` and `<=`; it is these rules that define the meaning of pattern-matching against overloaded constants.

When used as a translation, the identities in Figure 3 will generate a very inefficient program. This can be fixed by using further `case` or `let` expressions, but doing so would clutter the identities, which are intended only to convey the semantics.

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

(a) `case` e_0 `of {` p_1 $match_1$`;` ... `;` p_n $match_n$ `}`
$=$ `case` e_0 `of {` p_1 $match_1$ `;`
`_ -> ...` `case` e_0 `of {`
p_n $match_n$
`_ -> error "No match" }...}`
where each $match_i$ has the form:
`|` $g_{i,1}$ `->` $e_{i,1}$ `;` ... `; |` g_{i,m_i} `->` e_{i,m_i} `where {` $decls_i$ `}`

(b) `case` e_0 `of {` p `|` g_1 `->` e_1 `;` ...
`|` g_n `->` e_n `where {` $decls$ `}`
`_ ->` e' `}`
$=$ `let {` y `=` e' `}` (where y is a completely new variable)
`in case` e_0 `of {`
p `-> let {` $decls$ `} in`
`if` g_1 `then` e_1 ... `else if` g_n `then` e_n `else` y
`_ ->` y `}`

(c) `case` e_0 `of { ~`p `->` e`; _ ->` e' `}`
$=$ `let {` y `=` e_0 `} in`
`let {` x'_1 `= case` y `of {` p `->` x_1 `}} in` ...
`let {` x'_n `= case` y `of {` p `->` x_n `}} in` $e\,[x'_1/x_1, \ldots, x'_n/x_n]$
$x_1, \ldots, x_n$ are all the variables in p; $y, x'_1, \ldots, x'_n$ are completely new variables

(d) `case` e_0 `of {` x`@`p `->` e`; _ ->` e' `}`
$=$ `let {` y `=` e_0 `}` (where y is a completely new variable)
`in case` y `of {` p `-> ( \` x `->` e `)` y `; _ ->` e' `}`

(e) `case` e_0 `of { _ ->` e`; _ ->` e' `}` $=$ e

(f) `case` e_0 `of {` $K\ p_1 \ldots p_n$ `->` e`; _ ->` e' `}`
$=$ `let {` y `=` e' `}`
`in case` e_0 `of {`
$K\ x_1 \ldots x_n$ `-> case` x_1 `of {`
p_1 `-> ... case` x_n `of {` p_n `->` e `; _ ->` y `}` ...
`_ ->` y `}`
`_ ->` y `}` (where $y, x_1, \ldots, x_n$ are completely new variables)

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

(h) `case` e_0 `of {` x`+`k `->` e`; _ ->` e' `}`
$=$ `if (`e_0 `>=` k`) then let {` x `= (`e_0`-`k`) } in` e `else` e'

(i) `case` e_0 `of {` x `->` e`; _ ->` e' `}` $=$ `case` e_0 `of {` x `->` e `}`

(j) `case` e_0 `of {` x `->` e `}` $=$ `( \` x `->` e `)` e_0

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

(l) `case (`K e_1 ... e_n`) of {` K x_1 ... x_n `->` e`; _ ->` e' `}`
$=$ `case` e_1 `of {` x_1 `-> ... case` e_n `of {` x_n `->` e `}...}`
where K is a constructor of arity n

Figure 3: Semantics of Case Expressions

4 Declarations and Bindings

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

module	→	`module` *modid* [*exports*] `where` *body*	
	$\mid$	*body*	
body	→	`{` [*impdecls* `;`] [[*fixdecls* `;`] *topdecls* [`;`]] `}`	
	$\mid$	`{` *impdecls* [`;`] `}`	
topdecls	→	*topdecl*$_1$ `;` ... `;` *topdecl*$_n$	$(n \geq 1)$
topdecl	→	`type` *simple* `=` *type*	
	$\mid$	`data` [*context* `=>`] *simple* `=` *constrs* [`deriving` (*tycls* $\mid$ `(`*tyclses*`)`)]	
	$\mid$	`class` [*context* `=>`] *class* [`where` `{` *cbody* [`;`] `}`]	
	$\mid$	`instance` [*context* `=>`] *tycls* *inst* [`where` `{` *valdefs* [`;`] `}`]	
	$\mid$	`default` (*type* $\mid$ `(`*type*$_1$ `,` ... `,` *type*$_n$`)`)	$(n \geq 0)$
	$\mid$	*decl*	
decls	→	*decl*$_1$ `;` ... `;` *decl*$_n$	$(n \geq 0)$
decl	→	*vars* `::` [*context* `=>`] *type*	
	$\mid$	*valdef*	

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

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

Haskell has several primitive datatypes that are "hard-wired" (such as integers and arrays), but most "built-in" datatypes are defined in the standard prelude with normal Haskell code, using `type` and `data` declarations. These "built-in" datatypes are described in detail in Section 6.

4.1 Overview of Types and Classes

Haskell uses a traditional Hindley-Milner polymorphic type system to provide a static type semantics [4, 7], but the type system has been extended with *type classes* (or just *classes*) that provide a structured way to introduce *overloaded* functions. This is the major technical innovation in Haskell.

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 *methods*—instantiated on the named type.

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

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

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

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

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

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

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

More examples can be found in Wadler and Blott's paper [21].

4.1.1 Syntax of Types

type	$\rightarrow$	*atype*	
	$\mid$	*type*$_1$ `->` *type*$_2$	
	$\mid$	*tycon atype*$_1$ ... *atype*$_k$	(arity *tycon* $= k \geq 1$)
atype	$\rightarrow$	*tyvar*	
	$\mid$	*tycon*	(arity *tycon* $= 0$)
	$\mid$	`()`	(unit type)
	$\mid$	`(` *type* `)`	(parenthesised type)
	$\mid$	`(` *type*$_1$ `,` ... `,` *type*$_k$ `)`	(tuple type, $k \geq 2$)
	$\mid$	`[` *type* `]`	

The syntax for Haskell *type expressions* is given above. They are built in the usual way from type variables, function types, type constructors, tuple types, and list types. Type variables are identifiers beginning with a lower-case letter and type constructors are identifiers beginning with an upper-case letter. A type is one of:

1. A *function type* having form t_1 `->` t_2. Function arrows associate to the right.

2. A *constructed type* having form $T\ t_1\ \ldots\ t_k$, where T is a type constructor of arity k.

3. A *tuple type* having form `(`t_1 `,` ... `,` t_k`)` where $k \geq 2$. 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 3.6 and 6.5).

4. A *list type* has the form `[`t`]`. It denotes the type of lists with elements of type t (see Sections 3.5 and 6.4).

5. The *trivial type* having form `()`. It denotes the "nullary tuple" type, and has exactly one value, also written `()` (see Sections 3.7 and 6.6).

6. A *parenthesised type*, having form `(`t`)`, is identical to the type t.

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

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

With one exception, the type variables in a Haskell type expression are all assumed to be universally quantified; there is no explicit syntax for universal quantification [4, 17]. For example, the type expression `a -> a` denotes the type $\forall a.\ a \rightarrow a$. For clarity, however, we will often write quantification explicitly when discussing the types of Haskell programs.

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

4.1.2 Syntax of Class Assertions and Contexts

context	$\rightarrow$	*class*	
	|	`(` *class*$_1$ `,` ... `,` *class*$_n$ `)`	$(n \geq 1)$
class	$\rightarrow$	*tycls tyvar*	
tycls	$\rightarrow$	*aconid*	
tyvar	$\rightarrow$	*avarid*	

A *class assertion* has form *tycls tyvar*, and indicates the membership of the parameterised type *tyvar* in the class *tycls*. A class identifier begins with a capital letter.

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

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

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

4.1.3 Semantics of Types and Classes

In this subsection, we provide informal details of the type system. (Wadler and Blott [21] discuss type classes further.)

The Haskell type system attributes a *type* to each expression in the program. In general, a type is of the form $\forall \overline{u}.\ c \Rightarrow t$, where $\overline{u}$ is a set of type variables $u_1, \ldots, u_n$. In any such type, any of the universally-quantified type variables u_i which are free in c must also be free in t.

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 generalisation order (specified below); the most general type that can be assigned to a particular expression (in a given environment) is called its *principal type*. Haskell's extended Hindley-Milner type system can infer the principal type of all expressions, including the proper use of overloaded operations (although certain ambiguous overloadings could arise, as described in Section 4.3.4). Therefore, explicit typings (called *type signatures*) are optional (see Sections 3.12 and 4.4.1).

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

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

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

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

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

4.2 User-Defined Datatypes

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

4.2.1 Algebraic Datatype Declarations

topdecl	$\rightarrow$	`data` [*context* `=>`] *simple* `=` *constrs* [`deriving` (*tycls* | `(`*tyclses*`)`)]	
simple	$\rightarrow$	*tycon* $tyvar_1$ … $tyvar_k$	(arity *tycon* $= k \geq 0$)
constrs	$\rightarrow$	$constr_1$ `|` … `|` $constr_n$	($n \geq 1$)
constr	$\rightarrow$	*con* $atype_1$ … $atype_k$	(arity *con* $= k \geq 0$)
	|	$type_1$ *conop* $type_2$	(infix *conop*)
tyclses	$\rightarrow$	$tycls_1$ `,` … `,` $tycls_n$	($n \geq 0$)

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

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

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

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

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

where c_i is the largest subset of c that constrains only those type variables free in the types $t_{i1}, \ldots, t_{ik_i}$. The type variables u_1 through u_k must be distinct and may appear in c and the t_{ij}; it is a static error for any other type variable to appear in c or on the right-hand-side.

For example, the declaration

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

introduces a type constructor `Set`, and constructors `NilSet` and `ConsSet` with types

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

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`. The context in the `data` declaration has no other effect whatsoever. In particular, pattern matching is unaffected.

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 5.6.

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

4.2.2 Type Synonym Declarations

topdecl	$\rightarrow$	`type` *simple* `=` *type*	
simple	$\rightarrow$	*tycon* $tyvar_1$ ... $tyvar_k$	(arity *tycon* $= k \geq 0$)

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

`type` T u_1 ... u_k `=` t

which introduces a new type constructor, T. The type (T t_1 ... 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.

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

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

4.3 Type Classes and Overloading

4.3.1 Class Declarations

topdecl	$\rightarrow$	`class` [*context* `=>`] *class* [`where {` *cbody* [`;`] `}`]	
cbody	$\rightarrow$	[*csigns* `;`] [*valdefs*]	
csigns	$\rightarrow$	$csign_1$ `;` ... `;` $csign_n$	($n \geq 1$)
csign	$\rightarrow$	*vars* `::` [*context* `=>`] *type*	
vars	$\rightarrow$	var_1 `,` ... `,` var_n	($n \geq 1$)

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

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

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

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

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

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

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

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

Two classes in scope at the same time may not share any of the same methods.

Figure 4 shows some standard Haskell classes, including the use of superclasses; note the class inclusion diagram on the right. For example, `Eq` is a superclass of `Ord`, and thus in any context `Ord a` is equivalent to `(Eq a, Ord a)`.

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 operations in the original ones. For example:

```
class  (Ord a, Text a, Binary a) => Data 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 operations. The `instance` declaration must be given explicitly, and it must have an empty `where` part as well.

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

4.3.2 Instance Declarations

topdecl	$\rightarrow$	`instance` [*context* `=>`] *tycls* *inst* [`where` `{` *valdefs* [`;`] `}`]	
inst	$\rightarrow$	*tycon*	(arity *tycon* $= 0$)
	$\vert$	`(` *tycon* $tyvar_1$ … $tyvar_k$ `)`	($k \geq 1$, *tyvars* distinct)
	$\vert$	`(` $tyvar_1$ `,` … `,` $tyvar_k$ `)`	($k \geq 2$, *tyvars* distinct)
	$\vert$	`()`	
	$\vert$	`[` *tyvar* `]`	
	$\vert$	`(` $tyvar_1$ `->` $tyvar_2$ `)`	$tyvar_1$ and $tyvar_2$ distinct
valdefs	$\rightarrow$	$valdef_1$ `;` … `;` $valdef_n$	($n \geq 0$)

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

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

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

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

```
class  Eq a  where                                  --   Eq
        (==), (/=)  ::  a -> a -> Bool              --    |
                                                    --   Ord
        x /= y   =  not (x == y)                    --   / \
                                                    --  Ix Enum
class  (Eq a) => Ord a  where
        (<), (<=), (>=), (>) ::  a -> a -> Bool
        max, min             ::  a -> a -> a

        x <  y               =  x <= y && x /= y
        x >= y               =  y <= x
        x >  y               =  y <  x
        max x y | x >= y     =  x
                | y >= x     =  y
        min x y | x <= y     =  x
                | y <= x     =  y

class  Text a  where
        showsPrec :: Int -> a -> String -> String
        readsPrec :: Int -> String -> [(a,String)]
        showList  :: [a] -> String -> String
        readList  :: String -> [([a],String)]

        showList = ... -- see Appendix A
        readList = ... -- see Appendix A

class  Binary a  where
        showBin :: a -> Bin -> Bin
        readBin :: Bin -> (a,Bin)

class  (Ord a) => Ix a  where
        range   :: (a,a) -> [a]
        index   :: (a,a) -> a -> Int
        inRange :: (a,a) -> a -> Bool

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

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

Figure 4: Standard Classes and Associated Functions

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

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

The context c' must imply the context $c[(T\ u_1\ \ldots\ u_k)/u]$, and d may contain bindings only for the class methods of C. No type signatures may appear in d, as the signatures for the methods have already been given in the `class` declaration.

If no binding is given for some class method then the corresponding default method in the `class` declaration is used (if present); if such a default does not exist then the class method at this instance is implicitly bound to the completely undefined function (of the appropriate type) and no static 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 C-T instance declaration may only appear either in the module in which C is declared or in the module in which T is declared, and only where both C and T are in scope.
- A type may not be declared as an instance of a particular class more than once in the same scope.

Examples of `instance` declarations may be found in the next section on derived instances.

4.3.3 Derived Instances

As mentioned in Section 4.2.1, `data` 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. If a derived instance of a subclass is asked for, then each of the superclasses must either be asked for or an explicit instance declaration must be given for it.

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 for which derived instances are allowed are `Eq`, `Ord`, `Ix`, `Enum`, `Text`, and `Binary`, all defined in Figure 4, page 30. The precise details of how the derived instances are generated for each of these classes are provided in Appendix D, including a specification of when such derived instances are possible.

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

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

4.3.4 Defaults for Overloaded Operations

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

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

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

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

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

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

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

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

Overloading ambiguity, although rare, can only be circumvented by input from the user. One way is through the use of *expression type-signatures* as described in Section 3.12. For example, for the ambiguous expression given earlier, one could write:

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

which disambiguates the type.

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

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

(See Section 6.8.8.)

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

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

where $n \geq 0$ (the parentheses may be omitted when $n = 1$), and each t_i must be a monotype for which `Num` t_i holds. In situations where an ambiguous type is discovered, an ambiguous type variable is defaultable if at least one of its classes is a numeric class and if all of its classes are either numeric classes or standard classes. (Figures 8–10, pages 57–59, show the numeric classes, and Figure 4, page 30, shows the standard classes.) 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 defaults to:

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

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

4.4 Nested Declarations

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

4.4.1 Type Signatures

decl	$\rightarrow$	*vars* `::` [*context* `=>`] *type*	
vars	$\rightarrow$	var_1 `,` ... `,` var_n	$(n \geq 1)$

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

$$x_1,\ \ldots,\ x_n\ \texttt{::}\ c\ \texttt{=>}\ t$$

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

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

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

the `a`'s in the two type signatures are quite distinct. Indeed, these declarations contain a static error, since `x` does not have type $\forall a.\ a$.

A type signature for x may be more specific than the principal type derivable from the value binding of x (see Section 4.1.3), but it is an error to give a type that is more general than, or incomparable to, the principal type. If a more specific type is given then all occurrences of the variable must be used at the more specific type or at a more specific type still. 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 legal to declare a more specific type, such as

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

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

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

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

4.4.2 Function and Pattern Bindings

decl	$\rightarrow$	*valdef*	
valdef	$\rightarrow$	*lhs* `=` *exp* [`where {` *decls* [`;`] `}`]	
	$\vert$	*lhs gdrhs* [`where {` *decls* [`;`] `}`]	
lhs	$\rightarrow$	*apat*	
	$\vert$	*funlhs*	
funlhs	$\rightarrow$	*afunlhs*	
	$\vert$	$pat_1^{i+1}\ varop^{(\mathrm{n},i)}\ pat_2^{i+1}$	$(0 \leq i \leq 9)$
	$\vert$	$lpat^{i}\ varop^{(\mathrm{l},i)}\ pat^{i+1}$	$(0 \leq i \leq 9)$
	$\vert$	$pat^{i+1}\ varop^{(\mathrm{r},i)}\ rpat^{i}$	$(0 \leq i \leq 9)$
afunlhs	$\rightarrow$	*var apat*	
	$\vert$	`(` *funlhs* `)` *apat*	
	$\vert$	*afunlhs apat*	
gdrhs	$\rightarrow$	*gd* `=` *exp* [*gdrhs*]	
gd	$\rightarrow$	`|` *exp*	

We distinguish two cases within this syntax: a *pattern binding* occurs when *lhs* is *apat*; 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. The use of the nonterminal *apat* (rather than *pat*) in the production for *lhs* disallows top level n+k pattern bindings; otherwise, programs such as `x + 2 = 3` could be parsed either as a definition of `+` or as a pattern binding.

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:

$$= e \ \texttt{where} \ \{ \ decls \ \}$$

or

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

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:

$$\texttt{| True =} \ e \ \texttt{where} \ \{ \ decls \ \}$$

The set of patterns corresponding to each match must be *linear*—no variable is allowed to appear more than once in the entire set.

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

```
plus x y z = x+y+z
(x `plus` y) z = x+y+z
```

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

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

where the x_i are new identifiers.

Pattern bindings. A pattern binding binds variables to values. A *simple* pattern binding has form $p \ = \ e$. In both a `where` or `let` clause and at the top level of a module, the pattern p is matched "lazily" as an irrefutable pattern by default (as if there were an implicit `~` in front of it). See the translation in Section 3.10.

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:

```
p  | g1  = e1
   | g2  = e2
   ...
   | gm  = em
   where { decls }
```

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

```
p  =  let decls in
      if g1 then e1 else
      if g2 then e2 else
      ...
      if gm then em else error "Unmatched pattern"
```

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 is discussed in this section.

4.5.1 Dependency analysis

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

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

Careful application of the following rules causes each `let` or `where` construct to bind only the variables of a single declaration group, thus capturing the required dependency analysis:[2]

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

[2] A similar transformation is described in Peyton Jones' book [14].

4.5.2 Generalisation

The Hindley-Milner type system assigns types to a `let`-expression in two stages. First, the right-hand side of the declaration is typed, giving a type with no universal quantification. Second, all type variables which occur in this type are universally quantified unless they are associated with bound variables in the type environment; this is called *generalisation*. 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 $a \to (a, a)$. The generalisation step attributes to `g` the polymorphic type $\forall a.\ a \to (a, a)$, 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 $a \to a \to$ `String`, and the accumulated constraints are `Ord` a (arising from the use of `>`), and `Text` a (arising from the use of `show`). The type variables appearing in this collection of constraints are called the *constrained type variables*.

The generalisation step attributes to both `g1` and `g2` the type $\forall a.$ (`Ord` a, `Text` a) $\Rightarrow a \to a \to$ `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 Monomorphism

Sometimes it is not possible to generalise 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 $a \to b \to$ `(`$[a], b$`)`. The generalisation step attributes to `g` the type $\forall b.\ a \to b \to$ `(`$[a], b$`)`; only b can be universally quantified because a occurs in the type environment. We say that the type of `g` is *monomorphic in the type variable a*.

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

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

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

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

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

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

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

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

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

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

4.5.4 The monomorphism restriction

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

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

Two rules define the monomorphism restriction:

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

(a): every variable in the group is bound by a function binding or a simple pattern binding, *and*

(b): an explicit type signature is given for every variable in the group which is bound by simple pattern binding.

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

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

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

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

Now consider the following expression:

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

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

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

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

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

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

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

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

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

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

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

Lastly, Rule 2 is required because there is no way to enforce monomorphic use of an exported binding, except by performing type inference on the entire program at once.

The monomorphism rule has a number of consequences for the programmer. Anything defined with function syntax will usually generalize 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:

```
indices :: (Ix a) => Array a b -> [a]
indices =  range . bounds
```

5 Modules

A module defines a collection of values, datatypes, type synonyms, classes, etc. (see Section 4), and *exports* some of these resources, making them available to other modules. We use the term *entity* to refer to the values, types, and classes defined in and perhaps exported from a module.

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

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

The name-space for modules is flat, with each module being associated with a unique module name (which are Haskell identifiers beginning with a capital letter; i.e. *aconid*). There are two distinguished modules, `PreludeCore` and `Prelude`, both discussed in Section 5.4.

5.1 Overview

5.1.1 Interfaces and Implementations

A module consists of an *interface* and an *implementation* of that interface.

The interface of a module provides complete information about the static semantics of that module, including type signatures, class definitions, and type declarations for the various entities made available by the module. This information is complete in this sense: If a module M imports modules $M_1, \ldots, M_n$, then only the interfaces of $M_1, \ldots, M_n$ need be examined in order to perform static checking on the implementation of M. No implementations of $M_1, \ldots, M_n$ need to exist, nor need any further interfaces be consulted. Interfaces are discussed in Section 5.3.

An implementation "fills in" the information about a module missing from the interface. For example, for each value given a type signature in the interface the implementation either imports a module that defines the value or defines the value itself. Implementations are discussed in Section 5.2.

5.1.2 Original Names

It may be that a particular entity is imported into a module by more than one route—for example, because it is exported by two modules both of which are imported by a third module. It is important that benign name-clashes of this form are allowed, but that accidental name-clashes are detected and reported as errors. This is done as follows:

Each entity (class, type constructor, value, etc.) has an *original name* that is a pair consisting of the name of the module in which it was originally declared, and the name it was given in that declaration. The original name is carried with the entity wherever it is exported. Two entities are the same if and only if they have the same original name.

Renaming does *not* affect the original name; it is a purely syntactic operation that affects only the name by which the entity is currently known. For example, if a class is renamed and a type is declared to be an instance of the newly-named class, then it is also an instance of the original class—there is just one class, which happens to be known by different names in different parts of the program. Also, fixity is a property of the original name of an identifier or operator and is not affected by renaming; the new name has the same fixity as the old one.

A given entity may be known by at most one name in any scope. So, for example, a module may not import an entity twice and rename it differently on each occasion. Either it must be renamed in the same way on each import or else not imported twice (for example, by using a `hiding` clause).

As there are several name spaces, a single name may identify more than one entity. In a `renaming` clause, such as `renaming(`$\ldots, n_1$ `to` $n_2, \ldots$`)`, *all* the entities to which n_1 refers are renamed to n_2.

5.1.3 Closure

The implementation together with the interfaces of the modules it imports must be *statically closed* according to this rule: *every value, type, or class referred to in the text of an implementation together with the entities that it imports, must be declared in the implementation or in one of the imported interfaces.*

It is an error for a module to export a collection of entities that cannot possibly become closed. For example, if a module `A` declares both the type `T` and a value `t` of type `T`, it may not export `t` without also exporting `T`. But if another module `B` imported `T` from module `A`, and declared another value `s` of type `T`, it may export `s` without exporting `T`—but any module importing `B` must also import the type `T` by some other route, for example by also importing `A`.

5.1.4 The Compilation System

The task of checking consistency between interfaces and implementations must be done by the *compilation system*.

Haskell does not specify any particular association between implementations and interfaces on the one hand, and *files* on the other; nor does it specify how implementations and interfaces are produced. These matters are determined by the compilation system, and many variations are possible, depending on the programming environment. For example, a compilation system could insist that each implementation and each interface reside alone in a file, and that the module name is the same as that of the file, with the implementation and interface distinguished by a suffix.

Similarly, a compilation system may require the programmer to write the interface, or it may derive the interface from examination of the implementation, or some hybrid of the two. Haskell is defined so that, given the interfaces of all imported modules, it is always possible to perform a complete static check on the implementation, and, if it is well-typed, to derive its unique interface automatically. However, given a set of mutually recursive implementations, the compilation system may have to examine several modules at once to derive the interfaces, which will still be unique with one exception: because of the shorthand for exporting all entities from an imported module, the set of exports may not be unique. Any set satisfying the consistency constraints is a valid solution for a well-typed Haskell program, but if an implementation automatically derives the interface it must derive the smallest set of exports.

For optimisation across module boundaries, a compilation system may need more information (e.g., information about strictness, inlining, uncurrying, etc.) than is provided by the standard interface as defined in this report. Draft proposals exist for including such information as comments in interfaces; for details, contact the implementors listed in the preface (page iv).

5.2 Module Implementations

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

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

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

An abbreviated form of module is permitted, which consists only of the module body. If this is used, the header is assumed to be `module Main where`. If the first lexeme in the abbreviated module is not a `{`, then the layout rule applies for the top level of the module. It is inadvisable for a compilation system to permit an abbreviated module to appear in the same file as some unabbreviated modules.

5.2.1 Export Lists

exports	$\rightarrow$	`(` *export₁* `,` ... `,` *exportₙ* `)`	$(n \geq 1)$
export	$\rightarrow$	*entity*	
	$\mid$	*modid* `..`	
entity	$\rightarrow$	*varid*	
	$\mid$	*tycon*	
	$\mid$	*tycon* `(..)`	
	$\mid$	*tycon* `(` *conid₁* `,` ... `,` *conidₙ* `)`	$(n \geq 1)$
	$\mid$	*tycls* `(..)`	
	$\mid$	*tycls* `(` *varid₁* `,` ... `,` *varidₙ* `)`	$(n \geq 0)$

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

Entities in an export list may be named as follows:

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

2. A type synonym T declared by a `type` declaration may be named by simply giving the name of the type.

3. An algebraic datatype T with constructors $K_1, \ldots, K_n$ declared by a `data` declaration may be named in one of three ways:

   - The form T names the type *but not the constructors.* The ability to export a type without its constructors allows the construction of abstract datatypes (see Section 5.6).
   - The form T`(`K_1`,` ... `,`K_n`)`, where *all* and only the constructors are listed without duplications, names the type and *all* its constructors.
   - The abbreviated form T`(..)` also names the type and all its constructors.

   Data constructors may not be named in export lists in any other way.

4. A class C with operations $f_1, \ldots, f_n$ declared in a `class` declaration may be named in one of two ways, both of which name the class together with all its operations:

   - The form C`(`f_1`,` ... `,`f_n`)`, where all and only the operations in that class are listed without duplications.
   - The abbreviated form C`(..)`.

Operators in a class may not be named in export lists in any other way.

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

```
module Queue( 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`.

   It is a static error to have circular dependencies between imports/exports using this naming convention. For example, the following is not allowed:

```
module X( Y.. )     -- ILLEGAL
import Y            --
x = 1               --

module Y( X.. )     --
import X            --
y = 1               --
```

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

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

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

5.2.2 Import Declarations

impdecl	$\rightarrow$	`import` *modid* [*impspec*] [`renaming` *renamings*]	
impspec	$\rightarrow$	`(` $import_1$ `,` ... `,` $import_n$ `)`	$(n \geq 0)$
	$\vert$	`hiding (` $import_1$ `,` ... `,` $import_n$ `)`	$(n \geq 1)$
import	$\rightarrow$	*entity*	
renamings	$\rightarrow$	`(` $renaming_1$ `,` ... `,` $renaming_n$ `)`	$(n \geq 1)$
renaming	$\rightarrow$	$varid_1$ `to` $varid_2$	
	$\vert$	$conid_1$ `to` $conid_2$	

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, optionally specifies the entities to be imported, and optionally provides renamings for imported entities. A single module may be imported by more than one `import` declaration.

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

1. The set of entities to be imported can be specified explicitly by listing them in parentheses. Items in the list have the same form as those in export lists, except that the *modid* abbreviation is not permitted.

   The list must name a subset of the entities exported by the imported module. The list may be empty, in which case nothing is imported; this is only useful in the case of the module `Prelude` (see Section 5.4.3).

2. Specific 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 apart from those named in the list.

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

As instance declarations do not have names, their import cannot be controlled by the *impspec* list. Instead, the following rule is used: *A C-T instance declaration is imported from an interface if and only if C is imported or T is imported from that interface.*

Some or all of the imported entities may be renamed, thus allowing them to be known by a new name in the importing scope (see Section 5.1.2). This is done using the `renaming` keyword, with a renaming of the form *oldname* `to` *newname*.

5.3 Module Interfaces

Every module has an *interface* containing all the information needed to do static checks on any importing module. All static checks on a module implementation can be done by inspecting its text and the interfaces of the modules it imports.

$$
\begin{array}{lcll}
\mathit{interface} & \rightarrow & \texttt{interface}\ \mathit{modid}\ \texttt{where}\ \mathit{ibody} & \\
 & & & \\
\mathit{ibody} & \rightarrow & \texttt{\{}\ [\mathit{iimpdecls}\ \texttt{;}]\ [\mathit{fixdecls}\ \texttt{;}]\ \mathit{itopdecls}\ [\texttt{;}]\ \texttt{\}} & \\
 & | & \texttt{\{}\ \mathit{iimpdecls}\ [\texttt{;}]\ \texttt{\}} & \\
\mathit{iimpdecls} & \rightarrow & \mathit{iimpdecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{iimpdecl}_n & (n \geq 1) \\
\mathit{iimpdecl} & \rightarrow & \texttt{import}\ \mathit{modid}\ \texttt{(}\ \mathit{import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{import}_n\ \texttt{)} & \\
 & & [\texttt{renaming}\ \mathit{renamings}] & (n \geq 1) \\
\mathit{itopdecls} & \rightarrow & \mathit{itopdecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{itopdecl}_n & (n \geq 1) \\
\mathit{itopdecl} & \rightarrow & \texttt{type}\ \mathit{simple}\ \texttt{=}\ \mathit{type} & \\
 & | & \texttt{data}\ [\mathit{context}\ \texttt{=>}]\ \mathit{simple}\ [\texttt{=}\ \mathit{constrs}]\ [\texttt{deriving}\ (\mathit{tycls}\ |\ \texttt{(}\mathit{tyclses}\texttt{)})] & \\
 & | & \texttt{class}\ [\mathit{context}\ \texttt{=>}]\ \mathit{class}\ [\texttt{where \{}\ \mathit{icdecls}\ [\texttt{;}]\ \texttt{\}}] & \\
 & | & \texttt{instance}\ [\mathit{context}\ \texttt{=>}]\ \mathit{tycls}\ \mathit{inst} & \\
 & | & \mathit{vars}\ \texttt{::}\ [\mathit{context}\ \texttt{=>}]\ \mathit{type} & \\
\mathit{icdecls} & \rightarrow & \mathit{icdecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{icdecl}_n & (n \geq 1) \\
\mathit{icdecl} & \rightarrow & \mathit{vars}\ \texttt{::}\ \mathit{type} &
\end{array}
$$

The syntax of `interface` is similar to that of `module`, except:

- There is no export list: everything in the interface is exported.
- `import` declarations have a slightly different purpose from those in implementations (see Section 5.3.2). The list of entities to be imported is always specified explicitly.
- `data` declarations appear without their constructors if these are not exported.
- There is no implementation part to `instance` declarations.
- Value declarations do not appear at all; for exported values, type signatures take their place.

5.3.1 Consistency

The interface and implementation of a module must obey certain constraints. (In the following, the phrase "in the implementation" refers to something either declared within the implementation or imported by it.)

1. Every entity given a declaration in an interface must either have an import declaration for the entity in the interface (the import specifies the module that defines it) or have a definition of the entity in the implementation. Furthermore, if an interface A imports an entity X from module B (perhaps renaming it), then the interface for B must define X but not import it.
2. A class, type synonym, algebraic datatype, or value appears in the interface exactly when its name appears in the implementation's export list or, if the export list is omitted, when it is *declared* in the implementation.
3. A type signature appears in the interface for every value that the implementation exports. This type signature must be the same as that in the implementation (see Section 4.1.3), where the latter is obtained from the explicit type signature in the implementation (when present) or is the most general type inferred from the declaration of the value.
4. A `type` declaration in an interface must be identical to that in the implementation.
5. A `class` declaration in an interface must be identical to that in the implementation, except that default-method declarations are omitted.
6. If the constructors of a `data` declaration are not exported, then the `data` declaration in the interface differs from that in the implementation by omitting everything after (and including) the `=` sign. If the `data` declaration in the implementation uses the `deriving` mechanism to derive instance declarations for the type, a separate `instance` declaration must appear in the interface for each class of which the type is made an instance of. Hence, the information that certain instances are derived is hidden when the constructors are hidden, since in this case the type is abstract (see Section 5.6).

7. If the constructors of a `data` declaration are to be exported, then the `data` declaration in the interface is identical to that in the implementation including the `deriving` part.[3]

8. If a C-T instance is declared in a module or imported by it, then the instance declaration appears in the interface (omitting the `where` part) if *either* C is exported *or* T is exported. Instance declarations are not named explicitly in export or import lists. This rule ensures that, if C and T are both in scope, then the (unique) C-T instance declaration will also be in scope.[4]

   No explicit instance declaration should appear in the interface for instances that are specified by the `deriving` part of a `data` declaration in the interface.

9. A fixity declaration for a value or constructor appears in an interface exactly when (a) the value or constructor is declared by the interface, and (b) the identical fixity declaration appears either in the implementation or in an imported interface.

5.3.2 Imports and Original Names

The original-name information is carried in the interface file using `import` declarations in a special way.

Suppose that a module `A` exports an entity `x`; the interface for `A` will contain static information about `x`. If `x` was originally defined in `A`, then this is all that appears. But, suppose that `x` was imported by `A` from some other module `B` and that `x` was originally defined in module `C` with name `y`; this declaration must appear in the interface for `A`:

```
import C(y) renaming ( y to x )
```

No reference to `B` remains in the interface. *The* `import` *declaration in the interface serves only to convey to the importing module the original name of* `x`, and does *not* imply that module `C`'s interface must be consulted when reading module `A`'s interface. Multiple imports from a single original module may optionally be grouped in a single import declaration in the interface.

A module may export a value whose typing involves a type and/or class that is not exported. (Any importing module would have to import the type or class by some other route.) *Nevertheless, it is still required that the interface contain the import declaration required to give the original name of the type or class.*

In summary, for every entity `e1` mentioned in the interface of a module `M` whose original name is `e2` in module `N`, `M`'s interface must contain the `import` declaration

```
import N(e2) renaming ( e2 to e1 )
```

The word "mentioned" includes mention in the type signature of an exported value, as discussed above.

[3]It is important to retain the information about which instances are derived and which are not, because the importing module "knows" more about derived instances.

[4]The reverse also applies. For example, suppose that a new type T is declared and made an instance of an imported class C. The instance declaration will be exported along with T, and so the closure rule (Section 5.1.3) will require that C is also in scope in every importing scope.

This example illustrates most of these constraints; first, the interface:

```
interface A where
infixr 4 `sameShape`
import PreludeList(sum) renaming ( sum to oldSum )
data  BinTree a = Empty | Branch a (BinTree a) (BinTree a)
class Tree a where
        sameShape :: a -> a -> Bool
instance Tree (BinTree a)
sum :: Num a => BinTree a -> a
oldSum :: Num a => [a] -> a
```

Now the implementation:

```
module A( BinTree(..), Tree(..), sum, oldSum ) where
import Prelude renaming ( sum to oldSum )
infixr 4 `sameShape`
        -- `sameShape` is an operation of class C below

data BinTree a = Empty | Branch a (BinTree a) (BinTree a)

class Tree a where
      sameShape :: a -> a -> Bool
      t1 `sameShape` t2 = False     -- Default method

instance Tree (BinTree a) where
         Empty `sameShape` Empty  =  True
         (Branch _ t1 t2) `sameShape` (Branch _ t1' t2')
            =  (t1 `sameShape` t1') && (t2 `sameShape` t2')
         t1 `sameShape` t2 = False

sum  Empty           = 0
sum (Branch n t1 t2) = n + sum t1 + sum t2
```

5.4 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 standard prelude is specified as two distinct modules (in the technical sense of this chapter), `PreludeCore` and `Prelude`.

`PreludeCore` and `Prelude` differ from other modules in that *their interfaces, and the semantics of the entities defined by those interfaces, are part of the Haskell language definition.* This means, for example, that a compiler may optimise calls to functions in the standard prelude, because it knows their semantics as well as their interface.

Each of these modules is structured into submodules. To avoid name-clashes with these sub-modules, user-defined module names must not begin with the prefix `Prelude`.

5.4.1 The `PreludeCore` Module

The `PreludeCore` module contains *all the algebraic datatypes, type synonyms, classes and instance declarations* specified by the standard prelude.

`PreludeCore` is *always implicitly imported*, so it is not possible to import only part of it or to rename any of the entities that it defines.

The semantics of the entities defined by `PreludeCore` is specified by an implementation written in Haskell, in Appendix A.2. A Haskell system need not implement `PreludeCore` in this way. The interface for `PreludeCore` may be inferred from the implementation in Appendix A.2.

Some datatypes (such as `Int`) and functions (such as addition of `Int`s) cannot be specified directly in Haskell. This is expressed in the `PreludeCore` implementation by importing these built-in types and values from `PreludeBuiltin`. The semantics of the built-in datatypes and functions is given as English text in Appendix A.1.

The implementation for `PreludeCore` is incomplete in its treatment of tuples: there should be an infinite family of instance declarations for tuples, but the implementation only gives a scheme.

The alert reader may notice that the implementation of `PreludeCore` given in Appendix A.2 uses some functions defined in `Prelude` (see next section). There is no conflict; `PreludeCore` and `Prelude` are mutually recursive.

5.4.2 The `Prelude` Module

The `Prelude` module contains all the *value* declarations in the standard prelude.

The `Prelude` module is imported automatically if and only if it is not imported with an explicit `import` declaration. This provision for explicit import allows values defined in the standard prelude to be renamed or not imported at all.

The semantics of the entities in `Prelude` is specified by an implementation of `Prelude` written in Haskell, given in Appendix A. As for `PreludeCore`, a Haskell system may implement the `Prelude` module as it pleases, provided it maintains the semantics in Appendix A. The interface can be inferred from this implementation.

5.4.3 Shadowing Prelude Names and Non-Standard Preludes

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

```
module A where
import Prelude hiding (map)
map f x = x f
```

Module `A` redefines `map`, but it must indicate this by importing `Prelude` without `map`.

Furthermore, `A` exports `map`, but every module that imports `map` from `A` must also hide `map` from `Prelude` just as `A` does. Thus there is little danger of accidentally shadowing standard prelude names.

It is possible to construct and use a different `Prelude` module:

```
module B where
import Prelude()
import MyPrelude
...
```

`B` imports nothing from `Prelude`, but the explicit `import Prelude` declaration prevents the automatic import of `Prelude`. `import MyPrelude` brings the non-standard prelude into scope. As before, the standard prelude names are hidden explicitly.

5.5 Example

As an example, here are two small modules:

```
module A( Tree(..), depth ) where
data Tree a = Leaf a | Branch (Tree a) (Tree a)
depth (Leaf a)        =  0
depth (Branch xt yt)  =  (depth xt `max` depth yt) + 1
```

```
module B( leaves ) where
import A
leaves (Leaf a)        =  [a]
leaves (Branch xt yt)  =  leaves xt ++ leaves yt
```

Module `A` must export `Tree` because it exports `depth`, and `Tree` could not be made visible in any other way. However, `B` is not required to export `Tree`, since a module importing `B` could import `A` in order to satisfy the closure constraints.

Modules may be used to combine the resources of other modules. For example, one might use renaming to make trees available to French speakers:

```
module C( Arbre(..), fond, feuilles ) where
import A renaming ( Tree to Arbre, Leaf to Feuille, Branch to Branche,
                    depth to fond )
import B renaming ( leaves to feuilles )
```

5.6 Abstract Datatypes

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

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

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

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

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

Note 1. Every ADT must be a module (but a Haskell compilation system may allow multiple modules in a single file).

Note 2. Using a single-constructor single-field `data` declaration to create an isomorphic type introduces an unwanted extra element to the new type, namely (`Stk` $\perp$), with the risk of an accompanying small inefficiency in the implementation.

5.7 Fixity Declarations

fixdecls	$\rightarrow$	fix_1 `;` ... `;` fix_n	$(n \geq 1)$
fix	$\rightarrow$	`infixl` [*digit*] *ops*	
	|	`infixr` [*digit*] *ops*	
	|	`infix` [*digit*] *ops*	
ops	$\rightarrow$	op_1 `,` ... `,` op_n	$(n \geq 1)$
op	$\rightarrow$	*varop* | *conop*	

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

There are three kinds of fixity, non-, left- and right-associativity (`infix`, `infixl`, and `infixr`, respectively), and ten precedence levels, 0 through 9 (level 0 binds least tightly, and level 9 binds most tightly). If the *digit* is omitted, level 9 is assumed. Any operator lacking a fixity declaration is assumed to be `infixl 9` (See Section 3 for more on the use of fixities). Figure 5 lists the fixities and precedences of the operators defined in the standard prelude.

Fixity is a property of the original name of an identifier or operator (see Section 5.1.2). Fixity is not affected by renaming; the new name has the same fixity as the old one. The same fixity attaches to every occurrence of an operator name in a module, whether at the top level or rebound at an inner level. For example:

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

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

<table>
<tr><th>Precedence</th><th>Fixity</th><th>Operators</th></tr>
<tr><td>9</td><td>infixl</td><td>!, !!, //</td></tr>
<tr><td></td><td>infixr</td><td>.</td></tr>
<tr><td>8</td><td>infixr</td><td>**, ^, ^^</td></tr>
<tr><td>7</td><td>infixl</td><td>%, *, :%</td></tr>
<tr><td></td><td>infix</td><td>/, `div`, `mod`, `rem`</td></tr>
<tr><td>6</td><td>infixl</td><td>+, -</td></tr>
<tr><td></td><td>infix</td><td>:+</td></tr>
<tr><td>5</td><td>infixr</td><td>:, ++</td></tr>
<tr><td></td><td>infix</td><td>\\</td></tr>
<tr><td>4</td><td>infix</td><td>/=, <, <=, ==, >, >=, `elem`, `notElem`</td></tr>
<tr><td>3</td><td>infixr</td><td>&&</td></tr>
<tr><td>2</td><td>infixr</td><td>||</td></tr>
<tr><td>1</td><td>infix</td><td>:=</td></tr>
</table>

Figure 5: Precedences and fixities of prelude-defined operators

Here `` `op` `` has fixity 3 wherever it is in scope, provided `Bar` does not export the identifier `op`. If `Bar` does export `op`, then the example becomes illegal, because the fixity (or lack thereof) of `op` is defined in `Bar` (or wherever `Bar` imported `op` from).

```
data  Bool  =  False | True

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

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

otherwise               :: Bool
otherwise               = True
```

Figure 6: Standard functions on booleans

6 Basic Types

6.1 Booleans

The boolean type `Bool` is an enumeration; Figure 6 shows its definition and standard functions `&&`, `||`, `not`, and `otherwise`.

6.2 Characters and Strings

The character type `Char` is an enumeration, and consists of 256 values, of which the first 128 are the ASCII character set. The lexical syntax for characters is defined in Section 2.5; character literals are nullary constructors in the datatype `Char`. The standard prelude provides an instance declaration for `Char` in classes `Enum` and `Ix` and two functions relating characters to `Int`s in the range $[0, 255]$:

```
ord :: Char -> Int
chr :: Int  -> Char
```

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

A *string* is a list of characters:

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

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

```
[´A´,´ ´,´s´,´t´,´r´, ´i´,´n´,´g´]
```

6.3 Functions

Functions are defined via lambda abstractions and function definitions. Besides application, an infix composition operator is defined:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

The function `until` applies a function to an initial value zero or more times until the result satisfies a given predicate:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x | p x        =  x
            | otherwise  =  until p f (f x)
```

The function `flip`, applied to a binary function, reverses the order of the arguments:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y  =  f y x
```

6.4 Lists

Lists are an algebraic datatype of two constructors, although with special syntax, as described in Section 3.5. The first constructor is the null list, written `[]`, and the second is `:` ("cons"). See the standard prelude (Appendix A.5) for the definitions of the standard list functions. *Arithmetic sequences* and *list comprehensions*, two convenient syntaxes for special kinds of lists, are described in Sections 3.8 and 3.9, respectively.

6.5 Tuples

Tuples are also algebraic datatypes with special syntax, as defined in Section 3.6. Each tuple type has a single constructor. Six functions, named `zip`, `zip3`, ..., `zip7`, are provided by the standard prelude (Appendix A.5). These produce lists of n-tuples from n lists, for $2 \leq n \leq 7$. The resulting lists are as long as the shortest argument list; excess elements of other argument lists are ignored.

6.6 Unit Datatype

The unit datatype `()` has one member, the nullary constructor `()` (and thus an overloading of syntax)—see also Section 3.7.

6.7 Binary Datatype

The `Bin` datatype is a primitive abstract datatype including the value `nullBin` (the empty or nullary binary value), the function `appendBin`, and the predicate `isNullBin` (which returns `True` when applied to `nullBin` and `False` when applied to all other values of type `Bin`).

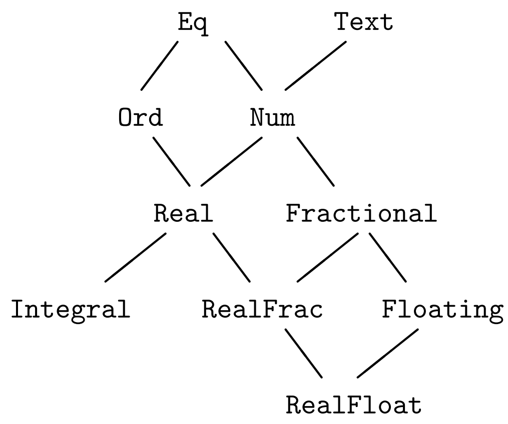

Figure 7: Numeric class inclusions (cf. Figure 4, page 30)

Also, derived instances of the `Binary` class generate definitions for `showBin` and `readBin`, as described in Section 4.3.3 and Appendix D. The `Bin` datatype is used primarily for efficient and transparent I/O, as described in Section 7.

6.8 Numbers

6.8.1 Introduction

Haskell provides several kinds of numbers; the numeric types and the operations upon them have been heavily influenced by Common Lisp [18] and Scheme [16]. Numeric function names and operators are usually overloaded, using several type classes with an inclusion relation shown in Figure 7 (cf. Figure 4, page 30). (Some classes are immediate subclasses of two other classes; there are pairs of classes with a nontrivial intersection.) 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. The class `Integral` contains both fixed- and arbitrary-precision integers; the class `Fractional` contains all nonintegral types; and the class `Floating` contains all floating-point types, both real and complex.

Table 1 lists the standard numeric types. The type `Int` is a fixed-precision type, covering at least the range $[-2^{29}+1, 2^{29}-1]$. The range chosen by an implementation must either be symmetric about zero or contain one more negative value than positive (to accommodate twos-complement representation) and should be large enough to serve as array indices. The constants `minInt` and `maxInt` (Figure 9, page 58) define the limits of `Int` in each implementation. `Float` is a floating-point type, also implementation-defined; it is desirable that this type be at least equal in range and precision to the IEEE single-precision type.

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 1: Standard numeric types

Similarly, `Double` should cover IEEE double-precision. An implementation may provide other numeric types, such as additional precisions of integer and floating-point. 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 interface text (Section 5.3) associated with the standard numeric classes, types, and operations is shown in Figures 8–10.

6.8.2 Numeric Literals

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

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

integer and floating literals have the typings `(Num a) => a` and `(Fractional a) => a`, respectively. Numeric literals are defined in this indirect way so that they may be interpreted as values of any appropriate numeric type. For example, `fromInteger` for complex numbers is defined as follows:

```
fromInteger n = fromInteger n :+ 0
```

See Section 4.3.4 for a discussion of overloading ambiguity.

6.8.3 Constructed Numbers

There are two kinds of numeric types formed by data constructors: namely, `Ratio` and `Complex`. For each `Integral` type t, there is a type `Ratio` t of rational pairs with components of type t. (The type name `Rational` is a synonym for `Ratio Integer`.) Similarly, for each real floating-point type t, `Complex` t is a type of complex numbers with real and imaginary components of type t.

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

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

class  (Real a) => Integral a  where
    div, rem, mod       :: a -> a -> a
    divRem              :: a -> a -> (a,a)
    even, odd           :: a -> Bool
    toInteger           :: a -> Integer
    x `div` y           =  q  where (q,r) = divRem x y
    x `rem` y           =  r  where (q,r) = divRem x y
    x `mod` y           =  if signum x == - (signum y) then r + y else r
                           where r = x `rem` y
    even x              =  x `rem` 2 == 0
    odd                 =  not . even

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

Figure 8: Numeric classes and related operations

```
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

    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

instance  Integral Int
instance  Integral Integer

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

data  (Integral a)      => Ratio a
type  Rational          =  Ratio Integer
instance  (Integral a)  => RealFrac (Ratio a)

(%)                     ::  (Integral a) => a -> a -> Ratio a
numerator, denominator  ::  (Integral a) => Ratio a -> a

instance  RealFloat Float
instance  RealFloat Double

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

truncate, round         ::  (RealFrac a, Integral b) => a -> b
ceiling, floor          ::  (RealFrac a, Integral b) => a -> b
atan2                   ::  (RealFloat a) => a -> a -> a
```

Figure 9: Numeric classes and related operations (continued)

```
data  (RealFloat a)    => Complex a = a :+ a  deriving (Eq, Binary, Text)
instance (RealFloat a) => Floating (Complex a)

realPart, imagPart     ::  (RealFloat a) => Complex a -> a
conjugate              ::  (RealFloat a) => Complex a -> Complex a
mkPolar                ::  (RealFloat a) => a -> a -> Complex a
cis                    ::  (RealFloat a) => a -> Complex a
polar                  ::  (RealFloat a) => Complex a -> (a,a)
magnitude, phase       ::  (RealFloat a) => Complex a -> a
```

Figure 10: Numeric classes and related operations (continued)

The operator `(%)` forms the ratio of two integral numbers. The functions `numerator` and `denominator` extract the components of a ratio; these are in reduced form with a positive denominator.

Complex numbers are an algebraic type:

```
data  (RealFloat a) => Floating (Complex a)  =  a :+ a
```

The constructor `(:+)` forms a complex number from its real and imaginary rectangular components. A complex number may also be formed from polar components of magnitude and phase by the function `mkPolar`. The function `cis` produces a complex number from an angle t:

```
cis t = cos t :+ sin t
```

Put another way, `cis` t is a complex value with magnitude 1 and phase t (modulo 2π).

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. Several component-extraction functions are provided:

```
realPart (x:+y) =  x
imagPart (x:+y) =  y
magnitude z     =  r  where (r,t) = polar z
phase z         =  t  where (r,t) = polar z
```

Also defined on complex numbers is the conjugate function:

```
conjugate (x:+y) =  x:+(-y)
```

6.8.4 Arithmetic and Number-Theoretic Operations

The infix operations `(+)`, `(*)`, `(-)` and the unary function `negate` (which can also be written as a prefix minus sign; see section 3.2) apply to all numbers. The operations `div`, `rem`, and `mod` apply only to integral numbers, while the operation `(/)` applies only to fractional ones. The `div` and `rem` operations satisfy the law:

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

The result of ``x `div` y`` has the same sign as `x * y` and is truncated toward zero. The modulo function differs from the remainder function when the signs of the dividend and divisor differ, the remainder always having the sign of the dividend, and the modulo having the sign of the divisor. For example,

```
-13 `rem` 4 == -1
-13 `mod` 4 == 3

13 `rem` -4 == 1
13 `mod` -4 == -3
```

The `divRem` operation takes a dividend and a divisor as arguments and returns a (quotient, remainder) pair:

```
divRem x y  =  (x `div` y, x `rem` y)
```

Also available on integers are the even and odd predicates:

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

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

6.8.5 Exponentiation and Logarithms

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

6.8.6 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
```

For complex numbers, the definitions are different:

```
abs z             =  magnitude z :+ 0
signum 0          =  0
signum z@(x:+y)   =  x/r :+ y/r  where r = magnitude z
```

That is, `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. (`abs` for a complex number differs from `magnitude` only in type. See Section 6.8.3.)

6.8.7 Trigonometric Functions

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

The precise definition of the above functions is as in Common Lisp [18], which in turn follows Penfield's proposal for APL [13]. See these references for discussions of branch cuts, discontinuities, and implementation.

6.8.8 Coercions and Component Extraction

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

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

Two functions convert numbers to type `Rational`: `toRational` returns the rational equivalent of its real argument with full precision; `approxRational` takes two real fractional arguments and returns an approximation to the first within the tolerance given by the second. Subject to the tolerance constraint, the result has the smallest denominator possible.

The operations 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.

Also available are the following coercion functions:

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

6.9 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.

Types that are instances of class `Ix` (see Section 4.3.2) may be indices of arrays; a one-dimensional array might have index type `Int`, a two-dimensional array `(Int,Char)` etc.

6.9.1 Array Construction

If `a` is an index type and `b` is any type, the type of arrays with indices in `a` and elements in `b` is written `Array a b`. An array may be created by the function `array`:

```
array :: (Ix a) => (a,a)  -> [Assoc a b] -> Array a b
data  Assoc a b  =  a := b
```

The first argument of `array` is 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))`.

The second argument of `array` is 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`. The array is undefined 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. 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 nonstrict 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

```
-- Scaling an array of numbers by a given number:
scale :: (Num a, Ix b) => a -> Array b a -> Array b a
scale x a = array b [i := a!i * x | i <- range b]
            where b = bounds a

-- Inverting an array that holds a permutation of its indices
invPerm :: (Ix a) => Array a a -> Array a a
invPerm a = array b [a!i := i | i <- range b]
            where b = bounds a

-- The inner product of two vectors
inner :: (Ix a, Num b) => Array a b -> Array a b -> b
inner v w = if b == bounds w
                then sum [v!i * w!i | i <- range b]
                else error "inconformable arrays for inner product"
            where b = bounds v
```

Figure 11: Array examples

values associated with indices that do not appear will be undefined. Figure 11 shows some examples that use the `array` constructor.

`(!)` denotes array subscripting; the `bounds` function applied to an array returns its bounds:

```
(!)    :: (Ix a) => Array a b -> a -> b
bounds :: (Ix a) => Array a b -> (a,a)
```

The functions `indices`, `elems`, and `assocs`, when applied to an array, return lists of the indices, elements, or associations, respectively, in index order:

```
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 -> [Assoc a b]
assocs a = [ i := a!i | i <- indices a]
```

An array may be constructed from a pair of bounds and a list of values in index order using the function `listArray`:

```
listArray:: (Ix a) => (a,a) -> [b] -> Array a b
listArray bnds xs = array bnds (zipWith (:=) (range bnds) xs)
```

6.9.2 Accumulated Arrays

Another array creation function, `accumArray`, relaxes the restriction that a given index may appear at most once in the association list, using an *accumulating function* which combines the values of associations with the same index [12, 20]:

```
accumArray::(Ix a) => (b->c->b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

The first argument of `accumArray` is the accumulating function; the second is an initial value; the remaining two arguments are a bounds pair and an association list, as for the `array` function. 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, accumulated arrays should not in general be recursive.

6.9.3 Incremental Array Updates

```
(//)  :: (Ix a) => Array a b -> [Assoc a b] -> Array a b
accum :: (Ix a) => (b -> c -> b) -> Array a b -> [Assoc a c] -> Array a b
```

The operator `(//)` takes an array and a list of `Assoc` pairs and returns an array identical to the left argument except that it has been updated by the associations in the right argument. (As with the `array` function, the indices in the association list must be unique for the updated elements to be defined.) 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.

`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])
```

6.9.4 Derived Arrays

The two functions `amap` and `ixmap` derive new arrays from existing ones; they may be thought of as providing function composition on the left and right, respectively, with the mapping that the original array embodies:

```
amap :: (Ix a) => (b -> c) -> Array a b -> Array a c
amap f a = array b [i := f (a!i) | i <- range b]
           where b = bounds a

ixmap :: (Ix a, Ix a') => (a',a') -> (a'->a) -> Array a b -> Array a' b
ixmap bnds f a = array bnds [i := a ! f i | i <- range bnds]
```

`amap` is the array analogue of the `map` function on lists, while `ixmap` allows for transformations on array indices. Figure 12 shows some examples.

```
-- A rectangular subarray
subArray :: (Ix a) => (a,a) -> Array a b -> Array a b
subArray bnds = ixmap bnds (\i->i)

-- A row of a matrix
row :: (Ix a, Ix b) => a -> Array (a,b) c -> Array b c
row i x = ixmap (l',u') (\j->(i,j)) x where ((l,l'),(u,u')) = bounds x

-- Diagonal of a square matrix
diag :: (Ix a) => Array (a,a) b -> Array a b
diag x = ixmap (l,u) (\i->(i,i)) x
         where ((l,l'),(u,u')) | l == l' && u == u'  = bounds x

-- Projection of first components of an array of pairs
firstArray :: (Ix a) => Array a (b,c) -> Array a b
firstArray = amap (\(x,y)->x)
```

Figure 12: Derived array examples

6.10 Errors

All errors in Haskell are semantically equivalent to $\perp$. `error:: String -> a` takes a string argument and returns $\perp$. An application of `error` terminates evaluation of the program and displays the string as appropriate.

7 Input/Output

Haskell's I/O system is based on the view that a program communicates to the outside world via *streams of messages*: a program issues a stream of *requests* to the operating system and in return receives a stream of *responses*. Since a stream in Haskell is only a lazy list, a Haskell program has the type:

```
type  Dialogue = [Response] -> [Request]
```

The datatypes `Response` and `Request` are defined below. Intuitively, `[Response]` is an ordered list of *responses* and `[Request]` is an ordered list of *requests*; the nth response is the operating system's reply to the nth request.

With this view of I/O, there is no need for any special-purpose syntax or constructs for I/O; the I/O system is defined entirely in terms of how the operating system responds to a program with the above type—i.e. what response it issues for each request. An abstract specification of this behaviour is defined by giving a definition of the operating system as a function that takes as input an initial state and a collection of Haskell programs, each with the above type. This specification appears in Appendix C, using standard Haskell syntax augmented with a single non-deterministic merge operator.

One can define a continuation-based version of I/O in terms of a stream-based version. Such a definition is provided in Section 7.5. The specific I/O requests available in each style are identical; what differs is the way they are expressed. This means that programs in either style may be combined with a well-defined semantics. In both cases arbitrary I/O requests within conventional operating systems may be induced while retaining referential transparency within a Haskell program.

The required requests for a valid implementation are:

```
data  Request =
     -- file system requests:
              ReadFile      String
            | WriteFile     String String
            | AppendFile    String String
            | ReadBinFile   String
            | WriteBinFile  String Bin
            | AppendBinFile String Bin
            | DeleteFile    String
            | StatusFile    String
     -- channel system requests:
            | ReadChan      String
            | AppendChan    String String
            | ReadBinChan   String
            | AppendBinChan String Bin
            | StatusChan    String
```

```
    -- environment requests:
            | Echo          Bool
            | GetArgs
            | GetEnv        String
            | SetEnv        String String

stdin       = "stdin"
stdout      = "stdout"
stderr      = "stderr"
stdecho     = "stdecho"
```

Conceptually the above requests can be organised into three groups: those relating to the *file system* component of the operating system (the first eight), those relating to the *channel system* (the next five), and those relating to the *environment* (the last four).

The file system is fairly conventional: a mapping of file names to contents. The channel system consists of a collection of *channels*, examples of which include standard input (`stdin`), standard output (`stdout`), standard error (`stderr`), and standard echo (`stdecho`) channels. A channel is a one-way communication medium—it either consumes values from the program (via `AppendChan` or `AppendBinChan`) or produces values for the program (by responding to `ReadChan` or `ReadBinChan`). Channels communicate to and from *agents* (a concept made more precise in Appendix C). Examples of agents include line printers, disk controllers, networks, and human beings. As an example of the latter, the *user* is normally the consumer of standard output and the producer of standard input. Channels cannot be deleted, nor is there a notion of creating a channel.

Apart from these required requests, several optional requests are described in Appendix C.1. Although not required for a valid Haskell implementation, they may be useful in particular implementations.

Requests to the file system are in general order-dependent; if $i > j$ then the response to the ith request may depend on the jth request. In the case of the channel system the nature of the dependencies is dictated by the agents. In all cases external effects may also be felt "between" internal effects.

Responses are defined by:

```
data  Response = Success
               | Str String
               | StrList [String]
               | Bn  Bin
               | Failure IOError

data  IOError  = WriteError   String
               | ReadError    String
               | SearchError  String
               | FormatError  String
               | OtherError   String
```

The response to a request is either `Success`, when no value is returned; `Str` s [`Bn` b], when a string [binary] value s [b] is returned; or `Failure` e, indicating failure with I/O error e.

The nature of a failure is defined by the `IOError` datatype, which captures the most common kinds of errors. The `String` components of these errors are implementation dependent, and may be used to refine the description of the error (for example, for `ReadError`, the string might be `"file locked"`, `"access rights violation"`, etc.). An implementation is free to extend `IOError` as required.

7.1 I/O Modes

The I/O requests `ReadFile`, `WriteFile`, `AppendFile`, `ReadChan`, and `AppendChan` all work with *text* values—i.e. strings. Any value whose type is an instance of the class `Text` may be written to a file (or communicated on a channel) by using the appropriate output request if it is first converted to a string, using `shows` (see Section 4.3.3). Similarly, `reads` can be used with the appropriate input request to read such a value from a file (or a channel). This is text mode I/O.

For both efficiency and transparency, Haskell also supports a corresponding set of *binary* I/O requests—`ReadBinFile`, `WriteBinFile`, `AppendBinFile`, `ReadBinChan`, and `AppendBinChan`. `showBin` and `readBin` are using analogously to `shows` and `reads` (see Section 4.3.3) for values whose types are instances of the class `Binary` (see Section 6.7).

Binary mode I/O ensures transparency *within* an implementation—i.e. "what is read is what was written." Implementations on conventional machines will probably be able to realise binary mode more efficiently than text mode. On the other hand, the `Bin` datatype itself is implementation dependent, and thus binary mode *should not* be used as a method to ensure transparency *between* implementations.

In the remainder of this section, various aspects of text mode will be discussed, including the behaviour of standard channels such as `stdin` and `stdout`.

7.1.1 Transparent Character Set

The *transparent character set* is defined by:

the 52 uppercase and lowercase alphabetic characters
the 10 decimal digits
the 32 graphic characters:
`! " # $ % & ´ ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`
the space character

(This is identical to the *any* syntactic category defined in Section 2.2, with *tab* excluded.)

A *transparent line* is a list of no more than 254 transparent characters followed by a `\n` character (i.e. no more than 255 characters in total). A *transparent string* is the finite concatenation of zero or more transparent lines.

Haskell's *text mode for files is transparent whenever the string being used is transparent.* An implementation must ensure that a transparent string written to a file in text mode is identical to the string read back from the same file in text mode (assuming there were no intervening external effects).

The transparent character set is restricted because of the inconsistent treatment of text files by operating systems. For example, some systems translate the newline character `\n` into `CR/LF`, and others into just `CR` or just `LF`—so none of these characters can be in the transparent character set. Similarly, some systems truncate lines exceeding a certain length, others do not. Haskell's transparent string is intended to provide a useful degree of portability of text file manipulating programs. Of course, an implementation is free to guarantee a higher degree of transparency than that defined here (such as longer lines or more character types).

Besides this definition of text mode transparency, the standard input and output channels carry with them notions of standard *presentation* and *acceptance*, as defined below.

7.1.2 Presentation

Standard text mode presentation guarantees a minimum kind of presentable output on standard output devices; thus it is only defined for `AppendChan` using the channels `stdout`, `stderr`, and `stdecho`. Abstractly, these channels are assumed to be attached to a sequence of rectangular grids of characters called *pages*; each page consists of a number of lines and columns, with the first line presented at the "top" and the first column presented to the "left." The width of a column is assumed to be constant. (On a paper printing device, we expect an abstract page to correspond to a physical page; on a terminal display, it will correspond to whatever abstraction is presented by the terminal, but at a minimum the terminal should support display of at least one full page.)

Characters obtained from `AppendChan` requests are written sequentially into these pages starting at the top left hand corner of the first page. The characters are written in order horizontally across the page until a newline character (`\n`) is processed, at which point the subsequent characters are written starting in column one of line two, and so on. If a form feed character (`\f`) is processed, writing starts at the top left hand corner of the second page, and so on.

Maximum line length and page length for the output channels `stdout`, `stdecho`, and `stderr` may be obtained via the `StatusChan` request as described in Section 7.3. These are implementation-dependent constants, but must be at least 40 characters and 20 lines, respectively. `AppendChan` may induce a `FormatError` if either of these limits is exceeded.

Presentation of the transparent character set may be in any readable font. Presentation of `\n` and `\f` is as defined above. Presentation of any other character is not defined—presentation of such a character may invalidate standard presentation of all subsequent characters. An implementation, of course, may guarantee other forms of useful presentation beyond what is specified here.

To facilitate processing of text to and from standard input/output channels, the auxiliary functions shown in Figure 13 are provided in the standard prelude.

7.1.3 Acceptance

Standard text mode acceptance guarantees a minimum kind of character input from standard

```
span, break              :: (a -> Bool) -> [a] -> ([a],[a])
span p xs                =  (takeWhile p xs, dropWhile p xs)
break p                  =  span (not . p)

lines                    :: String -> [String]
lines ""                 =  []
lines s                  =  l : (if null s' then [] else lines (tail s'))
                            where (l, s') = break ((==) '\n') s

words                    :: String -> [String]
words s                  =  case dropWhile isSpace s of
                                 "" -> []
                                 s' -> w : words s''
                                       where (w, s'') = break isSpace s'

unlines                  :: [String] -> String
unlines ls               = concat (map (\l -> l ++ "\n") ls)

unwords                  :: [String] -> String
unwords []               = ""
unwords [w]              = w
unwords (w:ws)           = w ++ concat (map (' ' :) ws)
```

Figure 13: Auxiliary Functions for Text Processing of Standard Output

input devices; thus it is only defined for `ReadChan` using the channel `stdin`. Abstractly, `stdin` is assumed to be attached to a *keyboard*. The only requirement of the keyboard is that it have keys to support the transparent character set plus the newline (`\n`) character.

7.1.4 Echoing

The channel `stdecho` is assumed connected to the display associated with the device to which `stdin` is connected. It may be possible for `stdout` and `stdecho` to be connected to the same device, but this is not required. It may be possible in some operating systems to redirect `stdout` to a file while still displaying information to the user on `stdecho`.

The `Echo` request (described in Section 7.4) controls echoing of `stdin` on `stdecho`. When echoing is enabled, characters typed at the terminal connected to `stdin` are echoed onto `stdecho`, with optional implementation-specific line-editing functions available. The list of characters returned by a read request to `stdin` should be the result of this processing. As an entire line may be erased by the user, a program will not see any of the line until a `\n` character is typed.

A display may receive data from four different sources: echoing from `stdin`, and explicit output to `stdecho`, `stdout`, and `stderr`. The result is an interleaving of these character streams, but it is not an arbitrary one, because of two constraints: (1) *explicit* output (via `AppendChan`) must appear as the concatenation of the individual streams; i.e. they cannot be interleaved (this is consistent with the hyperstrict nature of `AppendChan`), and (2) if echoing is on, characters from `stdin` that a program depends on for some I/O request must appear on the display before that I/O occurs. These constraints permit a user to type ahead, but prevent a system from printing a reply before echoing the user's request.

7.2 File System Requests

In this section, each request is described using the stream model—the corresponding behaviour using the continuation model should be obvious. Optional requests, not required of a valid Haskell implementation, are described in Appendix C.1.

- `ReadFile    name`
`ReadBinFile name`

  Returns the contents of file `name` treated as a text [binary] file. If successful, the response will be of the form `Str s` [`Bn b`], where `s` [`b`] is a string [binary] value. If the file is not found, the response `Failure (SearchError string)` is induced; if it is unreadable for some other reason, the `Failure (ReadError string)` error is induced.

- `WriteFile    name string`
`WriteBinFile name bin`

  Writes `string` [`bin`] to file `name`. If the file does not exist, it is created. If it already exists, it is overwritten. A successful response has form `Success`; the only failure possible has the form `Failure (WriteError string)`.

Both of these requests are "hyperstrict" in their second argument: no response is returned until the entire list of values is completely evaluated.

- `AppendFile    name string`
  `AppendBinFile name bin`

  Identical to `WriteFile` [`WriteBinFile`], except that (1) the `string` [`bin`] argument is appended to the current contents of the file named `name`; (2) if the I/O mode does not match the previous mode with which `name` was written, the behaviour is not specified; and (3) if the file does not exist, the response `Failure (SearchError string)` is induced. All other errors have form `Failure (WriteError string)`, and both requests are hyperstrict in their second argument.

- `DeleteFile name`

  Deletes file `name`, with successful response `Success`. If the file does not exist, the response `Failure (SearchError string)` is induced. If it cannot be deleted for some other reason, a response of the form `Failure (WriteError string)` is induced.

- `StatusFile name`

  Induces `Failure (SearchError string)` if an object `name` does not exist, otherwise induces `Str status` where `status` is a string containing, in this order: (1) either `'t'`, `'b'`, `'d'`, or `'u'` depending on whether the object is a text file, binary file, directory, or something else, respectively (if text and binary files cannot be distinguished, `'f'` indicates either text or binary file); (2) `'r'` if the object is readable by this program, `'-'` if not; and (3) `'w'` if the object is writable by this program, `'-'` if not. For example `"dr-"` denotes a directory that can be read but not written. An implementation is free to append more status information to this string.

Note 1. A proper implementation of `ReadFile` or `ReadBinFile` may have to make copies of files in order to preserve referential transparency—a successful read of a file returns a *lazy list* whose contents should be preserved, despite future writes to or deletions of that file, even if the lazy list has not yet been completely evaluated.

Note 2. Given the two juxtaposed requests:

```
[ ..., WriteFile name contents1, ReadFile name, ... ]
```

with the corresponding responses:

```
[ ..., Success, Str contents2, ... ]
```

then `contents1 == contents2` if `contents1` is a transparent string, assuming that there were no external effects. A similar result would hold if the binary versions were used.

7.3 Channel System Requests

Channels are inherently different from files—they contain ephemeral streams of data as opposed to persistent stationary values. The most common channels are standard input (`stdin`), standard output (`stdout`), standard error (`stderr`), and standard echo (`stdecho`); these four are the only required channels in a valid implementation.

- `ReadChan    name`
  `ReadBinChan name`

  Opens channel `name` for input. A successful response returns the contents of the channel as a lazy stream of characters [a binary value]. If the channel does not exist the response `Failure (SearchError string)` is induced; all other errors have form `Failure (ReadError string)`.

  Unlike files, once a `ReadChan` or `ReadBinChan` request has been issued for a particular channel, it cannot be issued again for the same channel in that program. This reflects the ephemeral nature of its contents and prevents a serious space leak.

- `AppendChan    name string`
  `AppendBinChan name bin`

  Writes `string` [`bin`] to channel `name`. The semantics is as for `AppendFile`, except: (1) the second argument is appended to whatever was previously written (if anything); (2) if `AppendChan` and `AppendBinChan` are both issued to the same channel, the resulting behaviour is not specified; (3) if the channel does not exist, the response `Failure (SearchError string)` is induced; and (4) if the maximum line or page length of `stdout`, `stderr`, or `stdecho` is exceeded, the response `Failure (FormatError string)` is induced (see Section 7.1.2). All other errors have form `Failure (WriteError string)`. Both requests are hyperstrict in their second argument.

- `StatusChan name`

  Induces `Failure (SearchError string)` if channel `name` does not exist, otherwise induces `Str status` where `status` is a string containing implementation-dependent information about the named channel. The only information required of a valid implementation is that for the output channels `stdout`, `stdecho`, and `stderr`: the beginning of the status string must contain two integers separated by a space, the first integer indicating the maximum line length (in characters) allowed on the channel, the second indicating the maximum page length (in lines) allowed (see Section 7.1.2). A zero length implies that there is no bound.

7.4 Environment Requests

- `Echo bool`

  `Echo True` enables echoing of `stdin` on `stdecho`; `Echo False` disables it (see Section 7.1.4). Either `Success` or `Failure (OtherError string)` is induced.

  The echo mode can only be set once by a particular program, and it must be done before any I/O operation involving `stdin`. If no `Echo` request is made, a valid implementation is expected to use the echoing mode of the OS at the time the program is run.

- `GetArgs`

  Induces the response `StrList str_list`, where `str_list` is a list of the program's command line arguments.

- `GetEnv name`

  Returns the value of environment variable `name`. If successful, the response will be of the form `Str s`, where `s` is a string. If the environment variable does not exist, a `SearchError` is induced.

- `SetEnv name string`

  Sets environment variable `name` to value `string`, with response `Success`. If the environment variable does not exist, it is created.

7.5 Continuation-based I/O

Haskell supports an alternative style of I/O called *continuation-based I/O*. Under this model, a Haskell program still has type `[Response]->[Request]`, but instead of the user manipulating the requests and responses directly, a collection of *transactions* defined in a continuation style, captures the effect of each request/response pair.

Transactions are functions. For each request `Req` there corresponds a transaction `req`, as shown in Figure 14. For example, `ReadFile` induces either a failure response `Failure msg` or success response `Str contents`. In contrast the transaction `readFile` would be used in continuation-based I/O, as for example,

```
readFile name (\ msg -> errorTransaction)
              (\ contents -> successTransaction)
```

where the second and third arguments are the *failure continuation* and *success continuation*, respectively. If the transaction fails then the error continuation is applied to the error message; if it succeeds then the success continuation is applied to the contents of the file. The following type synonyms and auxiliary functions are defined for continuation-based I/O:

```
type  Dialogue     =  [Response] -> [Request]
type  SuccCont     =                Dialogue
type  StrCont      =  String     -> Dialogue
type  StrListCont  =  [String]   -> Dialogue
type  BinCont      =  Bin        -> Dialogue
type  FailCont     =  IOError    -> Dialogue
```

```
strDispatch fail succ (resp:resps) =
            case resp of Str val     -> succ val resps
                         Failure msg -> fail msg resps
strListDispatch fail succ (resp:resps) =
            case resp of StrList val -> succ val resps
                         Failure msg -> fail msg resps
binDispatch fail succ (resp:resps) =
            case resp of Bn val      -> succ val resps
                         Failure msg -> fail msg resps
succDispatch fail succ (resp:resps) =
            case resp of Success     -> succ resps
                         Failure msg -> fail msg resps

abort     ::  FailCont
abort err  =  done

exit     ::  FailCont
exit err  =  appendChan stdout msg abort done
             where msg = case err of ReadError   s -> s
                                     WriteError  s -> s
                                     SearchError s -> s
                                     FormatError s -> s
                                     OtherError  s -> s

print           :: (Text a) => a -> Dialogue
print x         =  appendChan stdout (show x) abort done
prints          :: (Text a) => a -> String -> Dialogue
prints x s      =  appendChan stdout (shows x s) abort done

interact  ::  (String -> String) -> Dialogue
interact f = readChan stdin abort
                     (\x -> appendChan stdout (f x) abort done)
```

7.6 A Small Example

Both of the following programs prompt the user for the name of a file, and then look up and display the contents of the file on standard-output. The filename as typed by the user is also echoed. The first program uses the stream-based style (note the irrefutable patterns):

```
main ~(Success : ~((Str userInput) : ~(Success : ~(r4 : _)))) =
  [ AppendChan stdout "please type a filename\n",
    ReadChan stdin,
    AppendChan stdout name,
    ReadFile name,
    AppendChan stdout (case r4 of Str contents    -> contents
                                  Failure ioerror -> "can't open file")
  ] where (name : _) = lines userInput
```

```
done         ::                                                    Dialogue
readFile     :: String ->           FailCont -> StrCont     -> Dialogue
writeFile    :: String -> String -> FailCont -> SuccCont    -> Dialogue
appendFile   :: String -> String -> FailCont -> SuccCont    -> Dialogue
readBinFile  :: String ->           FailCont -> BinCont     -> Dialogue
writeBinFile :: String -> Bin    -> FailCont -> SuccCont    -> Dialogue
appendBinFile :: String -> Bin   -> FailCont -> SuccCont    -> Dialogue
deleteFile   :: String ->           FailCont -> SuccCont    -> Dialogue
statusFile   :: String ->           FailCont -> StrCont     -> Dialogue
readChan     :: String ->           FailCont -> StrCont     -> Dialogue
appendChan   :: String -> String -> FailCont -> SuccCont    -> Dialogue
readBinChan  :: String ->           FailCont -> BinCont     -> Dialogue
appendBinChan :: String -> Bin   -> FailCont -> SuccCont    -> Dialogue
statusChan   :: String ->           FailCont -> StrCont     -> Dialogue
echo         :: Bool   ->           FailCont -> SuccCont    -> Dialogue
getArgs      ::                     FailCont -> StrListCont -> Dialogue
getEnv       :: String ->           FailCont -> StrCont     -> Dialogue
setEnv       :: String -> String -> FailCont -> SuccCont    -> Dialogue

done resps = []
readFile name fail succ resps =           --similarly for readBinFile
   (ReadFile name) : strDispatch fail succ resps
writeFile name contents fail succ resps =  --similarly for writeBinFile
   (WriteFile name contents) : succDispatch fail succ resps
appendFile name contents fail succ resps = --similarly for appendBinFile
   (AppendFile name contents) : succDispatch fail succ resps
deleteFile name fail succ resps =
   (DeleteFile name) : succDispatch fail succ resps
statusFile name fail succ resps =         --similarly for statusChan
   (StatusFile name) : strDispatch fail succ resps
readChan name fail succ resps =           --similarly for readBinChan
   (ReadChan name) : strDispatch fail succ resps
appendChan name contents fail succ resps = --similarly for appendBinChan
   (AppendChan name contents) : succDispatch fail succ resps
echo bool fail succ resps =
   (Echo bool) : succDispatch fail succ resps
getArgs fail succ resps =
   GetArgs : strListDispatch fail succ resps
getEnv name fail succ resps =
   (GetEnv name) : strDispatch fail succ resps
setEnv name contents fail succ resps =
   (SetEnv name contents) : succDispatch fail succ resps
```

Figure 14: Transactions of continuation-based I/O.

The second program uses the continuation-based style:

```
main = appendChan stdout "please type a filename\n" abort (
       readChan stdin abort (\ userInput ->
       let (name : _) = lines userInput in
       appendChan stdout name abort (
       readFile name (\ ioerror -> appendChan stdout
                                   "can't open file" abort done)
                     (\ contents ->
       appendChan stdout contents abort done)))))
```

More examples and a general discussion of both forms of I/O may be found in a report by Hudak and Sundaresh [8].

7.7 An Example Involving Synchronisation

The following program reads two numbers and prints their sum. After the initial `readChan` request, the value of the input stream must be passed in and out of the functions which actually obtain the user input. The programmer must control the synchronisation between the `appendChan` requests and when the program stops to read input. The `readChan` request does not actually cause the program to stop and wait for the user to enter the entire input stream; only at demands for actual input characters will execution pause for input. This program assures that this demand is properly synchronised with the `appendChan` requests by verifying input values in the `readInt` function.

```
main :: Dialogue

main = readChan stdin abort (\ userInput -> readNums (lines userInput))

readNums :: [String] -> Dialogue

readNums inputLines =
   readInt "Enter first number: " inputLines
     (\ num1 inputLines1 ->
          readInt "Enter second number: " inputLines1
             (\ num2 _ -> reportResult num1 num2))
reportResult :: Int -> Int -> Dialogue

reportResult num1 num2 =
  appendChan stdout ("Their sum is: " ++ show (num1 + num2)) abort done

-- readInt prints a prompt and then reads a line of input.  If the
-- line contains an integer, the value of the integer is passed to the
-- success continuation.  If a line cannot be parsed as an integer,
-- an error message is printed and the user is asked to try again.
-- If EOF is detected, the program is aborted.

readInt :: String -> [String] -> (Int -> [String] -> Dialogue) -> Dialogue

readInt prompt inputLines succ =
  appendChan stdout prompt abort
    (case inputLines of
       (l1 : rest) -> case (reads l1) of
                        [(x,"")] -> succ x rest
                        _        -> appendChan stdout
                                      "Error - retype the number\n" abort
                                       (readInt prompt rest succ)
       _           -> appendChan stdout "Early EOF" abort done)
```

A Standard Prelude

In this appendix the entire Haskell prelude is given. It is organised into a root module and eight sub-modules.

```
-- Standard value bindings

module Prelude (
    PreludeCore.., PreludeRatio.., PreludeComplex.., PreludeList..,
    PreludeArray.., PreludeText.., PreludeIO..,
    nullBin, isNullBin, appendBin,
    (&&), (||), not, otherwise
    ord, chr,
    isAscii, isControl, isPrint, isSpace,
    isUpper, isLower, isAlpha, isDigit, isAlphanum,
    toUpper, toLower,
    minInt, maxInt, subtract, gcd, lcm, (^), (^^),
    truncate, round, ceiling, floor, fromIntegral, fromRealFrac, atan2,
    fst, snd, (.), flip, until, error, asTypeOf ) where

import PreludeBuiltin
import PreludeCore
import PreludeList
import PreludeArray
import PreludeRatio
import PreludeComplex
import PreludeText
import PreludeIO

infixr 9  .
infixr 8  ^, ^^
infixr 3  &&
infixr 2  ||

-- Binary functions

nullBin                 :: Bin
nullBin                 =  primNullBin

isNullBin               :: Bin -> Bool
isNullBin               =  primIsNullBin

appendBin               :: Bin -> Bin -> Bin
appendBin               =  primAppendBin
```

```
-- 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 functions

ord                     :: Char -> Int
ord                     =  primCharToInt

chr                     :: Int -> Char
chr                     =  primIntToChar

isAscii, isControl, isPrint, isSpace            :: Char -> Bool
isUpper, isLower, isAlpha, isDigit, isAlphanum  :: Char -> Bool

isAscii c               =  ord c < 128
isControl c             =  c < ' ' || c == '\DEL'
isPrint c               =  c >= ' ' && c <= '~'
isSpace c               =  c == ' ' || c == '\t' || c == '\n' ||
                           c == '\r' || c == '\f' || c == '\v'
isUpper c               =  c >= 'A' && c <= 'Z'
isLower c               =  c >= 'a' && c <= 'z'
isAlpha c               =  isUpper c || isLower c
isDigit c               =  c >= '0' && c <= '9'
isAlphanum c            =  isAlpha c || isDigit c

toUpper, toLower        :: Char -> Char
toUpper c | isLower c   =  chr (ord c - (ord 'a' + ord 'A'))
          | otherwise   =  c

toLower c | isUpper c   =  chr (ord c - (ord 'A' + ord 'a'))
          | otherwise   =  c

-- Numeric functions

minInt, maxInt  :: Int
minInt          =  primMinInt
maxInt          =  primMaxInt
```

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

gcd             :: (Integral a) => a -> a-> a
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 `div` (gcd x y)) * y)

(^)             :: (Num a, Integral b) => a -> b -> a
x ^ 0           =  1
x ^ (n+1)       =  f x n x
                   where f _ 0 y = y
                         f x n y = g x n  where
                                   g x n | even n  = g (x*x) (n`div`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 1/x^(-n)

truncate        :: (RealFrac a, Integral b) => a -> b
truncate x      =  m  where (m,_) = properFraction x

round           :: (RealFrac a, Integral b) => a -> b
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         :: (RealFrac a, Integral b) => a -> b
ceiling x       =  if r > 0 then n + 1 else n
                   where (n,r) = properFraction x

floor           :: (RealFrac a, Integral b) => a -> b
floor x         =  if r < 0 then n - 1 else n
                   where (n,r) = properFraction x

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

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

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

-- Some standard functions:
-- component projections for pairs:
fst                     :: (a,b) -> a
fst (x,y)               =  x

snd                     :: (a,b) -> b
snd (x,y)               =  y

-- 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

-- 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)

-- error is applied to a string, returns any type, and is everywhere undefined.
-- Operationally, the intent is that its application terminate execution of
-- the program and display the argument string in some appropriate way.
-- The following defines the semantics of error, but may or may not have
-- the intended operational effect, depending on the implementation.
error                   :: String -> a
error msg | False       =  error msg

-- asTypeOf returns its first argument, ignoring the value of the second.
-- Its typing, however, forces the first argument (which is usually
-- overloaded) to have the same type as the second.
asTypeOf                :: a -> a -> a
x `asTypeOf` _          =  x
```

A.1 Prelude PreludeBuiltin

```
interface PreludeBuiltin  where

infixr 5 :

-- The following are algebraic types with special syntax.  All of their
-- standard instances are derived here, except for class Text, for
-- which the special syntax must be taken into account.  See PreludeText
-- for the Text instances of lists and the trivial type and a scheme
-- for Tuple Text instances.
--
-- data [a] = [] | a : [a]  deriving (Eq, Ord, Binary)     Lists
-- data () = ()  deriving (Eq, Ord, Ix, Enum, Binary)      Trivial Type
-- data (a,b) = (a,b)  deriving (Eq, Ord, Ix, Binary)      Pairs
-- data (a,b,c) = (a,b,c)  deriving (Eq, Ord, Ix, Binary)  Triples
-- et cetera                                               Other Tuples

-- The primitive types:

data Char
data Int
data Integer
data Float
data Double
data Bin

instance Binary Char
instance Binary Int
instance Binary Integer
instance Binary Float
instance Binary Double

primMinInt, primMaxInt          :: Int
primCharToInt                   :: Char -> Int
primIntToChar                   :: Int -> Char
primIntToInteger                :: Int -> Integer
primIntegerToInt                :: Integer -> Int

primEqInt, primLeInt            :: Int -> Int -> Bool
primPlusInt, primMulInt         :: Int -> Int -> Int
primNegInt                      :: Int -> Int
primDivRemInt                   :: Int -> Int -> (Int,Int)

primEqInteger, primLeInteger    :: Integer -> Integer -> Bool
primPlusInteger, primMulInteger :: Integer -> Integer -> Integer
primNegInteger                  :: Integer -> Integer
primDivRemInteger               :: Integer -> Integer -> (Integer,Integer)
```

```
primFloatRadix                  :: Integer
primFloatDigits, primFloatMinExp,
    primFloatMaxExp             :: Int
primDecodeFloat                 :: Float -> (Integer,Int)
primEncodeFloat                 :: Integer -> Int -> Float
primEqFloat, primLeFloat        :: Float -> Float -> Bool
primPlusFloat, primMulFloat,
    primDivFloat                :: Float -> Float -> Float
primNegFloat                    :: Float -> Float

primPiFloat                     :: Float
primExpFloat, primLogFloat,
    primSqrtFloat, primSinFloat,
    primCosFloat, primTanFloat,
    primAsinFloat, primAcosFloat,
    primAtanFloat, primSinhFloat,
    primCoshFloat, primTanhFloat,
    primAsinhFloat, primAcoshFloat,
    primAtanhFloat              :: Float -> Float

primDoubleRadix                 :: Integer
primDoubleDigits, primDoubleMinExp,
    primDoubleMaxExp            :: Int
primDecodeDouble                :: Double -> (Integer,Int)
primEncodeDouble                :: Integer -> Int -> Double
primEqDouble, primLeDouble      :: Double -> Double -> Bool
primPlusDouble, primMulDouble,
    primDivDouble               :: Double -> Double -> Double
primNegDouble                   :: Double -> Double
primPiDouble                    :: Double
primExpDouble, primLogDouble,
    primSqrtDouble, primSinDouble,
    primCosDouble, primTanDouble,
    primAsinDouble, primAcosDouble,
    primAtanDouble, primSinhDouble,
    primCoshDouble, primTanhDouble,
    primAsinhDouble, primAcoshDouble,
    primAtanhDouble             :: Double -> Double

primNullBin                     :: Bin
primIsNullBin                   :: Bin -> Bool
primAppendBin                   :: Bin -> Bin -> Bin
```

A.2 Prelude PreludeCore

```
-- Standard types, classes, and instances

module PreludeCore (
    Eq((==), (/=)),
    Ord((<), (<=), (>=), (>), max, min),
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Integral(divRem, div, rem, mod, even, odd, toInteger),
    Fractional((/), fromRational),
    Floating(pi, exp, log, sqrt, (**), logBase,
             sin, cos, tan, asin, acos, atan,
             sinh, cosh, tanh, asinh, acosh, atanh),
    Real(toRational),
    RealFrac(properFraction, approxRational),
    RealFloat(floatRadix, floatDigits, floatRange,
              encodeFloat, decodeFloat, exponent, significand, scaleFloat),
    Ix(range, index, inRange),
    Enum(enumFrom, enumFromThen, enumFromTo, enumFromThenTo),
    Text(readsPrec, showsPrec, readList, showList),
    Binary(readBin, showBin),
--  List type: [_]((:), [])
--  Tuple types: (_,_), (_,_,_), etc.
--  Trivial type: ()
    Bool(True, False),
    Char, Int, Integer, Float, Double, Bin,
    Ratio, Complex((:+)), Assoc((:=)), Array,
    String, Rational )  where

import PreludeBuiltin
import Prelude(iterate)
import PreludeText(Text(readsPrec, showsPrec, readList, showList))
import PreludeRatio(Ratio, Rational)
import PreludeComplex(Complex((:+)))
import PreludeArray(Assoc((:=)), Array)
import PreludeIO(Name, Request, Response, IOError,
                 Dialogue, SuccCont, StrCont, BinCont, FailCont)

infixr 8  **
infixl 7  *
infix  7  /, `div`, `rem`, `mod`
infixl 6  +, -
infix  4  ==, /=, <, <=, >=, >
```

```
-- Equality and Ordered classes

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

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

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

    x <  y              =  x <= y && x /= y
    x >= y              =  y <= x
    x >  y              =  y <  x

    -- The following default methods are appropriate for partial orders.
    -- Note that the second guards in each function can be replaced
    -- by "otherwise" for total orders.
    max x y | x >= y    =  x
            | y >= x    =  y
            |otherwise  =  error "max{PreludeCore}: no ordering relation"
    min x y | x <= y    =  x
            | y <= x    =  y
            |otherwise  =  error "min{PreludeCore}: no ordering relation"

-- Numeric classes

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

    x - y               =  x + negate y

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

class  (Real a) => Integral a  where
    div, rem, mod       :: a -> a -> a
    divRem              :: a -> a -> (a,a)
    even, odd           :: a -> Bool
    toInteger           :: a -> Integer

    x `div` y           =  q  where (q,r) = divRem x y
    x `rem` y           =  r  where (q,r) = divRem x y
    x `mod` y           =  if signum r == - signum y then r + y else r
                           where r = x `rem` y
    even x              =  x `rem` 2 == 0
    odd                 =  not . even
```

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

    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)
    approxRational      :: a -> a -> Rational

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

    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

-- Index and Enumeration classes

class  (Ord a) => Ix a  where
    range               :: (a,a) -> [a]
    index               :: (a,a) -> a -> Int
    inRange             :: (a,a) -> a -> Bool
```

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

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

-- Binary class

class  Binary a  where
    readBin             :: Bin -> (a,Bin)
    showBin             :: a -> Bin -> Bin

-- Boolean type

data  Bool  =  False | True     deriving (Eq, Ord, Ix, Enum, Text, Binary)

-- Character type

instance  Eq Char  where
    c == c'             =  ord c == ord c'

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

instance  Ix Char  where
    range (c,c')        =  [c..c']
    index (c,c') ci     =  ord ci - ord c
    inRange (c,c') ci   =  ord c <= i && i <= ord c'
                           where i = ord ci

instance  Enum Char  where
    enumFrom c          =  map chr [ord c ..]
    enumFromThen c c'   =  map chr [ord c, ord c' ..]

type  String = [Char]

-- Standard Integral types

instance  Eq Int  where
    (==)                =  primEqInt

instance  Eq Integer  where
    (==)                =  primEqInteger

instance  Ord Int  where
    (<=)                =  primLeInt
```

```
instance  Ord Integer  where
    (<=)                = primLeInteger

instance  Num Int  where
    (+)                 = primPlusInt
    negate              = primNegInt
    (*)                 = primMulInt
    abs                 = absReal
    signum              = signumReal
    fromInteger         = primIntegerToInt

instance  Num Integer  where
    (+)                 = primPlusInteger
    negate              = primNegInteger
    (*)                 = primMulInteger
    abs                 = absReal
    signum              = signumReal
    fromInteger x       = x

absReal x    | x >= 0    =  x
             | otherwise =  - x

signumReal x | x == 0    =  0
             | x > 0     =  1
             | otherwise = -1

instance  Real Int  where
    toRational x        =  toInteger x % 1

instance  Real Integer  where
    toRational x        =  x % 1

instance  Integral Int  where
    divRem              =  primDivRemInt
    toInteger           =  primIntToInteger

instance  Integral Integer  where
    divRem              =  primDivRemInteger
    toInteger x         =  x

instance  Ix Int  where
    range (m,n)         =  [m..n]
    index (m,n) i       =  i - m
    inRange (m,n) i     =  m <= i && i <= n
```

```
instance  Ix Integer  where
    range (m,n)         =  [m..n]
    index (m,n) i       =  fromInteger (i - m)
    inRange (m,n) i     =  m <= i && i <= n

instance  Enum Int  where
    enumFrom            =  numericEnumFrom
    enumFromThen        =  numericEnumFromThen

instance  Enum Integer  where
    enumFrom            =  numericEnumFrom
    enumFromThen        =  numericEnumFromThen

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

-- Standard Floating types

instance  Eq Float  where
    (==)                =  primEqFloat

instance  Eq Double  where
    (==)                =  primEqDouble

instance  Ord Float  where
    (<=)                =  primLeFloat

instance  Ord Double  where
    (<=)                =  primLeDouble

instance  Num Float  where
    (+)                 =  primPlusFloat
    negate              =  primNegFloat
    (*)                 =  primMulFloat
    abs                 =  absReal
    signum              =  signumReal
    fromInteger n       =  encodeFloat n 0

instance  Num Double  where
    (+)                 =  primPlusDouble
    negate              =  primNegDouble
    (*)                 =  primMulDouble
    abs                 =  absReal
    signum              =  signumReal
    fromInteger n       =  encodeFloat n 0
```

```
instance  Real Float  where
    toRational          =  floatingToRational

instance  Real Double  where
    toRational          =  floatingToRational

floatingToRational x    =  (m%1)*(b%1)^^n
                           where (m,n) = decodeFloat x
                                 b     = floatRadix  x

instance  Fractional Float  where
    (/)                 =  primDivFloat
    fromRational        =  rationalToFloating

instance  Fractional Double  where
    (/)                 =  primDivDouble
    fromRational        =  rationalToFloating

rationalToFloating x    =  fromInteger (numerator x)
                                / fromInteger (denominator x)

instance  Floating Float  where
    pi                  =  primPiFloat
    exp                 =  primExpFloat
    log                 =  primLogFloat
    sqrt                =  primSqrtFloat
    sin                 =  primSinFloat
    cos                 =  primCosFloat
    tan                 =  primTanFloat
    asin                =  primAsinFloat
    acos                =  primAcosFloat
    atan                =  primAtanFloat
    sinh                =  primSinhFloat
    cosh                =  primCoshFloat
    tanh                =  primTanhFloat
    asinh               =  primAsinhFloat
    acosh               =  primAcoshFloat
    atanh               =  primAtanhFloat
```

```
instance  Floating Double  where
    pi                  =  primPiDouble
    exp                 =  primExpDouble
    log                 =  primLogDouble
    sqrt                =  primSqrtDouble
    sin                 =  primSinDouble
    cos                 =  primCosDouble
    tan                 =  primTanDouble
    asin                =  primAsinDouble
    acos                =  primAcosDouble
    atan                =  primAtanDouble
    sinh                =  primSinhDouble
    cosh                =  primCoshDouble
    tanh                =  primTanhDouble
    asinh               =  primAsinhDouble
    acosh               =  primAcoshDouble
    atanh               =  primAtanhDouble

instance  RealFrac Float  where
    properFraction      =  floatProperFraction
    approxRational      =  floatApproxRational

instance  RealFrac Double  where
    properFraction      =  floatProperFraction
    approxRational      =  floatApproxRational

floatProperFraction x   =  let (m,n) = decodeFloat x
                               b     = floatRadix x
                           in if n >= 0
                                 then (fromInteger m * fromInteger b ^ n, 0)
                                 else let d      = b ^ (-n)
                                          (m',k) = divRem m d
                                      in (fromInteger m',
                                          fromInteger k / fromInteger d)

floatApproxRational x eps =
    let (m,n) = decodeFloat x
        b     = floatRadix x
        (p,q) = if n < 0 then (m, b^(-n)) else (m*b^n, 1)
    in case dropWhile (\r -> abs (fromRational r - x) > eps)
                      (approximants p q)
       of (_:r:_) | denominator r == 1 -> r
          (r:_)                        -> r
```

```
instance  RealFloat Float  where
    floatRadix _        =  primFloatRadix
    floatDigits _       =  primFloatDigits
    floatRange _        =  (primFloatMinExp,primFloatMaxExp)
    decodeFloat         =  primDecodeFloat
    encodeFloat         =  primEncodeFloat

instance  RealFloat Double  where
    floatRadix _        =  primDoubleRadix
    floatDigits _       =  primDoubleDigits
    floatRange _        =  (primDoubleMinExp,primDoubleMaxExp)
    decodeFloat         =  primDecodeDouble
    encodeFloat         =  primEncodeDouble

instance  Enum Float  where
    enumFrom            =  numericEnumFrom
    enumFromThen        =  numericEnumFromThen

instance  Enum Double  where
    enumFrom            =  numericEnumFrom
    enumFromThen        =  numericEnumFromThen
```

A.3 Prelude PreludeRatio

```
-- Standard functions on rational numbers

module  PreludeRatio (
    Ratio, Rational, (%), numerator, denominator,
    approximants, partialQuotients ) where

infixl 7  %, :%

prec = 7

data  (Integral a)      => Ratio a = a :% a  deriving (Eq, Binary)
type  Rational          =  Ratio Integer

(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approximants            :: (Integral a) => a -> a -> [Ratio a]
partialQuotients        :: (Integral a) => a -> a -> [a]

reduce _ 0              =  error "(%){PreludeRatio}: zero denominator"
reduce x y              =  (x `div` d) :% (y `div` d)
                           where d = gcd x y

x % y                   =  reduce (x * signum y) (abs y)

numerator (x:%y)        =  x

denominator (x:%y)      =  y

approximants p q        =  zipWith (:%) ps qs
                           where
                           ps       = gen unit (unit*a)
                           qs       = gen 0 1
                           unit     = signum p * signum q
                           (a:as)   = partialQuotients (abs p) (abs q)
                           gen x x' = xs
                                      where
                                      xs = x' : zipWith3 next as (x:xs) xs
                                      next a x x' = x'*a + x

partialQuotients p q    =  a : (if r==0 then [] else partialQuotients q r)
                           where (a,r) = divRem p q

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')
    fromRational (x:%y) =  fromInteger x :% fromInteger y

instance  (Integral a)  => RealFrac (Ratio a)  where
    properFraction (x:%y) = (fromIntegral q, r:%y)
                            where (q,r) = divRem x y

    approxRational x@(p:%q) eps = toRational approx
        where approx    = case dropWhile (\r -> abs (r-x) > eps)
                                         (approximants p q)     of
                                _:r@(_:%1):_ -> r
                                r:_          -> r

instance  (Integral a)  => Enum (Ratio a)  where
    enumFrom            =  iterate ((+)1)
    enumFromThen n m    =  iterate ((+)(m-n)) n

instance  (Integral a, Text a) => Text (Ratio a)  where
    readsPrec p  =  readParen (p > prec)
                              (\r -> [(x%y,u) | (x,s)   <- reads r,
                                                ("%",t) <- [lex s],
                                                (y,u)   <- reads t ])

    showsPrec p (x:%y)  =  showParen (p > prec)
                               (shows x . showString " % " . shows y)
```

A.4 Prelude PreludeComplex

```
-- Complex Numbers

module PreludeComplex ( Complex((:+)) )  where

infix  6  :+

data  (RealFloat a)     => Complex a = a :+ a  deriving (Eq,Binary,Text)

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 (((-y+1):+x) * sqrt (1/(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 ((z+1) * sqrt (1 - 1/(z*z)))

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, phase :: (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 (x:+y)     =  atan2 y x
```

A.5 Prelude PreludeList

```
-- Standard list functions

module PreludeList  where

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

-- head and tail extract the first element and remaining elements,
-- respectively, of a list, which must be non-empty.  last and init
-- are the dual functions working from the end of a finite list,
-- rather than the beginning.

head                    :: [a] -> a
head (x:_)              =  x
head []                 =  error "head{PreludeList}: head []"

last                    :: [a] -> a
last [x]                =  x
last (_:xs)             =  last xs
last []                 =  error "last{PreludeList}: last []"

tail                    :: [a] -> [a]
tail (_:xs)             =  xs
tail []                 =  error "tail{PreludeList}: tail []"

init                    :: [a] -> [a]
init [x]                =  []
init (x:xs)             =  x : init xs
init []                 =  error "init{PreludeList}: init []"

-- null determines if a list is empty.
null                    :: [a] -> Bool
null []                 =  True
null (_:_)              =  False

-- list concatenation (right-associative)
(++)                    :: [a] -> [a] -> [a]
xs ++ ys                =  foldr (:) ys xs
```

```
-- 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.
(\\)                    :: (Eq a) => [a] -> [a] -> [a]
(\\)                    =  foldl del
                           where [] `del` _          = []
                                 (x:xs) `del` y
                                        | x == y    = xs
                                        | otherwise = x : xs `del` y

-- length returns the length of a finite list as an Int; it is an instance
-- of the more general genericLength, the result type of which may be
-- any kind of number.

genericLength           :: (Num a) => [b] -> a
genericLength           =  foldl (\n _ -> n+1) 0

length                  :: [a] -> Int
length                  =  genericLength

-- Array index (subscript) operator, 0-origin
(!!)                    :: (Integral a) => [b] -> a -> b
(x:_)  !! 0             =  x
(_:xs) !! (n+1)         =  xs !! n
(_:_)  !! _             =  error "(!!){PreludeList}: negative index"
[]     !! (_+1)         =  error "(!!){PreludeList}: index too large"

-- map f xs applies f to each element of xs; i.e., map f xs == [f x | x <- xs].
map                     :: (a -> b) -> [a] -> [b]
map f []                =  []
map f (x:xs)            =  f x : map f xs

-- 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].
filter                  :: (a -> Bool) -> [a] -> [a]
filter p                =  foldr (\x xs -> if p x then x:xs else xs) []

-- partition takes a predicate and a list and returns a pair of lists:
-- those elements of the argument list that do and do not satisfy the
-- predicate, respectively; i.e.,
-- partition p xs == (filter p xs, filter (not . p) xs).
partition               :: (a -> Bool) -> [a] -> ([a],[a])
partition p             =  foldr select ([],[])
                           where select x (ts,fs) | p x       = (x:ts,fs)
                                                  | otherwise = (ts,x:fs)
```

```
-- 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

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

-- 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)

scanr                   :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []           =  [q0]
scanr f q0 (x:xs)       =  f x q : qs
                           where qs@(q:_) = scanr f q0 xs

scanr1                  :: (a -> a -> a) -> [a] -> [a]
scanr1 f  [x]           =  [x]
scanr1 f  (x:xs)        =  f x q : qs
                           where qs@(q:_) = scanr1 f xs
```

```
-- 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

-- cycle ties a finite list into a circular one, or equivalently,
-- the infinite repetition of the original list.  It is the identity
-- on infinite lists.

cycle                   :: [a] -> [a]
cycle xs                =  xs' where xs' = xs ++ xs'

-- take n, applied to a list xs, returns the prefix of xs of length n,
-- or xs itself if n > length xs.  drop n xs returns the suffix of xs
-- after the first n elements, or [] if n > length xs.  splitAt n xs
-- is equivalent to (take n xs, drop n xs).

take                    :: (Integral a) => a -> [b] -> [b]
take  0     _           =  []
take  _     []          =  []
take (n+1) (x:xs)       =  x : take n xs

drop                    :: (Integral a) => a -> [b] -> [b]
drop  0     xs          =  xs
drop  _     []          =  []
drop (n+1) (_:xs)       =  drop n xs

splitAt                 :: (Integral a) => a -> [b] -> ([b],[b])
splitAt  0     xs       =  ([],xs)
splitAt  _     []       =  ([],[])
splitAt (n+1) (x:xs)    =  (x:xs',xs'') where (xs',xs'') = splitAt 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) where (ys,zs) = span p xs'
           | otherwise  =  ([],xs)
break p                 =  span (not . p)

-- lines breaks a string up into a list of strings at newline characters.
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.

lines                   :: String -> [String]
lines ""                =  []
lines s                 =  l : (if null s' then [] else lines (tail s'))
                           where (l, s') = break ((==) '\n') s

words                   :: String -> [String]
words s                 =  case dropWhile isSpace s of
                                "" -> []
                                s' -> w : words s''
                                      where (w, s'') = break isSpace s'

unlines                 :: [String] -> String
unlines ls              =  concat (map (\l -> l ++ "\n") ls)

unwords                 :: [String] -> String
unwords []              = ""
unwords ws              = foldr1 (\w s -> w ++ ' ':s) ws

-- nub (meaning "essence") removes duplicate elements from its list argument.
nub                     :: (Eq a) => [a] -> [a]
nub []                  =  []
nub (x:xs)              =  x : nub (filter (/= x) xs)

-- 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                    =  any . (==)
notElem                 =  all . (/=)

-- 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

-- sums and products give a list of running sums or products from
-- a list of numbers.  For example,  sums [1,2,3] == [0,1,3,6].
sums, products          :: (Num a) => [a] -> [a]
sums                    =  scanl (+) 0
products                =  scanl (*) 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                 =  foldl1 max
minimum                 =  foldl1 min

-- concat, applied to a list of lists, returns their flattened concatenation.
concat                  :: [[a]] -> [a]
concat                  =  foldr (++) []
```

```
-- transpose, applied to a list of lists, returns that list with the
-- "rows" and "columns" interchanged.  The input need not be rectangular
-- (a list of equal-length lists) to be completely transposable, but can
-- be "triangular":  Each successive component list must be not longer
-- than the previous one; any elements outside of the "triangular"
-- transposable region are lost.  The input can be infinite in either
-- dimension or both.
transpose               :: [[a]] -> [[a]]
transpose               =  foldr
                             (\xs xss -> zipWith (:) xs (xss ++ repeat []))
                             []

-- 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, etc.  Versions
-- of zip producing up to septuplets are defined here.

zip                     :: [a] -> [b] -> [(a,b)]
zip                     =  zipWith (\a b -> (a,b))

zip3                    :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3                    =  zipWith3 (\a b c -> (a,b,c))

zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (\a b c d -> (a,b,c,d))

zip5                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip5                    =  zipWith5 (\a b c d e -> (a,b,c,d,e))

zip6                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f]
                           -> [(a,b,c,d,e,f)]
zip6                    =  zipWith6 (\a b c d e f -> (a,b,c,d,e,f))

zip7                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
                           -> [(a,b,c,d,e,f,g)]
zip7                    =  zipWith7 (\a b c d e f g -> (a,b,c,d,e,f,g))

-- 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 _ _ _ _        =  []

zipWith4                :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith4 z (a:as) (b:bs) (c:cs) (d:ds)
                        =  z a b c d : zipWith4 z as bs cs ds
zipWith4 _ _ _ _ _      =  []

zipWith5                :: (a->b->c->d->e->f)
                           -> [a]->[b]->[c]->[d]->[e]->[f]
zipWith5 z (a:as) (b:bs) (c:cs) (d:ds) (e:es)
                        =  z a b c d e : zipWith5 z as bs cs ds es
zipWith5 _ _ _ _ _ _    =  []

zipWith6                :: (a->b->c->d->e->f->g)
                           -> [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith6 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs)
                        =  z a b c d e f : zipWith6 z as bs cs ds es fs
zipWith6 _ _ _ _ _ _ _  =  []

zipWith7                :: (a->b->c->d->e->f->g->h)
                           -> [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
zipWith7 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs) (g:gs)
                   =  z a b c d e f g : zipWith7 z as bs cs ds es fs gs
zipWith7 _ _ _ _ _ _ _ _ =  []
```

A.6 Prelude PreludeArray

```
module  PreludeArray ( Array, Assoc((:=)), array, listArray, (!), bounds,
                     indices, elems, assocs, accumArray, (//), accum, amap,
                     ixmap
                   ) where

-- This module specifies the semantics of arrays only: it is not
-- intended as an efficient implementation.

infixl 9  !
infixl 9  //
infix  1  :=

data  Assoc a b =  a := b  deriving (Eq, Ord, Ix, Text, Binary)
data  (Ix a)    => Array a b = MkArray (a,a) (a -> b) deriving ()

array           :: (Ix a) => (a,a) -> [Assoc a b] -> Array a b
listArray       :: (Ix a) => (a,a) -> [b] -> Array a b
(!)             :: (Ix a) => Array a b -> a -> b
bounds          :: (Ix a) => Array a b -> (a,a)
indices         :: (Ix a) => Array a b -> [a]
elems           :: (Ix a) => Array a b -> [b]
assocs          :: (Ix a) => Array a b -> [Assoc a b]
accumArray      :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [Assoc a c]
                             -> Array a b
(//)            :: (Ix a) => Array a b -> [Assoc a b] -> Array a b
accum           :: (Ix a) => (b -> c -> b) -> Array a b -> [Assoc a c]
                             -> Array a b
amap            :: (Ix a) => (b -> c) -> Array a b -> Array a c
ixmap           :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                             -> Array a c

array b ivs =
    if and [inRange b i | i:=_ <- ivs]
        then MkArray b
                     (\j -> case [v | (i := v) <- ivs, i == j] of
                            [v]   -> v
                            []    -> error "(!){PreludeArray}: \
                                           \undefined array element"
                            _     -> error "(!){PreludeArray}: \
                                           \multiply defined array element")
        else error "array{PreludeArray}: out-of-range array association"

listArray b vs        = array b (zipWith (:=) (range b) vs)

(!) (MkArray _ f)     = f
```

```
bounds (MkArray b _)  = b

indices               = range . bounds

elems a               = [a!i | i <- indices a]

assocs a              = [i := a!i | i <- indices a]

a // us               = array (bounds a)
                           ([i := a!i | i <- indices a \\ [i | i:=_ <- us]]
                            ++ us)

accum f               = foldl (\a (i := v) -> a // [i := f (a!i) v])

accumArray f z b      = accum f (array b [i := z | i <- range b])

amap f a              = array b [i := f (a!i) | i <- range b]
                        where b = bounds a

ixmap b f a           = array b [i := a ! f i | i <- range b]

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, Text a, Text b) => Text (Array a b)  where
    showsPrec p a = showParen (p > 9) (
                    showString "array " .
                    shows (bounds a) . showChar ' ' .
                    shows (assocs a)                  )

    readsPrec p = readParen (p > 9)
           (\r -> [(array b as, u) | ("array",s) <- [lex r],
                                     (b,t)       <- reads s,
                                     (as,u)      <- reads t   ]
                  ++
                  [(listArray b xs, u) | ("listArray",s) <- [lex r],
                                         (b,t)           <- reads s,
                                         (xs,u)          <- reads t ])

instance  (Ix a, Binary a, Binary b) => Binary (Array a b)  where
    showBin a = showBin (bounds a) . showBin (elems a)

    readBin bin = (listArray b vs, bin'')
                 where (b,bin')   = readBin bin
                       (vs,bin'') = readBin bin'
```

A.7 Prelude PreludeText

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

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

class  Text a  where
    readsPrec :: Int -> ReadS a
    showsPrec :: Int -> a -> ShowS
    readList  :: ReadS [a]
    showList  :: [a] -> ShowS

    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)    <- read t,
                                               (xs,v)   <- readl' u]
    showList [] = showString "[]"
    showList (x:xs)
                = showChar '[' . shows x . showl xs
                  where showl []     = showChar ']'
                        showl (x:xs) = showChar ',' . shows x . showl xs

reads           :: (Text a) => ReadS a
reads           =  readsPrec 0

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

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

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

showChar        :: Char -> ShowS
showChar        =  (:)
```

```
showString      :: String -> ShowS
showString      =  (++)

showParen       :: Bool -> ShowS -> ShowS
showParen b p   =  if b then showChar '(' . p . showChar ')' else p

readParen       :: Bool -> ReadS a -> ReadS a
readParen b g   =  if b then mandatory else optional
                   where optional r  = g r ++ mandatory r
                         mandatory r = [(x,u) | ("(",s) <- lex r,
                                                (x,t)   <- optional s,
                                                (")",u) <- lex t    ]

lex                     :: ReadS String
lex ""                  = [("","")]
lex (c:s) | isSpace c   = lex (dropWhile isSpace s)
lex ('-':'-':s)         = case dropWhile (/= '\n') s of
                                 '\n':t -> lex t
                                 _      -> [] -- unterminated end-of-line
                                              -- comment

lex ('{':'-':s)         = lexNest lex s
                          where
                          lexNest f ('-':'}':s) = f s
                          lexNest f ('{':'-':s) = lexNest (lexNest f) s
                          lexNest f (c:s)             = lexNest f s
                          lexNest _ ""          = [] -- unterminated
                                                     -- nested comment

lex ('-':'>':s)         = [("->",s)]
lex ('<':'-':s)         = [("<-",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)]
          | isSym1 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` ",;()[]{}_"
              isSym1 c    =  c `elem` "-~" || isSym c
              isSym c     =  c `elem` "!@#$%&*+./<=>?\\^|:"
              isIdChar c  =  isAlphanum c || c `elem` "_'"

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

              lexExp (e:s) | e `elem` "eE"
                       = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
                                                 (ds,u) <- lexDigits t] ++
                         [(e:ds,t)   | (ds,t) <- lexDigits s]
              lexExp s = [("",s)]

lexDigits               :: ReadS String
lexDigits               =  nonnull isDigit

nonnull                 :: (char -> Bool) -> ReadS String
nonnull p s             =  [(cs,t) | (cs@(_:_),t) <- [span p s]]

lexLitChar              :: ReadS String
lexLitChar ('\\':s)     =  [('\\':esc, t) | (esc,t) <- lexEsc s]
        where
        lexEsc (c:s)     | c `elem` "abfnrtv\\\"'" = [([c],s)]
        lexEsc ('^':c:s) | c >= '@' && c <= '_'  = [(['^',c],s)]
        lexEsc s@(d:_)   | isDigit d             = lexDigits s
        lexEsc ('o':s)  =  [('o':os, t) | (os,t) <- nonnull isOctDigit s]
        lexEsc ('x':s)  =  [('x':xs, t) | (xs,t) <- nonnull isHexDigit s]
        lexEsc s@(c:_)   | isUpper c
                        =  case [(mne,s') | mne <- "DEL" : elems asciiTab,
                                            ([],s') <- [match mne s]      ]
                           of (pr:_) -> [pr]
                              []     -> []
        lexEsc _        =  []
lexLitChar (c:s)        =  [([c],s)]

isOctDigit c  =  c >= '0' && c <= '7'
isHexDigit c  =  isDigit c || c >= 'A' && c <= 'F'
                           || c >= 'a' && c <= 'f'
```

```
match                   :: (Eq a) => [a] -> [a] -> ([a],[a])
match (x:xs) (y:ys) | x == y  =  match xs ys
match xs     ys               =  (xs,ys)

asciiTab = listArray ('\NUL', ' ')
           ["NUL", "SOH", "STX", "ETX", "EOT", "ENQ", "ACK", "BEL",
            "BS",  "HT",  "LF",  "VT",  "FF",  "CR",  "SO",  "SI",
            "DLE", "DC1", "DC2", "DC3", "DC4", "NAK", "SYN", "ETB",
            "CAN", "EM",  "SUB", "ESC", "FS",  "GS",  "RS",  "US",
            "SP"]

-- Trivial type

instance  Text ()  where
    readsPrec p    = readParen False
                            (\r -> [((),t) | ("(",s) <- lex r,
                                             (")",t) <- lex s ] )
    showsPrec p () = showString "()"

-- Binary type

instance  Text Bin  where
    readsPrec p s  =  error "readsPrec{PreludeText}: Cannot read Bin."
    showsPrec p b  =  showString "<<Bin>>"

-- Character type

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

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

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

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

```
readLitChar             :: ReadS Char
readLitChar ('\\':s)    =  readEsc s
        where
        readEsc ('a':s)  = [('\a',s)]
        readEsc ('b':s)  = [('\b',s)]
        readEsc ('f':s)  = [('\f',s)]
        readEsc ('n':s)  = [('\n',s)]
        readEsc ('r':s)  = [('\r',s)]
        readEsc ('t':s)  = [('\t',s)]
        readEsc ('v':s)  = [('\v',s)]
        readEsc ('\\':s) = [('\\',s)]
        readEsc ('"':s)  = [('"',s)]
        readEsc ('\'':s) = [('\'',s)]
        readEsc ('^':c:s) | c >= '@' && c <= '_'
                         = [(chr (ord c - ord '@'), s)]
        readEsc s@(d:_) | isDigit d
                         = [(chr n, t) | (n,t) <- readDec s]
        readEsc ('o':s)  = [(chr n, t) | (n,t) <- readOct s]
        readEsc ('x':s)  = [(chr n, t) | (n,t) <- readHex s]
        readEsc s@(c:_) | isUpper c
                         = let table = ('\DEL' := "DEL") : assocs asciiTab
                           in case [(c,s') | (c := mne) <- table,
                                             ([],s') <- [match mne s]]
                              of (pr:_) -> [pr]
                                 []     -> []
        readEsc _        = []
readLitChar (c:s)       =  [(c,s)]

showLitChar                :: Char -> ShowS
showLitChar c | c > '\DEL' =  protectEsc isDigit (showInt (ord c))
showLitChar '\DEL'         =  showString "\\DEL"
showLitChar '\\'           =  showString "\\\\"
showLitChar c | c >= ' '   =  showChar c
showLitChar '\a'           =  showString "\\a"
showLitChar '\b'           =  showString "\\b"
showLitChar '\f'           =  showString "\\f"
showLitChar '\n'           =  showString "\\n"
showLitChar '\r'           =  showString "\\r"
showLitChar '\t'           =  showString "\\t"
showLitChar '\v'           =  showString "\\v"
showLitChar '\SO'          =  protectEsc (== 'H') (showString "\\SO")
showLitChar c              =  showString ('\\' : asciiTab!c)

protectEsc p f             = f . cont
                             where cont s@(c:_) | p c = "\\&" ++ s
                                   cont s             = s
```

```
readDec, readOct, readHex :: (Integral a) => ReadS a
readDec = readInt 10 isDigit (\d -> ord d - ord '0')
readOct = readInt  8 isOctDigit (\d -> ord d - ord '0')
readHex = readInt 16 isHexDigit hex
            where hex d = ord d - (if isDigit d then ord '0'
                                   else ord (if isUpper d then 'A' else 'a')
                                        - 10)

readInt :: (Integral a) => a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readInt radix isDig digToInt s =
    [(foldl1 (\n d -> n * radix + d) (map (fromIntegral . digToInt) ds), r)
        | (ds,r) <- nonnull isDig s ]

showInt :: (Integral a) => a -> ShowS
showInt n = if n < 0 then showChar '-' . showInt' (-n) else showInt' n
            where showInt' n r = let (n',d) = divRem n 10
                                     r' = chr (ord '0' + fromIntegral d) : r
                                 in if n' == 0 then r' else showInt n' r'

-- Standard integral types

instance  Text Int  where
    readsPrec p = readSigned readDec
    showsPrec   = showSigned showInt

instance  Text Integer  where
    readsPrec p = readSigned readDec
    showsPrec   = showSigned showInt

readSigned:: (Real a) => ReadS a -> ReadS a
readSigned readPos = readParen False read'
                     where read' r  = read'' r ++
                                      [(-x,t) | ("-",s) <- lex r,
                                                (x,t)   <- read'' s]
                           read'' r = [(n,s)  | (str,s) <- lex r,
                                                (n,"")  <- readPos str]

showSigned:: (Real a) => (a -> ShowS) -> Int -> a -> ShowS
showSigned showPos p x = showParen (x < 0 && p > 6) (showPos x)

-- Standard real floating-point types

instance  Text Float  where
    readsPrec p = readSigned readFloat
    showsPrec   = showSigned showFloat

instance  Text Double  where
    readsPrec p = readSigned readFloat
    showsPrec   = showSigned showFloat
```

```
-- The functions readFloat and showFloat below use rational arithmetic
-- to insure correct conversion between the floating-point radix and
-- decimal.  It is often possible to use a higher-precision floating-
-- point type to obtain the same results.

readFloat r = [(fromRational ((n%1)*10^^(k-d)), t) | (n,d,s) <- readFix r,
                                                    (k,t)   <- readExp s]
              where readFix r = [(read (ds++ds'), length ds', t)
                                        | (ds,'.':s) <- lexDigits r,
                                          (ds',t)    <- lexDigits s ]

                    readExp (e:s) | e `elem` "eE" = readExp' s
                    readExp s                     = [(0,s)]

                    readExp' ('-':s) = [(-k,t) | (k,t) <- readDec s]
                    readExp' ('+':s) = readDec s
                    readExp' s       = readDec s

-- The number of decimal digits m below is chosen to guarantee
-- read(show x) = x.  See
--      Matula, D. W.  A formalization of floating-point numeric base
--      conversion.  IEEE Transactions on Computers C-19, 8 (1970 August),
--      681-692.

showFloat x =
    if x == 0 then showString ("0." ++ take (m-1) (repeat '0'))
              else if e >= m-1 || e < 0 then showSci else showFix
    where
    showFix     = showString whole . showChar '.' . showString frac
                  where (whole,frac) = splitAt (e+1) (show sig)
    showSci     = showChar d . showChar '.' . showString frac
                      . showChar 'e' . showInt e
                  where (d:frac) = show sig
    (m, sig, e) = if b == 10 then (w,   s,   n+w-1)
                             else (m', sig', e'   )
    m'          = ceiling ((fromInt w * log (fromInteger b))/log 10) + 1
    (sig', e')  = if      sig1 >= 10^m'     then (round (t/10), e1+1)
                  else if sig1 <  10^(m'-1) then (round (t*10), e1-1)
                                            else (sig1,          e1  )
    sig1        = round t
    t           = s%1 * (b%1)^^n * 10^^(m'-e1-1)
    e1          = floor (logBase 10 x)
    (s, n)      = decodeFloat x
    b           = floatRadix x
    w           = floatDigits x
```

```
-- Lists

instance  (Text a) => Text [a]  where
    readsPrec p = readList
    showsPrec p = showList

-- Tuples

instance  (Text a, Text b) => Text (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 ] )

    showsPrec p (x,y) = showChar '(' . shows x . showChar ',' .
                                       shows y . showChar ')'
-- et cetera

-- Functions

instance  Text (a -> b)  where
    readsPrec p s  =  error "readsPrec{PreludeText}: Cannot read functions."
    showsPrec p f  =  showString "<<function>>"
```

A.8 Prelude PreludeIO

```
-- I/O functions and definitions

module PreludeIO  where

-- File and channel names:

stdin       =  "stdin"
stdout      =  "stdout"
stderr      =  "stderr"
stdecho     =  "stdecho"

-- Requests and responses:

data Request =  -- file system requests:
                        ReadFile      String
                      | WriteFile     String String
                      | AppendFile    String String
                      | ReadBinFile   String
                      | WriteBinFile  String Bin
                      | AppendBinFile String Bin
                      | DeleteFile    String
                      | StatusFile    String
                -- channel system requests:
                      | ReadChan      String
                      | AppendChan    String String
                      | ReadBinChan   String
                      | AppendBinChan String Bin
                      | StatusChan    String
                -- environment requests:
                      | Echo          Bool
                      | GetArgs
                      | GetEnv        String
                      | SetEnv        String String
                deriving Text

data Response =         Success
                      | Str String
                      | StrList [String]
                      | Bn  Bin
                      | Failure IOError
                deriving Text
```

```
data IOError =            WriteError   String
                        | ReadError    String
                        | SearchError  String
                        | FormatError  String
                        | OtherError   String
                deriving Text

-- Continuation-based I/O:

type Dialogue    =  [Response] -> [Request]
type SuccCont    =                Dialogue
type StrCont     =  String     -> Dialogue
type StrListCont =  [String]   -> Dialogue
type BinCont     =  Bin        -> Dialogue
type FailCont    =  IOError    -> Dialogue

done          ::                                              Dialogue
readFile      :: String ->           FailCont -> StrCont     -> Dialogue
writeFile     :: String -> String -> FailCont -> SuccCont    -> Dialogue
appendFile    :: String -> String -> FailCont -> SuccCont    -> Dialogue
readBinFile   :: String ->           FailCont -> BinCont     -> Dialogue
writeBinFile  :: String -> Bin    -> FailCont -> SuccCont    -> Dialogue
appendBinFile :: String -> Bin    -> FailCont -> SuccCont    -> Dialogue
deleteFile    :: String ->           FailCont -> SuccCont    -> Dialogue
statusFile    :: String ->           FailCont -> StrCont     -> Dialogue
readChan      :: String ->           FailCont -> StrCont     -> Dialogue
appendChan    :: String -> String -> FailCont -> SuccCont    -> Dialogue
readBinChan   :: String ->           FailCont -> BinCont     -> Dialogue
appendBinChan :: String -> Bin    -> FailCont -> SuccCont    -> Dialogue
statusChan    :: String ->           FailCont -> StrCont     -> Dialogue
echo          :: Bool ->             FailCont -> SuccCont    -> Dialogue
getArgs       ::                     FailCont -> StrListCont -> Dialogue
getEnv        :: String ->           FailCont -> StrCont     -> Dialogue
setEnv        :: String -> String -> FailCont -> SuccCont    -> Dialogue

done resps    =  []

readFile name fail succ resps =
     (ReadFile name) : strDispatch fail succ resps

writeFile name contents fail succ resps =
    (WriteFile name contents) : succDispatch fail succ resps

appendFile name contents fail succ resps =
    (AppendFile name contents) : succDispatch fail succ resps

readBinFile name fail succ resps =
    (ReadBinFile name) : binDispatch fail succ resps
```

```
writeBinFile name contents fail succ resps =
    (WriteBinFile name contents) : succDispatch fail succ resps

appendBinFile name contents fail succ resps =
    (AppendBinFile name contents) : succDispatch fail succ resps

deleteFile name fail succ resps =
    (DeleteFile name) : succDispatch fail succ resps

statusFile name fail succ resps =
    (StatusFile name) : strDispatch fail succ resps

readChan name fail succ resps =
    (ReadChan name) : strDispatch fail succ resps

appendChan name contents fail succ resps =
    (AppendChan name contents) : succDispatch fail succ resps

readBinChan name fail succ resps =
    (ReadBinChan name) : binDispatch fail succ resps

appendBinChan name contents fail succ resps =
    (AppendBinChan name contents) : succDispatch fail succ resps

statusChan name fail succ resps =
    (StatusChan name) : strDispatch fail succ resps

echo bool fail succ resps =
    (Echo bool) : succDispatch fail succ resps

getArgs fail succ resps =
        GetArgs : strListDispatch fail succ resps

getEnv name fail succ resps =
        (GetEnv name) : strDispatch fail succ resps

setEnv name val fail succ resps =
        (SetEnv name val) : succDispatch fail succ resps

strDispatch fail succ (resp:resps) =
            case resp of Str val     -> succ val resps
                         Failure msg -> fail msg resps

strListDispatch fail succ (resp:resps) =
            case resp of StrList val -> succ val resps
                         Failure msg -> fail msg resps

binDispatch fail succ (resp:resps) =
            case resp of Bn val      -> succ val resps
                         Failure msg -> fail msg resps
```

```
succDispatch fail succ (resp:resps) =
            case resp of Success     -> succ resps
                         Failure msg -> fail msg resps

abort           :: FailCont
abort err       =  done

exit            :: FailCont
exit err        = appendChan stdout msg abort done
                  where msg = case err of ReadError s   -> s
                                          WriteError s  -> s
                                          SearchError s -> s
                                          FormatError s -> s
                                          OtherError s  -> s

print           :: (Text a) => a -> Dialogue
print x         =  appendChan stdout (show x) abort done
prints          :: (Text a) => a -> String -> Dialogue
prints x s      =  appendChan stdout (shows x s) abort done

interact        :: (String -> String) -> Dialogue
interact f      =  readChan stdin abort
                            (\x -> appendChan stdout (f x) abort done)
```

B Syntax

B.1 Notational Conventions

These notational conventions are used for presenting syntax:

[*pattern*]	optional
{*pattern*}	zero or more repetitions
(*pattern*)	grouping
$pat_1 \mid pat_2$	choice
$pat_{\{pat'\}}$	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 form:

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

There are some families of nonterminals indexed by precedence levels (written as a superscript). Similarly, the lexeme classes *op*, *varop*, and *conop* have a double index: a letter l, r, or n for left-, right- or nonassociativity and a precedence level. So, for example

$$exp^i \quad \rightarrow \quad exp^{i+1}\ [op^{(\mathrm{n},i)}\ exp^{i+1}]\ (0 \leq i \leq 9)$$

actually stands for 10 productions where *op* is non-associative.

B.2 Minor Syntax Changes in Version 1.1

This section is a list of the non-trivial changes to the Haskell syntax between versions 1.0 and 1.1 of this report, *excluding* those mentioned in the 1.1 preface (page). Other clarifications and corrections are reflected in the full syntax in the following sections.

1. Empty declarations and declaration lists ending with `;` have been added, to aid automatic program generation.
2. Guards have been eliminated from lambda expressions.
3. List comprehensions must have at least one qualifier.
4. `Case` expressions may have more than one guard per clause.
5. Instance declarations can only have *valdefs* in their body; in particular, they cannot have type signatures in their body.

B.3 Lexical Syntax

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

whitespace → *whitestuff* {*whitestuff*}
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *space* | *tab* | *vertab* | *formfeed*
newline → a newline (system dependent)
space → a space
tab → a horizontal tab
vertab → a vertical tab
formfeed → a form feed
comment → `--` {*any*} *newline*
ncomment → `{-` *ANYseq* {*ncomment* *ANYseq*} `-}`
ANYseq → $\{ANY\}_{\{ANY\}\ (\ \texttt{\{-}\ \mid\ \texttt{-\}}\ )\ \{ANY\}}$
ANY → *any* | *newline* | *vertab* | *formfeed*
any → *graphic* | *space* | *tab*
graphic → *large* | *small* | *digit*
| `!` | `"` | `#` | `$` | `%` | `&` | `´` | `(` | `)` | `*` | `+`
| `,` | `-` | `.` | `/` | `:` | `;` | `<` | `=` | `>` | `?` | `@`
| `[` | `\` | `]` | `^` | `_` | `` ` `` | `{` | `|` | `}` | `~`

small → `a` | `b` | ... | `z`
large → `A` | `B` | ... | `Z`
digit → `0` | `1` | ... | `9`

avarid → $(small\ \{small \mid large \mid digit \mid \texttt{´} \mid \texttt{\_}\})_{\{reservedid\}}$
varid → *avarid* | `(`*avarop*`)`
aconid → *large* {*small* | *large* | *digit* | `´` | `_`}
conid → *aconid* | `(`*aconop*`)`
reservedid → `case` | `class` | `data` | `default` | `deriving` | `else` | `hiding`
| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance` | `interface`
| `let` | `module` | `of` | `renaming` | `then` | `to` | `type` | `where`

avarop → $(\ (\ symbol \mid presymbol\ )\ \{symbol \mid \texttt{:}\}\ )_{\{reservedop\}}$
varop → *avarop* | `` `avarid` ``
aconop → $(\texttt{:}\ \{symbol \mid \texttt{:}\})_{\{reservedop\}}$
conop → *aconop* | `` `aconid` ``
presymbol → `-` | `~`
symbol → `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@` | `\` | `^` | `|`
reservedop → `..` | `::` | `=>` | `=` | `@` | `\` | `|` | `~` | `<-` | `->`

var → *varid* (*variables*)
con → *conid* (*constructors*)
tyvar → *avarid* (*type variables*)
tycon → *aconid* (*type constructors*)
tycls → *aconid* (*type classes*)
modid → *aconid* (*modules*)

integer → *digit*{*digit*}
float → *integer* . *integer*[(`e` | `E`)[`-` | `+`]*integer*]

char → ´ ($graphic_{\{´ \mid \backslash\}}$ | *space* | $escape_{\{\backslash\&\}}$) ´
string → `"` {$graphic_{\{" \mid \backslash\}}$ | *space* | *escape* | *gap*} `"`
escape → `\` (*charesc* | *ascii* | *integer* | `o` *octit*{*octit*} | `x` *hexit*{*hexit*})
charesc → `a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | ´ | `&`
ascii → `^`*cntrl* | `NUL` | `SOH` | `STX` | `ETX` | `EOT` | `ENQ` | `ACK`
| `BEL` | `BS` | `HT` | `LF` | `VT` | `FF` | `CR` | `SO` | `SI` | `DLE`
| `DC1` | `DC2` | `DC3` | `DC4` | `NAK` | `SYN` | `ETB` | `CAN`
| `EM` | `SUB` | `ESC` | `FS` | `GS` | `RS` | `US` | `SP` | `DEL`
cntrl → *large* | `@` | `[` | `\` | `]` | `^` | `_`
gap → `\` *whitechar* {*whitechar*} `\`
hexit → *digit* | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`
octit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

B.4 Layout

Definitions: The indentation of a lexeme is the column number indicating the start of that lexeme; the indentation of a line is the indentation of its leftmost lexeme. To determine the column number, assume a fixed-width font with this tab convention: tab stops are 8 characters apart, and a tab character causes the insertion of enough spaces to align the current position with the next tab stop.

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

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

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

The layout (or "off-side") rule takes effect whenever the open brace is omitted after the keyword `where`, `let` or `of`. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments). For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the declaration list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the declaration list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted. The layout rule will match only those open braces that it has inserted; an open brace that the user has inserted must be matched by a close brace inserted by the user.

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

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

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

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

B.5 Context-Free Syntax

$$
\begin{array}{lcll}
\mathit{module} & \rightarrow & \texttt{module}\ \mathit{modid}\ [\mathit{exports}]\ \texttt{where}\ \mathit{body} & \\
 & | & \mathit{body} & \\
\mathit{body} & \rightarrow & \texttt{\{}\ [\mathit{impdecls}\ \texttt{;}]\ [[\mathit{fixdecls}\ \texttt{;}]\ \mathit{topdecls}\ [\texttt{;}]]\ \texttt{\}} & \\
 & | & \texttt{\{}\ \mathit{impdecls}\ [\texttt{;}]\ \texttt{\}} & \\
 & & & \\
\mathit{modid} & \rightarrow & \mathit{aconid} & \\
\mathit{impdecls} & \rightarrow & \mathit{impdecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{impdecl}_n & (n \geq 1) \\
 & & & \\
\mathit{exports} & \rightarrow & \texttt{(}\ \mathit{export}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{export}_n\ \texttt{)} & (n \geq 1) \\
 & & & \\
\mathit{export} & \rightarrow & \mathit{entity} & \\
 & | & \mathit{modid}\ \texttt{..} & \\
 & & & \\
\mathit{impdecl} & \rightarrow & \texttt{import}\ \mathit{modid}\ [\mathit{impspec}]\ [\texttt{renaming}\ \mathit{renamings}] & \\
\mathit{impspec} & \rightarrow & \texttt{(}\ \mathit{import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{import}_n\ \texttt{)} & (n \geq 0) \\
 & | & \texttt{hiding}\ \texttt{(}\ \mathit{import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{import}_n\ \texttt{)} & (n \geq 1) \\
\mathit{import} & \rightarrow & \mathit{entity} &
\end{array}
$$

renamings	$\rightarrow$	`(` *renaming*$_1$ `,` … `,` *renaming*$_n$ `)`	$(n \geq 1)$
renaming	$\rightarrow$	*varid*$_1$ `to` *varid*$_2$	
	$\vert$	*conid*$_1$ `to` *conid*$_2$	
entity	$\rightarrow$	*varid*	
	$\vert$	*tycon*	
	$\vert$	*tycon* `(..)`	
	$\vert$	*tycon* `(` *conid*$_1$ `,` … `,` *conid*$_n$ `)`	$(n \geq 1)$
	$\vert$	*tycls* `(..)`	
	$\vert$	*tycls* `(` *varid*$_1$ `,` … `,` *varid*$_n$ `)`	$(n \geq 0)$
fixdecls	$\rightarrow$	*fix*$_1$ `;` … `;` *fix*$_n$	$(n \geq 1)$
fix	$\rightarrow$	`infixl` [*digit*] *ops*	
	$\vert$	`infixr` [*digit*] *ops*	
	$\vert$	`infix` [*digit*] *ops*	
ops	$\rightarrow$	*op*$_1$ `,` … `,` *op*$_n$	$(n \geq 1)$
op	$\rightarrow$	*varop* $\vert$ *conop*	
topdecls	$\rightarrow$	*topdecl*$_1$ `;` … `;` *topdecl*$_n$	$(n \geq 1)$
topdecl	$\rightarrow$	`type` *simple* `=` *type*	
	$\vert$	`data` [*context* `=>`] *simple* `=` *constrs* [`deriving` (*tycls* $\vert$ `(`*tyclses*`)`)]	
	$\vert$	`class` [*context* `=>`] *class* [`where` `{` *cbody* [`;`] `}`]	
	$\vert$	`instance` [*context* `=>`] *tycls* *inst* [`where` `{` *valdefs* [`;`] `}`]	
	$\vert$	`default` (*type* $\vert$ `(`*type*$_1$ `,` … `,` *type*$_n$`)`)	$(n \geq 0)$
	$\vert$	*decl*	
decls	$\rightarrow$	*decl*$_1$ `;` … `;` *decl*$_n$	$(n \geq 0)$
decl	$\rightarrow$	*vars* `::` [*context* `=>`] *type*	
	$\vert$	*valdef*	
type	$\rightarrow$	*atype*	
	$\vert$	*type*$_1$ `->` *type*$_2$	
	$\vert$	*tycon* *atype*$_1$ … *atype*$_k$	(arity *tycon* $= k \geq 1$)
atype	$\rightarrow$	*tyvar*	
	$\vert$	*tycon*	(arity *tycon* $= 0$)
	$\vert$	`()`	(unit type)
	$\vert$	`(` *type* `)`	(parenthesised type)
	$\vert$	`(` *type*$_1$ `,` … `,` *type*$_k$ `)`	(tuple type, $k \geq 2$)
	$\vert$	`[` *type* `]`	

context	$\rightarrow$	*class*	
	$\mid$	`(` $class_1$ `,` ... `,` $class_n$ `)`	$(n \geq 1)$
class	$\rightarrow$	*tycls tyvar*	
cbody	$\rightarrow$	[*csigns* `;`] [*valdefs*]	
csigns	$\rightarrow$	$csign_1$ `;` ... `;` $csign_n$	$(n \geq 1)$
csign	$\rightarrow$	*vars* `::` [*context* `=>`] *type*	
vars	$\rightarrow$	var_1 `,` ... `,` var_n	$(n \geq 1)$
simple	$\rightarrow$	*tycon* $tyvar_1$ ... $tyvar_k$	(arity $tycon = k \geq 0$)
constrs	$\rightarrow$	$constr_1$ `|` ... `|` $constr_n$	$(n \geq 1)$
constr	$\rightarrow$	*con* $atype_1$ ... $atype_k$	(arity $con = k \geq 0$)
	$\mid$	$type_1$ *conop* $type_2$	(infix *conop*)
tyclses	$\rightarrow$	$tycls_1$ `,` ... `,` $tycls_n$	$(n \geq 0)$
inst	$\rightarrow$	*tycon*	(arity $tycon = 0$)
	$\mid$	`(` *tycon* $tyvar_1$ ... $tyvar_k$ `)`	($k \geq 1$, *tyvars* distinct)
	$\mid$	`(` $tyvar_1$ `,` ... `,` $tyvar_k$ `)`	($k \geq 2$, *tyvars* distinct)
	$\mid$	`()`	
	$\mid$	`[` *tyvar* `]`	
	$\mid$	`(` $tyvar_1$ `->` $tyvar_2$ `)`	$tyvar_1$ and $tyvar_2$ distinct
valdefs	$\rightarrow$	$valdef_1$ `;` ... `;` $valdef_n$	$(n \geq 0)$
valdef	$\rightarrow$	*lhs* `=` *exp* [`where {` *decls* [`;`] `}`]	
	$\mid$	*lhs gdrhs* [`where {` *decls* [`;`] `}`]	
lhs	$\rightarrow$	*apat*	
	$\mid$	*funlhs*	
funlhs	$\rightarrow$	*afunlhs*	
	$\mid$	pat_1^{i+1} $varop^{(\mathrm{n},i)}$ pat_2^{i+1}	$(0 \leq i \leq 9)$
	$\mid$	$lpat^i$ $varop^{(\mathrm{l},i)}$ pat^{i+1}	$(0 \leq i \leq 9)$
	$\mid$	pat^{i+1} $varop^{(\mathrm{r},i)}$ $rpat^i$	$(0 \leq i \leq 9)$
afunlhs	$\rightarrow$	*var apat*	
	$\mid$	`(` *funlhs* `)` *apat*	
	$\mid$	*afunlhs apat*	
gdrhs	$\rightarrow$	*gd* `=` *exp* [*gdrhs*]	
gd	$\rightarrow$	`|` *exp*	

exp	$\rightarrow$	`\` $apat_1$ ... $apat_n$ `->` *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)
	|	exp^0 `::` [*context* `=>`] *atype*	(expression type signature)
	|	exp^0	
exp^i	$\rightarrow$	exp^{i+1} [$op^{(\mathrm{n},i)}$ exp^{i+1}]	($0 \leq i \leq 9$)
	|	$lexp^i$ $op^{(\mathrm{l},i)}$ exp^{i+1}	
	|	exp^{i+1} $op^{(\mathrm{r},i)}$ $rexp^i$	
$lexp^i$	$\rightarrow$	[$lexp^i$ $op^{(\mathrm{l},i)}$] exp^{i+1}	($0 \leq i \leq 9$)
$lexp^6$	$\rightarrow$	`-` exp^7	
$rexp^i$	$\rightarrow$	exp^{i+1} [$op^{(\mathrm{r},i)}$ $rexp^i$]	($0 \leq i \leq 9$)
exp^{10}	$\rightarrow$	exp^{10} *aexp*	(function application)
	|	*aexp*	

aexp	$\rightarrow$	*var*	(variable)
	|	*con*	(constructor)
	|	*literal*	
	|	`()`	(unit)
	|	`(` *exp* `)`	(parenthesised expression)
	|	`(` exp_1 `,` ... `,` exp_k `)`	(tuple, $k \geq 2$)
	|	`[` exp_1 `,` ... `,` exp_k `]`	(list, $k \geq 0$)
	|	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	|	`[` *exp* `|` $qual_1$ `,` ... `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	|	`(` exp^{i+1} $op^{(a,i)}$ `)`	(section, $0 \leq i \leq 9$, $a \in \{n, l, r\}$)
	|	`(` $op^{(a,i)}$ exp^{i+1} `)`	(section, $0 \leq i \leq 9$, $a \in \{n, l, r\}$)

qual	$\rightarrow$	*pat* `<-` *exp*	
	|	*exp*	
alts	$\rightarrow$	alt_1 `;` ... `;` alt_n	($n \geq 0$)
alt	$\rightarrow$	*pat* `->` *exp* [`where {` *decls* [`;`] `}`]	
	|	*pat* *gdpat* [`where {` *decls* [`;`] `}`]	
gdpat	$\rightarrow$	*gd* `->` *exp* [*gdpat*]	

pat	$\rightarrow$	pat^0	
pat^i	$\rightarrow$	pat_1^{i+1} [$conop^{(\mathrm{n},i)}$ pat_2^{i+1}]	($0 \leq i \leq 9$)
	|	$lpat^i$ $conop^{(\mathrm{l},i)}$ pat^{i+1}	
	|	pat^{i+1} $conop^{(\mathrm{r},i)}$ $rpat^i$	
$lpat^i$	$\rightarrow$	[$lpat^i$ $conop^{(\mathrm{l},i)}$] pat^{i+1}	($0 \leq i \leq 9$)
$lpat^6$	$\rightarrow$	$lpat^6$ `+` *integer*	(successor pattern)

	$\vert$	`-` $\{integer \mid float\}$	(negative literal)
$rpat^i$	$\rightarrow$	pat^{i+1} $[conop^{(\mathrm{r},i)}$ $rpat^i]$	$(0 \leq i \leq 9)$
pat^{10}	$\rightarrow$	$apat$	
	$\vert$	con $apat_1$ ... $apat_k$	(arity $con = k \geq 1$)

$apat$	$\rightarrow$	var [`@` $apat$]	(as pattern)
	$\vert$	con	(arity $con = 0$)
	$\vert$	$literal$	
	$\vert$	`_`	(wildcard)
	$\vert$	`()`	(unit pattern)
	$\vert$	`(` pat `)`	(parenthesised pattern)
	$\vert$	`(` pat_1 `,` ... `,` pat_k `)`	(tuple pattern, $k \geq 2$)
	$\vert$	`[` pat_1 `,` ... `,` pat_k `]`	(list pattern, $k \geq 0$)
	$\vert$	`~` $apat$	(irrefutable pattern)

$tycls$	$\rightarrow$	$aconid$
$tyvar$	$\rightarrow$	$avarid$
$tycon$	$\rightarrow$	$aconid$

B.6 Interface Syntax

$interface$	$\rightarrow$	`interface` $modid$ `where` $ibody$	
$ibody$	$\rightarrow$	`{` [$iimpdecls$ `;`] [$fixdecls$ `;`] $itopdecls$ [`;`] `}`	
	$\vert$	`{` $iimpdecls$ [`;`] `}`	
$iimpdecls$	$\rightarrow$	$iimpdecl_1$ `;` ... `;` $iimpdecl_n$	$(n \geq 1)$
$iimpdecl$	$\rightarrow$	`import` $modid$ `(` $import_1$ `,` ... `,` $import_n$ `)` [`renaming` $renamings$]	$(n \geq 1)$
$itopdecls$	$\rightarrow$	$itopdecl_1$ `;` ... `;` $itopdecl_n$	$(n \geq 1)$
$itopdecl$	$\rightarrow$	`type` $simple$ `=` $type$	
	$\vert$	`data` [$context$ `=>`] $simple$ [`=` $constrs$] [`deriving` ($tycls$ $\vert$ `(`$tyclses$`)`)]	
	$\vert$	`class` [$context$ `=>`] $class$ [`where` `{` $icdecls$ [`;`] `}`]	
	$\vert$	`instance` [$context$ `=>`] $tycls$ $inst$	
	$\vert$	$vars$ `::` [$context$ `=>`] $type$	
$icdecls$	$\rightarrow$	$icdecl_1$ `;` ... `;` $icdecl_n$	$(n \geq 1)$
$icdecl$	$\rightarrow$	$vars$ `::` $type$	

C Input/Output Semantics

The behaviour of a Haskell program performing I/O is given within the environment in which it is running. That environment will be described using standard Haskell code augmented with a non-deterministic merge operator.

The state of the operating system (OS state) that is relevant to Haskell programs is completely described by the file system and the channel system. The channel system is split into two subsystems, the input channel system and the output channel system.

```
type State = (FileSystem, ChannelSystem)
type FileSystem    = String -> Response
type ChannelSystem = (ICs, OCs)
type ICs   = String -> (Agent, Open)
type OCs   = String -> Response
type Agent = (FileSystem, OCs) -> Response
type Open  = PId -> Bool
type PId   = Int
type PList = [(PId,[Request->Response])]
```

An agent maps a list of OS states to responses. Those responses will be used as the contents of input channels, and thus can depend on output channels, other input channels, files, or any combination thereof. For example, a valid implementation must allow the user to act as agent between the standard output channel and standard input channel.

Each running process (i.e. program) has a unique `PId`. Elements of `PList` are lists of running programs.

```
os :: TagReqList -> State -> (TagRespList, State)
type TagRespList = [(PId,Response)]
type TagReqList  = [(PId,Request)]
```

The operating system is modeled as a (non-deterministic) function `os`. The `os` takes a tagged request list and an initial state, and returns a tagged response list and a final state. Given a list of programs `pList`, `os` must exhibit this behaviour:

```
(tagRespList, state') = os tagReqList state
tagReqList = merge [ zip [pId,pId..] (proc (untag pId tagRespList))
                     | (pId, proc) <- pList ]
```

where `merge` is a non-deterministic merge of a list of lists, and `untag` is:

```
untag n []               = []
untag n ((m,resp):resps) = if n==m then resp:(untag n resps)
                                   else untag n resps
```

This relationship can be generalised to include requests such as `CreateProcess`.

A valid implementation must ensure that the input channel system is defined at `stdin` and the output channel system is defined at `stdout`, `stderr`, and `stdecho`. If the agent attached to standard input is called `user` (i.e. `ics stdin` has form `(user, open)`), then `user` must depend at least on standard output. In other words, this constraint must hold:

```
user [..., (fs,(ics,ocs)), ...] = ... user' (ocs stdout) ...
```

where `user'` is a *strict*, but otherwise arbitrary, function modelling the user. Its strictness corresponds to the user's consumption of standard output whilst determining standard input.

The rest of this section specifies the required behaviour of `os` in response to each kind of request. This semantics is relatively abstract and omits any reference to hardware errors (e.g. "bad sector on disk") and system dependent errors (e.g. "access rights violation"). Implementation-specific requests (for example the environment requests) are not shown here. We describe only the text version of the requests: the binary version differs trivially. `os` is defined by:

```
os :: TagReqList -> State -> (TagRespList,State)

os [] state = ([], state)
os ((n, ReadChan name):es) state@(fs,(ics,ocs)) =
    (alist',state') where
          (agent,open) = ics name
          alist' = (n, (if open n
                        then fail
                        else (agent (fs,ocs)) )) : alist
          fail = Failure (OtherError "Channel already open\n")
          (alist,state') = os es (fs, (update ics name
                                          (agent, update open n true),
                                        ocs))
```

where the auxiliary function `update` is defined by:

```
update f x v x' = if x==x' then v else f x
```

If an attempt is made to read a non-existent channel, `ics` returns an agent that gives the appropriate error message when applied to its arguments. This definition is generalised in the obvious way for the behaviour of `ReadChannels`. In particular, `ack` must be created by non-deterministically merging the result of applying each agent to the stream of future states.

```
os ((n, AppendChan name contents):es) state@(fs,(ics,ocs)) =
    (alist',state') where
          alist' = ack:alist
          ack =
           (n,
            case (ocs name) of
             Failure msg -> Failure (SearchError "Nonexistent Channel")
             Str ochan -> Success
             Bn ochan -> Failure (FormatError "format error")
            )
          (alist,state') = os es (fs,(ics,
                                      case (ocs name) of
                                        Failure msg -> ocs
                                        Str ochan -> update ocs name
                                              (Str (ochan ++ contents))
                                        Bn ochan -> ocs
                                     ))

os ((n, ReadFile name):es) state@(fs,(ics,ocs)) =
    (alist',state') where
          alist' = ack : alist
          ack = (n,
                 case (fs name) of
                  Failure msg -> Failure (SearchError "File not found")
                  Str string -> Str string
                  Bn binary -> Failure (FormatError "")
                )
          (alist,state') = os es state

os ((n, WriteFile name contents):es) state@(fs,(ics,ocs)) =
    (alist',state') where
          alist' = (n, Success):alist
          (alist,state') = os es (update fs name (Str contents),
                                  (ics,ocs))
```

```
os ((n, AppendFile name contents):es) state@(fs,(ics,ocs)) =
    (alist',state') where
          alist' = ack:alist
          ack = (n,
                case (fs name) of
                 Failure msg -> Failure (SearchError "file not found")
                 Str s -> Success
                 Bn  b -> Failure (FormatError "")
                )
          (alist,state') = os es (newfs, (ics,ocs))  where
                           newfs = case (fs name) of
                                    Failure msg -> fs
                                    Str s ->
                                     update fs name (Str (s++contents))
                                    Bn  b -> fs

os ((n, DeleteFile name):es) state@(fs,(ics,ocs)) =
    (alist',state') where
          alist' = ack : alist
          ack = (n,
                 case (fs name) of
                  Failure msg -> Failure (SearchError "file not found")
                  Str s -> Success
                  Bn b -> Success
                )
          (alist,state') = os es (case (fs name) of
                                    Failure msg -> fs
                                    Str s -> update fs name fail
                                    Bn b -> update fs name fail,
                                  (ics,ocs))
          fail = Failure (SearchError "file not found")

os ((n,StatusFile name):es) state@(fs,(ics,ocs)) = (alist',state') where
          alist' = ack : alist
          ack = (n,
                 case (fs name) of
                  Failure msg -> Failure (SearchError "File not found")
                  Str string -> Str "t"++(rw n fs name)
                  Bn binary -> Str "b"++(rw n fs name)
                )
          (alist, state') = os es state
```

where `rw` is a function that determines the read and write status of a file for this particular process.

C.1 Optional Requests

These optional I/O requests may be useful in a Haskell implementation.

- `ReadChannels [cname1, ..., cnamek]`
  `ReadBinChannels [cname1, ..., cnamek]`

  Opens `cname1` through `cnamek` for input. A successful response has form `Tag vals` [`BinTag vals`] where `vals` is a list of values tagged with the name of the channel. These responses require an extension to the `Response` datatype:

  ```
  data  Response = ...
                 | Tag    [(String,Char)]
                 | BinTag [(String,Bin)]
  ```

  The tagged list of values is the non-deterministic merge of the values read from the individual channels. If an element of this list has form `(cnamei,val)`, then it came from channel `cnamei`.

  If any `cnamei` does not exist then the response `Failure (SearchError string)` is induced; all other errors induce `Failure (ReadError string)`.

- `CreateProcess prog`

  Introduces a new program `prog` into the operating system. `prog` must have type `[Response] -> [Request]`. Either `Success` and `Failure (OtherError string)` is induced.

- `CreateDirectory name string`
  `DeleteDirectory name`

  Create or delete directory `name`. The `string` argument to `CreateDirectory` is an implementation-dependent specification of the initial state of the directory.

- ```
  OpenFile     name inout
  OpenBinFile  name inout
  CloseFile    file
  ReadVal      file
  ReadBinVal   file
  WriteVal     file char
  WriteBinVal  file bin
  ```

  These requests emulate traditional file I/O in which characters are read and written one at a time.

  ```
  data  Response = ...
                 | Fil File
  data  File
  type  Bins     = [Bin]
  ```

  `OpenFile name inout` [`OpenBinFile name inout`] opens the file `name` in text [binary] mode with direction `inout` (`True` for input, `False` for output). The response `Fil file`
  ```

is induced, where `file` has type `File`, a primitive type that represents a handle to a file. Subsequent use of that file by other requests is via this handle.

`CloseFile file` closes `file`. `Failure (OtherError string)` is induced if `file` cannot be closed.

`ReadVal` [`ReadBinVal`] `file` reads `file`, inducing the response `Str val` [`Bins val`] or `Failure (ReadError string)`.

`WriteVal file char` [`WriteBinVal file bin`] writes `char` [`bin`] to `file`. The response `Success` or `Failure (WriteError string)` is induced.

`Failure (SearchError string)` is induced for `ReadVal`, `ReadBinVal`, `WriteVal`, and `WriteBinVal` if `file` is not a text or binary file, as appropriate.

D Specification of Derived Instances

If T is an algebraic datatype declared by:

$$\texttt{data}\ c\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \texttt{=}\ K_1\ t_{11}\ \ldots\ t_{1k_1}\ \texttt{|}\ \cdots\ \texttt{|}\ K_n\ t_{n1}\ \ldots\ t_{nk_n}$$
$$\texttt{deriving}\ \texttt{(}C_1\texttt{,}\ \ldots\texttt{,}\ C_m\texttt{)}$$

(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 and only if these conditions hold:

1. C is one of `Eq`, `Ord`, `Enum`, `Ix`, `Text`, or `Binary`.
2. There is a context c' such that $c' \Rightarrow C\ t_{ij}$ holds for each of the constituent types t_{ij}.
3. If C is either `Ix` or `Enum`, then further constraints must be satisfied as described under the paragraphs for `Ix` and `Enum` later in this section.
4. There must be no explicit instance declaration elsewhere in the module which makes $T\ u_1\ \ldots\ u_k$ an instance of C.

If the `deriving` form is present (as in the above general `data` declaration), an instance declaration is automatically generated for $T\ u_1\ \ldots\ u_k$ over each class C_i and each of C_i's superclasses. If the derived instance declaration is impossible for any of the C_i then a static error results. If no derived instances are required, the `deriving` form may be omitted or the form `deriving ()` may be used.

Each derived instance declaration will have the form:

$$\texttt{instance}\ \texttt{(}c\texttt{,}\ C'_1\ u'_1\texttt{,}\ \ldots\texttt{,}\ C'_j\ u'_j\ \texttt{)}\ \texttt{=>}\ C_i\ (T\ u_1\ \ldots\ u_k)\ \texttt{where \{}\ d\ \texttt{\}}$$

where d is derived automatically depending on C_i and the data type declaration for T (as will be described in the remainder of this section), and u'_1 through u'_j form a subset of u_1 through u_k. When inferring the context for the derived instances, type synonyms must be expanded out first. The free variables of the declarations d are all functions defined in the standard prelude. The remaining details of the derived instances for each of the six classes are now given.

Derived instances of `Eq` and `Ord`. The operations automatically introduced by derived instances of `Eq` and `Ord` are `(==)`, `(/=)`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min`. The latter six 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 instances of `Ix`. The derived instance declarations for the class `Ix` introduce the overloaded functions `range`, `index`, and `inRange`. The operation `range` takes a (lower, upper) bound pair, and returns a list of all indices in this range, in ascending order. The operation `inRange` is a predicate taking a (lower, upper) bound pair and an index and returning `True` if the index is contained within the specified range. The operation `index` takes a (lower, upper) bound pair and an index and returns an integer, the position of the index within the range.

Derived instance declarations for the class `Ix` are only possible for enumerations (i.e. datatypes having only nullary constructors) and single-constructor datatypes (including tuples) whose constituent types are instances of `Ix`.

- For an *enumeration*, the nullary constructors are assumed to be numbered left-to-right with the indices 0 through $n-1$. 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 created as shown for tuples in Figure 15.

Derived instances of `Enum`. Derived instance declarations for the class `Enum` are only possible for enumerations, using the same ordering assumptions made for `Ix`. They introduce the operations `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo`, which are used to define arithmetic sequences as described in Section 3.8.

`enumFrom n` returns a list corresponding to the complete enumeration of `n`'s type starting at the value `n`. Similarly, `enumFromThen n n'` is the enumeration starting at `n`, but with second element `n'`, and with subsequent elements generated at a spacing equal to the difference between `n` and `n'`. `enumFromTo` and `enumFromThenTo` are as defined by the default methods for `Enum` (see Figure 4, page 30).

Derived instances of `Binary`. The `Binary` class is used primarily for transparent I/O (see Section 7.1). The operations automatically introduced by derived instances of `Binary` are `readBin` and `showBin`. They coerce values to and from the primitive abstract type `Bin` (see Section 6.7). An implementation must be able to create derived instances of `Binary` for any type t not containing a function type.

`showBin` is analogous to `shows`, taking two arguments: the first is the value to be coerced, and the second is a `Bin` value to which the result is to be concatenated. `readBin` is analogous to `reads`, "parsing" its argument and returning a pair consisting of the coerced value and any remaining `Bin` value.

```
class  (Ord a) => Ix a where
        range           :: (a,a) -> [a]
        index           :: (a,a) -> a -> Int
        inRange         :: (a,a) -> a -> Bool

rangeSize               :: (Ix a) => (a,a) -> Int
rangeSize (l,u)         =  index (l,u) u + 1

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
```

Figure 15: Index classes and instances

Derived versions of `showBin` and `readBin` must obey this property:

$$\texttt{readBin (showBin } v\ b\texttt{) == (}v\texttt{,}b\texttt{)}$$

for any `Bin` value b and value v whose type is an instance of the class `Binary`.

Derived instances of `Text`. The operations automatically introduced by derived instances of `Text` are `showsPrec`, `readsPrec`, `showList` and `readList`. They are used to coerce values into strings and parse strings into values.

The function `showsPrec d x r` accepts a precedence level `d` (a number from `0` to `10`), a value `x`, and a string `r`. It returns a string representing `x` concatenated to `r`. `showsPrec` satisfies the law:

```
showsPrec d x r ++ s  ==  showsPrec d x (r ++ s)
```

The representation will be enclosed in parentheses if the precedence of the top-level constructor operator in `x` is less than `d`. Thus, if `d` is `0` then the result is never surrounded in parentheses; if `d` is `10` it is always surrounded in parentheses, unless it is an atomic expression. The extra parameter `r` is essential if tree-like structures are to be printed in linear time rather than time quadratic in the size of the tree.

The function `readsPrec d s` accepts a precedence level `d` (a number from `0` to `10`) and a string `s`, and returns a list of pairs `(x,r)` such that `showsPrec d x r == s`. `readsPrec` is a parse function, returning a list of (parsed value, remaining string) pairs. If there is no successful parse, the returned list is empty.

`showList` and `readList` allow lists of objects to be represented using non-standard denotations. This is especially useful for strings (lists of `Char`).

For convenience, the standard prelude provides the following auxiliary functions:

```
shows    =  showsPrec 0
reads    =  readsPrec 0
show x   =  shows x ""
read s   =  x  where [(x,"")] = reads s
```

`shows` and `reads` use a default precedence of 0, and `show` and `read` assume that the result is not being appended to an initial string.

The instances of `Text` for the standard types `Int`, `Integer`, `Float`, `Double`, `Char`, lists, tuples, and rational and complex numbers are defined in the standard prelude (see Appendix A). For characters and strings, the control characters that have special representations (`\n` etc.) are shown as such by `showsPrec`; otherwise, ASCII mnemonics are used. Non-ASCII characters are shown by decimal escapes. Floating point numbers are represented by decimal numbers of sufficient precision to guarantee `read . show` is an identity function. If b is the floating-point radix and there are w base-b digits in the floating-point significand, the number of decimal digits required is $d = \lceil w \log_{10} b \rceil + 1$ [10]. Numbers are shown in non-exponential format if this requires only d digits; otherwise, they are shown in

```
showsPrec d (e1 `Con` e2) = showParen (d > p) showStr
  where
          p = `the precedence of Con'
          lp = if `Con is left associative' then p else p+1
          rp = if `Con is right associative' then p else p+1
          cn = `the original name of Con'

          showStr = showsPrec lp e1 .
                    showChar ' ' . showString cn . showChar ' ' .
                    showsPrec rp e2
```

Figure 16: Specification of `showsPrec` for Constructors Declared in the Infix Style

exponential format, with one digit before the decimal point. `readsPrec` allows an exponent to be unsigned or signed with `+` or `-`; `showsPrec` shows a positive exponent without a sign.

`readsPrec` will parse any valid representation of the standard types apart from lists, for which only the bracketed form `[...]` is accepted. See Appendix A for full details.

D.1 Specification of `showsPrec`

As described in Section 4.3.3, `showsPrec` has the type

```
(Text a) => Int -> a -> String -> String
```

The first parameter is a precedence in the range 0 to 10, the second is the value to be converted into a string, and the third is the string to append to the end of the result.

For all constructors `Con` defined by some `data` declaration such as:

`data` c `=>` $T\ u_1\ \ldots\ u_k$ `= ... | Con` $t_1\ \ldots\ t_n$ `| ...`

the corresponding definition of `showsPrec` for `Con` is shown in Figure 16 for constructors declared in the infix style and Figure 17 for all other constructors. See Appendix A for details of `showParen`, `showChar`, etc.

D.2 Specification of `readsPrec`

A *lexeme* is exactly as in Section 2. `lex :: String -> [(String,String)]` reads the first lexeme from a string. If the string begins with a valid lexeme, the lexeme (with leading whitespace removed) and the remainder of the string are returned in a singleton list. If no lexeme is present or the lexeme is not syntacticly correct, `[]` is returned. A full definition is provided in Appendix A.7.

```
showsPrec d (Con e1 ... en) = showParen (d >= 10) showStr
  where
          showStr = showString cn . showChar ' ' .
                    showsPrec 10 e1 . showChar ' ' .
                    ...
                    showsPrec 10 en
          cn = 'the original name of Con'
```

Figure 17: General Specification of `showsPrec` for User-Defined Constructors

As described in Section 4.3.3, `readsPrec` has the type

`Text a => Int -> String -> [(a,String)]`

Its first parameter is a precedence in the range 0 to 10, its second is the string to be parsed. Figure 18 shows the specification of `readsPrec` for user-defined datatypes of the form:

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

D.3 An example

As a complete example, consider a tree datatype:

```
data Tree a = Leaf a | Tree a :^: Tree a
     deriving (Eq, Ord, Text, Binary)
instance (Eq a) => Eq (Tree a)
  where ...
instance (Ord a) => Ord (Tree a)
  where ...
instance (Text a) => Text (Tree a)
  where ...
instance (Binary a) => Binary (Tree a)
  where ...
```

Note the recursive context; the components of the datatype must themselves be instances of the class. Automatic derivation of instance declarations for `Ix` and `Enum` are not possible, as `Tree` is not an enumeration or single-constructor datatype. Except for `Binary`, the complete instance declarations for `Tree` are shown in Figure 19, Note the implicit use of default-method definitions—for example, only `<=` is defined for `Ord`, with the other operations (`<`, `>`, `>=`, `max`, and `min`) being defined by the defaults given in the class declaration shown in Figure 4 (page 30).

```
readsPrec d r = readCon K1 k1 ‘the original name of K1’ r ++
           ...
           readCon Kn kn ‘the original name of Kn’ r
  where
    readCon con n cn =                  -- if con is infix
        readParen (d > p) readVal
      where
                readVal r =  [(u ‘con‘ v, s2) |
                              (u,s0)   <- readsPrec lp r,
                              (tok,s1) <- lex s0, tok == cn,
                              (v,s2)   <- readsPrec rp s1]
                p = ‘the precedence of con’
                lp = if ‘con is left associative’ then p else p+1
                rp = if ‘con is right associative’ then p else p+1

    readCon con n cn =                  -- if con is not infix
        readParen (d > 9) readVal
      where
                 readVal r = [(con t1 ... tn, sn) |
                              (t0,s0) <- lex r, t0 == cn,
                              (t1,s1) <- readsPrec 10 s0,
                              ...
                              (tn,sn) <- readsPrec 10 s(n-1)]
```

Figure 18: Definition of `readsPrec` for User-Defined Types

```
infix 4 :^:
data Tree a =  Leaf a  |  Tree a :^: Tree a

instance (Eq a) => Eq (Tree a) where
        Leaf m == Leaf n  =  m==n
        u:^:v  == x:^:y   =  u==x && v==y
             _ == _       =  False

instance (Ord a) => Ord (Tree a) where
        Leaf m <= Leaf n  =  m<=n
        Leaf m <= x:^:y   =  True
        u:^:v  <= Leaf n  =  False
        u:^:v  <= x:^:y   =  u<x || u==x && v<=y

instance (Text a) => Text (Tree a) where

        showsPrec d (Leaf m) = showParen (d >= 10) showStr
          where
             showStr = showString "Leaf" . showChar ' ' . showsPrec 10 m

        showsPrec d (u :^: v) = showParen (d > 4) showStr
          where
             showStr = showsPrec 5 u .
                       showChar ' ' . showString ":^:" . showChar ' ' .
                       showsPrec 5 v

        readsPrec d r =  readParen (d > 4)
                         (\r -> [(u:^:v,w) |
                                 (u,s) <- readsPrec 5 r,
                                 (":^:",t) <- lex s,
                                 (v,w) <- readsPrec 5 t]) r

                      ++ readParen (d > 9)
                         (\r -> [(Leaf m,t) |
                                 ("Leaf",t) <- lex r,
                                 (m,t) <- readsPrec 10 t]) r
```

Figure 19: Example of Derived Instances

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[2] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. HOPE: An experimental applicative language. In *The 1980 LISP Conference*, pages 136–143, Stanford University, August 1980.

[3] H.B. Curry and R. Feys. *Combinatory Logic*. North-Holland Pub. Co., Amsterdam, 1958.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, N.M., January 1982.

[5] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings of 5th ACM Symposium on Principles of Programming Languages*, pages 119–130, 1978.

[6] K. Hammond and C. Hall. A natural dynamic semantics for Haskell (draft). Department of Computing Science, Glasgow University, February 1991.

[7] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[8] P. Hudak and R. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR665, Yale University, Department of Computer Science, December 1988.

[9] P.J. Landin. The next 700 programming languages. *CACM*, 9(3):157–166, March 1966.

[10] D.W. Matula. A formalization of floating-point numeric base conversion. *IEEE Transactions on Computers*, C-19(8):681–692, August 1970.

[11] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *CACM*, 3(4):184–195, April 1960.

[12] R.S. Nikhil. Id-Nouveau (version 88.0) reference manual. Technical report, MIT Laboratory for Computer Science, Cambridge, Mass., March 1988.

[13] P. Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings*, pages 248–256, San Francisco, September 1981.

[14] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

[15] S.L. Peyton Jones and P. Wadler. A static semantics for HASKELL. Department of Computing Science, Glasgow University, May 1991.

[16] J. Rees and W. Clinger (eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[17] J.C. Reynolds. Introduction to part II (polymorphic lambda calculus). In G. Huet, editor, *Logical Foundations of Functional Programming*, University of Texas Year of Programming Series. Addison-Wesley, 1990.

[18] G.L. Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, Mass., 1984.

[19] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 1985. Springer-Verlag.

[20] P. Wadler. A new array operation. In J.H. Fasel and R.M. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 328–335, Heidelberg, 1987. Springer-Verlag.

[21] 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 defined in the standard prelude (Appendix A) are shown in `typewriter` font. Ordinary index entries are shown in a roman font.