

Contents

I	The Haskell 98 Language	1
1	Introduction	3
1.1	Program Structure	3
1.2	The Haskell Kernel	4
1.3	Values and Types	4
1.4	Namespaces	5
2	Lexical Structure	7
2.1	Notational Conventions	7
2.2	Lexical Program Structure	8
2.3	Comments	9
2.4	Identifiers and Operators	10
2.5	Numeric Literals	11
2.6	Character and String Literals	12
2.7	Layout	13
3	Expressions	17
3.1	Errors	19
3.2	Variables, Constructors, Operators, and Literals	20
3.3	Curried Applications and Lambda Abstractions	21
3.4	Operator Applications	21
3.5	Sections	22
3.6	Conditionals	23
3.7	Lists	23
3.8	Tuples	24
3.9	Unit Expressions and Parenthesized Expressions	25
3.10	Arithmetic Sequences	25
3.11	List Comprehensions	25
3.12	Let Expressions	27
3.13	Case Expressions	27

3.14	Do Expressions	29
3.15	Datatypes with Field Labels	29
3.15.1	Field Selection	30
3.15.2	Construction Using Field Labels	30
3.15.3	Updates Using Field Labels	31
3.16	Expression Type-Signatures	32
3.17	Pattern Matching	32
3.17.1	Patterns	32
3.17.2	Informal Semantics of Pattern Matching	34
3.17.3	Formal Semantics of Pattern Matching	36
4	Declarations and Bindings	39
4.1	Overview of Types and Classes	40
4.1.1	Kinds	41
4.1.2	Syntax of Types	41
4.1.3	Syntax of Class Assertions and Contexts	43
4.1.4	Semantics of Types and Classes	44
4.2	User-Defined Datatypes	45
4.2.1	Algebraic Datatype Declarations	45
4.2.2	Type Synonym Declarations	47
4.2.3	Datatype Renamings	48
4.3	Type Classes and Overloading	49
4.3.1	Class Declarations	49
4.3.2	Instance Declarations	51
4.3.3	Derived Instances	53
4.3.4	Ambiguous Types, and Defaults for Overloaded Numeric Operations	53
4.4	Nested Declarations	55
4.4.1	Type Signatures	55
4.4.2	Fixity Declarations	56
4.4.3	Function and Pattern Bindings	58
4.4.3.1	Function bindings.	58
4.4.3.2	Pattern bindings.	59
4.5	Static Semantics of Function and Pattern Bindings	60
4.5.1	Dependency Analysis	60
4.5.2	Generalization	61
4.5.3	Context Reduction Errors	61
4.5.4	Monomorphism	62
4.5.5	The Monomorphism Restriction	63
4.6	Kind Inference	66
5	Modules	67
5.1	Module Structure	68
5.2	Export Lists	69
5.3	Import Declarations	71
5.3.1	What is Imported	72

5.3.2	Qualified Import	72
5.3.3	Local Aliases	73
5.3.4	Examples	73
5.4	Importing and Exporting Instance Declarations	74
5.5	Name Clashes and Closure	74
5.5.1	Qualified Names	74
5.5.2	Name Clashes	75
5.5.3	Closure	76
5.6	Standard Prelude	77
5.6.1	The <code>PreLude</code> Module	77
5.6.2	Shadowing Prelude Names	77
5.7	Separate Compilation	78
5.8	Abstract Datatypes	78
6	Predefined Types and Classes	81
6.1	Standard Haskell Types	81
6.1.1	Booleans	81
6.1.2	Characters and Strings	82
6.1.3	Lists	82
6.1.4	Tuples	82
6.1.5	The Unit Datatype	83
6.1.6	Function Types	83
6.1.7	The IO and IOError Types	83
6.1.8	Other Types	83
6.2	Strict Evaluation	84
6.3	Standard Haskell Classes	84
6.3.1	The Eq Class	86
6.3.2	The Ord Class	86
6.3.3	The Read and Show Classes	87
6.3.4	The Enum Class	88
6.3.5	The Functor Class	89
6.3.6	The Monad Class	90
6.3.7	The Bounded Class	91
6.4	Numbers	91
6.4.1	Numeric Literals	92
6.4.2	Arithmetic and Number-Theoretic Operations	92
6.4.3	Exponentiation and Logarithms	93
6.4.4	Magnitude and Sign	94
6.4.5	Trigonometric Functions	95
6.4.6	Coercions and Component Extraction	95
7	Basic Input/Output	97
7.1	Standard I/O Functions	97
7.2	Sequencing I/O Operations	99
7.3	Exception Handling in the I/O Monad	100

8	Standard Prelude	103
8.1	Module Prelude	104
8.2	Module PreludeList	114
8.3	Module PreludeText	119
8.4	Module PreludeIO	123
9	Syntax Reference	125
9.1	Notational Conventions	125
9.2	Lexical Syntax	126
9.3	Layout	128
9.4	Literate Comments	131
9.5	Context-Free Syntax	133
10	Specification of Derived Instances	139
10.1	Derived Instances of Eq and Ord	140
10.2	Derived Instances of Enum	140
10.3	Derived Instances of Bounded	141
10.4	Derived Instances of Read and Show	142
10.5	An Example	143
11	Compiler Pragmas	145
11.1	Inlining	145
11.2	Specialization	145
II	The Haskell 98 Libraries	147
12	Rational Numbers	149
12.1	Library Ratio	150
13	Complex Numbers	153
13.1	Library Complex	154
14	Numeric Functions	157
14.1	Showing Functions	158
14.2	Reading Functions	159
14.3	Miscellaneous	159
14.4	Library Numeric	160
15	Indexing Operations	169
15.1	Deriving Instances of Ix	170
15.2	Library Ix	171
16	Arrays	173
16.1	Array Construction	174
16.1.1	Accumulated Arrays	175
16.2	Incremental Array Updates	175

16.3	Derived Arrays	176
16.4	Library Array	176
17	List Utilities	179
17.1	Indexing Lists	181
17.2	“Set” Operations	181
17.3	List Transformations	182
17.4	unfoldr	182
17.5	Predicates	183
17.6	The “By” Operations	183
17.7	The “generic” Operations	184
17.8	Further “zip” Operations	184
17.9	Library List	184
18	Maybe Utilities	191
18.1	Library Maybe	192
19	Character Utilities	193
19.1	Library Char	195
20	Monad Utilities	199
20.1	Naming Conventions	200
20.2	Class MonadPlus	200
20.3	Functions	201
20.4	Library Monad	202
21	Input/Output	205
21.1	I/O Errors	207
21.2	Files and Handles	208
21.2.1	Standard Handles	209
21.2.2	Semi-Closed Handles	209
21.2.3	File Locking	210
21.3	Opening and Closing Files	210
21.3.1	Opening Files	210
21.3.2	Closing Files	210
21.4	Determining the Size of a File	211
21.5	Detecting the End of Input	211
21.6	Buffering Operations	211
21.6.1	Flushing Buffers	212
21.7	Repositioning Handles	213
21.7.1	Revisiting an I/O Position	213
21.7.2	Seeking to a New Position	213
21.8	Handle Properties	213
21.9	Text Input and Output	214
21.9.1	Checking for Input	214
21.9.2	Reading Input	214

21.9.3 Reading Ahead	214
21.9.4 Reading the Entire Input	214
21.9.5 Text Output	215
21.10 Examples	215
21.10.1 Summing Two Numbers	215
21.10.2 Copying Files	216
21.11 Library IO \square	216
22 Directory Functions	219
23 System Functions	223
24 Dates and Times	225
24.1 Library Time	227
25 Locales	231
25.1 Library Locale	231
26 CPU Time	233
27 Random Numbers	235
27.1 The RandomGen class, and the StdGen generator	236
27.2 The Random class	239
27.3 The global random number generator	240
Bibliography	241
Index	243

Preface

“Some half dozen persons have written technically on combinatory logic, and most of these, including ourselves, have published something erroneous. Since some of our fellow sinners are among the most careful and competent logicians on the contemporary scene, we regard this as evidence that the subject is refractory. Thus fullness of exposition is necessary for accuracy; and excessive condensation would be false economy here, even more than it is ordinarily.”

Haskell B. Curry and Robert Feys
in the Preface to *Combinatory Logic* [4], May 31 1956

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

Haskell 98: language and libraries

The committee intended that Haskell would serve as a basis for future research in language design, and hoped that extensions or variants of the language would appear, incorporating experimental features. Haskell has indeed evolved continuously since its original publication. By the middle of 1997, there had been four iterations of the language design (the latest at that point being Haskell 1.4). At the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed; this stable language is the subject of this book, and is called *Haskell 98*.

Haskell 98 was conceived as a relatively minor tidy-up of Haskell 1.4, making some simplifications, and removing some pitfalls for the unwary. It is intended to be a “stable” language in sense the *implementors are committed to supporting Haskell 98 exactly as specified, for the foreseeable future*.

The original Haskell Report covered only the language, together with a standard library called the `Prelude`. By the time Haskell 98 was stabilised, it had become clear that many programs need access to a larger set of library functions (notably concerning input/output and simple interaction with the operating system). If these programs were to be portable, a set of libraries would have to be standardised too. A separate effort was therefore begun by a distinct (but overlapping) committee to fix the Haskell 98 Libraries.

The Haskell 98 Language and Library Reports were published in February 1999.

Revising the Haskell 98 Reports

After a year or two, many typographical errors and infelicities had been spotted. I took on the role of gathering and acting on these corrections, with the following goals:

- Correct typographical errors.
- Clarify obscure passages.
- Resolve ambiguities.
- With reluctance, make small changes to make the overall language more consistent.

This task turned out to be much, much larger than I had anticipated. As Haskell becomes more widely used, the Report has been scrutinised by more and more people, and I have adopted hundreds of (mostly small) changes as a result of their feedback. The original committees ceased to exist when the original Haskell 98 Reports were published, so every change was instead proposed to the entire Haskell mailing list.

This book is the outcome of this process of refinement. It includes both the Haskell 98 Language Report and the Libraries Report, and constitutes the official specification of both. It is *not* a tutorial on programming in Haskell such as the “Gentle Introduction” [9], and some familiarity with functional languages is assumed.

The entire text of both Reports is available online (see ‘Haskell Resources’ on p. x).

Extensions to Haskell 98

Haskell continues to evolve, going well beyond Haskell 98. For example, at the time of writing there are Haskell implementations that support:

- **Syntactic sugar**, including:
 - pattern guards;
 - recursive do-notation;
 - lexically scoped type variables;
 - meta-programming facilities;
- **Type system innovations**, including:
 - multi-parameter type classes;
 - functional dependencies;
 - existential types;
 - local universal polymorphism and arbitrary rank-types;
- **Control extensions**, including:
 - monadic state;
 - exceptions;

- concurrency.

There is more besides. Haskell 98 does not impede these developments. Instead, it provides a stable point of reference, so that those who wish to write text books, or use Haskell for teaching, can do so in the knowledge that Haskell 98 will continue to exist.

Haskell Resources

The Haskell web site

`http://haskell.org`

gives access to many useful resources, including:

- Online versions of the language and library definitions, including a complete list of all the differences between Haskell 98 as published in February 1999 and this revised version.
- Tutorial material on Haskell.
- Details of the Haskell mailing list.
- Implementations of Haskell.
- Contributed Haskell tools and libraries.
- Applications of Haskell.

We welcome your comments, suggestions, and criticisms on the language or its presentation in the report, via the Haskell mailing list.

Building the language

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

Arvind (MIT)
Lennart Augustsson (Chalmers University)
Dave Barton (Mitre Corp)
Brian Boutel (Victoria University of Wellington)
Warren Burton (Simon Fraser University)
Jon Fairbairn (University of Cambridge)
Joseph Fasel (Los Alamos National Laboratory)
Andy Gordon (University of Cambridge)

Maria Guzman (Yale University)
 Kevin Hammond (University of Glasgow)
 Ralf Hinze University of Bonn
 Paul Hudak [editor] (Yale University)
 John Hughes [editor] (University of Glasgow; Chalmers University)
 Thomas Johnsson (Chalmers University)
 Mark Jones (Nottingham University)
 Dick Kieburtz (Oregon Graduate Institute)
 John Launchbury (University of Glasgow; Oregon Graduate Institute)
 Erik Meijer (Utrecht University)
 Rishiyur Nikhil (MIT)
 John Peterson (Yale University)
 Simon Peyton Jones [editor] (University of Glasgow; Microsoft Research Ltd)
 Mike Reeve (Imperial College)
 Alastair Reid (University of Glasgow)
 Colin Runciman (University of York)
 Philip Wadler [editor] (University of Glasgow)
 David Wise (Indiana University)
 Jonathan Young (Yale University)

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

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

Finally, aside from the important foundational work laid by Church, Rosser, Curry, and others on the lambda calculus, we wish to acknowledge the influence of many noteworthy programming

languages developed over the years. Although it is difficult to pinpoint the origin of many ideas, we particularly wish to acknowledge the influence of Lisp (and its modern-day incarnations Common Lisp and Scheme); Landin's ISWIM; APL; Backus's FP [1]; ML and Standard ML; Hope and Hope⁺; Clean; Id; Gofer; Sisal; and Turner's series of languages culminating in Miranda.¹ Without these forerunners Haskell would not have been possible.

Simon Peyton Jones
Cambridge, November 2002

¹Miranda is a trademark of Research Software Ltd.

Part I

The Haskell 98 Language

Chapter 1

Introduction

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

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

1.1 Program Structure

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

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

3. At the next lower level are *expressions*, described in Chapter 3. An expression denotes a *value* and has a *static type*; expressions are at the heart of Haskell programming “in the small.”
4. At the bottom level is Haskell’s *lexical structure*, defined in Chapter 2. The lexical structure captures the concrete representation of Haskell programs in text files.

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

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

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

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

“Holes” in program fragments representing arbitrary pieces of Haskell code are written in italics, as in `if e1 then e2 else e3`. Generally, the italicized names are mnemonic, such as *e* for expressions, *d* for declarations, *t* for types, etc.

1.2 The Haskell Kernel

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

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

1.3 Values and Types

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

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

1.4 Namespaces

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

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

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

