Язык и библиотеки Haskell 98
Исправленное описание

Саймон Пейтон Джонс (редактор)

Оглавление

Предисловие

> "*Около полдюжины человек формально написали комбинаторную логику, и большинство из них, включая нас, допустили ошибки. Поскольку некоторые из наших приятелей-грешников относятся к наиболее аккуратным и компетентным логикам современности, мы рассматриваем данный факт как доказательство того, что ошибок трудно избежать. Таким образом, полнота описания необходима для точности, и чрезмерная краткость была бы здесь ложной экономией, даже больше, чем обычно.*"
>
> Хаскелл Б. Карри (Haskell B. Curry) и Роберт Фейс (Robert Feys)
> в предисловии к *Комбинаторной Логике* [2], 31 мая 1956

В сентябре 1987 на конференции "Языки функционального программирования и компьютерная архитектура" (FPCA '87) в Портленде, штат Орегон, было организовано заседание для того, чтобы обсудить плачевную ситуацию, сложившуюся в сообществе функционального программирования: возникло более дюжины нестрогих чисто функциональных языков программирования, сходных в выразительной мощности и семантических основах. Участники встречи пришли к твердому мнению, что более широкому использованию этого класса функциональных языков препятствовало отсутствие общего языка. Было решено, что должен быть сформирован комитет для разработки такого языка, который обеспечил бы более быстрое средство связи новых идей, устойчивой основы для разработки реальных приложений и механизма поощрения других людей использовать функциональные языки. Этот документ описывает результат усилий этого комитета — чисто функциональный язык программирования, названный Haskell, в честь логика Хаскелла Б. Карри (Haskell B. Curry), чей труд обеспечивает логическую основу для многих наших работ.

Цели

Основная цель комитета заключалась в разработке языка, который удовлетворял бы следующим требованиям:

1. Язык должен быть пригоден для обучения, исследований и приложений, включая построение больших систем.
2. Язык должен полностью описываться с помощью формальных синтаксиса и семантики.
3. Язык должен находиться в свободном доступе. Следует разрешить свободную реализацию и распространение языка.

4. Язык должен базироваться на идеях, которые получат широкое одобрение.
5. Язык должен сократить излишнее многообразие языков функционального программирования.

Haskell 98: язык и библиотеки

Комитет планировал, что Haskell послужит основой для будущего исследования в разработке языков, и выражал надежду, что появятся расширения или варианты языка, включая экспериментальные функциональные возможности.

Haskell действительно непрерывно развивался с тех пор, как было опубликовано его первоначальное описание. К середине 1997 было выполнено четыре итерации в разработке языка (самая последняя на тот момент — Haskell 1.4). В 1997 на семинаре по Haskell в Амстердаме было решено, что необходим стабильный вариант Haskell, этот стабильный вариант языка является темой настоящего описания и называется "Haskell 98".

Haskell 98 задумывался как относительно незначительная модификация Haskell 1.4, выполненная за счет некоторых упрощений и удаления некоторых подводных камней. Он предназначен играть роль "стабильного" языка в том смысле, что *разработчики выполняют поддержку Haskell 98 в точности с его спецификацией, для обеспечения предсказуемости в будущем.*

Первоначальное описание Haskell описывало только язык и стандартную библиотеку, названную `Prelude`. К тому времени, когда Haskell 98 был признан стабильной версией, стало ясно, что многим программам необходим доступ к большому набору библиотечных функций (особенно это касается ввода - вывода и взаимодействия с операционной системой). Если эта программа должна быть переносимой, набор библиотек также должен быть стандартизирован. Поэтому отдельный комитет (в который входили некоторые из членов комитета по разработке языка) начал вносить исправления в библиотеки Haskell 98.

Описания языка Haskell 98 и библиотек были опубликованы в феврале 1999.

Внесение исправлений в описание Haskell 98

Через год или два было выявлено много типографских ошибок и погрешностей. Я взялся собирать и выполнять эти исправления со следующими целями:

- Исправить типографские ошибки.
- Разъяснить непонятные переходы.

- Разрешить неоднозначности.
- С неохотой сделать небольшие изменения, чтобы язык стал более последовательным.

Эта задача, оказалось намного, намного более масштабной, чем я ожидал. Поскольку Haskell все более широко использовался, его описание внимательно изучалось все большим и большим количеством людей, и я внес сотни (главным образом небольших) поправок в результате этой обратной связи. Первоначальные комитеты прекратили свое существование, когда было опубликовано первое описание Haskell 98, поэтому каждое изменение вместо этого вносилось на рассмотрение всего списка адресатов Haskell.

Этот документ является результатом описанного процесса усовершенствования. Он включает и описание языка Haskell 98, и описание библиотек и составляет официальную спецификацию обоих. Он *не* является учебным пособием по программированию на Haskell, как краткий вводный курс "Gentle Introduction" [6], и предполагает некоторое знакомство читателя с функциональными языками.

Полный текст обоих описаний доступен в режиме онлайн (см. ниже "Ресурсы Haskell").

Расширения Haskell 98

Haskell продолжает развиваться и сильно продвинулся дальше Haskell 98. Например, на момент написания этого документа имеются реализации Haskell, которые поддерживают:

Синтаксические возможности, включая:

- стражи образцов;
- рекурсивную do-нотацию;
- лексически ограниченные переменные типа;
- возможности для метапрограммирования.

Новации в системе типов, включая:

- классы типов, использующие множество параметров;
- функциональные зависимости;
- экзистенциальные типы;
- локальный универсальный полиморфизм и типы произвольного ранга.

Расширения управления, включая:

- монадическое состояние;

- исключения;
- параллелизм

и многое другое. Haskell 98 не препятствует этим разработкам. Вместо этого, он обеспечивает стабильную контрольную точку, чтобы те, кто хочет писать учебники или использовать Haskell для обучения, мог осуществить задуманное, зная, что Haskell 98 продолжит существование.

Ресурсы Haskell

Web-сайт Haskell

```
http://haskell.org
```

предоставляет доступ ко многим полезным ресурсам, включая:

- Online-версии определений языка и библиотек, включая полный список всех различий между Haskell 98, выпущенным в феврале 1999, и этой исправленной версией.
- Обучающий материал по Haskell.
- Детали рассылки Haskell.
- Реализации Haskell.
- Дополнительные средства и библиотеки Haskell.
- Приложения Haskell.

Приглашаем Вас комментировать, предлагать усовершенствования и критиковать язык или его представление в описании посредством рассылки Haskell.

Построение языка

Haskell создан и продолжает поддерживаться активным сообществом исследователей и прикладных программистов. Те, кто входил в состав комитетов по языку и библиотекам, в частности, посвятили огромное количество времени и энергии языку. Их имена, а также присоединившиеся к ним на тот период организации перечислены ниже:

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

Те, кто помечены [редактор], работали в качестве координирующих редакторов одной или более ревизий языка.

Кроме того, множество других людей внесли свой вклад, некоторые — небольшой, но многие — существенный. Это следующие люди: 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, Patrik Jansson, Robert Jeschofnik, Orjan Johansen, Simon B. Jones, Stef Joosten, Mike Joy, Stefan Kahrs, Antti-Juhani Kaijanaho, Jerzy Karczmarczuk, Wolfram Kahl, 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, Gary Memovich, Randy Michelsen, Rick

Mohr, Andy Moran, Graeme Moss, Henrik Nilsson, Arthur Norman, Nick North, Chris Okasaki, Bjarte M. Østvold, Paul Otto, Sven Panne, Dave Parrott, Ross Paterson, Larne Pekowsky, Rinus Plasmeijer, Ian Poole, Stephen Price, John Robson, Andreas Rossberg, George Russell, Patrick Sansom, Michael Schneider, Felix Schroeter, Julian Seward, Nimish Shah, Christian Sievers, Libor Skarvada, Jan Skibinski, Lauren Smith, Raman Sundaresh, Josef Svenningsson, Ken Takusagawa, 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 и Bonnie Yantis.

Наконец, кроме важной основополагающей работы Россера (Rosser), Карри (Curry) и других, положенной в основу лямбда-исчисления, будет правильным признать влияние многих заслуживающих внимания языков программирования, разработанных за эти годы. Хотя трудно точно определить происхождение многих идей, следующие языки были особенно важны: Lisp (и его современные воплощения Common Lisp и Scheme), ISWIM Ландина (Landin), APL, FP[1] Бэкуса (Backus), ML и Standard ML, Hope и Hope $^+$, Clean, Id, Gofer, Sisal и ряд языков Тернера (Turner), завершившиеся созданием Miranda [1]. Без этих предшественников Haskell был бы невозможен.

Саймон Пейтон Джонс (Simon Peyton Jones)
Кембридж, сентябрь 2002

[1] Miranda является торговой маркой Research Software Ltd.

Часть I

Язык Haskell 98

Глава 1

Введение

Haskell является чисто функциональным языком программирования общего назначения, который включает много последних инноваций в разработке языков программирования. Haskell обеспечивает функции высокого порядка, нестрогую семантику, статическую полиморфную типизацию, определяемые пользователем алгебраические типы данных, сопоставление с образцом, описание списков, модульную систему, монадическую систему ввода - вывода и богатый набор примитивных типов данных, включая списки, массивы, целые числа произвольной и фиксированной точности и числа с плавающей точкой. Haskell является и кульминацией, и кристаллизацией многих лет исследования нестрогих функциональных языков.

Это описание определяет синтаксис программ на Haskell и неформальную абстрактную семантику для понимания смысла таких программ. Мы не рассматриваем способы, которыми программы на Haskell управляются, интерпретируются, компилируются и т.д., поскольку они зависят от реализации, включая такие вопросы, как характер сред программирования и сообщения об ошибках, возвращаемые для неопределенных программ (т.е. программ, формальное вычисление которых приводит к $\perp$).

1.1 Структура программы

В этом разделе мы описываем абстрактную синтаксическую и семантическую структуру Haskell, а также то, как она соотносится с организацией остальной части описания.

1. Самый верхний уровень программы на Haskell представляет собой набор *модулей*, описанных в главе 5. Модули предоставляют средство управления пространствами имен и повторного использования программного обеспечения в больших программах.

2. Верхний уровень модуля состоит из совокупности *объявлений*, которых существует несколько видов, все они описаны в главе 4. Объявления определяют такие сущности, как обычные значения, типы данных, классы типов, ассоциативность и приоритеты операторов.

3. На следующем, более низком, уровне находятся *выражения*, описанные в главе 3. Выражение обозначает *значение* и имеет *статический тип*; выражения лежат в основе программирования на Haskell "в малом".

4. На нижнем уровне находится *лексическая структура* Haskell, определенная в главе 2. Лексическая структура охватывает конкретное представление программ на Haskell в текстовых файлах.

Данное описание направлено снизу вверх по отношению к синтаксической структуре Haskell.

Главы, которые не упомянуты выше, — это глава 6, которая описывает стандартные встроенные типы данных и классы в Haskell, и глава 7, в которой рассматривается средство ввода - вывода в Haskell (т.е. как программы на Haskell связываются со внешним миром). Также есть несколько глав, описывающих Prelude, конкретный синтаксис, грамотное программирование, подробное описание производных экземпляров и псевдокомментарии, поддерживаемые большинством компиляторов Haskell.

Примеры фрагментов программ на Haskell в данном тексте даны в машинописном шрифте:

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

"Дыры" во фрагментах программ, представляющие собой произвольные части кода на Haskell, написаны в курсиве, как, например, в `if` e_1 `then` e_2 `else` e_3. Вообще курсивные имена являются мнемоническими, например, e — для выражений (expressions), d — для объявлений (declarations), t — для типов (types) и т.д.

1.2 Ядро Haskell

Haskell заимствовал многие из удобных синтаксических структур, которые стали популярными в функциональном программировании. В этом описании значение такого синтаксического средства дается трансляцией в более простые конструкции. Если эти трансляции полностью применимы, результатом является программа, записанная в небольшом подмножестве Haskell, которое мы называем *ядром* Haskell.

Хотя ядро формально не определено, это по существу слегка смягченный вариант лямбда-исчисления с прямо обозначенной семантикой. Трансляция каждой

синтаксической структуры в ядро дается, когда вводится синтаксис. Эта модульная конструкция облегчает объяснение программ на Haskell и предоставляет полезные рекомендации для разработчиков реализаций языка.

1.3 Значения и типы

Результатом вычисления выражения является *значение*. Выражение имеет статический *тип*. Значения и типы не смешаны в Haskell. Тем не менее, система типов допускает определяемые пользователем типы данных различных видов и разрешает не только параметрический полиморфизм (используя традиционную структуру типов Хиндли-Милнера (Hindley-Milner)), но также *специальный* полиморфизм, или *перегрузку* (используя *классы типов*).

Ошибки в Haskell семантически эквивалентны $\perp$. С формальной точки зрения они не отличимы от незавершенного вычисления, поэтому язык не содержит механизма обнаружения или реагирования на ошибки. Тем не менее, реализации языка вероятно будут пытаться предоставить полезную информацию об ошибках (см. раздел 3.1).

1.4 Пространства имен

Есть шесть видов имен в Haskell: имена *переменных* и *конструкторов* обозначают значения; имена *переменных типов*, *конструкторов типов* и *классов типов* ссылаются на сущности, относящиеся к системе типов; *имена модулей* ссылаются на модули. Есть два ограничения на присваивание имен:

1. Именами переменных и переменных типов являются идентификаторы, которые начинаются со строчных букв или символа подчеркивания; остальные четыре вида имен являются идентификаторами, которые начинаются с заглавных букв.
2. Идентификатор нельзя использовать в качестве имени конструктора типа и класса в одной и той же области видимости.

Это единственные ограничения; например, `Int` может одновременно являться именем модуля, класса и конструктора в пределах одной области видимости.

Глава 2

Лексическая структура

В этой главе мы опишем лексическую структуру нижнего уровня языка Haskell. Большинство деталей может быть пропущено при первом прочтении этого описания.

2.1 Соглашения об обозначениях

Эти соглашения об обозначениях используются для представления синтаксиса:

$[pattern]$	необязательный
$\{pattern\}$	ноль или более повторений
$(pattern)$	группировка
$pat_1 \mid pat_2$	выбор
$pat_{\langle pat' \rangle}$	разность — элементы, порождаемые с помощью pat, за исключением элементов, порождаемых с помощью pat'
`fibonacci`	терминальный синтаксис в машинописном шрифте

Поскольку синтаксис в этом разделе описывает *лексический* синтаксис, все пробельные символы выражены явно; нет никаких неявных пробелов между смежными символами. Повсюду используется BNF-подобный синтаксис, чьи правила вывода имеют вид:

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

Перевод:
нетерминал →
  альтернатива$_1$
  | *альтернатива*$_2$
  | ...
  | *альтернатива*$_n$

Необходимо внимательно отнестись к отличию синтаксиса металогических символов, например, | и [...], от конкретного синтаксиса терминалов (данных в машинописном шрифте), например, `|` и `[...]`, хотя обычно это ясно из контекста.

Haskell использует кодировку символов Unicode [11]. Тем не менее, исходные программы в настоящее время написаны в основном в кодировке символов ASCII , используемой в более ранних версиях Haskell.

Этот синтаксис зависит от свойств символов Unicode, определяемых консорциумом Unicode. Ожидается, что компиляторы Haskell будут использовать новые версии Unicode, когда они станут доступными.

2.2 Лексическая структура программы

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

whitespace → *whitestuff* {*whitestuff*}
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *vertab* | *space* | *tab* | *uniWhite*
newline → *return linefeed* | *return* | *linefeed* | *formfeed*
return → возврат каретки
linefeed → перевод строки
vertab → вертикальная табуляция
formfeed → перевод страницы
space → пробел
tab → горизонтальная табуляция
uniWhite → любой пробельный символ Unicode

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

small → *ascSmall* | *uniSmall* | `_`
ascSmall → `a` | `b` | ... | `z`

uniSmall → любая буква Unicode нижнего регистра

large → *ascLarge* | *uniLarge*
ascLarge → `A` | `B` | ... | `Z`
uniLarge → любая буква Unicode верхнего регистра или заглавная
symbol → *ascSymbol* | *uniSymbol*$\langle$*special* | `_` | `:` | `"` | `'`$\rangle$

ascSymbol → `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | @
| `\` | `^` | `|` | `-` | `~`
uniSymbol → любой символ или знак пунктуации Unicode
digit → *ascDigit* | *uniDigit*
ascDigit → `0` | `1` | ... | `9`
uniDigit → любая десятичная цифра Unicode
octit → `0` | `1` | ... | `7`
hexit → *digit* | `A` | ... | `F` | `a` | ... | `f`

Перевод:
программа →
{ *лексема* | *пробельная-строка* }
лексема →
квалифицированный-идентификатор-переменной
| *квалифицированный-идентификатор-конструктора*
| *квалифицированный-символ-переменной*
| *квалифицированный-символ-конструктора*
| *литерал*
| *специальная-лексема*
| *зарезервированный-оператор*
| *зарезервированный-идентификатор*
литерал →
целый-литерал
| *литерал-с-плавающей-точкой*
| *символьный-литерал*
| *строковый-литерал*
специальная-лексема →
`(` | `)` | `,` | `;` | `[` | `]` | `` ` `` | `{` | `}`

пробельная-строка →
пробельный-элемент {*пробельный-элемент*}
пробельный-элемент →
пробельный-символ
| *комментарий*
| *вложенный-комментарий*
пробельный-символ →
новая-строка

| *вертикальная-табуляция*
| *пробел*
| *горизонтальная-табуляция*
| *пробельный-символ-Unicode*

новая-строка →
возврат-каретки перевод-строки
| *возврат-каретки*
| *перевод-строки*
| *перевод-страницы*

комментарий →
тире [*любой-символ*$_{\langle \textit{символ} \rangle}$ {*любой-символ*}] *новая-строка*
тире →
`--` {`-`}
начало-комментария →
`{-`
конец-комментария →
`-}`
вложенный-комментарий →
начало-комментария ЛЮБАЯ-последовательность
{*вложенный-комментарий ЛЮБАЯ-последовательность*} *конец-комментария*
ЛЮБАЯ-последовательность →
{*ЛЮБОЙ-символ*}$_{\langle \{\textit{ЛЮБОЙ-символ}\}\ (\ \textit{начало-комментария}\ |\ \textit{конец-комментария}\ )\ \{\textit{ЛЮБОЙ-символ}\} \rangle}$
ЛЮБОЙ-символ →
графический-символ
| *пробельный-символ*
любой-символ →
графический-символ
| *пробел*
| *горизонтальная-табуляция*
графический-символ →
маленькая-буква
| *большая-буква*
| *символ*
| *цифра*
| *специальная-лексема*
| `:` | `"` | `'`

маленькая-буква →
маленькая-буква-ASCII
| *маленькая-буква-Unicode*
| `_`
маленькая-буква-ASCII →
`a` | `b` | ... | `z`

большая-буква →
  большая-буква-ASCII
  | *большая-буква-Unicode*
большая-буква-ASCII →
  `A` | `B` | ... | `Z`
символ →*символ-ASCII*
  | *символ-Unicode*⟨*специальная-лексема* | `_` | `:` | `"` | `'`⟩

символ-ASCII →
  `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@`
  |`\` | `^` | `|` | `-` | `~`
символ-Unicode →
  любой символ или знак пунктуации Unicode
цифра →
  цифра-ASCII
  | *цифра-Unicode*
цифра-ASCII →
  `0` | `1` | ... | `9`
цифра-Unicode →
  любая десятичная цифра Unicode
восьмиричная-цифра →
  `0` | `1` | ... | `7`
шестнадцатиричная-цифра →
  цифра | `A` | ... | `F` | `a` | ... | `f`

Лексический анализ должен использовать правило “максимального потребления”: в каждой точке считывается наиболее длинная из возможных лексем, которая удовлетворяет правилу вывода *lexeme* (*лексемы*). Таким образом, несмотря на то, что `case` является зарезервированным словом, `cases` таковым не является. Аналогично, несмотря на то, что `=` зарезервировано, `==` и `~=` — нет.

Любой вид *whitespace* (*пробельной-строки*) также является правильным разделителем для лексем.

Символы, которые не входят в категорию *ANY* (*ЛЮБОЙ-символ*), недопустимы в программах на Haskell и должны приводить к лексической ошибке.

2.3 Комментарии

Комментарии являются правильными пробельными символами.

Обычный комментарий начинается с последовательности двух или более следующих друг за другом символов тире (например, `-`) и простирается до следующего символа

новой строки. *Последовательность тире не должна являться частью правильной лексемы.* Например, "`->`" или "`|-`" *не* являются началом комментария, потому что оба они являются правильными лексемами; но "`-foo`" начинает комментарий.

Вложенный комментарий начинается с "`{-`" и заканчивается "`-}`". Нет никаких правильных лексем, которые начинаются с "`{-`"; следовательно, например, "`{--`" начинает вложенный комментарий несмотря на замыкающие символы тире.

Сам комментарий не подвергается лексическому анализу. Вместо этого, первое, не имеющее соответствующей пары вхождение строки "`-}`" заканчивает вложенный комментарий. Вложенные комментарии могут быть вложенными на любую глубину: любое вхождение строки "`{-`" в пределах вложенного комментария начинает новый вложенный комментарий, заканчивающийся "`-}`". В пределах вложенного комментария каждый "`{-`" сопоставляется с соответствующим вхождением "`-}`".

В обычном комментарии последовательности символов "`{-`" и "`-}`" не имеют никакого специального значения, и во вложенном комментарии последовательность символов тире не имеет никакого специального значения.

Вложенные комментарии также используются для указаний компилятору, объясненных в главе 11.

Если некоторый код закомментирован с использованием вложенного комментария, то любое вхождение `{-` или `-}` в пределах строки или в пределах комментария до конца строки в этом коде будет влиять на вложенные комментарии.

2.4 Идентификаторы и операторы

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

Перевод:
идентификатор-переменной →
(*маленькая-буква* {*маленькая-буква* | *большая-буква* | *цифра* |
`'` })$_{\langle зарезервированный\text{-}идентификатор \rangle}$
идентификатор-конструктора →
большая-буква {*маленькая-буква* | *большая-буква* | *цифра* | `'` }
зарезервированный-идентификатор →
`case` | `class` | `data` | `default` | `deriving` | `do` | `else`
| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance`
| `let` | `module` | `newtype` | `of` | `then` | `type` | `where` | `_`

Идентификатор состоит из буквы, за которой следует ноль или более букв, цифр, символов подчеркивания и одинарных кавычек. Идентификаторы лексически делятся на два пространства имен (раздел 1.4): те, которые начинаются со строчной буквы (идентификаторы переменных), и те, которые начинаются с заглавной буквы (идентификаторы конструкторов). Идентификаторы зависят от регистра букв: `name`, `naMe` и `Name` — это три различных идентификатора (первые два являются идентификаторами переменных, последний — идентификатором конструктора).

Символ подчеркивания "`_`" рассматривается как строчная буква, и может появляться везде, где может появляться строчная буква. Тем не менее, "`_`" сам по себе является зарезервированным идентификатором, который используется для обозначения группы любых символов в образцах. Компиляторы, которые выводят предупреждения о неиспользованных идентификаторах, подавляют вывод таких предупреждений для идентификаторов, начинающихся с символа подчеркивания. Это позволяет программистам использовать "`_foo`" для параметра, который, как они ожидают, не будет использоваться.

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

Перевод:
символ-переменной →
(*символ* {*символ* | `:`})$_{\langle зарезервированный\text{-}оператор \mid тире \rangle}$
символ-конструктора →
(`:` {*символ* | `:`})$_{\langle зарезервированный\text{-}оператор \rangle}$
зарезервированный-оператор →
`..` | `:` | `::` | `=` | `\` | `|` | `<-` | `->` | `@` | `~` | `=>`

Символы операторов образуются из одного или более символов, в соответствии с приведенным выше определением, и лексически делятся на два пространства имен (раздел 1.4):

- Символ оператора, начинающийся с двоеточия, обозначает конструктор.
- Символ оператора, начинающийся с любого другого символа, обозначает обычный идентификатор.

Заметьте, что само по себе двоеточие "`:`" зарезервировано исключительно для использования в качестве конструктора списков в Haskell; это делает их интерпретацию единообразной с другими частями синтаксиса списка, как например "`[]`" и "`[a,b]`".

За исключением специального синтаксиса для префиксного отрицания, все остальные операторы являются инфиксными, хотя каждый инфиксный оператор можно

использовать в *сечении* для выполнения частичного применения операторов (см. раздел 3.5). Все стандартные инфиксные операторы являются просто предопределенными символами и могут использоваться в связывании имен.

В оставшейся части описания будут использоваться шесть различных видов имен:

varid			(переменные)
conid			(конструкторы)
tyvar	→	*varid*	(переменные типов)
tycon	→	*conid*	(конструкторы типов)
tycls	→	*conid*	(классы типов)
modid	→	*conid*	(модули)

Перевод:
идентификатор-переменной
(переменные)
идентификатор-конструктора
(конструкторы)
переменная-типа →
идентификатор-переменной
(переменные типов)
конструктор-типа →
идентификатор-конструктора
(конструкторы типов)
класс-типа →
идентификатор-конструктора
(классы типов)
идентификатор-модуля →
идентификатор-конструктора
(модули)

Переменные и переменные типов представляются идентификаторами, начинающимися с маленьких букв, а остальные четыре — идентификаторами, начинающимися с больших букв; также, переменные и конструкторы имеют инфиксные формы, остальные четыре — нет. Пространства имен также рассматриваются в разделе 1.4.

В определенных обстоятельствах имя может быть снабжено необязательным квалификатором, т.е. быть *квалифицировано*, посредством присоединения к нему слева идентификатора модуля. Это относится к именам переменных, конструкторов, конструкторов типов и классов типов, но не относится к переменным типов или именам модулей. Квалифицированные имена подробно рассматриваются в главе 5.

qvarid → [*modid* .] *varid*

qconid → [*modid* .] *conid*
qtycon → [*modid* .] *tycon*
qtycls → [*modid* .] *tycls*
qvarsym → [*modid* .] *varsym*
qconsym → [*modid* .] *consym*

Перевод:
квалифицированный-идентификатор-переменной →
[*идентификатор-модуля* .] *идентификатор-переменной*
квалифицированный-идентификатор-конструктора →
[*идентификатор-модуля* .] *идентификатор-конструктора*
квалифицированный-конструктор-типа →
[*идентификатор-модуля* .] *конструктор-типа*
квалифицированный-класс-типа →
[*идентификатор-модуля* .] *класс-типа*
квалифицированный-символ-переменной →
[*идентификатор-модуля* .] *символ-переменной*
квалифицированный-символ-конструктора →
[*идентификатор-модуля* .] *символ-конструктора*

Так как квалифицированное имя является лексемой, пробелы недопустимы между квалификатором и именем. Примеры лексического анализа приведены ниже.

Это	интерпретируется как
`f.g`	`f . g` (три токена)
`F.g`	`F.g` (квалифицированное ‘g’)
`f..`	`f ..` (два токена)
`F..`	`F..` (квалифицированный ‘.’)
`F.`	`F .` (два токена)

Квалификатор не изменяет синтаксическую интерпретацию имени; например, `Prelude.+` — это инфиксный оператор той же ассоциативности и того же приоритета, что и определение `+` в Prelude (раздел 4.4.2).

2.5 Числовые литералы

decimal → *digit*{*digit*}
octal → *octit*{*octit*}
hexadecimal → *hexit*{*hexit*}

integer → *decimal*
| `0o` *octal* | `0O` *octal*
| `0x` *hexadecimal* | `0X` *hexadecimal*

float → *decimal* . *decimal* [*exponent*]
| *decimal* *exponent*

exponent → (`e` | `E`) [`+` | `-`] *decimal*

Перевод:
десятичный-литерал →
цифра{цифра}
восьмиричный-литерал →
восьмиричная-цифра{восьмиричная-цифра}
шестнадцатиричный-литерал →
шестнадцатиричная-цифра{шестнадцатиричная-цифра}

целый-литерал →
десятичный-литерал
| `0o` *восьмиричный-литерал*
| `0O` *восьмиричный-литерал*
| `0x` *шестнадцатиричный-литерал*
| `0X` *шестнадцатиричный-литерал*

литерал-с-плавающей-точкой →
десятичный-литерал . *десятичный-литерал* [*экспонента*]
| *десятичный-литерал* *экспонента*

экспонента →
(`e` | `E`) [`+` | `-`] *десятичный-литерал*

Есть два различных вида числовых литералов: целые и с плавающей точкой. Целые литералы можно задавать в десятичной (по умолчанию), восьмиричной (начинается с `0o` или `0O`) или шестнадцатиричной записи (начинается с `0x` или `0X`). Литералы с плавающей точкой всегда являются десятичными. Литерал с плавающей точкой должен содержать цифры и перед, и после десятичной точки; это гарантирует, что десятичная точка не будет ошибочно принята за другое использование символа точка. Отрицательные числовые литералы рассматриваются в разделе 3.4. Типизация числовых литералов рассматривается в разделе 6.4.1.

2.6 Символьные и строковые литералы

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

Перевод:
символьный-литерал →
' (*графический-символ*$_{\langle ' \mid \backslash \rangle}$ | *пробел* | *эскейп-символ*$_{\langle \backslash \& \rangle}$) '
строковый-литерал →
" {*графический-символ*$_{\langle " \mid \backslash \rangle}$ | *пробел* | *эскейп-символ* | *разрыв*} "
эскейп-символ →
\ (*символ-эскейп* | *символ-ascii* | *десятичный-литерал* | o *восьмиричный-литерал* |
x *шестнадцатиричный-литерал*)
символ-эскейп →
a | b | f | n | r | t | v | \ | " | ' | &
символ-ascii →
^*управляющий-символ* | 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
управляющий-символ →
большая-буква-ASCII | @ | [| \ |] | ^ | _
разрыв →
\ *пробельный-символ* {*пробельный-символ*} \

Символьные литералы записываются между одинарными кавычками, как например `'a'`, а строки — между двойными кавычками, как например `"Hello"`.

Эскейп-коды можно использовать в символах и строках для представления специальных символов. Обратите внимание, что одинарную кавычку ' можно использовать внутри строки как есть, но для того, чтобы использовать ее внутри символа, необходимо записать перед ней символ обратной косой черты (\). Аналогично, двойную кавычку " можно использовать внутри символа как есть, но внутри строки она должна предваряться символом обратной косой черты. \ всегда должен предваряться символом обратной косой черты. Категория *charesc* (*символ-эскейп*)

также включает переносимые представления для символов “тревога” (`\a`), “забой” (`\b`), “перевод страницы” (`\f`), “новая строка” (`\n`), “возврат каретки” (`\r`), “горизонтальная табуляция” (`\t`) и “вертикальная табуляция” (`\v`).

Эскейп-символы для кодировки Unicode, включая символы управления, такие как `\^X`, также предусмотрены. Числовые эскейп-последовательности, такие как `\137`, используются для обозначения символа с десятичным представлением 137; восьмиричные (например, `\o137`) и шестнадцатиричные (например, `\x37`) представления также допустимы.

Согласующиеся с правилом “максимального потребления”, числовые эскейп-символы в строках состоят изо всех последовательных цифр и могут иметь произвольную длину. Аналогично, единственный неоднозначный эскейп-код ASCII `"\SOH"` при разборе интерпретируется как строка длины 1. Эскейп-символ `\&` предусмотрен как “пустой символ”, чтобы позволить создание таких строк, как `"\137\&9"` и `"\SO\&H"` (обе длины два). Поэтому `"\&"` эквивалентен , а символ `'\&'` недопустим. Дальнейшие эквивалентности символов определены в разделе 6.1.2.

Строка может включать “разрыв” — две обратные косые черты, окруженные пробельными символами, которые игнорируются. Это позволяет записывать длинные программные строки на более чем одной строке файла, для этого надо добавлять обратную косую черту (бэкслэш) в конец каждой строки файла и в начале следующей. Например,

```
"Это бэкслэш \\, так же как \137 --- \
    \ числовой эскейп-символ и \^X --- управляющий символ."
```

Строковые литералы в действительности являются краткой записью для списков символов (см. раздел 3.7).

2.7 Размещение

В Haskell разрешено опускать фигурные скобки и точки с запятой, используемые в нескольких правилах вывода грамматики, посредством использования определенного *размещения* текста программы с тем, чтобы отразить ту же самую информацию. Это позволяет использовать и зависящий от размещения текста, и не зависящий от размещения текста стили написания кода, которые можно свободно смешивать в одной программе. Поскольку не требуется располагать текст определенным образом, программы на Haskell можно непосредственно генерировать другими программами.

Влияние размещения текста программы на смысл программы на Haskell можно полностью установить путем добавления фигурных скобок и точек с запятой в местах, определяемых размещением. Смысл такого дополнения программы состоит в том, чтобы сделать ее не зависимой от размещения текста.

Будучи неофициально заявленными, фигурные скобки и точки с запятой добавляются следующим образом. Правило размещения текста (или правило "вне игры") вступает в силу всякий раз, когда открытая фигурная скобка пропущена после ключевого слова `where`, `let`, `do` или `of`. Когда это случается, отступ следующей лексемы (неважно на новой строке или нет) запоминается, и вставляется пропущенная открытая фигурная скобка (пробельные символы, предшествующие лексеме, могут включать комментарии). Для каждой последующей строки выполняется следующее: если она содержит только пробельные символы или больший отступ, чем предыдущий элемент, то это означает, что продолжается предыдущий элемент (ничего не добавляется); если строка имеет тот же отступ, это означает, что начинается новый элемент (вставляется точка с запятой); если строка имеет меньший отступ, то это означает, что эакончился список размещения (вставляется закрывающая фигурная скобка). Если отступ лексемы без фигурных скобок, которая следует непосредственно за `where`, `let`, `do` или `of`, меньше чем или равен текущему уровню углубления, то вместо начала размещения вставляется пустой список "`{}`" и обработка размещения выполняется для текущего уровня (т.е. вставляется точка с запятой или закрывающая фигурная скобка). Закрывающая фигурная скобка также вставляется всякий раз, когда заканчивается синтаксическая категория, содержащая список размещения; то есть если неправильная лексема встретится в месте, где была бы правильной закрывающая фигурная скобка, вставляется закрывающая фигурная скобка. Правило размещения добавляет закрывающие фигурные скобки, которые соответствуют только тем открытым фигурным скобкам, которые были добавлены согласно этому правилу; явная открытая фигурная скобка должна соответствовать явной закрывающей фигурной скобке. В пределах этих явных открытых фигурных скобок, *никакая* обработка размещения не выполняется для конструкций вне фигурных скобок, даже если строка выравнена левее (имеет меньший отступ) более ранней неявной открытой фигурной скобки.

В разделе 9.3 дано более точное определение правил размещения.

Согласно этим правилам, отдельный символ новой строки на самом деле может завершить несколько списков размещения. Также эти правила разрешают:

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

делая `a`, `b` и `g` частью одного и того же списка размещения.

В качестве примера на рис. 2.1 изображен (несколько запутанный) модуль, а на рис. 2.2 показан результат применения к нему правила размещения. Обратите внимание, в частности, на: (a) строку, начинающую `}};pop`, в которой завершение предыдущей строки вызывает три применения правила размещения, соответствующих глубине (3) вложенной инструкции `where`, (b) закрывающие фигурные скобки в инструкции `where`, вложенной в пределах кортежа и `case`-выражения, вставленные потому, что был обнаружен конец кортежа, и (c) закрывающую фигурную скобку в самом конце, вставленную из-за нулевого отступа лексемы конца файла.

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

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

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

pop :: Stack a -> (a, Stack a)
pop (MkStack x s)
  = (x, case s of r -> i r where i x = x) -- (pop Empty) является ошибкой

top :: Stack a -> a
top (MkStack x s) = x                     -- (top Empty) является ошибкой
```

Рис. 2.1: Пример программы

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

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

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

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

;top :: Stack a -> a
;top (MkStack x s) = x                      -- (top Empty) является ошибкой
}
```

Рис. 2.2: Пример программы, дополненной размещением

Глава 3

Выражения

В этой главе мы опишем синтаксис и неформальную семантику *выражений* Haskell, включая, где это возможно, их трансляцию в ядро Haskell. За исключением `let`-выражений эти трансляции сохраняют и статическую, и динамическую семантику. Свободные переменные и конструкторы, используемые в этих трансляциях, всегда ссылаются на сущности, определенные в `Prelude`. Например, "`concatMap`", используемая в трансляции описания списка (раздел 3.11), обозначает `concatMap`, определенную в `Prelude`, невзирая на то, находится ли идентификатор "`concatMap`" в области видимости, где используется описание списка, и (если находится в области видимости) к чему он привязан.

В синтаксисе, который следует далее, есть некоторые семейства нетерминалов, индексированные уровнями приоритета (записанными как верхний индекс). Аналогично, нетерминалы *op* (*оператор*), *varop* (*оператор-переменной*) и *conop* (*оператор-конструктора*) могут иметь двойной индекс: букву l, r или n соответственно для левоассоциативности, правоассоциативности или отсутствия ассоциативности и уровень приоритета. Переменная уровня приоритета i изменяется в пределах от 0 до 9, переменная ассоциативности a изменяется в диапазоне $\{l,\ r,\ n\}$. Например,

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

на самом деле обозначает 30 правил вывода с 10 подстановками для i и 3 для a.

$$\begin{array}{lcll}
exp & \rightarrow & exp^{0}\ \texttt{::}\ [context\ \texttt{=>}]\ type & \text{(сигнатура типа выражения)} \\
 & | & exp^{0} & \\
exp^{i} & \rightarrow & exp^{i+1}\ [qop^{(\mathrm{n},i)}\ exp^{i+1}] & \\
 & | & lexp^{i} & \\
 & | & rexp^{i} & \\
lexp^{i} & \rightarrow & (lexp^{i} \mid exp^{i+1})\ qop^{(\mathrm{l},i)}\ exp^{i+1} &
\end{array}$$

$lexp^6$	$\rightarrow$	`-` exp^7	
$rexp^i$	$\rightarrow$	exp^{i+1} $qop^{(\mathrm{r},i)}$ $(rexp^i \mid exp^{i+1})$	
exp^{10}	$\rightarrow$	`\` $apat_1$ ... $apat_n$ `->` exp	(лямбда-абстракция, $n \geq 1$)
	|	`let` $decls$ `in` exp	(let-выражение)
	|	`if` exp `then` exp `else` exp	(условное выражение)
	|	`case` exp `of` `{` $alts$ `}`	(case-выражение)
	|	`do` `{` $stmts$ `}`	(do-выражение)
	|	$fexp$	
$fexp$	$\rightarrow$	$[fexp]$ $aexp$	(применение функции)
$aexp$	$\rightarrow$	$qvar$	(переменная)
	|	$gcon$	(общий конструктор)
	|	$literal$	
	|	`(` exp `)`	(выражение в скобках)
	|	`(` exp_1 `,` ... `,` exp_k `)`	(кортеж, $k \geq 2$)
	|	`[` exp_1 `,` ... `,` exp_k `]`	(список, $k \geq 1$)
	|	`[` exp_1 `[,` exp_2`]` `..` `[`exp_3`]` `]`	(арифметическая последовательность)
	|	`[` exp `|` $qual_1$ `,` ... `,` $qual_n$ `]`	(описание списка, $n \geq 1$)
	|	`(` exp^{i+1} $qop^{(a,i)}$ `)`	(левое сечение)
	|	`(` $lexp^i$ $qop^{(l,i)}$ `)`	(левое сечение)
	|	`(` $qop^{(a,i)}_{\langle \texttt{-} \rangle}$ exp^{i+1} `)`	(правое сечение)
	|	`(` $qop^{(r,i)}_{\langle \texttt{-} \rangle}$ $rexp^i$ `)`	(правое сечение)
	|	$qcon$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(именованная конструкция, $n \geq 0$)
	|	$aexp_{\langle qcon \rangle}$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(именованное обновление, $n \geq 1$)

Перевод:

выражение $\rightarrow$
  выражение^0 `::` [*контекст* `=>`] *тип*
    (сигнатура типа выражения)
  | *выражение*^{0}

выражение^i $\rightarrow$
  выражение$^{i+1}$ [*квалифицированный-оператор*$^{(\mathrm{n},i)}$ *выражение*$^{i+1}$]
  | *левое-сечение-выражения*^{i}
  | *правое-сечение-выражения*^{i}

левое-сечение-выражения^i $\rightarrow$
  (*левое-сечение-выражения*^{i} | *выражение*$^{i+1}$) *квалифицированный-оператор*$^{(\mathrm{l},i)}$
    выражение$^{i+1}$

левое-сечение-выражения^6 $\rightarrow$
  `-` *выражение*^{7}

правое-сечение-выражения^i $\rightarrow$

выражение$^{i+1}$ *квалифицированный-оператор*$^{(\mathrm{r},i)}$
(*правое-сечение-выражения*^{i} | *выражение*$^{i+1}$)

выражение^{10} →
`\` *такой-как-образец*$_1$ ... *такой-как-образец*$_n$ `->` *выражение*
(лямбда-абстракция, $n \geq 1$)
| `let` *списки-объявлений* `in` *выражение*
(let-выражение)
| `if` *выражение* `then` *выражение* `else` *выражение*
(условное выражение)
| `case` *выражение* `of {` *список-альтернатив* `}`
(case-выражение)
| `do {` *список-инструкций* `}`
(do-выражение)
| *функциональное-выражение*

функциональное-выражение →
[*функциональное-выражение*] *выражение-аргумента*
(применение функции)

выражение-аргумента →
квалифицированная-переменная
(переменная)
| *общий-конструктор*
(общий конструктор)
| *литерал*
| `(` *выражение* `)`
(выражение в скобках)
| `(` *выражение*$_1$ `,` ... `,` *выражение*$_k$ `)`
(кортеж, $k \geq 2$)
| `[` *выражение*$_1$ `,` ... `,` *выражение*$_k$ `]`
(список, $k \geq 1$)
| `[` *выражение*$_1$ [`,` *выражение*$_2$] `..` [*выражение*$_3$] `]`
(арифметическая последовательность)
| `[` *выражение* `|` *квалификатор*$_1$ `,` ... `,` *квалификатор*$_n$ `]`
(описание списка, $n \geq 1$)
| `(` *выражение*$^{i+1}$ *квалифицированный-оператор*$^{(a,i)}$ `)`
(левое сечение)
| `(` *левое-сечение-выражения*^{i} *квалифицированный-оператор*$^{(l,i)}$ `)`
(левое сечение)
| `(` *квалифицированный-оператор*$^{(a,i)}_{\langle\text{-}\rangle}$ *выражение*$^{i+1}$ `)`
(правое сечение)
| `(` *квалифицированный-оператор*$^{(r,i)}_{\langle\text{-}\rangle}$ *правое-сечение-выражения*^{i} `)`
(правое сечение)
| *квалифицированный-конструктор*
`{` *связывание-имени-поля*$_1$ `,` ... `,` *связывание-имени-поля*$_n$ `}`

(именованная конструкция, $n \geq 0$)
| *выражение-аргумента*$_{\langle$*квалифицированный-конструктор*$\rangle}$
{ *связывание-имени-поля*$_1$, ... , *связывание-имени-поля*$_n$ }
(именованное обновление, $n \geq 1$)

Неоднозначность выражений, включая инфиксные операторы, разрешается с помощью ассоциативности и приоритета оператора (см. раздел 4.4.2). Следующие друг за другом операторы без скобок, имеющие один и тот же приоритет, должны быть оба либо левоассоциативными, либо правоассоциативными, во избежание синтаксической ошибки. Для заданного выражения без скобок "x *qop*$^{(a,i)}$ y *qop*$^{(b,j)}$ z", где *qop* — оператор, часть "x *qop*$^{(a,i)}$ y" или "y *qop*$^{(b,j)}$ z" следует взять в скобки, когда $i = j$, за исключением $a = b =$ l или $a = b =$ r.

Отрицание является единственным префиксным оператором в Haskell, он имеет тот же приоритет, что и инфиксный оператор -, определенный в Prelude (см. раздел 4.4.2, рис. 4.1).

Грамматика является неоднозначной по отношению к пространству лямбда-абстракций, let-выражений и условных выражений. Неоднозначность разрешается с помощью мета-правила, согласно которому каждая из этих конструкций рассматривается слева направо насколько это возможно.

Примеры интерпретации при разборе показаны ниже.

Это	интерпретируется как
`f x + g y`	`(f x) + (g y)`
`- f x + y`	`(- (f x)) + y`
`let { ... } in x + y`	`let { ... } in (x + y)`
`z + let { ... } in x + y`	`z + (let { ... } in (x + y))`
`f x y :: Int`	`(f x y) :: Int`
`\ x -> a+b :: Int`	`\ x -> ((a+b) :: Int)`

Замечание относительно разбора. Выражения, которые затрагивают взаимодействие ассоциативности и приоритетов с применением мета-правила для let/лямбда, могут оказаться трудными для разбора. Например, выражение

```
let x = True in x == x == True
```

не может означать

```
let x = True in (x == x == True)
```

потому что `(==)` является неассоциативным оператором, поэтому выражение должно быть интерпретировано таким образом:

```
(let x = True in (x == x)) == True
```

Тем не менее, реализации могут прекрасно обойтись дополнительным проходом после завершения разбора, чтобы обработать ассоциативность и приоритеты операторов, поэтому они могут спокойно передать предыдущий неправильный разбор. Мы советуем программистам избегать использования конструкций, чей разбор затрагивает взаимодействие (отсутствия) ассоциативности с применением мета-правила для let/лямбда.

Ради ясности остальная часть этого раздела описывает синтаксис выражений без указания их приоритетов.

3.1 Ошибки

Ошибки во время вычисления выражений, обозначаемые как $\perp$, в программе на Haskell не отличимы от незавершенного вычисления. Поскольку Haskell является языком с нестрогой типизацией данных, все типы Haskell включают $\perp$. Другими словами, значение любого типа может быть связано с вычислением, которое будет завершено, когда потребуется результат вычисления, и завершится с ошибкой. При своем вычислении ошибки вызывают немедленное завершение программы и не могут быть отловлены пользователем. Prelude содержит две функции, которые сразу вызывают такие ошибки:

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

Вызов функции `error` завершает выполнение программы и возвращает в операционную систему соответствующий признак ошибки. Он также должен вывести на экран строку некоторым, зависящим от системы, способом. Когда используется функция `undefined`, сообщение об ошибке создается компилятором.

Трансляции выражений Haskell используют `error` и `undefined` для явного указания мест, где могли возникнуть ошибки времени выполнения. Реальное поведение программы в случае возникновения ошибки зависит от реализации. Сообщения, передаваемые в функцию `error` при этих трансляциях, являются лишь указаниями, реализации могут выбирать между отображением большей или меньшей информации в случае возникновения ошибки.

3.2 Переменные, конструкторы, операторы и литералы

aexp	→	*qvar*	(переменная)
	|	*gcon*	(общий конструктор)
	|	*literal*	

gcon	→	`()`	
	|	`[]`	
	|	`(,{,})`	
	|	*qcon*	
var	→	*varid* | `(` *varsym* `)`	(переменная)
qvar	→	*qvarid* | `(` *qvarsym* `)`	(квалифицированная переменная)
con	→	*conid* | `(` *consym* `)`	(конструктор)
qcon	→	*qconid* | `(` *gconsym* `)`	(квалифицированный конструктор)
varop	→	*varsym* | ‘ *varid* ‘	(оператор переменной)
qvarop	→	*qvarsym* | ‘ *qvarid* ‘	(квалифицированный оператор переменной)
conop	→	*consym* | ‘ *conid* ‘	(оператор конструктора)
qconop	→	*gconsym* | ‘ *qconid* ‘	(квалифицированный оператор конструктора)
op	→	*varop* | *conop*	(оператор)
qop	→	*qvarop* | *qconop*	(квалифицированный оператор)
gconsym	→	`:` | *qconsym*	

Перевод:

выражение-аргумента →
  квалифицированная-переменная
    (переменная)
  | *общий-конструктор*
    (общий конструктор)
  | *литерал*

общий-конструктор →
  `()`
  | `[]`
  | `(,{,})`
  | *квалифицированный-конструктор*

переменная →
  идентификатор-переменной
  | `(` *символ-переменной* `)`
    (переменная)

квалифицированная-переменная →
  квалифицированный-идентификатор-переменной
  | `(` *квалифицированный-символ-переменной* `)`
    (квалифицированная переменная)

конструктор →
  идентификатор-конструктора
  | (*символ-конструктора*)
    (конструктор)
квалифицированный-конструктор →
  квалифицированный-идентификатор-конструктора
  | (*символ-общего-конструктора*)
    (квалифицированный конструктор)
оператор-переменной →
  символ-переменной
  | ‘ *идентификатор-переменной* ‘
    (оператор переменной)
квалифицированный-оператор-переменной →
  квалифицированный-символ-переменной
  | ‘ *квалифицированный-идентификатор-переменной* ‘
    (квалифицированный оператор переменной)
оператор-конструктора →
  символ-конструктора
  | ‘ *идентификатор-конструктора* ‘
    (оператор конструктора)
квалифицированный-оператор-конструктора →
  символ-общего-конструктора
  | ‘ *квалифицированный-идентификатор-конструктора* ‘
    (квалифицированный оператор конструктора)
оператор →
  оператор-переменной
  | *оператор-конструктора*
    (оператор)
квалифицированный-оператор →
  квалифицированный-оператор-переменной
  | *квалифицированный-оператор-конструктора*
    (квалифицированный оператор)
символ-общего-конструктора →
  : | *квалифицированный-символ-конструктора*

Для поддержки инфиксной записи в Haskell используется специальный синтаксис. *Оператор* — это функция, которая может быть применена, используя инфиксный синтаксис (раздел 3.4), или частично применена, используя *сечения* (раздел 3.5).

Оператор представляет собой *символ оператора*, например, `+` или `$$`, или обычный идентификатор, заключенный в обратные кавычки, например, ‘`op`‘. Вместо префиксной записи `op x y` можно использовать инфиксную запись `x ‘op‘ y`. Если ассоциативность и приоритет для ‘`op`‘ не заданы, то по умолчанию используется наивысший приоритет и левоассоциативность (см. раздел 4.4.2).

Наоборот, символ оператора может быть преобразован в обычный идентификатор,

если записать его в круглых скобках. Например, `(+) x y` эквивалентно `x + y`, а `foldr (*) 1 xs` эквивалентно `foldr (\x y -> x*y) 1 xs`.

Для обозначения некоторых конструкторов встроенных типов используется специальный синтаксис, как это видно из грамматики для *gcon* (*общего-конструктора*) и *literal* (*литерала*). Они описаны в разделе 6.1.

Целый литерал представляет собой применение функции `fromInteger` к соответствующему значению типа `Integer`. Аналогично, литерал с плавающей точкой обозначает применение функции `fromRational` к значению типа `Rational` (то есть `Ratio Integer`).

Трансляция: Целый литерал i эквивалентен `fromInteger` i, где `fromInteger` — метод класса `Num` (см. раздел 6.4.1).
Литерал с плавающей точкой f эквивалентен `fromRational` (n `Ratio.%` d), где `fromRational` — метод класса `Fractional`, а `Ratio.%` составляет из двух целых чисел рациональное число в соответствии с определением, заданным в библиотеке `Ratio`. Если заданы целые числа n и d, то $n/d = f$.

3.3 Производные функции и лямбда-абстракции

fexp	→	[*fexp*] *aexp*	(применение функции)
exp	→	\ $apat_1$ ... $apat_n$ -> *exp*	(лямбда-абстракция, $n \geq 1$)

Перевод:
функциональное-выражение →
  [*функциональное-выражение*] *выражение-аргумента*
    (применение функции)
выражение →
  \ *такой-как-образец*$_1$ ... *такой-как-образец*$_n$ -> *выражение*
    (лямбда-абстракция, $n \geq 1$)

Применение функции записывается в виде $e_1\ e_2$. Применение левоассоциативно, поэтому в `(f x) y` скобки можно опустить. Поскольку e_1 может являться конструктором данных, возможно частичное применение конструкторов данных.

Лямбда-абстракции записываются в виде \ p_1 ... p_n -> e, где p_i — *образцы*. Выражение вида `\x:xs->x` синтаксически неправильно, его можно правильно записать в виде `\(x:xs)->x`.

Набор образцов должен быть *линейным*: ни одна переменная не должна появляться в наборе более одного раза.

Трансляция: Выполняются следующие тождества:

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

где x_i — новые идентификаторы.

Используя эту трансляцию в комбинации с с семантикой case-выражений и сопоставлений с образцом, описанной в разделе 3.17.3, получим: если сопоставление с образцом завершится неудачно, то результатом будет $\bot$.

3.4 Применение операторов

exp	→	*exp₁ qop exp₂*	
	|	- *exp*	(префиксное отрицание)
qop	→	*qvarop* | *qconop*	(квалифицированный оператор)

Перевод:
выражение →
  выражение₁ квалифицированный-оператор выражение₂
  | - *выражение*
    (префиксное отрицание)
квалифицированный-оператор →
  квалифицированный-оператор-переменной
  | *квалифицированный-оператор-конструктора*
    (квалифицированный оператор)

Форма $e_1\ qop\ e_2$ представляет собой инфиксное применение бинарного оператора *qop* к выражениям e_1 и e_2.

Специальная форма `-`e обозначает префиксное отрицание, единственный префиксный оператор в Haskell, и является синтаксической записью `отрицания` (e). Бинарный оператор `-` необязательно ссылается на определение `-` в Prelude, он может быть переопределен системой модуля. Тем не менее, унарный оператор `-` будет всегда ссылаться на функцию `negate`, определенную в Prelude. Нет никакой связи между локальным значением оператора `-` и унарным отрицанием.

Префиксный оператор отрицания имеет тот же приоритет, что и инфиксный оператор `-`, определенный в Prelude (см. таблицу 4.1, стр. 79). Поскольку `e1-e2` при разборе интерпретируется как инфиксное применение бинарного оператора `-`, для альтернативной интерпретации нужно писать `e1(-e2)`. Аналогично, `(-)` является

синтаксической записью `(\ x y -> x-y)`, как и любой инфиксный оператор, и не обозначает `(\ x -> -x)` — для этого нужно использовать `negate`.

Трансляция: Выполняются следующие тождества:

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

3.5 Сечения

aexp	→	`(` exp^{i+1} $qop^{(a,i)}$ `)`	(левое сечение)
	|	`(` $lexp^{i}$ $qop^{(l,i)}$ `)`	(левое сечение)
	|	`(` $qop^{(a,i)}_{\langle\texttt{-}\rangle}$ exp^{i+1} `)`	(правое сечение)
	|	`(` $qop^{(r,i)}_{\langle\texttt{-}\rangle}$ $rexp^{i}$ `)`	(правое сечение)

Перевод:
выражение-аргумента →
`(` *выражение*$^{i+1}$ *квалифицированный-оператор*$^{(a,i)}$ `)`
(левое сечение)
| `(` *левое-сечение-выражения*^{i} *квалифицированный-оператор*$^{(l,i)}$ `)`
(левое сечение)
| `(` *квалифицированный-оператор*$^{(a,i)}_{\langle\texttt{-}\rangle}$ *выражение*$^{i+1}$ `)`
(правое сечение)
| `(` *квалифицированный-оператор*$^{(r,i)}_{\langle\texttt{-}\rangle}$ *правое-сечение-выражения*^{i} `)`
(правое сечение)

Сечения записываются в виде `(` *op e* `)` или `(` *e op* `)`, где *op* — бинарный оператор, а *e* — выражение. Сечения представляют собой удобный синтаксис для записи частичного применения бинарных операторов.

Синтаксические правила приоритетов применяются к сечениям следующим образом. `(`*op e*`)` допустимо, если и только если `(x` *op e*`)` при разборе интерпретируется так же, как и `(x` *op* `(`*e*`))`; аналогично для `(`*e op*`)`. Например, `(*a+b)` синтаксически недопустимо, но `(+a*b)` и `(*(a+b))` допустимы. Поскольку оператор `(+)` левоассоциативен, `(a+b+)` синтаксически правильно, а `(+a+b)` — нет, его можно правильно записать в виде `(+(a+b))`. В качестве другого примера рассмотрим выражение

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

которое является недопустимым в соответствии с мета-правилом для let/лямбда (раздел 3). Выражение

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

при разборе интерпретируется как

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

вместо

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

Поскольку `-` интерпретируется в грамматике специальным образом, `(-` *exp*`)` является не сечением, а применением префиксного оператора отрицания, в соответствии с описанием в предыдущем разделе. Тем не менее, имеется функция `subtract`, определенная в Prelude таким образом, что `(subtract` *exp*`)` эквивалентно недопустимому ранее сечению. Для той же цели может служить выражение `(+ (-` *exp*`))`.

Трансляция: Выполняются следующие тождества:

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

где *op* — бинарный оператор, *e* — выражение, а *x* — переменная, которая не является свободной в *e*.

3.6 Условные выражения

exp → `if` exp_1 `then` exp_2 `else` exp_3

Перевод:
выражение →
`if` $выражение_1$ `then` $выражение_2$ `else` $выражение_3$

Условное выражение имеет вид `if` e_1 `then` e_2 `else` e_3 и возвращает: значение e_2 — если значение e_1 равно `True`, e_3 — если e_1 равно `False`, и $\perp$ — иначе.

Трансляция: Выполняются следующие тождества:

`if` e_1 `then` e_2 `else` e_3 = `case` e_1 `of { True ->` e_2 `; False ->` e_3 `}`

где `True` и `False` — конструкторы с нулевым числом аргументов из типа `Bool`, определенные в Prelude. Тип e_1 должен быть `Bool`, e_2 и e_3 должны иметь тот же тип, который также является типом всего условного выражения.

3.7 Списки

exp	→	exp_1 *qop* exp_2	
aexp	→	`[` exp_1 `,` ... `,` exp_k `]`	$(k \geq 1)$
	|	*gcon*	
gcon	→	`[]`	
	|	*qcon*	
qcon	→	`(` *gconsym* `)`	
qop	→	*qconop*	
qconop	→	*gconsym*	
gconsym	→	`:`	

Перевод:
выражение →
  выражение$_1$ *квалифицированный-оператор выражение*$_2$
выражение-аргумента →
  `[` *выражение*$_1$ `,` ... `,` *выражение*$_k$ `]`
    $(k \geq 1)$
  | *общий-конструктор*
общий-конструктор →
  `[]`
  | *квалифицированный-конструктор*
квалифицированный-конструктор →
  `(` *символ-общего-конструктора* `)`
квалифицированный-оператор →
  квалифицированный-оператор-конструктора
квалифицированный-оператор-конструктора →
  символ-общего-конструктора
символ-общего-конструктора →
  `:`

Списки записываются в виде `[`e_1`,` ... `,` e_k`]`, где $k \geq 1$. Конструктором списка является `:`, пустой список обозначается `[]`. Стандартные операции над списками описаны в Prelude (см. раздел 6.1.3 и главу 8, особенно раздел 8.1).

Трансляция: Выполняются следующие тождества:

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

где : и [] — конструкторы списков, определенные в Prelude (см. раздел 6.1.3). Выражения e_1, ... , e_k должны быть одного типа (назовем его t), а типом всего выражения является [t] (см. раздел 4.1.2).

Конструктор ":" предназначен исключительно для построения списка; как и [], он является обоснованной частью синтаксиса языка и не может быть скрыт или переопределен. Он представляет собой правоассоциативный оператор с уровнем приоритета 5 (раздел 4.4.2).

3.8 Кортежи

aexp	→	(exp_1 , ... , exp_k)	$(k \geq 2)$
	|	*qcon*	
qcon	→	(,{,})	

Перевод:
выражение-аргумента →
(*выражение*$_1$, ... , *выражение*$_k$)
$(k \geq 2)$
| *квалифицированный-конструктор*
квалифицированный-конструктор →
(,{,})

Кортежи записываются в виде (e_1, ..., e_k) и могут быть произвольной длины $k \geq 2$. Конструктор для кортежа размера n обозначается (,...,), где $n-1$ запятых. Таким образом, (a,b,c) и (,,) a b c обозначают одно и то же значение. Стандартные операции над кортежами описаны в Prelude (см. раздел 6.1.4 и главу 8).

Трансляция: (e_1, ..., e_k) для $k \geq 2$ является экземпляром кортежа размера k, в соответствии с определением в Prelude, и не требует трансляции. Если t_1, ... , t_k — соответственно типы e_1, ... , e_k, то типом кортежа будет (t_1, ..., t_k) (см. раздел 4.1.2).

3.9 Единичные выражения и выражения в скобках

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

Перевод:
выражение-аргумента →
общий-конструктор
| `(` *выражение* `)`
общий-конструктор →
`()`

Выражение вида (e) представляет собой просто *выражение в скобках* и эквивалентно e. *Единичное выражение* `()` имеет тип `()` (см. раздел 4.1.2). Этот единственный член указанного типа, отличный от $\perp$, может рассматриваться как "кортеж нулевого размера" (см. раздел 6.1.5).

Трансляция: (e) эквивалентно e.

3.10 Арифметические последовательности

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

Перевод:
выражение-аргумента →
`[` $выражение_1$ [`,` $выражение_2$] `..` [$выражение_3$] `]`

Арифметическая последовательность `[`e_1 `,` e_2 `..` e_3`]` обозначает список значений типа t, где каждое из выражений e_i имеет тип t, и t является экземпляром класса `Enum`.

Трансляция: Для арифметических последовательностей выполняются следующие тождества:

`[` 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

где `enumFrom`, `enumFromThen`, `enumFromTo` и `enumFromThenTo` являются методами класса `Enum`, определенные в Prelude (см. рис. 6.1, стр. 112).

Семантика арифметических последовательностей поэтому полностью зависит от объявления экземпляра для типа t. Для получения более детальной информации о том, какие типы Prelude являются подтипами `Enum` и какова их семантика, смотрите раздел 6.3.4.

3.11 Описание списка

aexp	→	`[` *exp* `|` *qual*$_1$ `,` ... `,` *qual*$_n$ `]`	(описание списка, $n \geq 1$)
qual	→	*pat* `<-` *exp*	(генератор)
		`let` *decls*	(локальное объявление)
		exp	(страж)

Перевод:
выражение-аргумента →
`[` *выражение* `|` *квалификатор*$_1$ `,` ... `,` *квалификатор*$_n$ `]`
(описание списка, $n \geq 1$)
квалификатор →
образец `<-` *выражение*
(генератор)
| `let` *списки-объявлений*
(локальное объявление)
| *выражение*
(страж)

Описание списка имеет вид `[` e `|` q_1 `,` ... `,` q_n `]`, $n \geq 1$, где квалификаторы q_i являются

- или *генераторами* вида p `<-` e, где p — образец (см. раздел 3.17) типа t, а e — выражение типа `[`t`]`,
- или *стражами*, которые являются произвольными выражениями типа `Bool`,
- или *локальными связываниями имен*, которые обеспечивают новые определения, используемые в генерируемом выражении e или последующих стражах и генераторах.

Такое описание списка возвращает список элементов, порожденный путем вычисления e в последовательных окружениях, созданных вложенным вычислением вглубину генераторов в списке квалификаторов. Связывание имен переменных происходит согласно правилам обычного сопоставления с образцом (см. раздел 3.17), и если сопоставление завершится неудачей, то соответствующий элемент списка будет просто пропущен. Таким образом,

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

порождает список `[4,2]`. Если квалификатор является стражем, то, для того чтобы предшествующее сопоставление с образцом завершилось успешно, необходимо, чтобы значение квалификатора равнялось `True`. Как обычно, связывания имен в описаниях списков могут скрыть связывания имен во внешних областях видимости, например,

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

Трансляция: Для описаний списков выполняются следующие тождества, которые могут быть использованы в качестве трансляции в ядро:

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

где e — выражения, p — образцы, l — выражения, значениями которых являются списки, b — булевы выражения, *decls* — списки объявлений, q — квалификаторы, а Q — последовательности квалификаторов. `ok` — новая переменная. Функция `concatMap` и булево значение `True` определены в Prelude.

Как показывает трансляция описаний списков, переменные, связанные с помощью `let`, имеют полностью полиморфные типы, тогда как переменные, определенные с помощью `<-`, связаны лямбда-выражением и поэтому мономорфны (см. раздел 4.5.4).

3.12 Let-выражения

exp → `let` *decls* `in` *exp*

Перевод:
выражение →
`let` *списки-объявлений* `in` *выражение*

Let-выражения имеют общий вид `let {` d_1 `;` ... `;` d_n `} in` e и вводят вложенный, лексически ограниченный, взаимно рекурсивный список объявлений (в других языках `let` часто называют `letrec`). Областью видимости объявлений является выражение e и правая часть объявлений. Объявления описаны в главе 4. Сопоставление и связывание образцов выполняется лениво, неявная `~` делает эти образцы неопровержимыми. Например,

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

не вызовет ошибку времени выполнения до тех пор, пока `x` или `y` не будут вычислены.

Трансляция: Динамическая семантика выражения `let {` d_1 `;` ... `;` d_n `} в` e_0 охватывается следующей трансляцией: после удаления всех сигнатур типов каждое объявление d_i транслируется в уравнение вида p_i `=` e_i, где p_i и e_i — соответственно образцы и выражения, при этом используется трансляция в разделе 4.4.3. Однажды сделав это, эти тождества выполняются, и это можно использовать в качестве трансляции в ядро:

`let {`p_1`=`e_1`;` ... `;` p_n`=`e_n`} in` e_0	=	`let (~`p_1`,` ... `,~`p_n`) = (`e_1`,` ... `,`e_n`) in` e_0
`let` p `=` e_1 `in` e_0	=	`case` e_1 `of ~`p `->` e_0 где ни одна переменная в p не является свободной в e_1
`let` p `=` e_1 `in` e_0	=	`let` p `= fix (` `\ ~`p `->` e_1`) in` e_0

где `fix` — наименьший ассоциативный оператор. Обратите внимание на использование неопровержимого образца `~`p. Эта трансляция не сохраняет статическую семантику, потому что использование `case` препятствует полностью полиморфной типизации связанных переменных. Статическая семантика связываний имен в `let`-выражениях описана разделе 4.4.3.

3.13 Case-выражения

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

Перевод:
выражение →
  `case` *выражение* `of {` *список-альтернатив* `}`
список-альтернатив →
  $альтернатива_1$ `;` ... `;` $альтернатива_n$
    $(n \geq 1)$
альтернатива →
  образец `->` *выражение* [`where` *список-объявлений*]
  | *образец образец-со-стражами* [`where` *список-объявлений*]
  |

(*пустая альтернатива*)

образец-со-стражами →
  страж `->` *выражение* [*образец-со-стражами*]
страж →
  `|` *выражение*^{0}

Case-выражение имеет общий вид

$$\texttt{case}\ e\ \texttt{of}\ \{\ p_1\ match_1\ ;\ \ldots\ ;\ p_n\ match_n\ \}$$

где каждый $match_i$ имеет общий вид

$$\begin{array}{l} \texttt{|}\ g_{i1}\quad \texttt{->}\ e_{i1} \\ \ldots \\ \texttt{|}\ g_{im_i}\quad \texttt{->}\ e_{im_i} \\ \texttt{where}\ decls_i \end{array}$$

(Заметьте, что в синтаксическом правиле для *gd* "`|`" является терминальным символом, а не синтаксическим мета-символом для указания альтернатив.) Каждая альтернатива $p_i\ match_i$ состоит из образца p_i и его сопоставлений $match_i$. Каждое сопоставление, в свою очередь, состоит из последовательности пар стражей g_{ij} и тел e_{ij} (выражений), за которыми следуют необязательные связывания ($decls_i$), чья область видимости распространяется над всеми стражами и выражениями альтернативы. Альтернатива вида

$$pat\ \texttt{->}\ exp\ \texttt{where}\ decls$$

интерпретируется как краткая запись для

$$\begin{array}{l} pat\ \texttt{|}\ \texttt{True}\quad \texttt{->}\ exp \\ \texttt{where}\ decls \end{array}$$

Case-выражение должно иметь по крайней мере одну альтернативу, и каждая альтернатива должна иметь по крайней мере одно тело. Каждое тело должно иметь один и тот же тип, и все выражение должно быть того же типа.

Вычисление case-выражения выполняется посредством сопоставления выражения e отдельным альтернативам. Альтернативы проверяются последовательно, сверху вниз. Если e соответствует образцу в альтернативе, выполняется связывание переменных, сначала указанных в образце, а затем — с помощью $decls_i$ в операторе `where`, связанном с этой альтернативой. Если значение одного из вычисляемых стражей окажется `True`, в том же окружении, что и страж, будет вычислена соответствующая правая часть. Если значения всех стражей окажутся `False`, процесс сопоставления с образцом будет возобновлен со следующей альтернативы. Если не удастся сопоставить ни один образец,

результатом будет $\bot$. Сопоставление с образцом описано в разделе 3.17, а формальная семантика case-выражений — в разделе 3.17.3.

Замечание о разборе. Выражение

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

нелегко правильно интерпретировать при разборе. Оно имеет единственную однозначную интерпретацию, а именно:

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

Тем не менее, выражение `Bool -> a` является синтаксически правильным типом, и синтаксические анализаторы с ограниченным предварительным просмотром могут выбрать этот неправильный вариант, и тогда программа будет признана недопустимой. Поэтому мы советуем программистам избегать использования стражей, которые заканчиваются указанием сигнатуры типа, именно поэтому *gd* содержит exp^0, а не не *exp*.

3.14 Do-выражения

exp	→	`do {` *stmts* `}`	(do-выражение)
stmts	→	$stmt_1 \ \ldots \ stmt_n$ *exp* [;]	($n \geq 0$)
stmt	→	*exp* ;	
	|	*pat* `<-` *exp* ;	
	|	`let` *decls* ;	
	|	;	(*пустая инструкция*)

Перевод:
выражение →
  `do {` *список-инструкций* `}`
    (do-выражение)
список-инструкций →
  $инструкция_1 \ \ldots \ инструкция_n$ *выражение* [;]
    ($n \geq 0$)
инструкция →
  выражение ;
  | *образец* `<-` *выражение* ;
  | `let` *список-объявлений* ;
  | ;
    (*пустая инструкция*)

Do-выражения предоставляют более удобный синтаксис для монадического программирования. Оно позволяет записать такое выражение

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

в более традиционном виде:

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

Трансляция: Для do-выражений выполняются следующие тождества, которые, после удаления пустых *stmts*, можно использовать в качестве трансляции в ядро:

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

Пропуски `"..."` обозначают генерируемое компилятором сообщение об ошибке, передаваемое функции `fail`, желательно давая некоторое указание на местоположение ошибки сопоставления с образцом; функции `>>`, `>>=` и `fail` являются операциями в классе `Monad`, определенными в Prelude; `ok` является новым идентификатором.

Как показано в трансляции `do`, переменные, связанные `let`, имеют полностью полиморфные типы, тогда как те переменные, которые определены с помощью `<-`, являются связанными лямбда-выражением и поэтому являются мономорфными.

3.15 Типы данных с именованными полями

Объявление типа данных может содержать необязательные определения имен полей (см. раздел 4.2.1). Эти имена полей можно использовать для создания, извлечения и обновления полей способом, который не зависит от всей структуры типа данных.

Различные типы данных не могут совместно использовать общие имена полей в одной области видимости. Имя поля можно использовать не более одного раза в конструкторе. В пределах типа данных, тем не менее, имя поля можно использовать в более чем одном конструкторе, при условии, что поле имеет один и тот же тип во всех конструкторах. Для того чтобы проиллюстрировать последнее замечание, рассмотрим:

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

Здесь `S` является допустимым типом данных, а `T` — нет, потому что в последнем случае для `y` указан другой тип, противоречащий указанному ранее.

3.15.1 Извлечение полей

aexp → *qvar*

Перевод:
выражение-аргумента →
  квалифицированная-переменная

Имена полей используются в качестве селекторной функции. Когда имя поля используется в качестве переменной, оно действует как функция, которая извлекает поле из объекта. Селекторы являются связываниями верхнего уровня, и поэтому они могут быть перекрыты локальными переменными, но не могут конфликтовать с другими связываниями верхнего уровня с тем же именем. Это сокрытие затрагивает только селекторные функции, при создании записей (раздел 3.15.2) и их обновлении (раздел 3.15.3) имена полей не могут быть спутаны с обычными переменными.

Трансляция: Имя поля f представляет собой селекторную функцию в соответствии с определением:

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

где все $C_1\ \ldots\ C_n$ — конструкторы типа данных, содержащие поле с именем f, p_{ij} — это `y`, когда f именует собой j-ю компоненту C_i, или `_` иначе, а e_i — это `y`, когда некоторое поле в C_i имеет имя f, или `undefined` иначе.

3.15.2 Создание типов данных с использованием имен полей

aexp → *qcon* `{` $fbind_1$ `,` $\ldots$ `,` $fbind_n$ `}` (именованная конструкция, $n \geq 0$)

fbind → *qvar* `=` *exp*

Перевод:
выражение-аргумента →
  квалифицированный-конструктор
    `{` *связывание-имени-поля*$_1$ `,` $\ldots$ `,` *связывание-имени-поля*$_n$ `}`
    (именованная конструкция, $n \geq 0$)
связывание-имени-поля →
  квалифицированная-переменная `=` *выражение*

Конструктор с именованными полями может использоваться для создания значения, в котором компоненты задаются именем, а не позицией. В отличие от фигурных скобок, используемых в списках объявлений, здесь присутствие фигурных скобок не зависит от размещения текста, символы `{` и `}` должны использоваться явно. (То же самое относится к обновлению полей и образцам полей.) Создание объектов с использованием имен полей подчинено следующим ограничениям:

- Могут использоваться только имена полей, объявленные в заданном конструкторе.
- Имя поля не может быть использовано более одного раза.
- Поля, которые не используются, инициализируются значением $\bot$.
- Когда какое-нибудь из обязательных полей (полей, чьи типы объявлены с префиксом `!`) оказывается пропущенным во время создания объекта, возникает ошибка компиляции. Обязательные поля рассматриваются в разделе 4.2.1.

Выражение `F {}`, где `F` — конструктор данных, является допустимым независимо от того, *было или нет* `F` *объявлено с использованием синтаксиса записи* (при условии, что `F` не имеет обязательных полей, см. пункт третий в приведенном выше списке); оно обозначает `F` $\bot_1$ ... $\bot_n$, где n — число аргументов `F`.

Трансляция: В связывании f `=` v поле f именует v.

$$C \texttt{ \{ } bs \texttt{ \} } \quad = \quad C\ (pick_1^C\ bs\ \texttt{undefined})\ \ldots\ (pick_k^C\ bs\ \texttt{undefined})$$

где k — число аргументов C.
Вспомогательная функция $pick_i^C\ bs\ d$ определена следующим образом:

> Если i-ый компонент конструктора C имеет имя поля f и если $f = v$ появляется в списке связываний bs, то $pick_i^C\ bs\ d$ равно v. Иначе $pick_i^C\ bs\ d$ равно значению по умолчанию d.

3.15.3 Обновления с использованием имен полей

aexp → *aexp*$_{\langle qcon \rangle}$ `{` *fbind*$_1$ `,` ... `,` *fbind*$_n$ `}` (именованное обновление, $n \geq 1$)

Перевод:
выражение-аргумента →
выражение-аргумента$_{\langle квалифицированный\text{-}конструктор \rangle}$
`{` *связывание-имени-поля*$_1$ `,` ... `,` *связывание-имени-поля*$_n$ `}`
(именованное обновление, $n \geq 1$)

Значения, принадлежащие типу данных с именованными полями, можно обновлять, не боясь разрушить структуру данных. При этом создается новое значение, в котором заданные значения полей замещают те, что были в существующем значении. Обновления подчиняются следующим правилам:

- Все имена должны быть взяты из одного типа данных.
- По меньшей мере один конструктор должен определять все имена, упомянутые в обновлении.
- Ни одно имя не может быть упомянуто более одного раза.
- Если обновляемое значение не содержит все указанные имена, в ходе выполнения возникнет ошибка.

Трансляция: Используя предыдущее определение функции *pick*,

$$
\begin{aligned}
e\ \texttt{\{}\ bs\ \texttt{\}} \quad = \quad & \texttt{case}\ e\ \texttt{of} \\
& C_1\ v_1\ \ldots\ v_{k_1}\ \texttt{->}\ C_1\ (pick_1^{C_1}\ bs\ v_1)\ \ldots\ (pick_{k_1}^{C_1}\ bs\ v_{k_1}) \\
& \quad \ldots \\
& C_j\ v_1\ \ldots\ v_{k_j}\ \texttt{->}\ C_j\ (pick_1^{C_j}\ bs\ v_1)\ \ldots\ (pick_{k_j}^{C_j}\ bs\ v_{k_j}) \\
& \texttt{\_ -> error "Ошибка обновления"}
\end{aligned}
$$

где $\{C_1, \ldots, C_j\}$ — набор конструкторов, содержащих все имена в bs, а k_i — число аргументов C_i.

Вот некоторые примеры, использующие именованные поля:

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

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

Поле `f1` является общим для обоих конструкторов в T. Этот пример транслирует выражения, использующие конструкторы, записываемые с именами полей, в эквивалентные выражения, использующие те же самые конструкторы без имен полей. Если не будет единого конструктора, который определяет набор имен полей в обновлении, такого как `x {f2 = 1, f3 = 'x'}`, произойдет ошибка компиляции.

3.16 Сигнатуры типов выражений

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

Перевод:
выражение →
выражение `::` [*контекст* `=>`] *тип*

Сигнатуры типов выражений имеют вид e `::` t, где e — выражение, а t — тип (раздел 4.1.2); они используются для явного указания типа выражения, в частности, для того чтобы разрешить неоднозначность типов из-за перегрузки (см. раздел 4.3.4). Значением выражения является значение *exp*. Как и с обычными сигнатурами типов (см. раздел 4.4.1), объявленный тип может быть более частным, чем основной тип, выводимый из *exp*, но будет ошибкой указать тип, который окажется более общим или не сопоставимым с основным типом.

Трансляция:

$$e \texttt{ :: } t \quad = \quad \texttt{let \{ } v \texttt{ :: } t\texttt{; } \quad v \texttt{ = } e \texttt{ \} in } v$$

3.17 Сопоставление с образцом

Образцы появляются в лямбда-абстракциях, определениях функций, связываниях с образцом, описаниях списков, do-выражениях и case-выражениях. Тем не менее, первые пять из них в конечном счете транслируются в case-выражения, поэтому достаточно ограничиться определением семантики сопоставления с образцом для case-выражений.

3.17.1 Образцы

Образцы имеют следующий синтаксис:

pat	→	*var* `+` *integer*	(образец упорядочивания)
	|	pat^0	
pat^i	→	pat^{i+1} [$qconop^{(\mathrm{n},i)}$ pat^{i+1}]	
	|	$lpat^i$	
	|	$rpat^i$	
$lpat^i$	→	($lpat^i$ | pat^{i+1}) $qconop^{(\mathrm{l},i)}$ pat^{i+1}	
$lpat^6$	→	`-` (*integer* | *float*)	(отрицательный литерал)
$rpat^i$	→	pat^{i+1} $qconop^{(\mathrm{r},i)}$ ($rpat^i$ | pat^{i+1})	

pat^{10}	$\rightarrow$	*apat*	
	|	*gcon apat*$_1$ ... *apat*$_k$	(число аргументов конструктора *gcon* $=$ k, $k \geq 1$)
apat	$\rightarrow$	*var* [@ *apat*]	("такой как"-образец)
	|	*gcon*	(число аргументов конструктора *gcon* $=$ 0)
	|	*qcon* { *fpat*$_1$, ... , *fpat*$_k$ }	(именованный образец, $k \geq 0$)
	|	*literal*	
	|	_	(любые символы)
	|	(*pat*)	(образец в скобках)
	|	(*pat*$_1$, ... , *pat*$_k$)	(образец кортежа, $k \geq 2$)
	|	[*pat*$_1$, ... , *pat*$_k$]	(образец списка, $k \geq 1$)
	|	~ *apat*	(неопровержимый образец)
fpat	$\rightarrow$	*qvar* = *pat*	

Перевод:

образец $\rightarrow$
  переменная + *целый-литерал*
    (образец упорядочивания)
  | *образец*^{0}
образец^i $\rightarrow$
  образец$^{i+1}$ [*квалифицированный-оператор-конструктора*$^{(\mathrm{n},i)}$ *образец*$^{i+1}$]
  | *левый-образец*^{i}
  | *правый-образец*^{i}
левый-образец^i $\rightarrow$
  (*левый-образец*^{i} | *образец*$^{i+1}$) *квалифицированный-оператор-конструктора*$^{(\mathrm{l},i)}$
    образец$^{i+1}$
левый-образец^6 $\rightarrow$
  - (*целый-литерал* | *литерал-с-плавающей-точкой*)
    (отрицательный литерал)
правый-образец^i $\rightarrow$
  образец$^{i+1}$ *квалифицированный-оператор-конструктора*$^{(\mathrm{r},i)}$
    (*правый-образец*^{i} | *образец*$^{i+1}$)
образец^{10} $\rightarrow$
  такой-как-образец
  | *общий-конструктор такой-как-образец*$_1$ ... *такой-как-образец*$_k$

(число аргументов конструктора $gcon \ = \ k, \ k \geq 1$)

такой-как-образец →
  переменная [@ *такой-как-образец*]
    ("такой как"-образец)
  | *общий-конструктор*
    (число аргументов конструктора $gcon \ = \ 0$)
  | *квалифицированный-конструктор*
    { *образец-с-именем*$_1$, ... , *образец-с-именем*$_k$ }
    (именованный образец, $k \geq 0$)
  | *литерал*
  | _
    (любые символы)
  | (*образец*)
    (образец в скобках)
  | (*образец*$_1$, ... , *образец*$_k$)
    (образец кортежа, $k \geq 2$)
  | [*образец*$_1$, ... , *образец*$_k$]
    (образец списка, $k \geq 1$)
  | ~ *такой-как-образец*
    (неопровержимый образец)

образец-с-именем →
  квалифицированная-переменная = *образец*

Число аргументов конструктора должно соответствовать числу образцов, связанных с ним; нельзя сопоставлять частично примененный конструктор.

Все образцы должны быть *линейными* : ни одна переменная не может появляться более одного раза. Например, следующее определение недопустимо:

```
f (x,x) = x   -- ЗАПРЕЩЕНО; x дважды используется в образце
```

Образцы вида *var*@*pat* называются *"такими как"-образцами*, и позволяют использовать *var* в качестве имени для значения, сопоставляемого *pat*. Например,

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

эквивалентено

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

Образцы вида _ обозначают *группы любых символов* и полезны, когда некоторая часть образца не используется в правой части. Это как если бы идентификатор, не используемый где-либо в другом месте, был помещен на свое место. Например,

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

эквивалентно

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

3.17.2 Неформальная семантика сопоставления с образцом

Образцы сопоставляются значениям. Попытка сопоставить образец может иметь один из трех результатов: *потерпеть неудачу*, *иметь успех*, при этом каждая переменная в образце связывается с соответствующим значением, или *быть отклонена* (т.е. вернуть $\perp$). Сопоставление с образцом выполняется слева направо и извне вовнутрь, в соответствии со следующими правилами:

1. Сопоставление образца *var* значению v всегда имеет успех и связывает *var* с v.

2. Сопоставление образца `~`*apat* значению v всегда имеет успех. Свободные переменные в *apat* связываются с соответствующими значениями, если сопоставление *apat* с v завершится успешно, или с $\perp$, если сопоставление *apat* с v потерпит неудачу или будет отклонено. (Связывание *не* подразумевает вычисление.)

   С точки зрения операций, это означает, что никакое сопоставление не будет сделано с образцом `~`*apat* до тех пор, пока одна из переменных в *apat* не будет использована. В этот момент весь образец сопоставляется значению, и если сопоставление потерпит неудачу или будет отклонено, так выполняется все вычисление.

3. Сопоставление образца `_` любому значению всегда имеет успех, при этом никаких связываний не происходит.

4. Сопоставление образца *con pat* значению, где *con* — конструктор, определенный с помощью `newtype`, зависит от значения:

   - Если значение имеет вид *con* v, то *pat* сопоставляется v.
   - Если значением является $\perp$, то *pat* сопоставляется $\perp$.

   То есть конструкторы, связанные с `newtype`, служат только для того, чтобы изменить тип значения.

5. Сопоставление образца *con* $pat_1 \ldots pat_n$ значению, где *con* — конструктор, определенный с помощью `data`, зависит от значения:

   - Если значение имеет вид *con* $v_1 \ldots v_n$, части образца сопоставляются слева направо компонентам значения данных, если все сопоставления

завершатся успешно, результатом всего сопоставления будет успех; первая же неудача или отклонение приведут к тому, что сопоставление с образцом соответственно потерпит неудачу или будет отклонено.

- Если значение имеет вид $con'\ v_1\ \ldots\ v_m$, где con — конструктор, отличный от con', сопоставление потерпит неудачу.
- Если значение равно $\perp$, сопоставление будет отклонено.

6. Сопоставление с конструктором, использующим именованные поля, — это то же самое, что и сопоставление с обычным конструктором, за исключением того, что поля сопоставляются в том порядке, в котором они перечислены (названы) в списке полей. Все перечисленные поля должны быть объявлены конструктором, поля не могут быть названы более одного раза. Поля, которые не названы в образце, игнорируются (сопоставляются с `_`).

7. Сопоставление числового, символьного или строкового литерала k значению v имеет успех, если v `==` k, где `==` перегружен на основании типа образца. Сопоставление будет отклонено, если эта проверка будет отклонена.

   Интерпретация числовых литералов в точности описана в разделе 3.2, то есть перегруженная функция `fromInteger` или `fromRational` применяется к литералу типа `Integer` или `Rational` (соответственно) для преобразования его к соответствующему типу.

8. Сопоставление n`+`k-образца (где n — переменная, а k — положительный целый литерал) значению v имеет успех, если x `>=` k, при этом n связывается с x `-` k, и терпит неудачу иначе. Снова, функции `>=` и `-` являются перегруженными в зависимости от типа образца. Сопоставление будет отклонено, если сравнение будет отклонено.

   Интерпретация литерала k является точно такой же, как и для числовых литералов, за исключением того, что допустимы только целые литералы.

9. Сопоставление “такого как”-образца *var*`@`*apat* значению v является результатом сопоставления *apat* с v, дополненного связыванием *var* с v. Если сопоставление *apat* с v потерпит неудачу или будет отклонено, такой же результат будет у сопоставления с образцом.

Помимо очевидных ограничений статических типов (например, статической ошибкой является сопоставление символа с булевским значением), выполняются следующие ограничения статических классов:

- Целочисленный литеральный образец можно сопоставить только значению класса `Num`.
- Литеральный образец с плавающей точкой можно сопоставить только значению в классе `Fractional`.
- n`+`k-образец можно сопоставить только значению в классе `Integral`.

Многие люди считают, что n+k-образцы не следует использовать. Эти образцы могут быть удалены или изменены в будущих версиях Haskell.

Иногда полезно различать два вида образцов. Сопоставление с *неопровержимым образцом* не является строгим: образец сопоставляется, даже если сопоставляемое значение равно $\perp$. Сопоставление с *опровержимым* образцом является строгим: если сопоставляемое значение равно $\perp$, сопоставление будет отклонено. Неопровержимыми являются следующие образцы: переменная, символ подчеркивания, N *apat*, где N — конструктор, определенный с помощью `newtype`, а *apat* — неопровержимый образец (см. раздел 4.2.3), *var*`@`*apat*, где *apat* — неопровержимый образец, и образцы вида `~`*apat* (независимо от того, является ли *apat* неопровержимым образцом или нет). Все остальные образцы являются *опровержимыми*.

Приведем несколько примеров:

1. Если образец `['a','b']` сопоставляется `['x',`$\perp$`]`, то сопоставление `'a'` с `'x'` потерпит неудачу, и все сопоставление завершится неудачей. Но если `['a','b']` сопоставляется [$\perp$`,'x']`, то попытка сопоставить `'a'` с $\perp$ приведет к тому, что все сопоставление будет отклонено.

2. Эти примеры демонстрируют сопоставление опровержимых и неопровержимых образцов:

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

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

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

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

3. Рассмотрим следующие объявления:

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

   Эти примеры показывают различие между типами, определенными с помощью `data` и `newtype`, при сопоставлении с образцом:

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

   Дополнительные примеры вы найдете в разделе 4.2.3.

Образцы верхнего уровня в case-выражениях и набор образцов верхнего уровня в связываниях имен функций или образцах могут иметь ноль или более связанных с ними *стражей*. Страж — это булево выражение, которое вычисляется только после того, как все аргументы были успешно сопоставлены, и, для того чтобы все сопоставление с образцом имело успех, значением этого выражения должно быть истина. Окружением стража является то же окружение, что и для правой части альтернативы case-выражения, определения функции или связывания с образцом, к которому он прикреплен.

Семантика стража имеет очевидное влияние на характеристики строгости функции или case-выражения. В частности, из-за стража может быть вычислен иной неопровержимый образец . Например, в

```
f :: (Int,Int,Int) -> [Int] -> Int
f ~(x,y,z) [a] | (a == y) = 1
```

и `a`, и `y` будут вычислены с помощью `==` в страже.

3.17.3 Формальная семантика сопоставления с образцом

Семантика всех конструкций сопоставления с образцом, отличных от `case`-выражений, определена с помощью тождеств, которые устанавливают связь этих конструкций с `case` -выражениями. В свою очередь, семантика самих `case`-выражений задана в виде последовательностей тождеств на рис. 3.1–3.2. Любая реализация должна обеспечивать выполнение этих тождеств; при этом не ожидается, что она будет использовать их непосредственно, поскольку это могло бы привести к генерации довольно неэффективного кода.

На рис. 3.1 - 3.2: e, e' и e_i — выражения, g и g_i — булевы выражения, p и p_i — образцы, v, x и x_i — переменные, K и K' — конструкторы алгебраических типов данных (`data`) (включая конструкторы кортежей), а N — конструктор `newtype`.

Правило (b) соответствует основному `case`-выражению исходного языка, независимо от того, включает ли оно стражей: если стражи не указаны, то в формах $match_i$ вместо стражей $g_{i,j}$ будет подставлено `True`. Последующие тождества управляют полученным `case`-выражением все более и более простой формы.

Правило (h) на рис. 3.2 затрагивает перегруженный оператор `==`; именно это правило определяет смысл сопоставления образца с перегруженными константами.

Все эти тождества сохраняют статическую семантику. Правила (d), (e), (j), (q) и (s) используют лямбду, а не `let`; это указывает на то, что переменные, связанные с помощью `case`, являются мономорфными (раздел 4.1.4).

(a) `case` e `of { }` $= (\backslash v$ `->` `case` v `of { }`$)$ e
где v — новая переменная

(b) `case` v `of { }` p_1 $match_1$`;` $\ldots$ `;` p_n $match_n$ `}`
$=$ `case` v `of { }` p_1 $match_1$ `;`
`_ ->` $\ldots$ `case` v `of {`
p_n $match_n$ `;`
`_ -> error "Нет сопоставлений" }...}`
где каждое $match_i$ имеет вид:
`|` $g_{i,1}$ `->` $e_{i,1}$ `;` $\ldots$ `;` `|` g_{i,m_i} `->` e_{i,m_i} `where { }` $decls_i$ `}`

(c) `case` v `of { }` p `|` g_1 `->` e_1 `;` $\ldots$
`|` g_n `->` e_n `where { }` $decls$ `}`
`_ ->` e' `}`
$=$ `case` e' `of`
{y `->` (где y — новая переменная)
`case` v `of {`
p `->` `let { }` $decls$ `in`
`if` g_1 `then` e_1 $\ldots$ `else if` g_n `then` e_n `else` y `;`
`_ ->` y `}}`

(d) `case` v `of { }` `~`p `->` e`;` `_ ->` e' `}`
$=$ $(\backslash x_1 \ldots x_n$ `->` e `)` `(case` v `of { }` p`->` x_1 `})` $\ldots$ `(case` v `of { }` p `->` x_n`})`
где $x_1, \ldots, x_n$ — переменные в p

(e) `case` v `of { }` x`@`p `->` e`;` `_ ->` e' `}`
$=$ `case` v `of { }` p `->` `(` `\` x `->` e `)` v `;` `_ ->` e' `}`

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

Рис. 3.1: Семантика case-выражений, часть 1

(g) `case` v `of { `$K\ p_1 \ldots p_n$` -> `e`; _ -> `e'` }`
`= case` v `of {`
`  `$K\ x_1 \ldots x_n$` -> case `x_1` of {`
`  `p_1` -> ... case `x_n` of { `p_n` -> `e` ; _ -> `e'` } ...`
`  _ -> `e'` }`
`  _ -> `e'` }`
по меньшей мере один из $p_1, \ldots, p_n$ не является переменной;
$x_1, \ldots, x_n$ — новые переменные

(h) `case` v `of { `k` -> `e`; _ -> `e'` } = if (`v`==`k`) then `e` else `e'
где k — числовой, символьный или строковый литерал

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

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

(k) `case` $N\ v$ `of { `$N\ p$` -> `e`; _ -> `e'` }`
`= case` v `of { `p` -> `e`; _ -> `e'` }`
где N — конструктор `newtype`

(l) `case` $\bot$ `of { `$N\ p$` -> `e`; _ -> `e'` } = case` $\bot$ `of { `p` -> `e` }`
где N — конструктор `newtype`

(m) `case` v `of { `K` { `f_1` = `p_1` , `f_2` = `p_2` , ... } -> `e` ; _ -> `e'` }`
`= case` e' `of {`
`  `y` ->`
`  case` v `of {`
`  `K` { `f_1` = `p_1` } ->`
`  case` v `of { `K` { `f_2` = `p_2` , ... } -> `e` ; _ -> `y` };`
`  _ -> `y` }}`
где f_1, f_2, ... — поля конструктора K, y— новая переменная

(n) `case` v `of { `K` { `f` = `p` } -> `e` ; _ -> `e'` }`
`= case` v `of {`
`  `$K\ p_1 \ldots p_n$` -> `e` ; _ -> `e'` }`
где p_i равно p, если f именует i-ую компоненту K, _ иначе

(o) `case` v `of { `K` {} -> `e` ; _ -> `e'` }`
`= case` v `of {`
`  `K` _ ... _ -> `e` ; _ -> `e'` }`

(p) `case (`$K'\ e_1 \ldots e_m$`) of { `$K\ x_1 \ldots x_n$` -> `e`; _ -> `e'` } = `e'
где K и K' — различные конструкторы `data` соответственно с n и m аргументами

(q) `case (`$K\ e_1 \ldots e_n$`) of { `$K\ x_1 \ldots x_n$` -> `e`; _ -> `e'` }`
`= (\`$x_1 \ldots x_n$` -> `e`)` $e_1 \ldots e_n$
где K — конструктор `data` с n аргументами

(r) `case` $\bot$ `of { `$K\ x_1 \ldots x_n$` -> `e`; _ -> `e'` } = `$\bot$
где K — конструктор `data` с n аргументами

(s) `case` v `of { `x`+`k` -> `e`; _ -> `e'` }`
`= if `v` >= `k` then (\`x` -> `e`) (`v`-`k`) else `e'
где k — числовой литерал

Рис. 3.2: Семантика case-выражений, часть 2

Глава 4

Объявления и связывания имен

В этой главе мы опишем синтаксис и неформальную семантику *объявлений* Haskell.

module	→	`module` *modid* [*exports*] `where` *body*	
	|	*body*	
body	→	`{` *impdecls* `;` *topdecls* `}`	
	|	`{` *impdecls* `}`	
	|	`{` *topdecls* `}`	
topdecls	→	$topdecl_1$ `;` ... `;` $topdecl_n$	$(n \geq 1)$
topdecl	→	`type` *simpletype* `=` *type*	
	|	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
	|	`newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]	
	|	`class` [*scontext* `=>`] *tycls tyvar* [`where` *cdecls*]	
	|	`instance` [*scontext* `=>`] *qtycls inst* [`where` *idecls*]	
	|	`default` (*type*$_1$ `,` ... `,` *type*$_n$)	$(n \geq 0)$
	|	*decl*	
decls	→	`{` $decl_1$ `;` ... `;` $decl_n$ `}`	$(n \geq 0)$
decl	→	*gendecl*	
	|	(*funlhs* | pat^0) *rhs*	
cdecls	→	`{` $cdecl_1$ `;` ... `;` $cdecl_n$ `}`	$(n \geq 0)$
cdecl	→	*gendecl*	
	|	(*funlhs* | *var*) *rhs*	
idecls	→	`{` $idecl_1$ `;` ... `;` $idecl_n$ `}`	$(n \geq 0)$
idecl	→	(*funlhs* | *var*) *rhs*	
	|		(пусто)

gendecl	→	*vars* `::` [*context* `=>`] *type*	(сигнатура типа)
	|	*fixity* [*integer*] *ops*	(infix-объявление)
	|		(пустое объявление)
ops	→	op_1 `,` ... `,` op_n	$(n \geq 1)$
vars	→	var_1 `,` ... `,` var_n	$(n \geq 1)$
fixity	→	`infixl` | `infixr` | `infix`	

Перевод:

модуль →
  `module` *идентификатор-модуля* [*список-экспорта*] `where` *тело*
  | *тело*

тело →
  `{` *список-объявлений-импорта* `;` *список-объявлений-верхнего-уровня* `}`
  | `{` *список-объявлений-импорта* `}`
  | `{` *список-объявлений-верхнего-уровня* `}`

список-объявлений-верхнего-уровня →
  $объявление\text{-}верхнего\text{-}уровня_1$ `;` ... `;` $объявление\text{-}верхнего\text{-}уровня_n$
    $(n \geq 1)$

объявление-верхнего-уровня →
  `type` *простой-тип* `=` *тип*
  | `data` [*контекст* `=>`] *простой-тип* `=` *список-конструкций* [*deriving-инструкция*]
  | `newtype` [*контекст* `=>`] *простой-тип* `=` *новая-конструкция*
    [*deriving-инструкция*]
  | `class` [*простой-контекст* `=>`] *класс-типа переменная-типа*
    [`where` *список-объявлений-классов*]
  | `instance` [*простой-контекст* `=>`] *квалифицированный-класс-типа экземпляр*
    [`where` *список-объявлений-экземпляров*]
  | `default` (*тип*$_1$ `,` ... `,` *тип*$_n$)
    $(n \geq 0)$
  | *объявление*

список-объявлений →
  `{` $объявление_1$ `;` ... `;` $объявление_n$ `}`
    $(n \geq 0)$

объявление →
  общее-объявление
  | (*левая-часть-функции* | $образец^0$) *правая-часть*

список-объявлений-классов →
  `{` $объявление\text{-}класса_1$ `;` ... `;` $объявление\text{-}класса_n$ `}`
    $(n \geq 0)$

объявление-класса →
  общее-объявление
  | (*левая-часть-функции* | *переменная*) *правая-часть*

список-объявлений-экземпляров →
  { *объявление-экземпляра*$_1$; ... ; *объявление-экземпляра*$_n$ }
    ($n \geq 0$)
объявление-экземпляра →
  (*левая-часть-функции* | *переменная*) *правая-часть*
  |
    (пусто)

общее-объявление →
  список-переменных :: [*контекст* =>] *тип*
    (сигнатура типа)
  | *ассоциативность* [*целый-литерал*] *список-операторов*
    (infix-объявление)
  |
    (пустое объявление)

список-операторов →
  оператор$_1$, ... , *оператор*$_n$
    ($n \geq 1$)
список-переменных →
  переменная$_1$, ... , *переменная*$_n$
    ($n \geq 1$)
ассоциативность →
  `infixl` | `infixr` | `infix`

Объявления в синтаксической категории *topdecls* (*список-объявлений-верхнего-уровня*) допустимы только на верхнем уровне модуля Haskell (см. главу 5), тогда как *decls* (*список-объявлений*) можно использовать или на верхнем уровне, или во вложенных областях видимости (т.е. в пределах конструкций `let` или `where`).

Для описания мы разделили объявления на три группы: группу определяемых пользователем типов данных, состоящую из объявлений `type`, `newtype` и `data` (раздел 4.2), группу классов типов и перегрузок операторов, состоящую из объявлений `class`, `instance` и `default` (раздел 4.3), и группу вложенных объявлений, состоящую из связываний значений, сигнатур типов и infix-объявлений (раздел 4.4).

В Haskell есть несколько примитивных типов данных, которые являются "зашитыми" (такие как целые числа и числа с плавающей точкой), но большинство "встроенных" типов данных определено с помощью обычного кода на Haskell с использованием обычных объявлений `type` и `data`. Эти "встроенные" типы данных подробно описаны в разделе 6.1.

4.1 Обзор типов и классов

Haskell использует традиционную систему полиморфных типов Хиндли-Милнера (Hindley-Milner) для того, чтобы обеспечить статическую семантику типов [3, 5], но система типов была расширена с помощью *классов типов* (или просто *классов*), которые обеспечивают структурированный способ ввести *перегруженные* функции.

Объявление `class` (раздел 4.3.1) вводит новый *класс типа* и перегруженные операции, которые должны поддерживаться любым типом, который является экземпляром этого класса. Объявление `instance` (раздел 4.3.2) объявляет, что тип является *экземпляром* класса и включает определения перегруженных операций, называемых *методами класса*, инстанцированных на названном типе.

Например, предположим, что мы хотим перегрузить операции `(+)` и `negate` на типах `Int` и `Float`. Мы вводим новый класс типов, названный `Num`:

```
class Num a  where          -- упрощенное объявление класса Num
  (+)    :: a -> a -> a     -- (Класс Num определен в Prelude)
  negate :: a -> a
```

Это объявление можно толковать так: "тип `a` является экземпляром класса `Num`, если есть методы класса `(+)` и `negate` заданных типов, определенные на этом типе."

Затем мы можем объявить `Int` и `Float` экземплярами этого класса:

```
instance Num Int  where     -- упрощенный экземпляр Num Int
  x + y       =  addInt x y
  negate x    =  negateInt x

instance Num Float  where   -- упрощенный экземпляр Num Float
  x + y       =  addFloat x y
  negate x    =  negateFloat x
```

где предполагается, что `addInt`, `negateInt`, `addFloat` и `negateFloat` — примитивные, в данном случае, функции, но вообще могут быть любыми определяемыми пользователем функциями. Первое сверху объявление можно толковать так: "`Int` является экземпляром класса `Num` в соответствии с этими определениями (т.е. методами класса) для `(+)` и `negate`. "

Большее количество примеров классов типов можно найти в работах Джонса (Jones) [7] или Уодлера и Блотта (Wadler и Blott) [12]. Термин "класс типа" использовался для описания системы типов в исходном языке Haskell 1.0, а термин "конструктор класса" — для описания расширения исходных классов типов. Больше нет никакой причины использовать два различных термина: в этом описании "класс типа" включает и классы типов исходного языка Haskell, и конструируемые классы, введенные Джонсом (Jones).

4.1.1 Виды

Для того чтобы гарантировать их допустимость, выражения с типами разделены на классы различных *видов*. Каждый вид имеет одну из двух возможных форм:

- Символ $*$ представляет вид, к которому относятся все конструкторы типов с нулевым числом аргументов.
- Если κ_1 и κ_2 являются видами, то $\kappa_1 \rightarrow \kappa_2$ — вид, к которому относятся типы, у которых тип принимаемого аргумента относится к виду κ_1, а тип возвращаемого значения — к виду κ_2.

Правильность выражений с типами проверяется с помощью вывода вида подобно тому, как правильность выражений со значениями проверяется с помощью вывода типа. Однако, в отличие от типов, виды являются полностью неявными и не являются видимой частью языка. Вывод видов рассматривается в разделе 4.6.

4.1.2 Синтаксис типов

type	→	*btype* [`->` *type*]	(тип функции)
btype	→	[*btype*] *atype*	(наложение типов)
atype	→	*gtycon*	
	|	*tyvar*	
	|	`(` $type_1$ `,` ... `,` $type_k$ `)`	(тип кортежа, $k \geq 2$)
	|	`[` *type* `]`	(тип списка)
	|	`(` *type* `)`	(конструктор в скобках)
gtycon	→	*qtycon*	
	|	`()`	(тип объединения)
	|	`[]`	(конструктор списка)
	|	`(->)`	(конструктор функции)
	|	`(,{,})`	(конструкторы кортежей)

Перевод:
mun →
  b-mun [`->` *mun*]
    (тип функции)

b-mun →
  [*b-mun*] *a-mun*

(наложение типов)

а-тип →
  общий-конструктор-типа
  | *переменная-типа*
  | `(` *тип*$_1$ `,` ... `,` *тип*$_k$ `)`
    (тип кортежа, $k \geq 2$)
  | `[` *тип* `]`
    (тип списка)
  | `(` *тип* `)`
    (конструктор в скобках)

общий-конструктор-типа →
  квалифицированный-конструктор-типа
  | `()`
    (тип объединения)
  | `[]`
    (конструктор списка)
  | `(->)`
    (конструктор функции)
  | `(,{,})`
    (конструкторы кортежей)

Синтаксис для выражений с типами в Haskell описан выше. Подобно тому, как значения данных построены с использованием конструкторов данных, значения типов построены из *конструкторов типов*. Как и с конструкторами данных, имена конструкторов типов начинаются с заглавных букв. В отличие от конструкторов данных, инфиксные конструкторы типов не допускаются (отличные от `(->)`).

Основными видами выражений с типами являются следующие:

1. Переменные типов, которые обозначаются идентификаторами, начинающимися со строчной буквы. Вид, к которому относится переменная, определяется неявно из контекста, в котором она появилась.

2. Конструкторы типов. Большинство конструкторов типов обозначаются идентификаторами, начинающимися с заглавной буквы. Например:

   - `Char`, `Int`, `Integer`, `Float`, `Double` и `Bool` являются константами типов и относятся к виду $*$.
   - `Maybe` и `IO` являются конструкторами типов с одним аргументом и рассматриваются как типы, относящиеся к виду $* \rightarrow *$.
   - Объявления `data T ...` или `newtype T ...` добавляют в список типов конструктор типа `T`. Вид, к которому относится тип `T`, определяется с помощью вывода вида.

Для конструкторов определенных встроенных типов предусмотрен специальный синтаксис:

- *Тривиальный тип* обозначается `()` и относится к виду $*$. Он обозначает тип "кортеж с нулевым числом аргументов" и имеет ровно одно значение, которое также обозначается `()` (см. разделы 3.9 и 6.1.5).
- *Тип функции* обозначается `(->)` и относится к виду $* \to * \to *$.
- *Тип списка* обозначается `[]` и относится к виду $* \to *$.
- *Типы кортежей* обозначаются `(,)`, `(,,)` и так далее. Они относятся к видам $* \to * \to *$, $* \to * \to * \to *$ и так далее .

Использование констант `(->)` и `[]` более подробно описано ниже.

3. Наложение типов. Если t_1 — тип, относящийся к виду $\kappa_1 \to \kappa_2$, а t_2 — тип, относящийся к виду κ_1, то $t_1\ t_2$ является выражением с типом, относящимся к виду κ_2.

4. *Тип в скобках* вида `(`t`)` идентичен типу t.

Например, выражение типа `IO a` можно воспринимать как применение константы `IO` к переменной `a`. Поскольку конструктор типа `IO` относится к виду $* \to *$, из этого следует, что и переменная `a`, и все выражение `IO a` должно относиться к виду $*$. Вообще, процесс *вывода вида* (см. раздел 4.6) необходим для того, чтобы установить соответствующие виды для определяемых пользователем типов данных, синонимов типов и классов.

Поддерживается специальный синтаксис, который позволяет записывать выражения с определенными типами с использованием более традиционного стиля:

1. *Тип функции* имеет вид t_1 `->` t_2 и эквивалентен типу `(->)` $t_1\ t_2$. Стрелки функций являются правоассоциативными операциями. Например, `Int -> Int -> Float` означает `Int -> (Int -> Float)`.

2. *Тип кортежа* имеет вид `(`t_1`, ... ,` t_k`)`, где $k \geq 2$, и эквивалентен типу `(,...,)` $t_1\ \ldots\ t_k$, где имеются $k-1$ запятых между круглыми скобками. Он обозначает тип k-кортежей, у которых первая компонента имеет тип t_1, вторая компонента — тип t_2 и так далее (см. разделы 3.8 и 6.1.4).

3. *Тип списка* имеет вид `[`t`]` и эквивалентен типу `[]` t. Он обозначает тип списков с элементами типа t (см. разделы 3.7 и 6.1.3).

Эти специальные синтаксические формы всегда обозначают конструкторы встроенных типов для функций, кортежей и списков, независимо от того, что находится в области видимости. Аналогично, префиксные конструкторы типов `(->)`, `[]`, `()`, `(,)` и так далее всегда обозначают конструкторы встроенных типов; их нельзя ни использовать с квалификаторами, ни указывать в списках импорта или экспорта (глава 5). (Отсюда специальное правило вывода *gtycon* (*общего-конструктора-типа*), описанное выше.)

Несмотря на то, что у типов списков и кортежей специальный синтаксис, их семантика — такая же, как и у эквивалентных определяемых пользователем алгебраических типов данных.

Отметим, что выражения и типы имеют согласующийся синтаксис. Если t_i — тип выражения или образца e_i, то выражения `(\` e_1 `->` e_2`)`, `[`e_1`]` и `(`e_1`,`e_2`)` имеют соответственно типы `(`t_1 `->` t_2`)`, `[`t_1`]` и `(`t_1`,`t_2`)`.

За одним исключением (переменной типа в объявлении класса (раздел 4.3.1)), все переменные типов в выражении с типами Haskell предполагаются стоящими под квантором всеобщности; квантор всеобщности [3] не указывается явно, для этого нет специального синтаксиса. Например, выражение `a -> a` обозначает тип $\forall\ a.\ a \rightarrow a$. Тем не менее, для ясности, мы часто записываем кванторы явно при обсуждении типов программ на Haskell. Когда мы записываем тип с явным использованием квантора, область действия квантора $\forall$ (*для всех*) простирается вправо насколько возможно, например, $\forall\ a.\ a \rightarrow a$ означает $\forall\ a.\ (a \rightarrow a)$.

4.1.3 Синтаксис утверждений классов и контекстов

context	→	*class*	
	|	(*class*$_1$, ... , *class*$_n$)	$(n \geq 0)$
class	→	*qtycls tyvar*	
	|	*qtycls* (*tyvar atype*$_1$ ... *atype*$_n$)	$(n \geq 1)$
qtycls	→	[*modid* .] *tycls*	
tycls	→	*conid*	
tyvar	→	*varid*	

Перевод:

контекст →
  класс
  | (*класс*$_1$, ... , *класс*$_n$)
    $(n \geq 0)$
класс →
  квалифицированный-класс-типа переменная-типа
  | *квалифицированный-класс-типа* (*переменная-типа а-тип*$_1$ ... *а-тип*$_n$)
    $(n \geq 1)$
квалифицированный-класс-типа →
  [*идентификатор-модуля* .] *класс-типа*
класс-типа →
  идентификатор-конструктора
переменная-типа →
  идентификатор-переменной

Утверждение класса имеет вид *qtycls tyvar* и указывает на то, что тип *tyvar* является элементом класса *qtycls*. Идентификатор класса начинается с заглавной буквы. *Контекст* состоит из нуля или более утверждений класса и имеет общий вид

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

где C_1, ..., C_n — идентификаторы класса, и каждый из u_1, ..., u_n является или переменной типа, или применением переменной типа к одному или более типам. Внешние круглые скобки можно опустить при $n = 1$. Вообще, мы используем cx для обозначения контекста и мы записываем cx `=>` t для указания того, что тип t ограничен контекстом cx. Контекст cx должен содержать только переменные типов, упомянутые в t. Для удобства мы записываем cx `=>` t, даже если контекст cx пуст, хотя в этом случае конкретный синтаксис не содержит `=>`.

4.1.4 Семантика типов и классов

В этом разделе мы дадим неформальные детали системы типов. (Уодлер (Wadler) и Блотт (Blott) [12] и Джонс (Jones) [7] рассматривают соответственно классы типов и конструируемые классы более подробно.)

Система типов в Haskell приписывает *тип* каждому выражению в программе. Вообще, тип имеет вид $\forall\ \overline{u}.\ cx\ \Rightarrow\ t$, где $\overline{u}$ — набор переменных типов u_1, ..., u_n. В любом таком типе любая из стоящих под квантором всеобщности переменных типов u_i, которая является свободной в cx, должна также быть свободной в t. Кроме того, контекст cx должен иметь вид, описанный выше в разделе 4.1.3. В качестве примера приведем некоторые допустимые типы:

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

В третьем типе ограничение `Eq (f a)` не может быть упрощено, потому что `f` стоит под квантором всеобщности.

Тип выражения e зависит от *окружения типа*, которое задает типы для свободных переменных в e, и *окружения класса*, которое объявляет, какие типы являются экземплярами каких классов (тип становится экземпляром класса только посредством наличия объявления `instance` или инструкции `deriving`).

Типы связаны прямым порядком обобщения (указан ниже); наиболее общий тип, с точностью до эквивалентности, полученный посредством обобщения, который можно присвоить конкретному выражению (в заданном окружении), называется его *основным типом*. В системе типов в языке Haskell, которая является расширением системы Хиндли-Милнера (Hindley-Milner), можно вывести основной тип всех выражений, включая надлежащее использование перегруженных методов класса (хотя, как описано в разделе 4.3.4, при этом могут возникнуть определенные неоднозначные перегрузки).

Поэтому использование явных указаний типов (называемые *сигнатурами типов*) обычно является необязательным (см. разделы 3.16 и 4.4.1).

Тип $\forall\ \overline{u}.\ cx_1 \Rightarrow t_1$ является *более общим чем* тип $\forall\ \overline{w}.\ cx_2 \Rightarrow t_2$, если и только если существует подстановка S с областью определения $\overline{u}$, такая что:

- t_2 идентично $S(t_1)$.
- Всякий раз, когда cx_2 выполняется в окружении класса, $S(cx_1)$ также выполняется.

Значение типа $\forall\ \overline{u}.\ cx \Rightarrow t$ можно инстанцировать на типах $\overline{s}$, если и только если выполняется контекст $cx[\overline{s}/\overline{u}]$. Например, рассмотрим функцию `double`:

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

Наиболее общим типом для `double` является $\forall\ a.\ \texttt{Num}\ a \Rightarrow a \rightarrow a$. `double` можно применять к значениям типа `Int` (инстанцирование a к `Int`), так как выполняется `Num Int`, потому что `Int` является экземпляром класса `Num`. Однако `double` обычно нельзя применять к значениям типа `Char`, потому что `Char` обычно не является экземпляром класса `Num`. Пользователь может предпочесть объявить такой экземпляр, в этом случае `double` можно действительно применить к `Char`.

4.2 Типы данных, определяемые пользователем

В этом разделе мы опишем алгебраические типы данных (объявления `data`), переименованные типы данных (объявления `newtype`) и синонимы типов (объявления `type`). Эти объявления можно использовать только на верхнем уровне модуля.

4.2.1 Объявления алгебраических типов данных

topdecl	→	`data` [*context* `=>`] *simpletype* `=` *constrs* [*deriving*]	
simpletype	→	*tycon tyvar*$_1$ ... *tyvar*$_k$	($k \geq 0$)
constrs	→	*constr*$_1$ `|` ... `|` *constr*$_n$	($n \geq 1$)
constr	→	*con* [`!`] *atype*$_1$ ... [`!`] *atype*$_k$	(число аргументов конструктора *con* $= k$, $k \geq 0$)
		(*btype* | `!` *atype*) *conop* (*btype* | `!` *atype*)	(инфиксный оператор *conop*)

	|	*con* `{` *fielddecl*$_1$ `,` ... `,` *fielddecl*$_n$ `}` $(n \geq 0)$
fielddecl	$\rightarrow$	*vars* `::` (*type* | `!` *atype*)
deriving	$\rightarrow$	`deriving` (*dclass* | (*dclass*$_1$ `,` ... `,` *dclass*$_n$)) $(n \geq 0)$
dclass	$\rightarrow$	*qtycls*

Перевод:

объявление-верхнего-уровня $\rightarrow$
  `data` [*контекст* `=>`] *простой-тип* `=` *список-конструкций*
    [*deriving-инструкция*]

простой-тип $\rightarrow$
  конструктор-типа переменная-типа$_1$ ... *переменная-типа*$_k$
    $(k \geq 0)$

список-конструкций $\rightarrow$
  конструкция$_1$ `|` ... `|` *конструкция*$_n$
    $(n \geq 1)$
конструкция $\rightarrow$
  конструктор [`!`] *a-тип*$_1$ ... [`!`] *a-тип*$_k$
    (число аргументов конструктора *con* $= k$, $k \geq 0$)
  | (*b-тип* | `!` *a-тип*) *оператор-конструктора* (*b-тип* | `!` *a-тип*)
    (инфиксный оператор *сопор*)
  |*конструктор* `{` *объявление-поля*$_1$ `,` ... `,` *объявление-поля*$_n$ `}`
    $(n \geq 0)$
объявление-поля $\rightarrow$
  список-переменных `::` (*тип* | `!` *a-тип*)

deriving-инструкция $\rightarrow$
  `deriving` (*производный-класс* |
    (*производный-класс*$_1$, ... , *производный-класс*$_n$))
    $(n \geq 0)$
производный-класс $\rightarrow$
  квалифицированный-класс-типа

Приоритет для *constr* (*конструкции*) — тот же, что и для выражений: применение обычного конструктора имеет более высокий приоритет, чем применение инфиксного конструктора (таким образом, `a : Foo a` интерпретируется при разборе как `a : (Foo a)`).

Объявление алгебраического типа данных имеет вид:

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

где cx — контекст. Это объявление вводит новый *конструктор типа* T с одной или более составляющими *конструкторами данных* K_1, ..., K_n. В этом описании неуточненный термин "конструктор" всегда означает "конструктор данных".

Типы конструкторов данных задаются следующим образом:

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

где cx_i — наибольшее подмножество cx , которое содержит только те переменные типов, котрые являются свободными в типах t_{i1}, ..., t_{ik_i}. Переменные типов u_1, ..., u_k должны быть различными и могут появляться в cx и t_{ij}; если любая другая переменная типа появится в cx или в правой части — возникнет статическая ошибка. Новая константа типа T относится к виду $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow *$, где виды κ_i, к которым относятся переменные аргументов u_i, устанавливаются с помощью выводы вида, описанного в разделе 4.6. Это означает, что T можно использовать в выражениях с типами с любым числом аргументов от *0* до k.

Например, объявление

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

вводит конструктор типа `Set`, который относится к виду $* \rightarrow *$, и конструкторы `NilSet` и `ConsSet` с типами

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

В данном примере перегруженный тип для `ConsSet` гарантирует, что `ConsSet` можно применить только к значениям типа, который является экземпляром класса `Eq`. Сопоставление с образцом `ConsSet` также приводит к ограничению `Eq a`. Например:

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

Функция `f` имеет установленный с помощью вывода тип `Eq a => Set a -> a`. Контекст в объявлении `data` вообще не имеет никакого другого результата.

Видимость конструкторов типов данных (т.е. "абстрактность" типа данных) вне модуля, в котором определен тип данных, управляется посредством указания имени типа данных в списке экспорта, описанном в разделе 5.8.

Необязательная в объявлении `data` часть `deriving` должна относиться к *производным экземплярам* и описана в разделе 4.3.3.

Именованные поля Конструктор данных с k аргументами создает объект с k компонентами. К этим компонентам обычно обращаются исходя из их позиций, как с аргументами конструктора в выражениях или образцах. Для больших типов данных полезно присваивать *имена полей* компонентам объекта данных. Это

позволяет ссылаться на определенное поле независимо от его размещения в пределах конструктора.

Определение конструктора в объявлении `data` может присвоить имена полям конструктора, используя синтаксис записи (`C { ... }`). Конструкторы, использующие имена полей, можно свободно смешивать с конструкторами без них. Конструктор со связанными именами полей можно по-прежнему использовать как обычный конструктор; использование имен просто является краткой записью для операций, использующих лежащий в основе позиционный конструктор. Аргументы позиционного конструктора находятся в том же порядке, что и именованные поля. Например, объявление

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

определяет тип и конструктор, идентичный тому, который соответствует определению

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

Операции, использующие имена полей, описаны в разделе 3.15. Объявление `data` может использовать то же самое имя поля во множестве конструкторов до тех пор, пока тип поля, после раскрытия синонимов, остается одним и тем же во всех случаях. Одно и то же имя не могут совместно использовать несколько типов в области видимости. Имена полей совместно используют пространство имен верхнего уровня вместе с обычными переменными и методами класса и не должны конфликтовать с другими именами верхнего уровня в области видимости.

Образец `F {}` соответствует любому значению, построенному конструктором `F`, *независимо от того, был объявлен* `F` *с использованием синтаксиса записи или нет.*

Флажки строгости Всякий раз, когда применяется конструктор данных, каждый аргумент конструктора вычисляется, если и только если соответствующий тип в объявлении алгебраического типа данных имеет флажок строгости, обозначаемый восклицательным знаком "`!`". С точки зрения лексики, "`!`" — обычный *varsym* (*символ-переменной*), а не *reservedop* (*зарезервированный-оператор*), он имеет специальное значение только в контексте типов аргументов объявления data.

Трансляция: Объявление вида

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

где каждый s_i имеет вид $\texttt{!}t_i$ или t_i, замещает каждое вхождение K в выражении на

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

где op_i — нестрогое применение функции `$`, если s_i имеет вид t_i, а op_i — строгое применение функции `$!` (см. раздел 6.2), если s_i имеет вид $\texttt{!}\ t_i$. На сопоставление с образцом в K не влияют флажки строгости.

4.2.2 Объявление синонимов типов

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

Перевод:
объявление-верхнего-уровня →
  `type` *простой-тип* `=` *тип*
простой-тип →
  конструктор-типа переменная-типа$_1$ ... *переменная-типа*$_k$
    $(k \geq 0)$

Объявление синонима типа вводит новый тип, который эквивалентен старому типу. Объявление имеет вид

`type` $T\ u_1\ \ldots\ u_k$ `=` t

и вводит конструктор нового типа — T. Тип $(T\ t_1\ \ldots t_k)$ эквивалентен типу $t[t_1/u_1,\ \ldots,\ t_k/u_k]$. Переменные типа $u_1, \ldots, u_k$ должны быть различными и находятся в области видимости только над t; если любая другая переменная типа появится в t — возникнет статическая ошибка. Конструктор нового типа T относится к виду $\kappa_1 \to \ldots \to \kappa_k \to \kappa$, где виды κ_i, к которым относятся аргументы u_i, и κ в правой части t устанавливаются с помощью вывода вида, описанного в разделе 4.6. Например, следующее определение можно использовать для того, чтобы обеспечить альтернативный способ записи конструктора типа список:

```
type List = []
```

Символы конструкторов типов T, введенные с помощью объявлений синонимов типов, нельзя применять частично; если использовать T без полного числа аргументов — возникнет статическая ошибка.

Хотя рекурсивные и взаимно рекурсивные типы данных допустимы, это не так для синонимов типов, *пока не вмешаются алгебраические типы данных.* Например,

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

допустимы, тогда как

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

— нет. Аналогично, `type Rec a = [Rec a]` недопустим.

Синонимы типов представляют собой удобный, но строго синтаксический механизм, который делает сигнатуры более читабельными. Синоним и его определение — полностью взаимозаменяемы, за исключением типа экземпляра в объявлении `instance` (раздел 4.3.2).

4.2.3 Переименования типов данных

topdecl → `newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]
newconstr → *con atype*
| *con* `{` *var* `::` *type* `}`
simpletype → *tycon tyvar*$_1$ ... *tyvar*$_k$ $(k \geq 0)$

Перевод:
объявление-верхнего-уровня →
`newtype` [*контекст* `=>`] *простой-тип* `=` *новая-конструкция*
[*deriving-инструкция*]
новая-конструкция →
конструктор а-тип
| *конструктор* `{` *переменная* `::` *тип* `}`
простой-тип →
конструтор-типа переменная-типа$_1$ ... *переменная-типа*$_k$
$(k \geq 0)$

Объявление вида

`newtype` *cx* `=>` *T u*$_1$ ... *u*$_k$ `=` *N t*

вводит новый тип, который имеет то же самое представление, что и существующий тип. Тип `(`*T u*$_1$ ... *u*$_k$`)` переименовывает тип данных *t*. Он отличается от синонима типа тем, что создает отдельный тип, который должен явно приводиться к исходному типу или обратно. Также, в отличие от синонимов типа, `newtype` может использоваться для определения рекурсивных типов. Конструктор *N* в выражении приводит значение типа *t* к типу `(`*T u*$_1$ ... *u*$_k$`)`. Использование *N* в образце приводит значение типа `(`*T u*$_1$ ... *u*$_k$`)` к типу *t*. Эти приведения типов могут быть реализованы без накладных расходов времени выполнения, `newtype` не меняет лежащее в основе представление объекта.

Новые экземпляры (см. раздел 4.3.2) можно определить для типа, определенного с помощью `newtype`, но нельзя определить для синонима типа. Тип, созданный с помощью `newtype`, отличается от алгебраического типа данных тем, что представление алгебраического типа данных имеет дополнительный уровень непрямого доступа. Это отличие может сделать доступ к представлению менее эффективным. Различие отражено в различных правилах для сопоставления с образцом (см. раздел 3.17). В отличие от алгебраического типа данных, конструктор нового типа данных *N* является *неповышенным*, так что *N* $\perp$ — то же самое, что и $\perp$.

Следующие примеры разъясняют различия между `data` (алгебраическими типами данных), `type` (синонимами типов) и `newtype` (переименованными типами). При данных объявлениях

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

выражения `( d1` $\perp$ `)`, `( d2` $\perp$ `)` и `(d2 (D2` $\perp$ `) )` эквивалентны $\perp$, поскольку `( n` $\perp$ `)`, `( n ( N` $\perp$ `) )`, `( d1 ( D1` $\perp$ `) )` и `( s` $\perp$ `)` эквивалентны `42`. В частности, `( N` $\perp$ `)` эквивалентно $\perp$, тогда как `( D1` $\perp$ `)` неэквивалентно $\perp$.

Необязательная часть deriving объявления `newtype` трактуется так же, как и компонента deriving объявления `data` (см. раздел 4.3.3).

В объявлении `newtype` можно использовать синтаксис для именования полей, хотя, конечно, может быть только одно поле. Таким образом,

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

вводит в область видимости и конструктор, и деструктор:

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

4.3 Классы типов и перегрузка

4.3.1 Объявления классов

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

Перевод:
объявление-верхнего-уровня $\rightarrow$
`class` [*простой-контекст* `=>`] *класс-типа переменная-типа*
`where` *список-объявлений-классов*]

простой-контекст →
  простой-класс
  | (*простой-класс*$_1$, ... , *простой-класс*$_n$)
    $(n \geq 0)$
простой-класс →
  квалифицированный-класс-типа переменная-типа
список-объявлений-классов →
  { *объявление-класса*$_1$; ... ; *объявление-класса*$_n$ }
    $(n \geq 0)$
объявление-класса →
  общее-объявление
| (*левая-часть-функции* | *переменная*) *правая-часть*

Объявление класса вводит новый класс и операции (*методы класса*) над ним. Объявление класса имеет общий вид:

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

Это объявление вводит новый класс C; переменная типа u находится только в области видимости над сигнатурами методов класса в теле класса. Контекст cx определяет суперклассы C, как описано ниже; единственной переменной типа, на которую можно ссылаться в cx, является u.

Отношение суперкласса не должно быть циклическим; т.е. оно должно образовывать ориетированный ациклический граф.

Часть *cdecls* (*список-объявлений-классов*) в объявлении `class` содержит три вида объявлений:

- Объявление класса вводит новые *методы класса* v_i, чья область видимости простирается за пределы объявления `class`. Методы класса в объявлении класса являются в точности v_i, для которых есть явная сигнатура типа

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

  в *cdecls*. Методы класса совместно со связанными переменными и именами полей используют пространство имен верхнего уровня; они не должны конфликтовать с другими именами в области видимости. То есть метод класса не может иметь то же имя, что и определение верхнего уровня, имя поля или другой метод класса.

  Типом метода класса верхнего уровня v_i является:

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

  t_i должен ссылаться на u; оно может ссылаться на переменные типов $\overline{w}$, отличные от u, в этом случае тип v_i является полиморфным в u и $\overline{w}$. cx_i может ограничивать только $\overline{w}$, в частности, cx_i не может ограничивать u. Например,

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

Здесь типом `op` является $\forall\ a,\ b.\ (\texttt{Foo}\ a,\ \texttt{Num}\ b) \Rightarrow a \rightarrow b \rightarrow a$.

- *cdecls* может также содержать *infix-объявления* для любого из методов класса (но не для других значений). Тем не менее, так как методы класса объявлены значениями верхнего уровня, infix-объявления для методов класса могут, в качестве альтернативы, появляться на верхнем уровне, вне объявления класса.

- Наконец, *cdecls* может содержать *методы класса по умолчанию* для любого из v_i. Метод класса по умолчанию для v_i используется, если для него не задано связывание в конкретном объявлении `instance` (см. раздел 4.3.2). Объявление метода по умолчанию является обычным определением значения, за исключением того, что левая часть может быть только переменной или определением функции. Например, объявление

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

  недопустимо, потому что левой частью объявления метода по умолчанию является образец.

В остальных случаях, отличных от описанных здесь, никакие другие объявления не допустимы в *cdecls*.

Объявление `class` без части `where` может оказаться полезным для объединения совокупности классов в больший класс, который унаследует все методы исходных классов. Например,

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

В таком случае, если тип является экземпляром всех суперклассов, это не означает, что он *автоматически* является экземпляром подкласса, даже если подкласс не имеет непосредственных методов класса. Объявление `instance` должно быть задано явно без части `where`.

4.3.2 Объявления экземпляров

topdecl	→	`instance` [*scontext* `=>`] *qtycls inst* [`where` *idecls*]	
inst	→	*gtycon*	
	|	`(` *gtycon* $tyvar_1$ ... $tyvar_k$ `)`	($k \geq 0$, все *tyvar* различны)
	|	`(` $tyvar_1$ `,` ... `,` $tyvar_k$ `)`	($k \geq 2$, все *tyvar* различны)
	|	`[` *tyvar* `]`	
	|	`(` $tyvar_1$ `->` $tyvar_2$ `)`	($tyvar_1$ и $tyvar_2$

различны)

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

объявление-верхнего-уровня →
  `instance` [*простой-контекст* `=>`] *квалифицированный-класс-типа экземпляр*
    [`where` *список-объявлений-экземпляров*]
экземпляр →
  общий-конструктор-типа
  | `(` *общий-конструктор-типа* $переменная\text{-}типа_1$ ... $переменная\text{-}типа_k$ `)`
    ($k \geq 0$, все *переменные-типа* различны)
  | `(` $переменная\text{-}типа_1$ `,` ... `,` $переменная\text{-}типа_k$ `)`
    ($k \geq 2$, все *переменные-типа* различны)
  | `[` *переменная-типа* `]`
  | `(` $переменная\text{-}типа_1$ `->` $переменная\text{-}типа_2$ `)`
    ($переменная\text{-}типа_1$ и $переменная\text{-}типа_2$ различны)
список-объявлений-экземпляров →
  `{` $объявление\text{-}экземпляра_1$ `;` ... `;` $объявление\text{-}экземпляра_n$ `}`
    ($n \geq 0$)
объявление-экземпляра →
  (*левая-часть-функции* | *переменная*) *правая-часть*
  |
    (*пусто*)

Объявление экземпляра вводит экземпляр класса. Пусть

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

является объявлением `class`. Тогда соответствующее объявление экземпляра в общем виде:

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

где $k \geq 0$. Тип $(T\ u_1\ \ldots\ u_k)$ должен иметь вид конструктора типа T, примененного к простым переменным типа $u_1, \ldots\ u_k$; кроме того, T не должен являться синонимом типа, и все u_i должны быть различными.

Поэтому такие объявления экземпляров запрещены:

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

Объявления d могут содержать связывания имен только для методов класса C. Неправильно задавать связывание имен для метода класса, который не находится

в области видимости, но имя, в области которого он находится, несущественно; в частности, это может быть имя с квалификатором. (Это правило идентично тому, что используется для подчиненных имен в списках экспорта, см. раздел 5.2.) Например, это допустимо, даже если `range` находится в области видимости только с именем с квалификатором `Ix.range`.

```
module A where
  import qualified Ix

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

Объявления не могут содержать сигнатуры типов или infix-объявления, поскольку они уже были заданы в объявлении `class`. Как и в случае методов класса по умолчанию (раздел 4.3.1), объявления методов должны иметь вид переменной или определения функции.

Если для некоторого метода класса не заданы связывания имен, тогда используется соответствующий метод класса по умолчанию, заданный в объявлении `class` (если таковое имеется); если такой метод по умолчанию не существует, тогда метод класса этого экземпляра связывается с `undefined`, и ошибка компиляции не возникнет.

Объявление `instance`, которое объявляет тип T экземпляром класса C, называется *объявлением экземпляра* C-T и подчиняется следующим статическим ограничениям:

- Тип не может быть объявлен экземпляром конкретного класса более одного раза в программе.
- Класс и тип должны относиться к одному и тому же виду, его можно установить, используя вывод вида, описанный в разделе 4.6.
- Предположим, что переменные типов в экземпляре типа $(T\ u_1\ \ldots\ u_k)$ удовлетворяют ограничениям в контексте экземпляра cx'. При этом предположении также должны выполняться следующие два условия:

  1. Ограничения, выраженные контекстом $cx[(T\ u1\ \ldots\ uk)/u]$ суперкласса C, должны выполняться. Другими словами, T должен быть экземпляром каждого из суперклассов класса C, и контексты всех экземпляров суперклассов должны следовать из cx'.
  2. Любые ограничения на переменные типов в типе экземпляра, которые требуют, чтобы объявления методов класса в d были хорошо типизированы, также должны выполняться.

  В действительности, за исключением неправильных случаев, возможно вывести из объявления экземпляра самый общий контекст экземпляра cx', при котором будут выполняться два вышеупомянутых ограничения, но, тем не менее, обязательно нужно записать явный контекст экземпляра.

Следующий пример иллюстрирует ограничения, налагаемые экземплярами суперкласса:

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

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

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

Этот пример является правильным в Haskell. Так как `Foo` является суперклассом `Bar`, второе объявление экземпляра будет правильным, только если `[a]` является экземпляром `Foo` исходя из предположения `Num a`. Первое объявление экземпляра действительно сообщает, что `[a]` является экземпляром `Foo` исходя из этого предположения, потому что `Eq` и `Show` являются суперклассами `Num`.

Если, вместо этого, объявления двух экземпляров оказались бы подобны этим:

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

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

тогда программа была бы неправильной. Объявление второго экземпляра будет правильным, только если `[a]` является экземпляром `Foo` исходя из предположений `(Eq a, Show a)`. Но это не выполняется, так как `[a]` является экземпляром `Foo` только исходя из более сильного предположения `Num a`.

Дополнительные примеры объявлений `instance` можно найти в главе 8.

4.3.3 Производные экземпляры

Как упомянуто в разделе 4.2.1, объявления `data` и `newtype` содержат необязательную форму `deriving` . Если форма включена, то для каждого из названных классов типов данных автоматически генерируются *объявления производных экземпляров*. Эти экземпляры подчинены тем же самым ограничениям, что и определяемые пользователем экземпляры. При выведении класса C для типа T, для T должны существовать экземпляры для всех суперклассов класса C, либо посредством явного объявления `instance`, либо посредством включения суперкласса в инструкцию `deriving`.

Производные экземпляры предоставляют удобные, широко используемые операции для определяемых пользователем типов данных. Например, производные экземпляры для типов данных в классе `Eq` определяют операции `==` и `/=`, освобождая программиста от необходимости определять их.

Классами в Prelude, для которых разрешены производные экземпляры, являются: `Eq`, `Ord`, `Enum`, `Bounded`, `Show` и `Read`. Все они приведены на рис. 6.1, стр. 112. Точные детали того, как генерируются производные экземпляры для каждого из этих

классов, даны в главе 10, включая подробное описание того, когда такие производные экземпляры возможны. Классы, определенные стандартными библиотеками, также можно использовать для порождения производных.

Если невозможно произвести объявление `instance` над классом, названном в форме `deriving`, — возникнет статическая ошибка. Например, не все типы данных могут должным образом поддерживать методы класса `Enum`. Также статическая ошибка возникнет в случае, если задать явное объявление `instance` для класса, который также является производным.

Если форма `deriving` в объявлении `data` или `newtype` опущена, то *никакие* объявления экземпляров для этого типа данных не производятся, то есть отсутствие формы `deriving` эквивалентно включению пустой формы: `deriving ()`.

4.3.4 Неоднозначные типы и значения по умолчанию для перегруженных числовых операций

topdecl → `default (`$type_1$ `, ... ,` $type_n$`)` $(n \geq 0)$

Перевод:
объявление-верхнего-уровня →
`default (`mun_1 `, ... ,` mun_n`)`
$(n \geq 0)$

Проблема, связанная с перегрузкой в Haskell, состоит в возможности *неоднозначного типа*. Например, возьмем функции `read` и `show`, определенные в главе 10, и предположим, что именно `Int` и `Bool` являются членами `Read` и `Show`, тогда выражение

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

является неоднозначным, потому что условия, налагаемые на типы для `show` и `read`

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

можно выполнить путем инстанцирования `a` как `Int` или `Bool` в обоих случаях. Такие выражения считаются неправильно типизированными, возникнет статическая ошибка.

Мы говорим, что выражение `e` имеет *неоднозначный тип*, если в его типе $\forall\, \overline{u}.\ cx \Rightarrow t$ есть переменная типа u в $\overline{u}$, которая встречается в cx, но не в t. Такие типы недопустимы.

Например, ранее рассмотренное выражение, включающее `show` и `read`, имеет неоднозначный тип, так как его тип $\forall\ a.$ `Show` a, `Read` $a \Rightarrow$ `String`.

Неоднозначные типы можно обойти с помощью ввода пользователя. Один способ заключается в использовании *сигнатур типов выражений*, описанных в разделе 3.16. Например, для неоднозначного выражения, данного ранее, можно записать:

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

Это устраняет неоднозначность типа.

Иногда предпочтительнее, чтобы неоднозначное выражение было того же типа, что и некоторая переменная, а не заданного с помощью сигнатуры фиксированного типа. В этом состоит назначение функции `asTypeOf` (глава 8): x `` `asTypeOf` `` y имеет значение x, но заставляет x и y иметь один и тот же тип. Например,

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

(Описание `encodeFloat` и `exponent` см. в разделе 6.4.6.)

Неоднозначности в классе `Num` наиболее распространены, поэтому Haskell предоставляет другой способ разрешить их — с помощью *default-объявления*:

`default` (t_1 `,` ... `,` t_n)

где $n \geq 0$, и каждая t_i должна иметь тип, для которого выполняется `Num` t_i. В ситуациях, когда обнаружен неоднозначный тип, переменная неоднозначного типа v является умолчательной, если:

- v появляется только в ограничениях вида C v, где C — класс, и,
- по меньшей мере, один из этих классов является числовым классом (то есть `Num` или подклассом `Num`), и
- все эти классы определены в Prelude или стандартной библиотеке. (На рис. 6.2 - 6.3, стр. 121 - 122 изображены числовые классы, а на рис. 6.1, стр. 112 изображены классы, определенные в Prelude.)

Каждая умолчательная переменная замещается первым типом в default-списке, который является экземпляром всех классов неоднозначной переменной. Если такой тип не будет найден — возникнет статическая ошибка.

В модуле может быть только одно default-объявление, и его влияние ограничено этим модулем. Если в модуле default-объявление не задано, то предполагается, что задано объявление

```
default (Integer, Double)
```

Пустое default-объявление `default ()` отменяет все значения по умолчанию в модуле.

4.4 Вложенные объявления

Следующие объявления можно использовать в любом списке объявлений, включая верхний уровень модуля.

4.4.1 Сигнатуры типов

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

Перевод:
общее-объявление →
  список-переменных `::` [*контекст* `=>`] *тип*
список-переменых →
  $переменная_1$ `,` ... `,` $переменная_n$
  $(n \geq 1)$

Сигнатура типа определяет типы для переменных, возможно по отношению к контексту. Сигнатура типа имеет вид:

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

который эквивалентен утверждению v_i `::` cx `=>` t для каждого i от *1* до n. Каждая v_i должна иметь связанное с ним значение в том же списке объявлений, который содержит сигнатуру типа, т.е. будет неправильным задать сигнатуру типа для переменной, связанной во внешней области видимости. Кроме того, будет неправильным задать более одной сигнатуры типа для одной переменной, даже если сигнатуры идентичны.

Как упомянуто в разделе 4.1.2, каждая переменная типа, появляющаяся в сигнатуре, находится под квантором всеобщности над сигнатурой, и, следовательно, область видимости переменной типа ограничена сигнатурой типа, которая ее содержит. Например, в следующих объявлениях

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

`a` в двух сигнатурах типа совершенно различны. Действительно, эти объявления содержат статическую ошибку, так как `x` не имеет тип $\forall\ a.\ a$. (Тип `x` зависит от типа `f`; в настоящее время в Haskell нет способа указать сигнатуру для переменной с зависимым типом, это раэъясняется в разделе 4.5.4.)

Если данная программа включает сигнатуру для переменной f, тогда каждое использование f трактуется как f, имеющая объявленный тип. Если тот же тип нельзя также вывести для определяемого вхождения f — возникнет статическая ошибка.

Если переменная f определена без соответствующей сигнатуры типа, тогда при использовании f вне его собственной группы объявлений (см. раздел 4.5) переменная трактуется как имеющая соответствующий выведенный или *основной* тип. Тем не менее, для того чтобы гарантировать, что вывод типа еще возможен, определяемое вхождение и все использования f в пределах его группы объявлений должны иметь один и тот же мономорфный тип (из которого основной тип получается путем обобщения, описанного в разделе 4.5.2).

Например, если мы определим

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

тогда основным типом будет `sqr` :: $\forall\, a.$ `Num` $a \Rightarrow a \rightarrow a$. Этот тип позволяет такое применение, как `sqr 5` или `sqr 0.1`. Также правильным будет объявить более специализированный тип, например

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

но теперь такое применение, как `sqr 0.1`, будет неправильным. Такие сигнатуры типов, как

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

являются неправильными, поскольку они являются более общими, чем основной тип `sqr`.

Сигнатуры типов можно также использовать для того, чтобы обеспечить *полиморфную рекурсию*. Следующее определение является неправильным, зато показыает, как можно использовать сигнатуру типа для указания типа, который является более общим, чем тот, который был бы выведен:

```
data T a  =  K (T Int) (T a)

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

Если мы уберем объявление сигнатуры, тип `f` будет выведен как `T Int -> Int` благодаря первому рекурсивному вызову, для которого аргументом `f` является `T Int`. Полиморфная рекурсия позволяет пользователю указать более общую сигнатуру типа `T a -> a`.

4.4.2 Infix-объявления

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

Перевод:
общее-объявление →
  ассоциативность [*целый-литерал*] *список-операторов*
ассоциативность →
  `infixl` | `infixr` | `infix`
список-операторов →
  оператор$_1$, ... , *оператор*$_n$
    ($n \geq 1$)
оператор →
  оператор-переменной
  | *оператор-конструктора*

infix-объявление задает ассоциативность и приоритет (силу связывания) одного или более операторов. Целое число *integer* в infix-объявлении должно быть в диапазоне от *0* до *9*. infix-объявление можно разместить всюду, где можно разместить сигнатуру типа. Как и сигнатура типа, infix-объявление задает свойства конкретного оператора. Так же, как и сигнатура типа, infix-объявление можно разместить только в той же последовательности объявлений, что и объявление самого оператора, и для любого оператора можно задать не более одного infix-объявления. (Методы класса являются небольшим исключением: их infix-объявления можно размещать в самом объявлении класса или на верхнем уровне.)

По способу ассоциативности операторы делятся на три вида: неассоциативные, левоассоциативные и правоассоциативные (`infix`, `infixl` и `infixr` соответственно). По приоритету (силе связывания) операторы делятся на десять групп, в соответствии с уровнем приоритета от 0 до 9 включительно (уровень 0 связывает операнды наименее сильно, а уровень 9 — наиболее сильно). Если целое число *integer* не указано, оператору присваивается уровень приоритета 9. Любой оператор, для которого нет infix-объявления, считается объявленным `infixl 9` (более подробную информацию об использовании infix-объявлений см. в разделе 3). В таблице 4.1 перечислены ассоциативности и приоритеты операторов, определенных в Prelude.

Ассоциативность является свойством конкретного объекта (конструктора или переменной), как и его тип; ассоциативность не является свойством *имени* объекта. Например,

Приоритет	Левоассоциативные операторы	Неассоциативные операторы	Правоассоциативные операторы
9	`!!`		`.`
8			`^`, `^^`, `**`
7	`*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` ``		
6	`+`, `-`		
5			`:`, `++`
4		`==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` ``, `` `notElem` ``	
3			`&&`
2			`||`
1	`>>`, `>>=`		
0			`$`, `$!`, `` `seq` ``

Таблица 4.1: Приоритеты и ассоциативности операторов в Prelude

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

module Foo where
  import qualified Bar
  infix 3 `op`

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

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

Здесь `` `Bar.op` `` — оператор с `infixr 7`, `` `Foo.op` `` — оператор с `infix 3`, а оператор `op` во вложенном определении в правой части `f` имеет заданные по умолчанию `infixl 9`. (Свойства оператора `` `op` `` во вложенном определении можно было бы также задать с помощью вложенного infix-объявления.)

4.4.3 Связывание имен в функциях и образцах

$$
\begin{array}{lcl}
\textit{decl} & \rightarrow & (\textit{funlhs} \mid \textit{pat}^0)\ \textit{rhs} \\
\\
\textit{funlhs} & \rightarrow & \textit{var}\ \textit{apat}\ \{\ \textit{apat}\ \} \\
 & \mid & \textit{pat}^{i+1}\ \textit{varop}^{(a,i)}\ \textit{pat}^{i+1} \\
 & \mid & \textit{lpat}^{i}\ \textit{varop}^{(\mathrm{l},i)}\ \textit{pat}^{i+1} \\
 & \mid & \textit{pat}^{i+1}\ \textit{varop}^{(\mathrm{r},i)}\ \textit{rpat}^{i}
\end{array}
$$

	|	`(` *funlhs* `)` *apat* { *apat* }
rhs	→	`=` *exp* [`where` *decls*]
	|	*gdrhs* [`where` *decls*]
gdrhs	→	*gd* `=` *exp* [*gdrhs*]
gd	→	`|` exp^0

Перевод:
объявление →
  (*левая-часть-функции* | *образец*^{0}) *правая-часть*

левая-часть-функции →
  переменная такой-как-образец { *такой-как-образец* }
  | *образец*$^{i+1}$ *оператор-переменной*$^{(a,i)}$ *образец*$^{i+1}$
  | *левый-образец*^{i} *оператор-переменной*$^{(\mathrm{l},i)}$ *образец*$^{i+1}$
  | *образец*$^{i+1}$ *оператор-переменной*$^{(\mathrm{r},i)}$ *правый-образец*^{i}
  | `(` *левая-часть-функции* `)` *такой-как-образец* { *такой-как-образец* }

правая-часть →
  `=` *выражение* [`where` *список-объявлений*]
  | *правая-часть-со-стражами* [`where` *список-объявлений*]

правая-часть-со-стражами →
  страж `=` *выражение* [*правая-часть-со-стражами*]

страж →
  `|` *выражение*^{0}

Мы различаем два случая использования этого синтаксиса: *связывание имен в образцах* происходит, когда левой частью является pat^0, в противном случае это *связывание имен в функциях*. Связывание имен может иметь место на верхнем уровне модуля или в пределах конструкций `where` или `let`.

4.4.3.1 Связывание имен в функциях

Связывание имен в функции связывает переменную со значением функции. Связывание имен в функции для переменной x в общем виде выглядит так:

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

где каждое p_{ij} — образец, а каждое $match_i$ в общем виде выглядит так:

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

или

$$\begin{array}{l} \texttt{| } g_{i1} \quad \texttt{= } e_{i1} \\ \ldots \\ \texttt{| } g_{im_i} \quad \texttt{= } e_{im_i} \\ \qquad\quad \texttt{where \{ } decls_i \texttt{ \}} \end{array}$$

$n \geq 1,\ 1 \leq i \leq n,\ m_i \geq 1$. Первый из двух вариантов рассматривается как краткая запись для особого случая второго варианта, а именно:

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

Отметим, что все инструкции, определяющие функцию, должны следовать непосредственно друг за другом, и число образцов в каждой инструкции должно быть одно и то же. Набор образцов, соответствующий каждому сопоставлению, должен быть *линейным*: никакая переменная не может появиться во всем наборе более одного раза.

Для связывания значений функций с инфиксными операторами имеется альтернативный синтаксис. Например, все эти три определения функции эквивалентны:

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

Трансляция: Связывание имен для функций в общем виде семантически эквивалентно уравнению (т.е. простому связыванию имен в образцах):

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

где x_i — новые идентификаторы.

4.4.3.2 Связывание имен в образцах

Связывание имен в образцах связывает переменные со значениями. *Простое* связывание имен в образцах имеет вид $p = e$. Образец p "лениво" сопоставляется значению, как неопровержимый образец, как если бы впереди него был указана `~` (см. трансляцию в разделе 3.12).

В *общем* виде связывание имен в образцах выглядит так: p *match*, где *match* имеет ту же структуру, что для описанного выше связывания имен в функциях, другими словами, связывание имен в образцах имеет вид:

p | g_1 = e_1
| g_2 = e_2
...
| g_m = e_m
where { *decls* }

Трансляция: Описанное выше связывание имен в образцах семантически эквивалентно этому простому связыванию имен в образцах:

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

Замечание о синтаксисе. Обычно просто отличить, является ли связывание имен связыванием имен в образце или в функции, но наличие n+k-образцов иногда сбивает с толку. Рассмотрим четыре примера:

```
x + 1 = ...           -- Связывание имен в функции, определяет (+)
                      -- Эквивалентно   (+) x 1 = ...

(x + 1) = ...         -- Связывание имен в образце, определяет x

(x + 1) * y = ...     -- Связывание имен в функции, определяет (*)
                      -- Эквивалентно   (*) (x+1) y = ...

(x + 1) y = ...       -- Связывание имен в функции, определяет (+)
                      -- Эквивалентно   (+) x 1 y = ...
```

Первые два связывания имен можно различить, потому что связывание имен в образце имеет в левой части pat^0, а не *pat*, связывание имен в первом примере не может быть n+k-образцом без скобок.

4.5 Статическая семантика связываний имен в функциях и образцах

В этом разделе рассматривается статическая семантика связываний имен в функциях и образцах в let-выражении или инструкции where.

4.5.1 Анализ зависимостей

Вообще статическая семантика задается обычными правилами вывода Хиндли-Милнера (Hindley-Milner). *Преобразование на основе анализа зависимостей* — первое, что выполняется для того, чтобы расширить полиморфизм. Две переменные, связанные посредством объявлений со значениями, находятся в одной *группе объявлений*, если

1. они связаны одним и тем же связыванием имен в образце или
2. их связывания имен взаимно рекурсивны (возможно, посредством некоторых других объявлений, которые также являются частью группы).

Применение следующих правил служит причиной того, что каждая `let`- или `where`-конструкция (включая `where`-конструкцию, которая задает связывание имен верхнего уровня в модуле) связывает переменные лишь одной группы объявлений, охватывая, таким образом, необходимый анализ зависимостей: [1]

1. Порядок объявлений в `where`/`let`-конструкциях несущественен.
2. `let {`d_1`;` d_2`} in` e = `let {`d_1`} in (let {`d_2`} in` e`)`
   (когда нет идентификатора, связанного в d_2, d_2 является свободным в d_1)

4.5.2 Обобщение

Система типов Хиндли-Милнера (Hindley-Milner) устанавливает типы в `let`-выражении в два этапа. Сначала определяется тип правой части объявления, результатом является тип без использования квантора всеобщности. Затем все переменные типа, которые встречаются в этом типе, помещаются под квантор всеобщности, если они не связаны со связанными переменными в окружении типа; это называется *обобщением*. В заключение определяется тип тела `let`-выражения.

Например, рассмотрим объявление

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

Типом определения `g` является $a \rightarrow (a, a)$. На шаге обобщения `g` будет приписан полиморфный тип $\forall\ a.\ \ a \rightarrow (a, a)$, после чего можно переходить к определению типа части "`...`".

При определении типа перегруженных определений все ограничения на перегрузки из одной группы объявлений собираются вместе для того, чтобы создать контекст для типа каждой переменной, объявленной в группе. Например, в определении

[1] Сходное преобразование описано в книге Пейтона Джонса (Peyton Jones) [10].

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

Типом определений `g1` и `g2` является $a \rightarrow a \rightarrow$ `String`, а собранные ограничения представлют собой `Ord` a (ограничение, возникшее из использования `>`) и `Show` a (ограничение, возникшее из использования `show`). Переменные типа, встречающиеся в этой совокупности ограничений, называются *переменными ограниченного типа.*

На шаге обобщения `g1` и `g2` будет приписан тип

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

Заметим, что `g2` перегружен так же, как и `g1`, хотя `>` и `show` находятся в определении `g1`.

Если программист укажет явные сигнатуры типов для более чем одной переменной в группе оъявлений, контексты этих сигнатур должны быть идентичны с точностью до переименования переменных типа.

4.5.3 Ошибки приведения контекста

Как сказано в разделе 4.1.4, контекст типа может ограничивать только переменную типа или применение переменной типа одним или более типами. Следовательно, типы, полученные при обобщении, должны иметь вид, в котором все ограничения контекста приведены к этой "главной нормальной форме". Рассмотрим, к примеру, определение

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

Его типом является

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

а не

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

Даже если равенство имеет место в типе списка, перед обобщением необходимо упростить контекст, используя объявление экземпляра для `Eq` на списках. Если в области видимости нет такого экземпляра — возникнет статическая ошибка.

Рассмотрим пример, который показывает необходимость ограничения вида $C\ (m\ t)$, где m — одна из переменных типа, которая подвергается обобщению, то есть где класс C применяется к выражению с типами, которое не является переменной типа или конструктором типа. Рассмотрим

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

Типом `return` является `Monad m => a -> m a`, типом `(==)` является `Eq a => a -> a -> Bool`. Следовательно, типом `f` должен являться `(Monad m, Eq (m a)) => a -> m a -> Bool`, и контекст не может быть более упрощен.

Объявление экземпляра, полученное из инструкции `deriving` типа данных (см. раздел 4.3.3) должно, как любое объявление экземпляра, иметь *простой* контекст, то есть все ограничения должны иметь вид $C\ a$, где a — переменная типа. Например, в типе

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

выведенный экземпляр класса Show создаст контекст `Show (a b)`, который нельзя привести и который не является простым контекстом, поэтому возникнет статическая ошибка.

4.5.4 Мономорфизм

Иногда невозможно выполнить обобщение над всеми переменными типа, используемыми в типе определения. Например, рассмотрим объявление

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

В окружении, где `x` имеет тип a, типом определения `g` является $a \to b \to (\texttt{[}a\texttt{]}, b)$. На шаге обобщения `g` будет приписан тип $\forall\ b.\ \ a \to b \to (\texttt{[}a\texttt{]}, b)$; только b можно поставить под квантор всеобщности, потому что a встречается в окружении типа. Мы говорим, что тип `g` является *мономорфным по переменной типа* a.

Следствием такого мономорфизма является то, что первый аргумент всех применений `g` должен быть одного типа. Например, это выполняется, если "`...`" будет иметь тип

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

(это, кстати, привело бы к тому, что `x` будет иметь тип `Bool`), но это не выполнится, если выражение будет иметь тип

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

Вообще, говорят, что тип $\forall\ \overline{u}.\ cx \ \Rightarrow\ t$ является *мономорфным* по переменной типа a, если a является свободной в $\forall\ \overline{u}.\ cx \ \Rightarrow\ t$.

Стоит отметить, что предоставляемые Haskell явные сигнатуры типов не являются достаточно мощным средством для того, чтобы выразить типы, которые включают мономорфные переменные типов. Например, мы не можем записать

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

потому что это утверждало бы, что `g` является полиморфным по `a` и `b` (раздел 4.4.1). В этой программе для `g` можно задать сигнатуру типа, только если ее первый параметр ограничен типом, не содержащим переменные типа, например

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

Эта сигнатура также привела бы к тому, что `x` должен иметь тип `Int`.

4.5.5 Ограничение мономорфизма

Помимо стандартного ограничения Хиндли-Милнера (Hindley-Milner), описанного выше, Haskell устанавливает некоторые дополнительные ограничения на шаге обобщения, которые позволяют в отдельных случаях дальнейшее приведение полиморфизма.

Ограничение мономорфизма зависит от синтаксиса связывания переменной. Вспомним, что переменная связывается посредством *связывания имен в функциях* или *связывания имен в образцах*, и что связывание имен в *простом* образце — это связывание имен в образце, в котором образец состоит только из одной переменной (раздел 4.4.3).

Следующие два правила определяют ограничение мономорфизма:

Ограничение мономорфизма

Правило 1. Мы говорим, что данная группа объявлений является *неограниченной*, если и только если:

(a): каждая переменная в группе связана посредством связывания имен в функциях или посредством связывания имен в простых образцах (раздел 4.4.3.2), *и*

(b): для каждой переменной в группе, которая связана посредством связывания имен в простых образцах, явно указана сигнатура типа.

Обычное ограничение полиморфизма Хиндли-Милнера (Hindley-Milner) заключается в том, что только переменные типа, которые являются свободными в окружении, могут быть подвергнуты обобщению. Кроме того, *переменные ограниченного типа из группы ограниченных объявлений нельзя подвергать обобщению* на шаге обобщения для этой группы. (Вспомним, что переменная типа ограничена, если она должна принадлежать некоторому классу типа, см. раздел 4.5.2.)

Правило 2. Любые переменные мономорфного типа, которые остаются после завершения вывода типа для всего модуля, считаются *неоднозначными*, и разрешение неоднозначности с определением конкретных типов выполняется с использованием правил по умолчанию (раздел 4.3.4).

Обоснование Правило 1 требуется по двум причинам, обе из них довольно тонкие.

- *Правило 1 предотвращает непредвиденные повторы вычислений.* Например, `genericLength` является стандартной функцией (в библиотеке `List`) с типом

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

  Теперь рассмотрим следующее выражение:

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

  Оно выглядит так, будто `len` должно быть вычислено только один раз, но без Правила 1 оно могло быть вычислено дважды, по одному разу при каждой из двух различных перегрузок. Если программист действительно хочет, чтобы вычисление было повторено, можно явно указать сигнатуру типа:

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

- *Правило 1 предотвращает неоднозначность.* Например, рассмотрим группу объявлений

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

  Вспомним, что `reads` — стандартная функция, чей тип задается сигнатурой

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

  Без Правила 1 `n` был бы присвоен тип $\forall\ a$. `Read` $a\ \Rightarrow\ a$, а `s` — тип $\forall\ a$. `Read` a $\Rightarrow$ `String`. Последний тип является неправильным, потому что по сути он неоднозначен. Невозможно определить, ни в какой перегрузке использовать `s`, ни можно ли это решить путем добавления сигнатуры типа для `s`. Поэтому, когда используется связывание имен в образце, *не являющимся простым* образцом (раздел 4.4.3.2), выведенные типы всегда являются мономорфными по своим переменным ограниченного типа, независимо от того, была ли указана сигнатура типа. В этом случае `n` и `s` являются мономорфными по a.

  То же ограничение применимо к связыванным с образцами функциям. Например, в

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

  `f` и `g` мономорфны независимо от того, какая сигнатура типа будет указана для `f` или `g`.

Правило 2 требуется потому, что нет никакого иного способа предписать мономорфное использование *экспортируемого* связывания, кроме как выполняя вывод типов на модулях вне текущего модуля. Правило 2 устанавливает, что точные типы всех переменных, связанных в модуле, должны быть определены самим модулем, а не какими-либо модулями, которые импортируют его.

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

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

Когда вывод типа в модуле `M1` закончится, `len1` будет иметь мономорфный тип `Num a => a` (по Правилу 1). Теперь Правило 2 усатанавливает, что переменная мономорфного типа `a` является неоднозначной, и неоднозначность должна быть разрешена путем использования правил по умолчанию раздела 4.3.4. Поэтому `len1` получит тип `Int`, и его использование в `len2` является неправильным из-за типа. (Если вышеупомянутый код в действительности именно то, что требуется, то сигнатура типа для `len1` решила бы проблему.)

Эта проблема не возникает для вложенных связываний, потому что их область видимости видна компилятору.

Следствия Правило мономорфизма имеет множество последствий для программиста. Все, что определено с использованием функционального синтаксиса, обычно обобщается, поскольку ожидается функция. Таким образом, в

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

функция `f` может использоваться при любой перегрузке в классе `Num`. Здесь нет никакой опасности перевычисления. Тем не менее, та же функция, определенная с использованием синтаксиса образца

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

требует указания сигнатуры типа, если `f` должна быть полностью перегружена. Многие функции наиболее естественно определяются посредством использования связывания имен в простых образцах; пользователь должен быть внимателен, добавляя к ним сигнатуры типов, чтобы сохранить полную перегрузку. Стандартное начало (Prelude) содержит много таких примеров:

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

Правило 1 применяется к определениям верхнего уровня и к вложенным определениям. Рассмотрим пример:

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

Здесь с помощью вывода типа устанавливаем, что `len1` имеет мономорфический тип (`Num a => a`); при выполнении вывода типа для `len2` определяем, что переменная типа `a` имеет тип `Rational`.

4.6 Вывод вида

В этом разделе описываются правила, которые используются для того, чтобы выполнить *вывод вида*, т.е. вычислить подходящий вид для каждого конструктора типа и класса, фигурирующего в данной программе.

Первый шаг в процессе вывода вида заключается в разделении набора определений типов данных, синонимов и классов на группы зависимостей. Этого можно достичь почти таким же способом, как анализ зависимостей для объявлений значений, который был описан в разделе 4.5. Например, следующий фрагмент программы включает определение конструктора типа данных `D`, синонима `S` и класса `C`, все они будут включены в одну группу зависимостей:

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

Виды, к которым относятся переменные, конструкторы и классы в пределах каждой группы, определяются с использованием стандартных методов вывода типа и сохраняющей вид унификации (объединения) [7]. Например, в приведенном выше определении параметр `a` является аргументом конструктора функции `(->)` в типе `bar` и поэтому должен относиться к виду $*$. Из этого следует, что и `D`, и `S` должны относиться к виду $* \to *$ и что каждый экземпляр класса `C` должен относиться к виду $*$.

Возможно, что некоторые части выведенного вида не могут быть полностью определены исходя из соответствующих определений; в таких случаях принимается значение по умолчанию вида $*$. Например, мы могли принять произвольный вид κ для параметра `a` в каждом из следующих примеров:

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

Тогда мы получили бы виды $(\kappa \to *) \to \kappa \to *$ и $\kappa \to *$ соответственно для `App` и `Tree` для любого вида κ. Это также потребовало бы, чтобы расширение допускало полиморфные виды. Вместо этого, используя по умолчанию связывание $\kappa = *$, действительными видами для этих двух конструкторов являются соответственно $(* \to *) \to * \to *$ и $* \to *$.

Значения по умолчанию применяются к каждой группе зависимостей, независимо от того, как конкретные константы конструктора типа или классов используются в более поздних группах зависимостей или где-либо в другом месте в программе.

Например, добавление следующего определения к приведенным выше не влияет на вид, выведенный для `Tree` (путем изменения его на $(* \to *) \to *$, например), и вместо этого приводит к статической ошибке, потому что вид, которому принадлежит `[]`, $* \to *$, не соответствует виду $*$, который ожидается для аргумента `Tree`:

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

Это важно, потому что гарантирует, что каждый конструктор и класс используются в соответствии с одним и тем же видом всякий раз, когда они находятся в области видимости.

Глава 5

Модули

Модуль определяет совокупность значений, типов данных, синонимов типов, классов и т.д. (см. главу 4) в окружении, созданном набором *списков импорта* (введенных в область видимости ресурсов других модулей). Он *экспортирует* некоторые из этих ресурсов, делая их доступными другим модулям. Мы используем термин *сущность* для ссылки на значение, тип или класс, определенный, импортированный или, возможно, экспортированный из модуля.

Программа на Haskell — это совокупность модулей, один из которых условно должен называться `Main` и должен экспортировать значение `main`. *Значением* программы является значение идентификатора `main` в модуле `Main`, которое должно иметь тип `IO` τ для некоторого типа τ (см. главу 7). Когда выполняется программа, вычисляется значение `main` и результат (типа τ) отбрасывается.

Модули могут ссылаться на другие модули посредством явных объявлений `import`, каждое из которых задает имя импортируемого модуля и его сущности, которые будут импортированы. Модули могут быть взаимно рекурсивны.

Модули используются для управления пространством имен и не являются главными значениями класса. Многомодульная программа на Haskell может быть преобразована в программу с одним модулем, если дать каждой сущности уникальное имя, соответственно заменить все вхождения, ссылающиеся на эти имена, и затем объединить все тела модулей. [1] Например, рассмотрим программу с тремя модулями:

[1] Есть два незначительных исключения из этого утверждения. Первое — объявления `default` видны в области видимости одного модуля (раздел 4.3.4). Второе — Правило 2 ограничения мономорфизма (раздел 4.5.5) влияет на границы модулей.

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

module A where
  f = ...

module B where
  f = ...
```

Она эквивалентна следующей программе с одним модулем:

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

  af = ...

  bf = ...
```

Поскольку модули могут быть взаимно рекурсивными, с помощью модулей можно свободно разделить программу на части, не обращая внимания на зависимости.

Пространство имен для самих модулей является плоским, оно связывает каждый модуль с уникальным именем модуля (которые являются идентификаторами Haskell, начинающимися с заглавной буквы, т.е. *modid*). Есть один, отличный от остальных, модуль `Prelude`, который импортируется во все модули по умолчанию (см. раздел 5.6), плюс набор модулей стандартной библиотеки, которые можно импортировать по требованию (см. часть II).

5.1 Структура модуля

Модуль определяет взаимно рекурсивную область видимости, содержащую объявления для связывания значений, типов данных, синонимов типов, классов и т.д. (см. главу 4).

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

Перевод:
модуль →
  `module` *идентификатор-модуля* [*список-экспорта*] `where` *тело*
  | *тело*
тело →
  `{` *список-объявлений-импорта* `;` *список-объявлений-верхнего-уровня* `}`
  | `{` *список-объявлений-импорта* `}`
  | `{` *список-объявлений-верхнего-уровня* `}`

идентификатор-модуля →
  идентификатор-конструктора
список-объявлений-импорта →
  объявление-импорта$_1$ `;` ... `;` *объявление-импорта*$_n$
    $(n \geq 1)$
список-объявлений-верхнего-уровня →
  объявление-верхнего-уровня$_1$ `;` ... `;` *объявление-верхнего-уровня*$_n$
    $(n \geq 1)$

Модуль начинается с заголовка — ключевого слова `module`, имени модуля и списка экспортируемых сущностей (заключенного в круглые скобки). За заголовком следует возможно пустой список объявлений `import` (*impdecls*, раздел 5.3), который задает импортируемые модули, необязательно ограничивая импортируемые связывания имен. За ним следует возможно пустой список объявлений верхнего уровня (*topdecls*, глава 4).

Разрешена сокращенная форма модуля, состоящая только из тела модуля. Если используется сокращенная форма, то предполагается заголовок '`module Main(main) where`'. Если первая лексема в сокращенном модуле не является `{`, то для верхнего уровня модуля применяется правило размещения.

5.2 Списки экспорта

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

Перевод:
список-экспорта →

(*экспорт*$_1$, ... , *экспорт*$_n$ [,])
($n \geq 0$)

экспорт →
квалифицированная-переменная
| *квалифицированный-конструктор-типа* [(..) | (*с-имя*$_1$, ... , *с-имя*$_n$)]
($n \geq 0$)
| *квалифицированный-класс-типа* [(..) |
(*квалифицированная-переменная*$_1$, ... , *квалифицированная-переменная*$_n$)]
($n \geq 0$)
| `module` *идентификатор-модуля*

с-имя →
переменная
| *конструктор*

Список экспорта определяет сущности, которые экспортируются посредством объявления модуля. Реализация модуля может экспортировать только ту сущность, которую он объявляет или которую он импортирует из некоторого другого модуля. Если список экспорта пропущен, все значения, типы и классы, определенные в модуле, экспортируются, *кроме тех, что были импортированы.*

Сущности в списке экспорта можно перечислить следующим образом:

1. Значение, имя поля или метод класса, объявленные в теле модуля или импортированные, можно указать, задав имя значения в качестве *qvarid*, которое должно находиться в области видимости. Операторы должны быть заключены в круглые скобки, чтобы превратить их в *qvarid*.

2. Алгебраический тип данных T, объявленный посредством объявления `data` или `newtype`, можно указать одним из трех способов:

   - Форма T указывает тип, *но не конструкторы или имена полей.* Способность экспортировать тип без его конструкторов позволяет конструировать абстрактные типы данных (см. раздел 5.8).
   - Форма T`(`c_1`,...,`c_n`)` указывает тип и некоторые или все его конструкторы и имена полей.
   - Сокращенная форма T`(..)` указывает тип и все его конструкторы и имена полей, которые в настоящее время находятся в области видимости (квалифицированные или нет).

   Во всех случаях (возможно квалифицированный) конструктор типа T должен находиться в области видимости. Конструктор и имена полей c_i во второй форме являются неквалифицированными; одно из этих подчиненных имен является правильным, если и только если (a) оно именует собой конструктор или поле

T и (b) конструктор или поле находится в области видимости в теле модуля, при этом *неважно, находится он в области видимости под квалифицированным или неквалифицированном именем.* Например, следующее объявление является правильным:

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

Конструкторы данных нельзя указывать в списках экспорта, кроме как с помощью подчиненных имен, потому что иначе они не могут быть отличимы от конструкторов типов.

3. Синоним типа T, объявленный в объявлении `type`, можно указать с помощью формы T, где T находится в области видимости.

4. Класс C с операциями $f_1, \ldots, f_n$, объявленный в объявлении `class`, можно указать одним из трех способов:

   - Форма C указывает класс, *но не методы класса.*
   - Форма C`(`f_1`, ..., `f_n`)`, указывает класс и некоторых или все методы.
   - Сокращенная форма C`(..)` указывает класс и все его методы, которые находятся в области видимости (квалифицированные или нет).

   Во всех случаях C должен находиться в области видимости. Во второй форме одно из (неквалифицированных) подчиненных имен f_i является правильным, если и только если (a) оно именует собой метод класса C и (b) метод класса находится в области видимости в теле модуля, неважно, находится он в области видимости под квалифицированным или неквалифицированным именем.

5. Форма "`module M`" указывает набор всех сущностей, которые находятся в области видимости с неквалифицированным именем "`e`" и квалифицированным именем "`M.e`". Этот набор может быть пуст. Например:

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

   Здесь модуль `Queue` использует имя модуля `Stack` в своем списке экспорта, чтобы сократить имена всех сущностей, импортированных из `Stack`.

   Модуль может указать свои собственные локальные определения в своем списке экспорта, используя свое собственное имя в синтаксисе "`module M`", потому что локальное объявление вводит в область видимости и квалифицированное, и неквалифицированное имя (раздел 5.5.1). Например:

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

   Здесь модуль `Mod1` экспортирует все локальные определения, а также импортированные из `Mod2`, но не импортированные из `Mod3`.

Будет ошибкой использовать `module M` в списке экспорта, если `M` не является модулем, обладающим списком экспорта, или `M` не импортирован по меньшей мере посредством одного объявления импорта (квалифицированным или неквалифицированным).

Списки экспорта являются общими: набор сущностей, экспортируемых посредством списка экспорта является объединением сущностей, экспортируемых отдельными элементами списка.

Нет никакого различия для импортируемого модуля, как сущность была экспортирована. Например, имя поля `f` из типа данных `T` можно экспортировать отдельно (`f`, пункт (1) выше) или как явно указанный член его типа данных (`T(f)`, пункт (2)), или как неявно указанный член (`T(..)`, пункт (2)), или посредством экспорта всего модуля (`module M`, пункт (5)).

Неквалифицированные имена сущностей, экспортируемые модулем, должны отличаться друг от друга (в пределах их соответствующего пространства имен). Например,

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

Непосредственно в пределах модуля `A` конфликтов имен нет, но есть конфликт имен в списке экспорта между `C.g` и `g` (предположим, что `C.g` и `g` — различные сущности, вспомните, что модули могут импортировать друг друга рекурсивно) и между `module B` и `C.f` (предположим, что `B.f` и `C.f` — различные сущности).

5.3 Объявления импорта

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

Перевод:
объявление-импорта →
`import` [`qualified`] *идентификатор-модуля* [`as` *идентификатор-модуля*]

[*спецификатор-импорта*]
| (*пустое объявление*)

спецификатор-импорта →
(*импорт*$_1$, ... , *импорт*$_n$ [,])
($n \geq 0$)
| `hiding` (*импорт*$_1$, ... , *импорт*$_n$ [,])
($n \geq 0$)

импорт →
переменная
| *конструктор-типа* [`(..)` | (*с-имя*$_1$, ... , *с-имя*$_n$)]
($n \geq 0$)
| *класс-типа* [`(..)` | (*переменная*$_1$, ... , *переменная*$_n$)]
($n \geq 0$)
с-имя →
переменная
| *конструктор*

Сущности, экспортируемые модулем, можно ввести в область видимости другого модуля посредством объявления `import` в начале модуля. В объявлении `import` указывается импортируемый модуль и необязательно задаются импортируемые сущности. Один модуль можно импортировать с помощью более чем одного объявления `import`. Импортированные имена служат в качестве объявлений верхнего уровня: их область видимости простирается над всем телом модуля, но может быть сокрыта локальными связываниями имен отличного от верхнего уровня.

Влияние многократных объявлений `import` строго кумулятивно: сущность находится в области видимости, если она импортирована посредством любого из объявлений `import` в модуле. Порядок объявлений импорта не существенен.

С точки зрения лексики, каждый из терминальных символов "`as`", "`qualified`" и "`hiding`" является *varid (идентификатором-переменной)*, а не *reservedid (зарезервированным-идентификатором)*. Они имеют специальное значение только в контексте объявления `import`; их также можно использовать в качестве переменных.

5.3.1 Что такое импортирование

Какие точно сущности должны быть импортированы, можно задать одним из следующих трех способов:

1. Импортируемые сущности можно задать явно, перечислив их в круглых скобках. Элементы списка имеют ту же форму, что элементы в списках экспорта,

за исключением того, что нельзя использовать квалификаторы и нельзя использовать сущность '`module` *modid*'. Когда форма `(..)` импорта используется для типа или класса, `(..)` ссылается на все конструкторы, методы или имена полей, экспортированные из модуля.

В списке должны быть указаны только сущности, экспортированные импортируемым модулем. Список может быть пуст, в этом случае ничто, кроме экземпляров, не будет импортировано.

2. Сущности могут быть исключены посредством использования формы `hiding(`$import_1$ `,` ... `,` $import_n$ `)`, которая указывает, что все объекты, экспортированные названным модулем, должны быть импортированы, за исключением указанных в списке. Конструкторы данных можно указать непосредственно в списках hiding без использования в префиксе связанного с ним типа. Таким образом, в

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

   любой конструктор, класс, или тип, названный `C`, исключен. Напротив, используя `C` в списке импорта, вы укажете лишь класс или тип.

   Будет ошибкой указать в списке hiding сущность, которая на самом деле не экспортируется импортируемым модулем.

3. Наконец, если *impspec* пропущен, то все сущности, экспортируемые указанным модулем, будут импортированы.

5.3.2 Импортирование с использованием квалификаторов

Для каждой сущности, импортируемой в соответствии с правилами раздела 5.3.1, расширяется окружение верхнего уровня. Если объявление импорта использует ключевое слово `qualified`, то только *квалифицированное имя* сущности вводится в область видимости. Если ключевое слово `qualified` опущено, то *оба* имени: квалифицированное *и* неквалифицированное имя сущности — вводятся в область видимости. В разделе 5.5.1 квалифицированные имена описаны более подробно.

Квалификатор импортированного имени является именем импортированного модуля или локальным синонимом, заданным с помощью инструкции `as` (раздел 5.3.3) в инструкции `import`. Следовательно, *квалификатор необязательно является именем модуля, в котором первоначально была объявлена сущность.*

Возможность исключить неквалифицированные имена позволяет программисту осуществлять полное управление пространством неквалифицированных имен: локально определенная сущность может совместно использовать то же имя, что и импортируемая сущность с квалифицированным именем:

```
module Ring where
import qualified Prelude    -- Все имена из Prelude должны быть
                           -- квалифицированными
import List( nub )

l1 + l2 = l1 Prelude.++ l2  -- Этот + отличается от + в Prelude
l1 * l2 = nub (l1 + l2)     -- Эта * отличается от * в Prelude

succ = (Prelude.+ 1)
```

5.3.3 Локальные синонимы

Импортированным модулям можно присвоить локальный синоним в модуле, который осуществляет импортирование, для этого используется инструкция `as`. Например, в

```
import qualified VeryLongModuleName as C
```

к импортированным сущностям можно обращаться, используя в качестве квалификатора '`C.`' вместо '`VeryLongModuleName.`'. Это также позволяет другому модулю быть замененным на `VeryLongModuleName` без изменения квалификаторов, используемых для импортированного модуля. Более чем один модуль в области видимости может использовать тот же самый квалификатор, при условии, что все имена по-прежнему могут быть однозначно разрешены. Например:

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

Этот модуль является правильным только при условии, что и `Foo`, и `Baz` не экспортируют `f`.

Инструкцию `as` можно также использовать в инструкции `import` без `qualified`:

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

Это объявление вводит в область видимости `f` и `A.f`.

5.3.4 Примеры

Для того чтобы разъяснить вышеупомянутые правила импорта, предположим, что модуль `A` экспортирует `x` и `y`. Тогда эта таблица показывает, какие имена будут введены в область видимости с помощью заданного объявления импорта:

Объявление импорта	Имена, введенные в область видимости
`import A`	`x`, `y`, `A.x`, `A.y`
`import A()`	(ничего)
`import A(x)`	`x`, `A.x`
`import qualified A`	`A.x`, `A.y`
`import qualified A()`	(ничего)
`import qualified A(x)`	`A.x`
`import A hiding ()`	`x`, `y`, `A.x`, `A.y`
`import A hiding (x)`	`y`, `A.y`
`import qualified A hiding ()`	`A.x`, `A.y`
`import qualified A hiding (x)`	`A.y`
`import A as B`	`x`, `y`, `B.x`, `B.y`
`import A as B(x)`	`x`, `B.x`
`import qualified A as B`	`B.x`, `B.y`

Во всех случаях все объявления экземпляров в области видимости в модуле `A` будут импортированы (раздел 5.4).

5.4 Импортирование и экспортирование объявлений экземпляров

Объявления экземпляров нельзя явно указать в списках импорта или экспорта. Все экземпляры в области видимости модуля *всегда* экспортируются, и любое объявление импорта вводит в область видимости *все* экземпляры импортируемого модуля. Таким образом, объявление экземпляра находится в области видимости, если и только если цепочка объявлений `import` ведет к модулю, содержащему объявление экземпляра.

Например, `import M()` не вводит никакие новые имена из модуля `M` в область видимости, но вводит все экземпляры, которые видны в `M`. Модуль, чья единственная цель состоит в том, чтобы обеспечить объявления экземпляров, может иметь пустой список экспорта. Например,

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

5.5 Конфликт имен и замыкание

5.5.1 Квалифицированные имена

Квалифицированное имя имеет вид *modid*`.`*name* (*идентификатор-модуля*`.`*имя*) (раздел 2.4). Квалифицированное имя вводится в область видимости:

- *Посредством объявления верхнего уровня.* Объявление верхнего уровня вводит в область видимости и неквалифицированное, *и* квалифицированное имя определяемой сущности. Так:

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

  является правильным объявлением. *Определяемое* вхождение должно ссылаться на *неквалифицированное* имя; поэтому будет неправильным писать

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

- *Посредством объявления* `import`. Объявление `import`, с инструкцией `qualified` или без, всегда вводит в область видимости квалифицированное имя импортированной сущности (раздел 5.3). Это позволяет заменить объявление импорта с инструкцией `qualified` на объявление без инструкции `qualified` без изменений ссылок на импортированные имена.

5.5.2 Конфликты имен

Если модуль содержит связанное вхождение имени, например, `f` или `A.f`, должна быть возможность однозначно решить, на какую сущность при этом ссылаются; то есть должно быть только одно связывание для `f` или `A.f` соответственно.

Ошибки *не* будет, если существуют имена, которые нельзя так разрешить, при условии, что программа не содержит ссылок на эти имена. Например:

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

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

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

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

Рассмотрим определение `tup`.

- Ссылки на `b` и `c` можно однозначно разрешить: здесь подразумевается соответственно `b`, объявленный в `B`, и `c`, объявленный в `C`.
- Ссылка на `d` однозначно разрешается: здесь подразумевается `d`, объявленный в `D`. В этом случае та же сущность вводится в область видимости двумя путями (импорт `B` и импорт `C`), и на нее можно ссылаться в `A` посредством имен `d`, `B.d` и `C.d`.
- Ссылка на `x` является неоднозначной: она может означать `x`, объявленный в `B`, или `x`, объявленный в `C`. Неоднозначность может быть разрешена путем замены `x` на `B.x` или `C.x`.
- Нет ни одной ссылки на `y`, поэтому нет ошибки в том, что различные сущности с именем `y` экспортируют и `B`, и `C`. Сообщение об ошибке появится только в том случае, если будет ссылка на `y`.

Имя, встречающееся в сигнатуре типа или infix-объявлениях, всегда является неквалифицированным и однозначно ссылается на другое объявление в том же списке объявлений (за исключением того, что infix-объявление для метода класса может встречаться на верхнем уровне, см. раздел 4.4.2). Например, следующий модуль является правильным:

```
module F where

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

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

Локальное объявление `sin` является правильным, даже если `sin` из Prelude неявно находится в области видимости. Ссылки на `Prelude.sin` и `F.sin` должны быть обе квалифицированными для того, чтобы однозначно определить, какой подразумевается `sin`. Тем не менее, неквалифицированное имя `sin` в сигнатуре типа в первой строке `F` однозначно ссылается на локальное объявление `sin`.

5.5.3 Замыкание

Каждый модуль в программе на Haskell должен быть *замкнутым*. То есть каждое имя, явно указанное в исходном тексте, должно быть локально определено или импортировано из другого модуля. Тем не менее, нет необходимости в том, чтобы сущности, которые требуются компилятору для контроля типов или другого анализа времени компиляции, были импортированы, если к ним нет обращений по имени. Система компиляции Haskell несет ответственность за нахождение любой информации, необходимой для компиляции без помощи программиста. То есть импорт переменной `x` не требует, чтобы типы данных и классы в сигнатуре `x` были введены в модуль наряду с `x`, если к этим сущностям не обращаются по имени в пользовательской программе. Система Haskell молча импортирует любую информацию, которая должна сопровождать сущность для контроля типов или любых других целей. Такие сущности не требуется даже явно экспортировать: следующая программа является правильной, хотя `T` не избегает `M1`:

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

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

В этом примере нет способа указать явную сигнатуру типа для `y`, т.к. `T` не находится в области видимости. Независимо от того, экспортируется `T` явно или нет, модуль `M2` знает достаточно о `T`, чтобы правильно выполнить контроль соответствия типов программы.

На тип экспортируемой сущности не влияет неэкспортируемые синонимы типов. Например, в

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

типом `x` является и `T`, и `Int`; они взаимозаменяемы, даже когда `T` не находится в области видимости. То есть определение `T` доступно любому модулю, который сталкивается с ним, независимо от того, находится имя `T` в области видимости или нет. Единственная причина экспортировать `T` состоит в том, чтобы позволить другим модулям обращаться

к нему по имени; контроль типов находит определение `T`, если оно необходимо, независимо от того, было оно экспортировано или нет.

5.6 Стандартное начало (Prelude)

Многие возможности Haskell определены в самом Haskell в виде библиотеки стандартных типов данных, классов и функций, называемой "стандартным началом (prelude)." В Haskell стандартное начало содержится в модуле `Prelude`. Есть также много предопределенных модулей библиотеки, которые обеспечивают менее часто используемые функции и типы. Например, комплексные числа, массивы, и большинство операций ввода - вывода являются частью стандартной библиотеки. Они описаны в части II. Отделение библиотеки от Prelude имеет преимущество в виде сокращения размера и сложности Prelude, позволяя ему более легко становиться частью программы и расширяя пространство полезных имен, доступных программисту.

Prelude и модули библиотеки отличаются от других модулей тем, что их семантика (но не их реализация) является неизменной частью определения языка Haskell. Это означает, например, что компилятор может оптимизировать вызовы функций Haskell, не принимая во внимание исходный текст Prelude.

5.6.1 Модуль `Prelude`

Модуль `Prelude` автоматически импортируется во все модули, как если бы была инструкция '`import Prelude`', если и только если он не импортируется посредством явного объявления `import`. Это условие для явного импорта позволяет выборочно импортировать сущности, определенные в Prelude, точно так же, как сущности из любого другого модуля.

Семантика сущностей в `Prelude` задана ссылочной реализацией `Prelude`, написанной на Haskell, данной в главе 8. Некоторые типы данных (например, `Int`) и функции (например, сложение `Int`) нельзя задать непосредственно на Haskell. Так как обработка таких сущностей зависит от реализации, они формально не описаны в главе 8. Реализация `Prelude` также является неполной при обработке кортежей: должно быть бесконечное семейство кортежей и объявлений их экземпляров, но реализация лишь задает схему.

В главе 8 дано определение модуля `Prelude` с использованием нескольких других модулей: `PreludeList`, `PreludeIO` и так далее. Эти модули *не* являются частью Haskell 98, и их нельзя импортировать отдельно. Они просто помогают объяснить структуру модуля `Prelude`; их следует рассматривать как часть ее реализации, а не часть определения языка.

5.6.2 Сокрытие имен из Prelude

Правила о Prelude были разработаны так, чтобы имелась возможность использовать имена из Prelude для нестандартных целей; тем не менее, каждый модуль, который так делает, должен иметь объявление `import`, которое делает это нестандартное использование явным. Например:

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

Модуль `A` переопределяет `null` и содержит неквалифицированную ссылку на `null` в правой части `nonNull`. Последнее было бы неоднозначно без наличия инструкции `hiding(null)` в объявлении `import Prelude`. Каждый модуль, который импортирует неквалифицированное имя `A` и затем создает неквалифицированную ссылку на `null`, должен также разрешить неоднозначное использование `null` так же, как это делает `A`. Таким образом, есть небольшая опасность случайно скрыть имена из Prelude.

Имеется возможность создать и использовать другой модуль, который будет служить вместо Prelude. За исключением того факта, что модуль Prelude неявно импортируется в любой модуль, Prelude является обычным модулем Haskell; он является особенным только в том, что обращение к некоторым сущностям Prelude происходит посредством специальных синтаксических конструкций. Переопределение имен, используемых Prelude, не влияет на значение этих специальных конструкций. Например, в

```
module B where
  import Prelude()
  import MyPrelude
  f x = (x,x)
  g x = (,) x x
  h x = [x] ++ []
```

явное объявление `import Prelude()` предотвращает автоматический импорт `Prelude`, в то время как объявление `import MyPrelude` вводит в область видимости нестандартное начало (prelude). Специальный синтаксис для кортежей (например, `(x,x)` и `(,)`) и списков (например, `[x]` и `[]`) продолжает обращаться к кортежам и спискам, определенным стандартным `Prelude`; не существует способа переопределить значение `[x]`, например, в терминах другой реализации списков. С другой стороны, использование `++` не является специальным синтаксисом, поэтому он обращается к `++`, импортированному из `MyPrelude`.

Невозможно, тем не менее, скрыть объявления `instance` в `Prelude`. Например, нельзя определить новый экземпляр для `Show Char`.

5.7 Раздельная компиляция

В зависимости от используемой реализации Haskell, раздельная компиляция взаимно рекурсивных модулей может потребовать, чтобы импортированные модули содержали дополнительную информацию с тем, чтобы к ним можно было обратиться прежде, чем они будут скомпилированы. Явные сигнатуры типов для всех экспортированных значений могут быть необходимы для того, чтобы работать со взаимной рекурсией. Точные детали раздельной компиляции в этом описании не описаны.

5.8 Абстрактные типы данных

Способность экспортировать тип данных без его конструкторов позволяет конструировать абстрактные типы данных (ADT). Например, ADT для стеков можно определить так:

```
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
```

Модули, импортирующие `Stack`, не могут создавать значения типа `StkType`, потому что они не имеют доступа к конструкторам типа. Вместо этого они должны использовать `push`, `pop` и `empty`, чтобы создать такие значения.

Также имеется возможность строить ADT на верхнем уровне существующего типа посредством использования объявления `newtype`. Например, стеки можно определить через списки:

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

Глава 6

Предопределенные типы и классы

Haskell Prelude содержит предопределенные классы, типы и функции, которые неявно импортируются в каждую программу на Haskell. В этой главе мы опишем типы и классы, находящиеся в Prelude. Большинство функций не описаны здесь подробно, поскольку их назначение легко можно понять исходя из их определений, данных в главе 8. Другие предопределенные типы, такие как массивы, комплексные и рациональные числа, описаны в части II.

6.1 Стандартные типы Haskell

Эти типы определены в Haskell Prelude. Числовые типы описаны в разделе 6.4. Там, где это возможно, дается определение типа на Haskell. Некоторые определения могут не быть полностью синтаксически правильными, но они верно передают смысл лежащего в основе типа.

6.1.1 Булевский тип

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

Булевский тип `Bool` является перечислением. Основные булевские функции — это `&&` (и), `||` (или) и `not` (не). Имя `otherwise` (иначе) определено как `True`, чтобы сделать выражения, использующие стражи, более удобочитаемыми.

6.1.2 Символы и строки

Символьный тип `Char` является перечислением, чьи значения представляют собой символы Unicode [11]. Лексический синтаксис для символов определен в разделе 2.6;

символьные литералы — это конструкторы без аргументов в типе данных `Char`. Тип `Char` является экземпляром классов `Read`, `Show`, `Eq`, `Ord`, `Enum` и `Bounded`. Функции `toEnum` и `fromEnum`, которые являются стандартными функциями из класса `Enum`, соответственно отображают символы в тип Int и обратно.

Обратите внимание, что каждый символ управления ASCII имеет несколько представлений в символьных литералах: в виде числовой эскейп-последовательности, в виде мнемонической эскейп-последовательности ASCII, и представление в виде `\^`X. Кроме того, равнозначны следующие литералы: `\a` и `\BEL`, `\b` и `\BS`, `\f` и `\FF`, `\r` и `\CR`, `\t` и `\HT`, `\v` и `\VT` и `\n` и `\LF`.

Строка — это список символов:

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

Строки можно сократить, используя лексический синтаксис, описанный в разделе 2.6. Например, `"A string"` является сокращением (аббревиатурой)

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

6.1.3 Списки

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

Списки — это алгебраический тип данных для двух конструкторов, имеющих специальный синтаксис, описанных в разделе 3.7. Первый конструктор — это пустой список, который обозначается '`[]`' ("nil"), второй — это '`:`' ("cons"). Модуль `PreludeList` (см. раздел 8.1) определяет множество стандартных функций над списком. Арифметические последовательности и описание списка — это два удобных синтаксиса, используемых для записи специальных видов списков, они описаны в разделах 3.10 и 3.11 соответственно. Списки являются экземпляром классов `Read`, `Show`, `Eq`, `Ord`, `Monad`, `Functor` и `MonadPlus`.

6.1.4 Кортежи

Кортежи — это алгебраический тип данных со специальным синтаксисом, описанным в разделе 3.8. Тип каждого кортежа имеет один конструктор. Все кортежи являются экземплярами классов `Eq`, `Ord`, `Bounded`, `Read`, и `Show` (конечно, при условии, что все их составляющие типы являются экземплярами этих классов).

Нет никакой верхней границы размера кортежа, но некоторые реализации Haskell могут содержать ограничения на размер кортежей и на экземпляры, связанные с большими кортежами. Тем не менее, каждая реализация Haskell должна поддерживать кортежи вплоть до 15 размера, наряду с экземплярами классов `Eq`, `Ord`, `Bounded`, `Read` и `Show`.

Prelude и библиотеки содержат определения функций над кортежами, таких как `zip`, для кортежей вплоть до 7 размера.

Конструктор для кортежа записывается посредством игнорирования выражений, соседствующих с запятыми; таким образом, `(x,y)` и `(,) x y` обозначают один и тот же кортеж. То же самое относится к конструкторам типов кортежей; таким образом, `(Int,Bool,Int)` и `(,,) Int Bool Int` обозначают один и тот же тип.

Следующие функции определены для пар (кортежей 2 размера): `fst`, `snd`, `curry` и `uncurry`. Для кортежей большего размера подобные функции не являются предопределенными.

6.1.5 Единичный тип данных

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

Единичный тип данных `()` имеет единственный, отличный от $\perp$, член — конструктор без аргументов `()`. См. также раздел 3.9.

6.1.6 Типы функций

Функции — это абстрактный тип: никакие конструкторы непосредственно не создают значения функций. В Prelude описаны следующие простые функции: `id`, `const`, `(.)`, `flip`, `($)` и `until`.

6.1.7 Типы IO и IOError

Тип `IO` указывает на операции, которые взаимодействуют с внешним миром. Тип `IO` является абстрактным: никакие конструкторы не видны для пользователя. `IO` является экземпляром классов `Monad` и `Functor`. В главе 7 описываются операции ввода - вывода.

Тип `IOError` является абстрактным типом, который представляет ошибки, вызванные операциями ввода - вывода. Он является экземпляром классов `Show` и `Eq`. Значения этого типа создаются различными функциями ввода - вывода. Более подробная информация о значениях в этом описании не представлена. Prelude содержит несколько функций ввода - вывода (определены в разделе 8.3). Гораздо больше функций ввода - вывода содержит часть II.

6.1.8 Другие типы

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

Тип `Maybe` является экземпляром классов `Functor`, `Monad` и `MonadPlus`. Тип `Ordering` используется функцией `compare` в классе `Ord`. Функции `maybe` и `either` описаны в Prelude.

6.2 Строгое вычисление

Применение функций в Haskell не является строгим, то есть аргумент функции вычисляется только тогда, когда требуется значение. Иногда желательно вызвать вычисление значения, используя функцию `seq` :

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

Функция `seq` определена уравнениями:

$$\texttt{seq}\ \bot\ b\ =\ \bot$$
$$\texttt{seq}\ a\ b\ =\ b,\ \mathit{if}\ a\ \neq\ \bot$$

Функция `seq` обычно вводится для того, чтобы улучшить производительность за счет избежания ненужной ленивости. Строгие типы данных (см. раздел 4.2.1) определены в терминах оператора `$!`. Тем не менее, наличие функции `seq` имеет важные семантические последствия, потому что эта функция доступна *для каждого типа*. Как следствие, $\bot$ — это не то же самое, что `\x -> ` $\bot$, так как можно использовать `seq` для того, чтобы отличить их друг от друга. По той же самой причине существование `seq` ослабляет параметрические свойства Haskell.

Оператор `$!` является строгим (вызываемым по значению) применением, он определен в терминах `seq`. Prelude также содержит определение оператора `$` для выполнения нестрогих применений.

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

Наличие оператора нестрогого применения `$` может казаться избыточным, так как обычное применение `(f x)` означает то же самое, что `(f $ x)`. Тем не менее, `$` имеет низкий приоритет и правую ассоциативность, поэтому иногда круглые скобки можно опустить, например:

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

Это также полезно в ситуациях более высокого порядка, таких как `map ($ 0) xs` или `zipWith ($) fs xs`.

6.3 Стандартные классы Haskell

На рис. 6.1 изображена иерархия классов Haskell, определенных в Prelude, и типы из Prelude, которые являются экземплярами этих классов.

Для многих методов в стандартных классах предусмотрены заданные по умолчанию объявления методов класса (раздел 4.3). Комментарий, данный для каждого объявления `class` в главе 8, определяет наименьшую совокупность определений методов, которые вместе с заданными по умолчанию объявлениями обеспечивают разумное определение для всех методов класса. Если такого комментария нет, то для того, чтобы полностью определить экземпляр, должны быть заданы все методы класса.

6.3.1 Класс Eq

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

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

Класс `Eq` предоставляет методы для сравнения на равенство (`==`) и неравенство (`/=`). Все основные типы данных, за исключением функций и `IO`, являются экземплярами этого класса. Экземпляры класса `Eq` можно использовать для выведения любого определяемого пользователем типа данных, чьи компоненты также являются экземплярами класса `Eq`.

Это объявление задает используемые по умолчанию объявления методов `/=` и `==`, каждый из которых определен в терминах другого. Если объявление экземпляра класса Eq не содержит описания ни одного из перечисленных методов, тогда оба метода образуют петлю. Если определен один из методов, то другой, заданный по умолчанию метод, будет использовать тот, который определен. Если оба метода определены, заданные по умолчанию методы использоваться не будут.

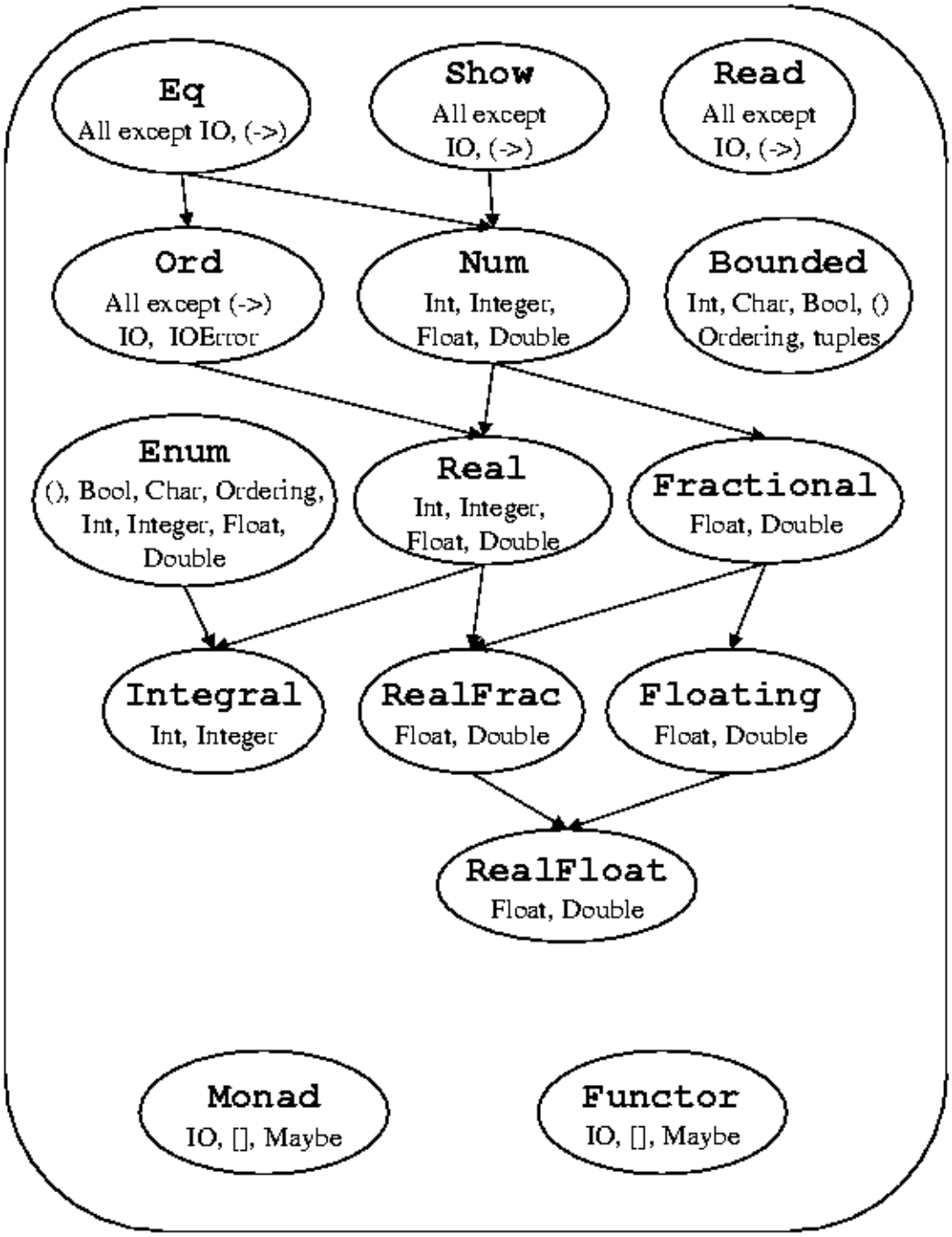

Рис. 6.1: Стандартные классы Haskell

6.3.2 Класс Ord

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

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

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

  -- Заметьте, что (min x y, max x y) = (x,y) или (y,x)
  max x y | x <= y    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

Класс `Ord` используется для полностью упорядоченных типов данных. Все основные типы данных, за исключением функций, `IO` и `IOError`, являются экземплярами этого класса. Экземпляры класса `Ord` можно использовать для выведения любого определяемого пользователем типа данных, чьи компоненты находятся в `Ord`. Объявленный порядок конструкторов в объявлении данных определяет порядок в производных экземплярах класса `Ord`. Тип данных `Ordering` позволяет использовать единообразное сравнение для определения точного порядка двух объектов.

Заданные по умолчанию объявления позволяют пользователю создавать экземпляры класса `Ord` посредством функции `compare` с определенным типом или функций `==` и `<=` с определенным типом.

6.3.3 Классы Read и Show

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

class  Read a  where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]
    -- ... объявление readList по умолчанию дано в Prelude

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

    showsPrec _ x s   = show x ++ s
    show x            = showsPrec 0 x
    -- ... объявление для showList по умолчанию дано в Prelude
```

Классы `Read` и `Show` используются для преобразования значений к типу строка или преобразования строк к другим значениям. Аргумент типа `Int` в функциях `showsPrec` и `readsPrec` задает приоритет внешнего контекста (см. раздел 10.4).

`showsPrec` и `showList` возвращают функцию, действующую из `String` в `String`, которая обеспечивает постоянную конкатенацию их результатов посредством использования композиции функций. Также имеется специализированный вариант `show`, который использует нулевой приоритет контекста и возвращает обычный `String`. Метод `showList` предназначен для того, чтобы предоставить программисту возможность задать специализированный способ представления списков значений. Это особенно полезно для типа `Char`, где значения типа `String` должны быть представлены в двойных кавычках, а не в квадратных скобках.

Производные экземпляры классов `Read` и `Show` копируют стиль, в котором объявлен конструктор: для ввода и вывода используются инфиксные конструкторы и имена полей. Строки, порождаемые `showsPrec`, обычно могут быть прочитаны `readsPrec`.

Все типы `Prelude`, за исключением функциональных типов и типов `IO` , являются экземплярами классов `Show` и `Read`. (Если желательно, программист может легко сделать функции и типы `IO` (пустыми) экземплярами класса `Show`, обеспечив объявление экземпляра.)

Для удобства использования Prelude обеспечивает следующие вспомогательные функции:

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

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

read    :: (Read a) => String -> a
read s  =  case [x | (x,t) <- reads s, (,) <- lex t] of
              [x] -> x
              []  -> error "PreludeText.read: нет разбора"
              _   -> error "PreludeText.read: неоднозначный разбор"
```

`shows` и `reads` используют заданный по умолчанию нулевой приоритет. Функция `read` считывает ввод из строки, которая должна быть полностью потреблена процессом ввода.

Функция `lex :: ReadS String`, используемая функцией `read`, также является частью Prelude. Она считывает из ввода одну лексему, игнорируя пробельные символы перед лексемой, и возвращает символы, которые составляют лексему. Если входная строка содержит только пробельные символы, `lex` возвращает одну успешно считанную "лексему", состоящую из пустой строки. (Таким образом `lex  = [(,)]`.) Если в начале входной строки нет допустимой лексемы, `lex` завершается с ошибкой (т.е. возвращает `[]`).

6.3.4 Класс Enum

```
class  Enum a  where
    succ, pred     :: a -> a
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
    enumFrom       :: a -> [a]             -- [n..]
    enumFromThen   :: a -> a -> [a]        -- [n,n'..]
    enumFromTo     :: a -> a -> [a]        -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]   -- [n,n'..m]

    -- Заданные по умолчанию объявления даны в Prelude
```

Класс `Enum` определяет операции над последовательно упорядоченными типами. Функции `succ` и `pred` возвращают соответственно последующий и предшествующий элемент заданного значения. Функции `fromEnum` и `toEnum` преобразуют соответственно значения типа `Enum` к типу `Int` и значения типа `Int` к типу `Enum`. Методы, начинающиеся с `enumFrom` ..., используются при преобразовании арифметических последовательностей (раздел 3.10).

Экземпляры класса `Enum` можно использовать для выведения любого перечислимого типа (типы, чьи конструкторы не имеют полей), см. главу 10.

Для любого типа, который является экземпляром класса `Bounded`, а также экземпляром класса `Enum`, должны выполняться следующие условия:

- Вызовы `succ maxBound` и `pred minBound` должны завершаться с ошибкой времени выполнения программы.
- `fromEnum` и `toEnum` должны завершаться с ошибкой времени выполнения программы, если значение результата не представимо в указанном типе результата. Например, `toEnum 7 :: Bool` является ошибкой.
- `enumFrom` и `enumFromThen` должны быть определены с неявным указанием границы, так:

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

Следующие типы `Prelude` являются экземплярами класса `Enum`:

- Перечислимые типы: `()`, `Bool` и `Ordering`. Семантика этих экземпляров описана в главе 10. Например, `[LT..]` — список `[LT,EQ,GT]`.
- `Char`: экземпляр описан в главе 8, базируется на примитивных функциях, которые осуществляют преобразование между `Char` и `Int`. Например, `enumFromTo 'a' 'z'` обозначает список строчных букв в алфавитном порядке.
- Числовые типы: `Int`, `Integer`, `Float`, `Double`. Семантика этих экземпляров описана далее.

Для всех четырех числовых типов `succ` добавляет 1, а `pred` вычитает 1. Преобразования `fromEnum` и `toEnum` осуществляют преобразование между заданным типом и типом `Int`. В случае `Float` и `Double` цифры после точки могут быть потеряны. Что вернет `fromEnum`, будучи примененной к значению, которое слишком велико для того, чтобы уместиться в `Int`, — зависит от реализации.

Для типов `Int` и `Integer` функции перечисления имеют следующий смысл:

- Последовательность `enumFrom` e_1 — это список $[e_1, e_1 + 1, e_1 + 2, \ldots]$.
- Последовательность `enumFromThen` e_1 e_2 — это список $[e_1, e_1 + i, e_1 + 2i, \ldots]$, где приращение i равно $e_2 - e_1$. Приращение может быть нулевое или отрицательное. Если приращение равно нулю, все элементы списка совпадают.
- Последовательность `enumFromTo` e_1 e_3 — это список $[e_1, e_1 + 1, e_1 + 2, \ldots e_3]$. Список пуст, если $e_1 > e_3$.

- Последовательность `enumFromThenTo` e_1 e_2 e_3 — это список $[e_1, e_1 + i, e_1 + 2i, \ldots e_3]$, где приращение i равно $e_2 - e_1$. Если приращение является положительным или нулевым, список заканчивается, когда следующий элемент будет больше чем e_3; список пуст, если $e_1 > e_3$. Если приращение является отрицательным, список заканчивается, когда следующий элемент будет меньше чем e_3; список пуст, если $e1 < e_3$.

Для `Float` и `Double` семантика семейства функций `enumFrom` задается с помощью правил, описанных выше для `Int`, за исключением того, что список заканчивается, когда элементы станут больше чем $e_3 + i/2$ для положительного приращения i или когда они станут меньше чем $e_3 + i/2$ для отрицательного i.

Для всех четырех числовых типов из Prelude все функции семейства `enumFrom` являются строгими по всем своим параметрам.

6.3.5 Класс Functor

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

Класс `Functor` используется для типов, для которых можно установить соответствие (задать отображение). Списки, `IO` и `Maybe` входят в этот класс.

Экземпляры класса `Functor` должны удовлетворять следующим условиям:

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

Все экземпляры класса `Functor`, определенные в Prelude, удовлетворяют этим условиям.

6.3.6 Класс Monad

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

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

Класс `Monad` определяет основные операции над *монадами*. Для получения дополнительной информации о монадах смотрите главу 7.

"do"-выражения предоставляют удобный синтаксис для записи монадических выражений (см. раздел 3.14). Метод `fail` вызывается при ошибке сопоставления с образцом в do-выражении.

В Prelude списки, `Maybe` и `IO` являются экземплярами класса `Monad`. Метод `fail` для списков возвращает пустой список `[]`, для `Maybe` возвращает `Nothing`, а для `IO` вызывает заданное пользователем исключение в монаде IO (см. раздел 7.3).

Экземпляры класса `Monad` должны удовлетворять следующим условиям:

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

Экземпляры классов `Monad` и `Functor` должны дополнительно удовлетворять условию:

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

Все экземпляры класса `Monad`, определенные в Prelude, удовлетворяют этим условиям.

Prelude обеспечивает следующие вспомогательные функции:

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_     :: Monad m => (a -> m b) -> [a] -> m ()
(=<<)     :: Monad m => (a -> m b) -> m a -> m b
```

6.3.7 Класс Bounded

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

Класс `Bounded` используется для именования верхней и нижней границ типа. Класс `Ord` не является суперклассом класса `Bounded`, так как типы, которые не являются полностью упорядоченными, могут также иметь верхнюю и нижнюю границы. Типы `Int`, `Char`, `Bool`, `()`, `Ordering` и все кортежи являются экземплярами класса `Bounded`. Класс `Bounded` можно использовать для выведения любого перечислимого типа; `minBound` является первым в списке конструкторов объявления `data`, а `maxBound` — последним. Класс `Bounded` можно также использовать для выведения типов данных, у которых один конструктор и типы компонентов находятся в `Bounded`.

6.4 Числа

Haskell предоставляет несколько видов чисел; на числовые типы и операции над ними сильно повлияли Common Lisp и Scheme. Имена числовых функций и

операторы обычно перегружены посредством использования нескольких классов типов с отношением включения, которые изображены на рис. 6.1, стр. 112. Класс `Num` числовых типов является подклассом класса `Eq`, так как все числа можно сравнить на равенство; его подкласс `Real` также является подклассом класса `Ord`, так как остальные операции сравнения применимы ко всем числам, за исключением комплексных (определенных в библиотеке `Complex`). Класс `Integral` содержит целые числа ограниченного и неограниченного диапазона; класс `Fractional` содержит все нецелые типы; а класс `Floating` содержит все числа с плавающей точкой, действительные и комплексные.

В Prelude определены только наиболее основные числовые типы: целые числа фиксированной точности (`Int`), целые числа произвольной точности (`Integer`), числа с плавающей точкой одинарной точности (`Float`) и двойной точности (`Double`). Остальные числовые типы, такие как рациональные и комплексные числа, определены в библиотеках. В частности тип `Rational` — это отношение двух значений типа `Integer`, он определен в библиотеке `Ratio`.

Заданные по умолчанию операции над числами с плавающей точкой, определенные в Haskell Prelude, не соответствуют текущим стандартам независимой от языка арифметики (LIA). Эти стандарты требуют значительно большей сложности в числовой структуре и потому были отнесены к библиотеке. Некоторые, но не все, аспекты стандарта IEEE чисел с плавающей точкой были учтены в классе `RealFloat` из Prelude.

Стандартные числовые типы перечислены в таблице 6.1. Тип `Int` целых чисел конечной точности охватывает по меньшей мере диапазон $[-2^{29}, 2^{29}-1]$. Поскольку `Int` является экземпляром класса `Bounded`, для определения точного диапазона, заданного реализацией, можно использовать `maxBound` и `minBound`. `Float` определяется реализацией; желательно, чтобы этот тип был по меньшей мере равен по диапазону и точности типу IEEE одинарной точности. Аналогично, тип `Double` должен охватывать диапазон чисел IEEE двойной точности. Результаты исключительных ситуаций (таких как выход за верхнюю или нижнюю границу) для чисел фиксированной точности не определены; в зависимости от реализации это может быть ошибка ($\perp$), усеченное значение или специальное значение, такое как бесконечность, неопределенность и т.д.

Стандартные классы чисел и другие числовые функции, определенные в Prelude, изображены на рис. 6.2-6.3. На рис. 6.1 показаны зависимости между классами и встроенными типами, которые являются экземплярами числовых классов.

6.4.1 Числовые литералы

Синтаксис числовых литералов описан в разделе 2.5. Целые литералы представляет собой применение функции `fromInteger` к соответствующему значению типа `Integer`. Аналогично, литералы с плавающей точкой обозначают применение `fromRational` к значению типа `Rational` (то есть `Ratio Integer`). С учетом заданных типов

Тип	Класс	Описание
`Integer`	`Integral`	Целые числа произвольной точности
`Int`	`Integral`	Целые числа фиксированной точности
`(Integral a) => Ratio a`	`RealFrac`	Рациональные числа
`Float`	`RealFloat`	Действительные числа с плавающей точкой одинарной точности
`Double`	`RealFloat`	Действительные числа с плавающей точкой двойной точности
`(RealFloat a) => Complex a`	`Floating`	Комплексные числа с плавающей точкой

Таблица 6.1: Стандартные числовые типы

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

целые литералы и литералы с плавающей точкой имеют соответственно тип `(Num a) => a` и `(Fractional a) => a`. Числовые литералы определены косвенным образом для того, чтобы их можно было рассматривать как значения любого подходящего числового типа. В разделе 4.3.4 рассматривается неоднозначность перегрузки.

6.4.2 Арифметические и теоретико-числовые операции

Инфиксные методы класса `(+)`, `(*)`, `(-)` и унарная функция `negate` (которая также может быть записана как знак *минус*, стоящий перед аргументом, см. раздел 3.4) применимы ко всем числам. Методы класса `quot`, `rem`, `div` и `mod` применимы только к целым числам, тогда как метод класса `(/)` применим только к дробным. Методы класса `quot`, `rem`, `div` и `mod` удовлетворяют следующим условиям, если `y` отличен от нуля:

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

`` `quot` `` — это деление нацело с округлением в сторону нуля, тогда как результат `` `div` `` округляется в сторону отрицательной бесконечности. Метод класса `quotRem` принимает в качестве аргументов делимое и делитель и возвращает пару (частное, остаток); `divMod` определен аналогично:

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

Также для целых чисел определены предикаты even (четный) и odd (нечетный):

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

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

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

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

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

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

Рис. 6.2: Стандартные классы чисел и связанные с ними операции, часть 1

Наконец, имеются функции, которые возвращают наибольший общий делитель и наименьшее общее кратное. `gcd` x y вычисляет наибольшее (положительное) целое число, которое является делителем и x, и y, например, `gcd (-3) 6` = `3`, `gcd (-3) (-6)` = `3`, `gcd 0 4` = `4`. `gcd 0 0` вызывает ошибку времени выполнения программы.

`lcm` x y вычисляет наименьшее положительное целое число, для которого и x, и y являются делителями.

6.4.3 Возведение в степень и логарифмы

Показательная функция `exp` и логарифмическая функция `log` принимают в качестве аргумента число с плавающей точкой и используют при вычислении основание e. `logBase` a x возвращает логарифм x по основанию a. `sqrt` возвращает арифметическое значение квадратного корня числа с плавающей точкой. Имеются три операции возведения в степень, каждая из которых принимает по два аргумента: `(^)` возводит

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

class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix          :: a -> Integer
    floatDigits         :: a -> Int
    floatRange          :: a -> (Int,Int)
    decodeFloat         :: a -> (Integer,Int)
    encodeFloat         :: Integer -> Int -> a
    exponent            :: a -> Int
    significand         :: a -> a
    scaleFloat          :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                        :: a -> Bool
    atan2               :: a -> a -> a

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

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

Рис. 6.3: Стандартные классы чисел и связанные с ними операции, часть 2

любое число в неотрицательную целую степень, `(^^)` возводит дробное число в любую целую степень и `(**)` принимает два аргумента с плавающей точкой. Значение x`^0` или x`^^0` равно `1` для любого x, включая ноль; значение `0**`y не определено.

6.4.4 Абсолюная величина и знак

Число имеет *абсолютную величину* и *знак*. Функции `abs` и `signum` применимы к любому числу и удовлетворяют условию:

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

Для действительных чисел эти функции определены следующим образом:

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

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

6.4.5 Тригонометрические функции

Класс `Floating` предоставляет функции для вычисления кругового и гиперболического синуса, косинуса, тангенса и обратных функций. Имеются реализации `tan`, `tanh`, `logBase`, `**` и `sqrt`, заданные по умолчанию, но разработчики могут реализовать свои, более точные функции.

Класс `RealFloat` предоставляет версию функции для вычисления арктангенса, которая принимает два действительных аргумента с плавающей точкой. Для действительных чисел с плавающей точкой x и y `atan2` y x вычисляет угол (от положительной оси X) вектора, проведенного из начала координат в точку (x, y). `atan2` y x возвращает значение в диапазоне `[-pi, pi]`. При этом, в соответствии с семантикой Common Lisp для начала координат, поддерживются нули со знаком. `atan2` y `1`, где y находится в типе `RealFloat`, должен вернуть то же самое значение, что и `atan` y. Имеется заданное по умолчанию определение `atan2`, но разработчики могут реализовать свою, более точную функцию.

Точное определение вышеупомянутых функций такое же, как и в Common Lisp, которое, в свою очередь, соответствует предложению Пенфилда (Penfield) для APL [9]. Для подробного обсуждения ветвей, разрывностей и реализации смотрите эти ссылки.

6.4.6 Приведение и извлечение компонент

Каждая из функций `ceiling`, `floor`, `truncate` и `round` принимает в качестве аргумента действительное дробное число и возвращает целое число. `ceiling` x возвращает наименьшее целое число, которое не меньше чем x, `floor` x возвращает наибольшее целое число, которое не больше чем x. `truncate` x возвращает ближайшее к x целое число, которое находится между 0 и x включительно. `round` x возвращает ближайшее к x целое число, результат округляется в сторону четного числа, если x находится на одинаковом расстоянии от двух целых чисел.

Функция `properFraction` принимает в качестве аргумента действительное дробное число x и возвращает пару (n, f), такую, что $x = n + f$, где n — целое число с тем же знаком, что и x, f — дробное число с тем же типом и знаком, что и x, и с абсолютным значением меньше 1. Функции `ceiling`, `floor`, `truncate` и `round` можно определить в терминах `properFraction`.

Имеются две функции, которые осуществляют преобразование чисел к типу `Rational`: `toRational` возвращает рациональный эквивалент действительного аргумента с полной точностью; `approxRational` принимает два действительных дробных аргумента x и ϵ и возвращает простейшее рациональное число, которое отличается от x не более чем на ϵ, где рациональное число p/q, находящееся в приведенном виде, считается *более простым*, чем другое число p'/q', если $|p| \leq |p'|$ и $q \leq q'$. Каждый действительный интервал содержит единственное простейшее рациональное число, в частности, обратите внимание, что $0/1$ является простейшим рациональным числом из всех.

Методы класса `RealFloat` предоставляют эффективный, машинонезависимый способ получить доступ к компонентам числа с плавающей точкой. Функции `floatRadix`, `floatDigits` и `floatRange` возвращают параметры типа с плавающей точкой: соответственно основание числового представления, количество цифр этого основания в мантиссе (значащей части числа) и наибольшее и наименьшее значения, которое может принимать экспонента. Функция `decodeFloat`, будучи примененной к действительному числу с плавающей точкой, возвращает мантиссу в виде числа типа `Integer` и соответствующую экспоненту (в виде числа типа `Int`). Если `decodeFloat x` возвращает `(`m`,`n`)`, то `x` равно по значению mb^n, где b — основание с плавающей точкой, и, кроме того, либо m и n равны нулю, либо $b^{d-1} \leq m < b^d$, где d — значение `floatDigits x`. `encodeFloat` выполняет обратное преобразование. Функции `significand` и `exponent` вместе предоставляют ту же информацию, что и `decodeFloat`, но более точную, чем `Integer`, `significand x` возвращает значение того типа, что и `x`, но лежащее в пределах открытого интервала $(-1, 1)$. `exponent 0` равно нулю. `scaleFloat` умножает число с плавающей точкой на основание, возведенное в целую степень.

Функции `isNaN`, `isInfinite`, `isDenormalized`, `isNegativeZero` и `isIEEE` поддерживают числа, представимые в соответствии со стандартом IEEE. Для чисел с плавающей точкой, не соответствующих стандарту IEEE, они могут вернуть ложное значение.

Также имеются следующие функции приведения:

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

Глава 7

Основные операции ввода - вывода

Система ввода - вывода в Haskell является чисто функциональной, но при этом обладает выразительной мощью обычных языков программирования. Чтобы достичь этого, Haskell использует *монаду* для интеграции операций ввода - вывода в чисто функциональный контекст.

Монада ввода - вывода используется в Haskell как связующее звено между *значениями*, присущими функциональному языку, и *действиями*, характеризующими операции ввода - вывода и императивное программирование в общем. Порядок вычисления выражений в Haskell ограничен только зависимостями данных; реализация обладает значительной свободой в выборе этого порядка. Действия, тем не менее, должны быть упорядочены определенным образом для выполнения программы и, в частности, ввода - вывода, для того чтобы быть правильно интерпретированы. В Haskell монада ввода - вывода предоставляет пользователю способ указать последовательное связывание действий, и реализация обязана соблюдать этот порядок.

Термин *монада* происходит из отрасли математики, известной как *теория категорий*. Однако, с точки зрения программиста Haskell, лучше думать о монаде как об *абстрактном типе данных*. В случае монады ввода - вывода абстрактными значениями являются упомянутые выше *действия*. Некоторые операции являются примитивными действиями, соответствующими обычным операциям ввода - вывода. Специальные операции (методы в классе `Monad`, см. раздел 6.3.6) последовательно связывают действия, соответствующие последовательным операторам (таким как точка с запятой) в императивных языках.

7.1 Стандартные функции ввода - вывода

Хотя Haskell обеспечивает довольно сложные средства ввода - вывода, определенные в библиотеке `IO`, многие программы на Haskell можно писать, используя лишь несколько

простых функций, которые экспортируются из Prelude и которые описаны в этом разделе.

Все функции ввода - вывода, описанные здесь, имеют дело с символами. Обработка символа новой строки будет различаться в различных системах. Например, два символа ввода, возврат каретки и перевод строки, могут быть считаны как один символ новой строки. Эти функции нельзя использовать в переносимых программах для бинарного ввода - вывода.

Далее, вспомним, что `String` является синонимом для `[Char]` (раздел 6.1.2).

Функции вывода Эти функции записывают в стандартное устройство вывода (обычно это пользовательский терминал).

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()  -- добавляет символ новой строки
print    :: Show a => a -> IO ()
```

Функция `print` выводит значение любого пригодного для печати типа на стандартное устройство вывода. Пригодные для печати типы — это те типы, которые являются экземплярами класса `Show`; `print` преобразует значения в строки для вывода, используя операцию `show`, и добавляет символ новой строки.

Например, программа для печати первых 20 целых чисел и их степеней 2 могла быть записана так:

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

Функции ввода Эти функции считывают данные из стандартного устройства ввода (обычно это пользовательский терминал).

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

Операция `getChar` вызывает исключение (раздел 7.3) при появлении признака конца файла, а предикат `isEOFError`, который распознает это исключение, определен в библиотеке `IO`. Операция `getLine` вызывает исключение при тех же обстоятельствах, что и `hGetLine`, определенная в библиотеке `IO`.

Операция `getContents` возвращает весь пользовательский ввод в виде одной строки, которая считывается лениво, по мере надобности. Функция `interact` принимает в качестве аргумента функцию типа `String->String`. Весь ввод из стандартного устройства ввода передается этой функции в качестве аргумента, а результирующая строка выводится на стандартное устройство вывода.

Обычно операция `read` из класса `Read` используется для преобразования строки в значение. Функция `readIO` похожа на `read`, за исключением того, что она предупреждает монаду ввода - вывода об ошибке разбора вместо завершения программы. Функция `readLn` объединяет `getLine` и `readIO`.

Следующая программа просто удаляет все символы, не являющиеся ASCII, из своего стандартного ввода и отображает результат на своем стандартном выводе. (Функция `isAscii` определена в библиотеке.)

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

Файлы Эти функции оперируют файлами символов. Файлы указываются посредством строк, используя некоторый, зависящий от реализации, метод разрешения строк как имен файлов.

Функции `writeFile` и `appendFile` соответственно записывают или добавляют в конец строку, свой второй аргумент, в файл, свой первый аргумент. Функция `readFile` считывает файл и возвращает содержимое файла в виде строки. Файл считывается лениво, по требованию, как в `getContents`.

```
type FilePath =  String

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

Обратите внимание, что `writeFile` и `appendFile` записывают литеральную строку в файл. Для того чтобы записать значение любого пригодного для печати типа, как в `print`, сначала используется функция `show` для преобразования значения в строку.

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

7.2 Последовательные операции ввода - вывода

Конструктор типа `IO` является экземпляром класса `Monad`. Две монадические связывающие функции, методы в классе `Monad`, используются для составления

последовательностей операций ввода - вывода. Функция >> используется там, где результат первой операции не представляет интереса, например, когда он представляет собой (). Операция >>= передает результат первой операции в качестве аргумента второй операции.

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

Например, программа

```
main = readFile "input-file"                        >>= \ s ->
       writeFile "output-file" (filter isAscii s)   >>
       putStr "Фильтрация завершилась успешно\n"
```

похожа на предыдущий пример, использующий interact, но получает свой ввод из "input-file" и записывает свой вывод в "output-file". Перед завершением программы на стандартный вывод распечатывается сообщение.

Нотация do позволяет программировать в более императивном синтаксическом стиле. Слегка более сложной версией предыдущего примера была бы программа:

```
main = do
        putStr "Файл ввода: "
        ifile <- getLine
        putStr "Файл вывода: "
        ofile <- getLine
        s <- readFile ifile
        writeFile ofile (filter isAscii s)
        putStr "Фильтрация завершилась успешно\n"
```

Функция return используется для определения результата операции ввода - вывода. Например, getLine определена в терминах getChar, используя return для определения результата:

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

7.3 Обработка исключений в монаде ввода - вывода

Монада ввода - вывода включает простую систему обработки исключений. Любая операция ввода - вывода может вызвать исключение вместо возвращения результата.

Исключения в монаде ввода - вывода представлены значениями типа `IOError`. Это абстрактный тип: его конструкторы скрыты от пользователя. Библиотека `IO` определяет функции, которые конструируют и изучают значения `IOError` . Единственной функцией Prelude, которая создает значение `IOError`, является `userError`. Пользовательские значения ошибок включают строку с описанием ошибки.

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

Исключения вызываются и отлавливаются с помощью следующих функций:

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

Функция `ioError` вызывает исключение; функция `catch` устанавливает обработчик, который получает любое исключение, вызванное действием, защищенным `catch`. Исключение отлавливается самым последним обработчиком, установленным `catch`. Эти обработчики не действуют выборочно: они отлавливают все исключения. Распространение исключения нужно явно обеспечить в обработчике путем повторного вызова любого нежелательного исключения. Например, в

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

функция `f` возвращает `[]`, когда в `g` возникает исключение конца файла, иначе исключение передается следующему внешнему обработчику. Функция `isEOFError` является частью библиотеки `IO`.

Когда исключение передается за пределы главной программы, система Haskell выводит связанное с ним значение `IOError` и выходит из программы.

Метод `fail` экземпляра `IO` класса `Monad` (раздел 6.3.6) вызывает `userError` так:

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

  fail s = ioError (userError s)
```

Исключения, вызванные функциями ввода - вывода в Prelude, описаны в главе 21.

Глава 8

Стандартное начало (Prelude)

В этой главе дается описание всего Haskell Prelude. Это описание составляет *спецификацию* Prelude. Многие определения записаны с точки зрения ясности, а не эффективности, и необязательно, что спецификация реализована так, как показано здесь.

Заданные по умолчанию определения методов, данные в объявлениях `class`, составляют спецификацию *только* заданного по умолчанию метода. Они не составляют спецификацию значения метода во всех экземплярах. Рассмотрим один конкретный пример: заданный по умолчанию метод для `enumFrom` в классе `Enum` не будет работать должным образом для типов, чей диапазон превышает диапазон `Int` (потому что `fromEnum` не может отображать все значения типа в различные значения `Int`).

Показанное здесь Prelude образовано из корневого модуля `Prelude` и трех подмодулей: `PreludeList`, `PreludeText` и `PreludeIO`. Эта структура является чисто репрезентативной. Реализация не обязана использовать эту организацию для Prelude, также эти три модуля не доступны для импорта по отдельности. Только список экспорта модуля `Prelude` является значимым.

Некоторые из этих модулей импортируют модули библиотеки, такие как `Char`, `Monad`, `IO` и `Numeric`. Эти модули полностью описаны в части II. Эти списки импорта, конечно, не являются частью спецификации `Prelude`. То есть реализация свободна в выборе импортировать больше или меньше модулей библиотеки, по своему усмотрению.

Примитивы, которые не не определимы на Haskell, обозначаются именами, начинающимися с "`prim`"; они определены системнозависимым способом в модуле `PreludeBuiltin` и полностью не показаны. Объявления экземпляров, которые просто связывают примитивы с методами класса, пропущены. Некоторые из наиболее подробных экземпляров с очевидными функциональными возможностями были пропущены ради краткости.

Объявления специальных типов, таких как `Integer` или `()`, включены в Prelude для полноты, даже если объявление может оказаться неполным или синтаксически

недопустимым. Пропуски "`...`" часто используются в местах, где остаток определения не может быть задан на Haskell.

Для того чтобы сократить возникновение неожиданных ошибок неоднозначности и улучшить эффективность, множество общеупотребительных функций над списками чаще используют тип `Int`, чем более общий числовой тип, такой как `Integral a` или `Num a`. Этими функциями являются: `take`, `drop`, `!!`, `length`, `splitAt` и `replicate`. Более общие версии заданы в библиотеке `List` и имеют приставку "`generic`", например, `genericLength`.

```
module Prelude (
    module PreludeList, module PreludeText, module PreludeIO,
    Bool(False, True),
    Maybe(Nothing, Just),
    Either(Left, Right),
    Ordering(LT, EQ, GT),
    Char, String, Int, Integer, Float, Double, Rational, IO,

--      Эти встроенные типы определены в Prelude, но
--      обозначены встроенным синтаксисом и не могут
--      появляться в списке экспорта.
--  Списочный тип: []((:), [])
--  Типы кортежей: (,)((,)), (,,)((,,)), etc.
--  Тривиальный тип: ()(())
--  Функциональный тип: (->)

    Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
    Bounded(minBound, maxBound),
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Real(toRational),
    Integral(quot, rem, div, mod, quotRem, divMod, toInteger),
    Fractional((/), recip, fromRational),
    Floating(pi, exp, log, sqrt, (**), logBase, sin, cos, tan,
             asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh),
    RealFrac(properFraction, truncate, round, ceiling, floor),
    RealFloat(floatRadix, floatDigits, floatRange, decodeFloat,
              encodeFloat, exponent, significand, scaleFloat, isNaN,
              isInfinite, isDenormalized, isIEEE, isNegativeZero, atan2),
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    maybe, either,
    (&&), (||), not, otherwise,
    subtract, even, odd, gcd, lcm, (^), (^^),
    fromIntegral, realToFrac,
    fst, snd, curry, uncurry, id, const, (.), flip, ($), until,
    asTypeOf, error, undefined,
    seq, ($!)
  ) where

import PreludeBuiltin                       -- Содержит все ‘примитивные’
                                            -- значения
import UnicodePrims( primUnicodeMaxChar )   -- Примитивы Unicode
import PreludeList
import PreludeText
import PreludeIO
import Ratio( Rational )
```

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

-- Оператор (:) является встроенным синтаксисом и не может быть задан с помощью
-- infix-объявления; но его ассоциативность и приоритет заданы:
--   infixr 5  :

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

-- Стандартные типы, классы, экземпляры и относящиеся к ним функции

-- Классы равенства (Eq) и упорядочивания (Ordering)

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

        -- Минимальное полное определение:
        --      (==) or (/=)
    x /= y     =  not (x == y)
    x == y     =  not (x /= y)

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

        -- Минимальное полное определение:
        --      (<=) или compare
        -- Использование compare может оказаться более эффективным для сложных
        -- типов.
    compare x y
         | x == y    =  EQ
         | x <= y    =  LT
         | otherwise =  GT

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

-- обратите внимание, что (min x y, max x y) = (x,y) или (y,x)
    max x y
         | x <= y    =  y
         | otherwise =  x
    min x y
         | x <= y    =  x
         | otherwise =  y
```

```
-- Классы перечисления (Enum) и границ (Bounded)

class  Enum a  where
    succ, pred       :: a -> a
    toEnum           :: Int -> a
    fromEnum         :: a -> Int
    enumFrom         :: a -> [a]             -- [n..]
    enumFromThen     :: a -> a -> [a]        -- [n,n'..]
    enumFromTo       :: a -> a -> [a]        -- [n..m]
    enumFromThenTo   :: a -> a -> a -> [a]   -- [n,n'..m]

        -- Минимальное полное определение:
        --      toEnum, fromEnum
        --
        -- ЗАМЕЧАНИЕ: эти методы по умолчанию только делают вид,
        --            что инъективно отображают типы в Int, используя
        --            fromEnum и toEnum.
    succ             =  toEnum . (+1) . fromEnum
    pred             =  toEnum . (subtract 1) . fromEnum
    enumFrom x       =  map toEnum [fromEnum x ..]
    enumFromTo x y   =  map toEnum [fromEnum x .. fromEnum y]
    enumFromThen x y =  map toEnum [fromEnum x, fromEnum y ..]
    enumFromThenTo x y z =
                        map toEnum [fromEnum x, fromEnum y .. fromEnum z]

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

-- Числовые классы

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

        -- Минимальное полное определение:
        --      Все, за исключением negate или (-)
    x - y            =  x + negate y
    negate x         =  0 - x

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

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

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

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

        -- Минимальное полное определение:
        --      fromRational и (recip или (/))
    recip x          =  1 / x
    x / y            =  x * recip y

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

        -- Минимальное полное определение:
        --      pi, exp, log, sin, cos, sinh, cosh
        --      asin, acos, atan
        --      asinh, acosh, atanh
    x ** y           =  exp (log x * y)
    logBase x y      =  log y / log x
    sqrt x           =  x ** 0.5
    tan  x           =  sin  x / cos  x
    tanh x           =  sinh x / cosh x
```

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

        -- Минимальное полное определение:
        --      properFraction
    truncate x       =  m  where (m,_) = properFraction x

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

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

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

```
class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix       :: a -> Integer
    floatDigits      :: a -> Int
    floatRange       :: a -> (Int,Int)
    decodeFloat      :: a -> (Integer,Int)
    encodeFloat      :: Integer -> Int -> a
    exponent         :: a -> Int
    significand      :: a -> a
    scaleFloat       :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                     :: a -> Bool
    atan2            :: a -> a -> a
        -- Минимальное полное определение:
        --      Все, за исключением exponent, significand,
        --                  scaleFloat, atan2
    exponent x       =  if m == 0 then 0 else n + floatDigits x
                        where (m,n) = decodeFloat x
    significand x    =  encodeFloat m (- floatDigits x)
                        where (m,_) = decodeFloat x
    scaleFloat k x   =  encodeFloat m (n+k)
                        where (m,n) = decodeFloat x
    atan2 y x
      | x>0           =  atan (y/x)
      | x==0 && y>0   =  pi/2
      | x<0  && y>0   =  pi + atan (y/x)
      |(x<=0 && y<0)  ||
       (x<0 && isNegativeZero y) ||
       (isNegativeZero x && isNegativeZero y)
                      = -atan2 (-y) x
      | y==0 && (x<0 || isNegativeZero x)
                      =  pi    -- должен быть после предыдущей проверки y на нуль
      | x==0 && y==0  =  y     -- должен быть после других двойных проверок
                               -- на нуль
      | otherwise     =  x + y -- x или y равен NaN, возвращает NaN
                               -- (посредством +)

-- Числовые функции

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

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

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

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

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

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

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

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

-- Монадические классы

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

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

        -- Минимальное полное определение:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s

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

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

-- Функции вида xxxM работают со списком аргументов, но повышают
-- тип функции или элемента списка до монадического типа
mapM            :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as       =  sequence (map f as)

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

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

```
-- Тривиальный тип
data  ()  =  ()  deriving (Eq, Ord, Enum, Bounded)
        -- Недопустимо в Haskell; только для примера

-- Функциональный тип

-- идентичная функция
id                :: a -> a
id x              =  x

-- константная функция
const             :: a -> b -> a
const x _         =  x

-- композиция функций
(.)               :: (b -> c) -> (a -> b) -> a -> c
f . g             =  \ x -> f (g x)

-- flip f  принимает свои (первые) два аргумента в обратном порядке для f.
flip              :: (a -> b -> c) -> b -> a -> c
flip f x y        =  f y x

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

-- правоассоциативное инфиксное применение операторов
-- (полезно в стиле с возобновляющейся передачей)
($), ($!) :: (a -> b) -> a -> b
f $  x    =  f x
f $! x    =  x `seq` f x

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

-- Булевы функции
(&&), (||)        :: 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

-- Символьный тип
data Char = ... 'a' | 'b' ... -- значения Unicode

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

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

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

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

type  String = [Char]

-- Тип "может быть" (Maybe)
data  Maybe a  =  Nothing | Just a      deriving (Eq, Ord, Read, Show)

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

instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)

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

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

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

-- Тип ввода - вывода
data IO a = ...         -- абстрактный

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

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

```
-- Тип упорядочивания

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

-- Стандартные числовые типы.  Объявления данных для этих типов нельзя
-- выразить непосредственно на Haskell, т.к. конструируемые списки были бы
-- слишком длинными.

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

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

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

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

```
-- Экземпляры Enum для Float и Double слегка необычны.
-- Функция ‘toEnum’ выполняет усечение числа до Int.  Определения
-- enumFrom и enumFromThen позволяют использовать числа с плавающей точкой
-- в арифметических последовательностях: [0,0.1 .. 0.95].  Тем не менее,
-- ошибки roundoff делают это несколько сомнительным.
-- В этом примере может быть 10 или 11 элементов, в зависимости от того,
-- как представлено 0.1.
instance  Enum Float  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- может вызвать переполнение
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo

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

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

-- Списки
data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
        -- Недопустимо в Haskell; только для примера

instance Functor [] where
    fmap = map

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

```
-- Кортежи

data  (a,b)   =  (a,b)    deriving (Eq, Ord, Bounded)
data  (a,b,c) =  (a,b,c)  deriving (Eq, Ord, Bounded)
        -- Недопустимо в Haskell; только для примера

-- проекции компонент для пары:
-- (NB: не предусмотрено для кортежей размера 3, 4 и т.д.)
fst             :: (a,b) -> a
fst (x,y)       =  x

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

-- curry преобразует развернутую функцию (функцию двух аргументов) в свернутую
-- функцию (функцию над парой аргументов)
-- uncurry преобразует свернутую функцию в развернутую функцию
curry           :: ((a, b) -> c) -> a -> b -> c
curry f x y     =  f (x, y)

uncurry         :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p     =  f (fst p) (snd p)

-- Разные функции

-- until p f получает результат применения f до тех пор, пока p выполняется.

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

-- asTypeOf является версией const с ограниченным набором типов.  Она обычно
-- используется в качестве инфиксного оператора, и ее типизация приводит к тому,
-- что ее первый аргумент (который обычно является перегруженным) должен иметь
-- тот же тип, что второй аргумент.
asTypeOf        :: a -> a -> a
asTypeOf        =  const

-- error останавливает вычисление и выводит на экран сообщение об ошибке

error           :: String -> a
error           =  primError

-- Ожидается, что компиляторы распознают это и вставят
-- более подходящие сообщения об ошибках для контекста, в котором возник
-- undefined.

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

8.1 Prelude PreludeList

```
-- Стандартные функции над списками

module PreludeList (
    map, (++), filter, concat, concatMap,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum,
    zip, zip3, zipWith, zipWith3, unzip, unzip3)
  where

import qualified Char(isSpace)

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

-- Отображение (map) и добавление в конец (append)
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

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

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

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

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

-- head и tail извлекают соответственно первый и последний элементы
-- конечного списка.  last и init являются
-- двойственными функциями, которые выполняются с конца конечного списка,
-- а не с начала.

head             :: [a] -> a
head (x:_)       =  x
head []          =  error "Prelude.head: пустой список"

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

```
last             :: [a] -> a
last [x]         =  x
last (_:xs)      =  last xs
last []          =  error "Prelude.last: пустой список"

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

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

-- length возвращает длину конечного списка в виде Int.
length           :: [a] -> Int
length []        =  0
length (_:l)     =  1 + length l

-- Оператор доступа к элементам списка по индексу, начало --- в 0.
(!!)                :: [a] -> Int -> a
xs     !! n | n < 0 =  error "Prelude.!!: отрицательный индекс"
[]     !! _         =  error "Prelude.!!: слишком большой индекс"
(x:_)  !! 0         =  x
(_:xs) !! n         =  xs !! (n-1)

-- foldl, будучи примененной к своим аргументам: бинарному оператору, начальному
-- значению (обычно левому аргументу из тождества оператора) и списку,
-- сокращает список, используя бинарный оператор слева направо:
--  foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
-- foldl1 является вариантом предыдущей функции, она не имеет аргумента с
-- начальным значением, и поэтому должна применяться к непустым спискам.
-- scanl похожа на foldl, но возвращает список успешно сокращенных значений слева:
--      scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
-- Обратите внимание, что last (scanl f z xs) == foldl f z xs.
-- scanl1 похожа на предыдущую функцию, но без начального элемента:
--      scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs

foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  =  foldl f x xs
foldl1 _ []      =  error "Prelude.foldl1: пустой список"

scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs     =  q : (case xs of
                           []   -> []
                           x:xs -> scanl f (f q x) xs)

scanl1           :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs)  =  scanl f x xs
scanl1 _ []      =  []
```

```
-- foldr, foldr1, scanr и scanr1 являются двойственными дополнениями
-- описанных выше функций; они действуют справа налево.
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)
foldr1 _ []      =  error "Prelude.foldr1: пустой список"

scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     =  [q0]
scanr f q0 (x:xs) =  f x q : qs
                     where qs@(q:_) = scanr f q0 xs

scanr1          :: (a -> a -> a) -> [a] -> [a]
scanr1 f []     =  []
scanr1 f [x]    =  [x]
scanr1 f (x:xs) =  f x q : qs
                   where qs@(q:_) = scanr1 f xs

-- iterate f x возвращает бесконечный список повторных применений f к 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 представляет собой бесконечный список, где каждый элемент равен x.
repeat           :: a -> [a]
repeat x         =  xs where xs = x:xs

-- replicate n x представляет собой список длины n, где каждый элемент равен x.
replicate        :: Int -> a -> [a]
replicate n x    =  take n (repeat x)

-- cycle связывает конечный список в круговой или, что то же самое,
-- бесконечно повторяет исходный список. Он идентичен
-- бесконечным спискам.
cycle            :: [a] -> [a]
cycle []         =  error "Prelude.cycle: пустой список"
cycle xs         =  xs' where xs' = xs ++ xs'

-- take n, будучи примененной к списку xs, возвращает префикс xs длины n,
-- или сам xs, если n > length xs.  drop n xs возвращает суффикс xs
-- после первых n элементов, или [], если n > length xs.  splitAt n xs
-- эквивалентна (take n xs, drop n xs).
take                   :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

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

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

-- takeWhile, будучи примененной к предикату p и списку xs, возвращает самый
-- длинный префикс (возможно пустой) xs, элементы которого удовлетворяют p.
-- dropWhile p xs возвращает оставшийся суффикс.  span p xs эквивалентна
-- (takeWhile p xs, dropWhile p xs), тогда как break p использует отрицание p.

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

dropWhile               :: (a -> Bool) -> [a] -> [a]
dropWhile p []          =  []
dropWhile p xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  xs

span, break             :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p xs@(x:xs')
            | p x       =  (x:ys,zs)
            | otherwise =  ([],xs)
                           where (ys,zs) = span p xs'

break p                 =  span (not . p)

-- lines разбивает строку в местах символов новой строки на список строк.
-- Полученные строки не содержат символов новой строки. Аналогично, words
-- разбивает строку на список строк в местах пробельных символов.
-- unlines и unwords являются обратными операциями.
-- unlines соединяет строки, добавляя в конец каждой символ новой строки,
-- а unwords соединяет слова, отделяя их друг от друга пробелами.

lines            :: String -> [String]
lines ""         =  []
lines s          =  let (l, s') = break (== '\n') s
                        in  l : case s' of
                                  []      -> []
                                  (_:s'') -> lines s''

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

```
unlines          :: [String] -> String
unlines          =  concatMap (++ "\n")

unwords          :: [String] -> String
unwords []       =  ""
unwords ws       =  foldr1 (\w s -> w ++ ' ':s) ws

-- reverse xs возвращает элементы списка xs в обратном порядке.
-- Список xs должен быть конечным.
reverse          :: [a] -> [a]
reverse          =  foldl (flip (:)) []

-- and возвращает конъюнкцию списка булевых значений.  Для того чтобы результат
-- был истиной, список должен быть конечным; False является результатом наличия
-- значения False в элементе с конечным индексом конечного или
-- бесконечного списка.
-- or является двойственной к and функцией, в ней используется дизъюнкция.
and, or          :: [Bool] -> Bool
and              =  foldr (&&) True
or               =  foldr (||) False

-- Будучи примененной к предикату и списку, any определяет, есть ли хотя бы один
-- элемент списка, который удовлетворяет предикату.  Аналогично all определяет,
-- все ли элементы списка удовлетворяет предикату.
any, all         :: (a -> Bool) -> [a] -> Bool
any p            =  or . map p
all p            =  and . map p

-- elem является предикатом, который определяет, является ли аргумент элементом
-- списка; обычно он записывается в инфиксной форме,
-- например, x `elem` xs. notElem является отрицанием предыдущей функции.
elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           =  any (== x)
notElem x        =  all (/= x)

-- lookup key assocs ищет ключ в ассоциативном списке.
lookup           :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []    =  Nothing
lookup key ((x,y):xys)
    | key == x   =  Just y
    | otherwise  =  lookup key xys

-- sum и product вычисляют соответственно сумму и произведение чисел из конечного
-- списка.
sum, product     :: (Num a) => [a] -> a
sum              =  foldl (+) 0
product          =  foldl (*) 1

-- maximum и minimum возвращают соответственно максимальное и минимальное значение
-- из списка, который должен быть непуст, конечен и содержать элементы, которые
-- можно упорядочить.
maximum, minimum :: (Ord a) => [a] -> a
maximum []       =  error "Prelude.maximum: пустой список"
maximum xs       =  foldl1 max xs
```

```
minimum []       =  error "Prelude.minimum: пустой список"
minimum xs       =  foldl1 min xs

-- zip принимает в качестве аргументов два списка и возвращает список
-- соответствующих пар. Если один из входных списков короче,
-- дополнительные элементы более длинного списка игнорируются.
-- zip3 принимает в качестве аргументов три списка и возвращает список кортежей
-- размера 3. Функции zip для более длинных кортежей находятся в библиотеке List.
zip              :: [a] -> [b] -> [(a,b)]
zip              =  zipWith (,)

zip3             :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3             =  zipWith3 (,,)

-- Семейство функций zipWith является обобщением семейства функций zip; они
-- упаковывают элементы списков с помощью функции, заданной в качестве
-- первого аргумента, вместо функции создания кортежей.
-- Например, zipWith (+), будучи примененной к двум спискам, порождает список
-- соответствующих сумм.
zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 =  z a b : zipWith z as bs
zipWith _ _ _    =  []

zipWith3         :: (a->b->c->d) -> [a]->[b]->[c]->[d]
zipWith3 z (a:as) (b:bs) (c:cs)
                 =  z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _ =  []

-- unzip преобразует список пар в пару списков.
unzip            :: [(a,b)] -> ([a],[b])
unzip            =  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

unzip3           :: [(a,b,c)] -> ([a],[b],[c])
unzip3           =  foldr (\(a,b,c) ~(as,bs,cs) -> (a:as,b:bs,c:cs))
                          ([],[],[])
```

8.2 Prelude PreludeText

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

-- Экземпляры классов Read и Show для
--      Bool, Maybe, Either, Ordering
-- созданы посредством инструкций "deriving" в Prelude.hs

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

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

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

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

        -- Минимальное полное определение:
        --      readsPrec
    readList         = readParen False (\r -> [pr | ("[",s)  <- lex r,
                                                   pr       <- readl s])
                       where readl  s = [([],t)   | ("]",t)  <- lex s] ++
                                        [(x:xs,u) | (x,t)    <- reads s,
                                                    (xs,u)   <- readl' t]
                             readl' s = [([],t)   | ("]",t)  <- lex s] ++
                                        [(x:xs,v) | (",",t)  <- lex s,
                                                    (x,u)    <- reads t,
                                                    (xs,v)   <- readl' u]

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

        -- Минимальное полное определение:
        --      show или showsPrec
    showsPrec _ x s   = show x ++ s

    show x            = showsPrec 0 x ""

    showList []       = showString "[]"
    showList (x:xs)   = showChar '[' . shows x . showl xs
                        where showl []     = showChar ']'
                              showl (x:xs) = showChar ',' . shows x .
                                             showl xs
```

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

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

read            :: (Read a) => String -> a
read s          =  case [x | (x,t) <- reads s, ("","") <- lex t] of
                        [x] -> x
                        []  -> error "Prelude.read: нет разбора"
                        _   -> error "Prelude.read: неоднозначный разбор"

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    ]

-- Этот лексический анализатор не полностью соответствует лексическому синтаксису
-- Haskell.
-- Текущие ограничения:
--    Квалифицированные имена не управляются должным образом
--    Восьмиричные и шестнадцатиричные цифры не распознаются как отдельный токен
--    Комментарии не обрабатываются должным образом

lex             :: ReadS String
lex ""          =  [("","")]
lex (c:s)
   | isSpace c  =  lex (dropWhile isSpace s)
lex ('\":s)     =  [('\":ch++"'", t) | (ch,'\":t)  <- lexLitChar s,
                                       ch /= \" ]
lex ('\:s)      =  [('\:str, t)      | (str,t) <- lexString s]
                   where
                   lexString ('\:s) = [("\"",s)]
                   lexString s = [(ch++str, u)
                                       | (ch,t)  <- lexStrItem s,
                                         (str,u) <- lexString t  ]

                   lexStrItem ('\\':'&':s) =  [("\\&",s)]
                   lexStrItem ('\\':c:s) | isSpace c
                                       =  [("\\&",t) |
                                           '\\':t <-
                                               [dropWhile isSpace s]]
                   lexStrItem s        =  lexLitChar s
```

```
lex (c:s) | isSingle c = [([c],s)]
          | isSym c    = [(c:sym,t)       | (sym,t) <- [span isSym s]]
          | isAlpha c  = [(c:nam,t)       | (nam,t) <- [span isIdChar s]]
          | isDigit c  = [(c:ds++fe,t)    | (ds,s)  <- [span isDigit s],
                                            (fe,t)  <- lexFracExp s     ]
          | otherwise  = []    -- плохой символ
             where
              isSingle c =  c `elem` ",;()[]{}_`"
              isSym c    =  c `elem` "!@#$%&*+./<=>?\\^|:-~"
              isIdChar c =  isAlphaNum c || c `elem` "_'"

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

              lexExp (e:s) | e `elem` "eE"
                       = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
                                                 (ds,u) <- lexDigits t] ++
                         [(e:ds,t)   | (ds,t) <- lexDigits s]
              lexExp s = [("",s)]

instance  Show Int  where
    showsPrec n = showsPrec n . toInteger
        -- Преобразование к Integer позволяет избежать
        -- возможного противоречия с minInt

instance  Read Int  where
  readsPrec p r = [(fromInteger i, t) | (i,t) <- readsPrec p r]
        -- Считывание в тип Integer позволяет избежать
        -- возможного противоречия с minInt

instance  Show Integer  where
    showsPrec           = showSigned showInt

instance  Read Integer  where
    readsPrec p         = readSigned readDec

instance  Show Float  where
    showsPrec p         = showFloat

instance  Read Float  where
    readsPrec p         = readSigned readFloat

instance  Show Double  where
    showsPrec p         = showFloat

instance  Read Double  where
    readsPrec p         = readSigned readFloat

instance  Show ()  where
    showsPrec p () = showString "()"
```

```
instance Read () where
    readsPrec p    = readParen False
                           (\r -> [((),t) | ("(",s) <- lex r,
                                            (")",t) <- lex s ] )
instance  Show Char  where
    showsPrec p '\'' = showString "'\\''"
    showsPrec p c    = showChar '\'' . showLitChar c . showChar '\''

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

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

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

instance  (Show a) => Show [a]  where
    showsPrec p      = showList

instance  (Read a) => Read [a]  where
    readsPrec p      = readList

-- Кортежи

instance  (Show a, Show b) => Show (a,b)  where
    showsPrec p (x,y) = showChar '(' . shows x . showChar ',' .
                                       shows y . showChar ')'

instance  (Read a, Read b) => Read (a,b)  where
    readsPrec p       = readParen False
                           (\r -> [((x,y), w) | ("(",s) <- lex r,
                                                (x,t)   <- reads s,
                                                (",",u) <- lex t,
                                                (y,v)   <- reads u,
                                                (")",w) <- lex v ] )

-- Другие кортежи имеют сходные экземпляры классов Read и Show
```

8.3 Prelude PreludeIO

```
module PreludeIO (
    FilePath, IOError, ioError, userError, catch,
    putChar, putStr, putStrLn, print,
    getChar, getLine, getContents, interact,
    readFile, writeFile, appendFile, readIO, readLn
  ) where

import PreludeBuiltin

type  FilePath = String

data IOError    -- Внутреннее представление этого типа зависит от системы

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

ioError    ::  IOError -> IO a
ioError    =   primIOError

userError  ::  String -> IOError
userError  =   primUserError

catch      ::  IO a -> (IOError -> IO a) -> IO a
catch      =   primCatch

putChar    :: Char -> IO ()
putChar    =  primPutChar

putStr     :: String -> IO ()
putStr s   =  mapM_ putChar s

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

print      :: Show a => a -> IO ()
print x    =  putStrLn (show x)

getChar    :: IO Char
getChar    =  primGetChar

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

getContents :: IO String
getContents =  primGetContents
```

```
interact    ::  (String -> String) -> IO ()
-- hSetBuffering гарантирует ожидаемое интерактивное поведение
interact f  =  do hSetBuffering stdin  NoBuffering
                  hSetBuffering stdout NoBuffering
                  s <- getContents
                  putStr (f s)

readFile   :: FilePath -> IO String
readFile   =  primReadFile

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

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

  -- вместо ошибки вызывает исключение
readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: нет разбора")
              _   -> ioError (userError "Prelude.readIO: неоднозначный разбор")

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

Глава 9

Синтаксический справочник

9.1 Соглашения об обозначениях

Эти соглашения об обозначениях используются для представления синтаксиса:

$[pattern]$	необязательный
$\{pattern\}$	ноль или более повторений
$(pattern)$	группировка
$pat_1 \mid pat_2$	выбор
$pat_{\langle pat' \rangle}$	разность — элементы, порождаемые с помощью pat, за исключением элементов, порождаемых с помощью pat'
`fibonacci`	терминальный синтаксис в машинописном шрифте

Повсюду используется BNF-подобный синтаксис, чьи правила вывода имеют вид:

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

Перевод:
нетерминал →
  альтернатива$_1$
  | *альтернатива*$_2$
  | ...
  | *альтернатива*$_n$

В синтаксисе, который следует далее, есть некоторые семейства нетерминалов, индексированные уровнями приоритета (записанными как верхний индекс). Аналогично, нетерминалы *op* (*оператор*), *varop* (*оператор-переменной*) и *conop* (*оператор-конструктора*) могут иметь двойной индекс: букву l, r или n

соответственно для левоассоциативности, правоассоциативности или отсутствия ассоциативности и уровень приоритета. Переменная уровня приоритета i изменяется в пределах от 0 до 9, переменная ассоциативности a изменяется в диапазоне $\{l,\ r,\ n\}$. Например,

aexp → `(` exp^{i+1} $qop^{(a,i)}$ `)`

на самом деле обозначает 30 правил вывода с 10 подстановками для i и 3 для a.

И в лексическом, и в контекстно-свободном синтаксисе есть некоторые неоднозначности, которые разрешаются посредством создания грамматических фраз наибольшей из возможных длины, действуя слева направо (в восходящем синтаксическом анализе при конфликте сдвиг - свертка выполняется сдвиг). В лексическом синтаксисе это правило “максимального потребления”. В контекстно-свободном синтаксисе это означает, что условные выражения, let-выражения и лямбда-абстракции продолжаются вправо насколько возможно.

9.2 Лексический синтаксис

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

whitespace → *whitestuff* {*whitestuff*}
whitestuff → *whitechar* | *comment* | *ncomment*
whitechar → *newline* | *vertab* | *space* | *tab* | *uniWhite*
newline → *return linefeed* | *return* | *linefeed* | *formfeed*
return → возврат каретки
linefeed → перевод строки
vertab → вертикальная табуляция
formfeed → перевод страницы
space → пробел
tab → горизонтальная табуляция
uniWhite → любой пробельный символ Unicode

comment → *dashes* [$any_{\langle symbol \rangle}$ {*any*}] *newline*
dashes → `--` {`-`}
opencom → `{-`
closecom → `-}`
ncomment → *opencom ANYseq* {*ncomment ANYseq*} *closecom*
ANYseq → $\{ANY\}_{\langle \{ANY\}\ (\ opencom\ |\ closecom\ )\ \{ANY\} \rangle}$

ANY → *graphic* | *whitechar*
any → *graphic* | *space* | *tab*
graphic → *small* | *large* | *symbol* | *digit* | *special* | `:` | `"` | `'`

small → *ascSmall* | *uniSmall* | `_`
ascSmall → `a` | `b` | ... | `z`
uniSmall → любая буква Unicode нижнего регистра

large → *ascLarge* | *uniLarge*
ascLarge → `A` | `B` | ... | `Z`
uniLarge → любая буква Unicode верхнего регистра или заглавная
symbol → *ascSymbol* | *uniSymbol*$_{\langle special \mid \_ \mid : \mid " \mid ' \rangle}$

ascSymbol → `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@`
| `\` | `^` | `|` | `-` | `~`
uniSymbol → любой символ или знак пунктуации Unicode
digit → *ascDigit* | *uniDigit*
ascDigit → `0` | `1` | ... | `9`
uniDigit → любая десятичная цифра Unicode
octit → `0` | `1` | ... | `7`
hexit → *digit* | `A` | ... | `F` | `a` | ... | `f`

Перевод:
программа →
{ *лексема* | *пробельная-строка* }
лексема →
квалифицированный-идентификатор-переменной
| *квалифицированный-идентификатор-конструктора*
| *квалифицированный-символ-переменной*
| *квалифицированный-символ-конструктора*
| *литерал*
| *специальная-лексема*
| *зарезервированный-оператор*
| *зарезервированный-идентификатор*
литерал →
целый-литерал
| *литерал-с-плавающей-точкой*
| *символьный-литерал*
| *строковый-литерал*
специальная-лексема →
`(` | `)` | `,` | `;` | `[` | `]` | `` ` `` | `{` | `}`

пробельная-строка →
пробельный-элемент {*пробельный-элемент*}

пробельный-элемент →
  пробельный-символ
  | *комментарий*
  | *вложенный-комментарий*
пробельный-символ →
  новая-строка
  | *вертикальная-табуляция*
  | *пробел*
  | *горизонтальная-табуляция*
  | *пробельный-символ-Unicode*
новая-строка →
  возврат-каретки перевод-строки
  | *возврат-каретки*
  | *перевод-строки*
  | *перевод-страницы*

комментарий →
  тире [*любой-символ*$_{\langle символ \rangle}$ {*любой-символ*}] *новая-строка*
тире →
  `--` {`-`}
начало-комментария →
  `{-`
конец-комментария →
  `-}`
вложенный-комментарий →
  начало-комментария ЛЮБАЯ-последовательность
    {*вложенный-комментарий ЛЮБАЯ-последовательность*} *конец-комментария*
ЛЮБАЯ-последовательность →
  {*ЛЮБОЙ-символ*}$_{\langle \{ЛЮБОЙ\text{-}символ\}\ (\ начало\text{-}комментария\ |\ конец\text{-}комментария\ )\ \{ЛЮБОЙ\text{-}символ\}\rangle}$
ЛЮБОЙ-символ →
  графический-символ
  | *пробельный-символ*
любой-символ →
  графический-символ
  | *пробел*
  | *горизонтальная-табуляция*
графический-символ →
  маленькая-буква
  | *большая-буква*
  | *символ*
  | *цифра*
  | *специальная-лексема*
  | `:` | `"` | `'`

маленькая-буква →
  маленькая-буква-ASCII
  | *маленькая-буква-Unicode*
  | `_`
маленькая-буква-ASCII →
  `a` | `b` | ... | `z`

большая-буква →
  большая-буква-ASCII
  | *большая-буква-Unicode*
большая-буква-ASCII →
  `A` | `B` | ... | `Z`
символ →*символ-ASCII*
  | *символ-Unicode*$_{\langle \textit{специальная-лексема} \mid \texttt{\_} \mid \texttt{:} \mid \texttt{"} \mid \texttt{'} \rangle}$

символ-ASCII →
  `!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | @
  |\ | `^` | `|` | `-` | `~`
символ-Unicode →
  любой символ или знак пунктуации Unicode
цифра →
  цифра-ASCII
  | *цифра-Unicode*
цифра-ASCII →
  `0` | `1` | ... | `9`
цифра-Unicode →
  любая десятичная цифра Unicode
восьмиричная-цифра →
  `0` | `1` | ... | `7`
шестнадцатиричная-цифра →
  цифра | `A` | ... | `F` | `a` | ... | `f`

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

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

varid (переменные)
conid (конструкторы)
tyvar → *varid* (переменные типов)

tycon → *conid* (конструкторы типов)
tycls → *conid* (классы типов)
modid → *conid* (модули)

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

decimal → *digit*{*digit*}
octal → *octit*{*octit*}
hexadecimal → *hexit*{*hexit*}

integer → *decimal*
| `0o` *octal* | `0O` *octal*
| `0x` *hexadecimal* | `0X` *hexadecimal*
float → *decimal* `.` *decimal* [*exponent*]
| *decimal exponent*
exponent → (`e` | `E`) [`+` | `-`] *decimal*

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

Перевод:
идентификатор-переменной →
(*маленькая-буква* {*маленькая-буква* | *большая-буква* | *цифра* | `'` })$_{\langle \text{зарезервированный-идентификатор} \rangle}$
идентификатор-конструктора →
большая-буква {*маленькая-буква* | *большая-буква* | *цифра* | `'` }
зарезервированный-идентификатор →
`case` | `class` | `data` | `default` | `deriving` | `do` | `else`
| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance`
| `let` | `module` | `newtype` | `of` | `then` | `type` | `where` | `_`

символ-переменной →
  (*символ* {*символ* | :})$_{\langle \textit{зарезервированный-оператор} \mid \textit{тире} \rangle}$
символ-конструктора →
  (: {*символ* | :})$_{\langle \textit{зарезервированный-оператор} \rangle}$
зарезервированный-оператор →
  `..` | `:` | `::` | `=` | `\` | `|` | `<-` | `->` | `@` | `~` | `=>`

идентификатор-переменной
    (переменные)
идентификатор-конструктора
    (конструкторы)
переменная-типа →
  идентификатор-переменной
    (переменные типов)
конструктор-типа →
  идентификатор-конструктора
    (конструкторы типов)
класс-типа →
  идентификатор-конструктора
    (классы типов)
идентификатор-модуля →
  идентификатор-конструктора
    (модули)

квалифицированный-идентификатор-переменной →
  [*идентификатор-модуля* .] *идентификатор-переменной*
квалифицированный-идентификатор-конструктора →
  [*идентификатор-модуля* .] *идентификатор-конструктора*
квалифицированный-конструктор-типа →
  [*идентификатор-модуля* .] *конструктор-типа*
квалифицированный-класс-типа →
  [*идентификатор-модуля* .] *класс-типа*
квалифицированный-символ-переменной →
  [*идентификатор-модуля* .] *символ-переменной*
квалифицированный-символ-конструктора →
  [*идентификатор-модуля* .] *символ-конструктора*

десятичный-литерал →
  цифра{*цифра*}
восьмиричный-литерал →
  восьмиричная-цифра{*восьмиричная-цифра*}
шестнадцатиричный-литерал →
  шестнадцатиричная-цифра{*шестнадцатиричная-цифра*}

целый-литерал →

десятичный-литерал
| `0o` *восьмиричный-литерал*
| `0O` *восьмиричный-литерал*
| `0x` *шестнадцатиричный-литерал*
| `0X` *шестнадцатиричный-литерал*

литерал-с-плавающей-точкой →
десятичный-литерал `.` *десятичный-литерал* [*экспонента*]
| *десятичный-литерал экспонента*

экспонента →
(`e` | `E`) [`+` | `-`] *десятичный-литерал*

символьный-литерал →
`'` (*графический-символ*$_{\langle \text{'} \mid \backslash \rangle}$ | *пробел* | *эскейп-символ*$_{\langle \backslash \& \rangle}$) `'`

строковый-литерал →
`"` {*графический-символ*$_{\langle \text{"} \mid \backslash \rangle}$ | *пробел* | *эскейп-символ* | *разрыв*} `"`

эскейп-символ →
`\` (*символ-эскейп* | *символ-ascii* | *десятичный-литерал* | `o` *восьмиричный-литерал* |
`x` *шестнадцатиричный-литерал*)

символ-эскейп →
`a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | `'` | `&`

символ-ascii →
`^`*управляющий-символ* | `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`

управляющий-символ →
большая-буква-ASCII | `@` | `[` | `\` | `]` | `^` | `_`

разрыв →
`\` *пробельный-символ* {*пробельный-символ*} `\`

9.3 Размещение

В разделе 2.7 дается неформальное определение правила размещения. В этом разделе это правило определено более точно.

Смысл программы на Haskell может зависеть от *размещения* ее текста. Влияние размещения на ее смысл может быть полностью описано путем добавления фигурных скобок и точек с запятой в местах, определяемых размещением. Смысл такого дополнения программы состоит в том, чтобы сделать ее не зависимой от размещения текста.

Влияние размещения задано в этом разделе посредством описания того, как добавить фигурные скобки и точки с запятой в текст программы. Спецификация принимает вид функции L, которая выполняет трансляцию. Входом для L являются:

- Поток лексем, заданных лексическим синтаксисом описания Haskell, со следующими дополнительными токенами:
  - Если за ключевым словом `let`, `where`, `do` или `of` не следует лексема `{`, то токен $\{n\}$ вставляется после ключевого слова, где n — отступ следующей лексемы, если она есть, или 0, если был достигнут конец файла.
  - Если первая лексема модуля не является `{` или `module`, то она предваряется $\{n\}$, где n — отступ лексемы.
  - Там, где начало лексемы предваряется только пробельными символами на той же строке, эта лексема предваряется $< n >$, где n — отступ лексемы, при условии что она, как следствие первых двух правил, не предваряется $\{n\}$. (NB: строковый литерал может простираться на несколько строк (см. раздел 2.6). Поэтому во фрагменте

    ```
    f = ("Здравствуйте \
            \Билл", "Джейк")
    ```

    $< n >$ не вставляются ни перед *Билл*, потому что она не является началом законченной лексемы, ни перед `,`, потому что она не предваряется только пробельными символами.)

- Стек "контекстов размещения", в котором каждый элемент является:
  - Или нулем, который означает, что внешний контекст является явным (т.е. программист поставил открывающую фигурную скобку). Если самый внутренний контекст равен 0, то никаких размещающих токенов не будет добавлено, пока не завершится внешний контекст или новый контекст не будет помещен в стек.
  - Или целым положительным числом, которое является отступом для внешнего контекста размещения.

"Отступом" лексемы является номер колонки для первого символа этой лексемы; отступом строки является отступ его крайней слева лексемы. Для того чтобы определить номер колонки, предположим, что используется шрифт фиксированной ширины со следующими условностями:

- Символы *новая строка*, *возврат каретки*, *перевод строки* и *перевод страницы* начинают новую строку.
- Первая колонка обозначается колонка 1, а не 0.
- При табуляции пропускается 8 символов.
- Символ табуляции вызывает вставку достаточного количества пробелов для выравнивания текущей позиции до следующей позиции табуляции.

С целью соблюдения правил размещения, символы Unicode в исходной программе рассматриваются как те же символы фиксированной ширины, что и символы ASCII. Тем не менее, чтобы избежать визуальной путаницы, программистам следует избегать написания программ, в которых смысл неявного размещения зависит от ширины непробельных символов.

Применение

$$L \text{ токены } []$$

передает не зависящую от размещения трансляцию *токенов*, где *токены* являются результатом лексического анализа модуля и добавления к нему указателей номеров колонок, как описано выше. Определение L заключается в следующем: где мы используем ":" в качестве оператора конструирования потока и "[]" для пустого потока.

$L\ (< n >: ts)\ (m : ms)$	$=$	`;` $:$ $(L\ ts\ (m : ms))$	если $m = n$
	$=$	`}` $:$ $(L\ (< n >: ts)\ ms)$	если $n < m$
$L\ (< n >: ts)\ ms$	$=$	$L\ ts\ ms$	
$L\ (\{n\} : ts)\ (m : ms)$	$=$	`{` $:$ $(L\ ts\ (n : m : ms))$	если $n > m$ (*Замечание* 1)
$L\ (\{n\} : ts)\ []$	$=$	`{` $:$ $(L\ ts\ [n])$	если $n > 0$ (*Замечание* 1)
$L\ (\{n\} : ts)\ ms$	$=$	`{` $:$ `}` $:$ $(L\ (< n >: ts)\ ms)$	(*Замечание* 2)
$L\ (\texttt{}\} : ts)\ (0 : ms)$	$=$	`}` $:$ $(L\ ts\ ms)$	(*Замечание* 3)
$L\ (\texttt{}\} : ts)\ ms$	$=$	ошибка-разбора	(*Замечание* 3)
$L\ (\texttt{\{} : ts)\ ms$	$=$	`{` $:$ $(L\ ts\ (0 : ms))$	(*Замечание* 4)
$L\ (t : ts)\ (m : ms)$	$=$	`}` $:$ $(L\ (t : ts)\ ms)$	если $m/ = 0$ и ошибка-разбора(t) (*Замечание* 5)
$L\ (t : ts)\ ms$	$=$	t $:$ $(L\ ts\ ms)$	
$L\ []\ []$	$=$	$[]$	
$L\ []\ (m : ms)$	$=$	`}` $:$ $L\ []\ ms$	если $m \neq 0$ (*Замечание* 6)

Замечание 1. Вложенный контекст должен иметь больший отступ, чем внешний контекст ($n > m$). Если нет — L завершается с ошибкой, и компилятору следует указать на ошибку размещения. Пример:

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

Здесь определение `p` имеет меньший отступ, чем отступ внешнего контекста, который устанавливается в этом случае путем определения `h`.

Замечание 2. Если первый токен после `where` (скажем) не имеет отступа, большего чем внешний контекст размещения, то блок должен быть пуст, поэтому вставляются пустые фигурные скобки. Токен $\{n\}$ заменяется на $< n >$, чтобы сымитировать ситуацию как если бы пустые фигурные скобки были явными.

Замечание 3. Посредством сопоставления с 0 текущего контекста размещения, мы гарантируем, что явная закрывающая фигурная скобка может быть сопоставлена только явной открывающей фигурной скобке. Если явная закрывающая фигурная скобка будет сопоставлена неявной открывающей фигурной скобке — возникнет ошибка разбора.

Замечание 4. Это утверждение означает, что все пары фигурных скобок трактуются как явные контексты размещения, включая именованные создание типов данных и их обновление (раздел 3.15). В этом заключается разница между этой формулировкой и Haskell 1.4.

Замечание 5. Дополнительное условие ошибка-разбора(t) интерпретируется следующим образом: если токены, порожденные до сих пор L вместе со следующим токеном t представляет недопустимый префикс в грамматике Haskell, а токены, порожденные к этому времени L, за которым следует токен "`}`", представляют правильный префикс в грамматике Haskell, то ошибка-разбора(t) равна истине.

Проверка $m/ = 0$ контролирует, что неявно добавленная закрывающая фигурная скобка будет сопоставлена неявной открывающей фигурной скобке.

Замечание 6. В конце ввода добавляются все незаконченные закрывающие фигурные скобки. Будет ошибкой оказаться здесь в пределах контекста без размещения (т.е. $m = 0$).

Если ни одно из данных выше правил не подойдет, то алгоритм завершится неудачей. Он может завершиться неудачей, например, когда будет достигнут конец ввода, и контекст без размещения будет активен, так как закрывающая фигурная скобка пропущена. Некоторые сбойные ситуации не обнаруживаются алгоритмом, хотя они могут быть: например, `let }`.

Замечание 1 реализует свойство, при котором обработка размещения может быть остановлена преждевременно из-за ошибки разбора. Например,

```
let x = e; y = x in e'
```

правильно, потому что оно транслируется в

```
let { x = e; y = x } in e'
```

Закрывающая фигурная скобка вставляется вследствие описанного выше правила ошибки разбора. Правило ошибки разбора трудно реализовать в его полной

применимости ко всему, потому что выполнение этого влечет применение ассоциативностей. Например, выражение

```
do a == b == c
```

имеет единственный однозначный (хотя, возможно, неправильный с точки зрения типов) разбор, а именно:

```
(do { a == b }) == c
```

потому что `(==)` является неассоциативным. Поэтому программистам советуют избегать написания кода, который требует, чтобы синтаксический анализатор вставлял закрывающую фигурную скобку в таких ситуациях.

9.4 Грамотные комментарии

Соглашение о "грамотных комментариях", впервые разработанное Ричардом Бердом (Richard Bird) и Филиппом Уодлером (Philip Wadler) для Orwell, и позаимствованное в свою очередь Дональдом Кнутом (Donald Knuth) для "грамотного программирования", является альтернативным стилем программирования исходного кода на Haskell. Грамотный стиль поощряет комментарии, создавая их по умолчанию. Строка, в которой "`>`" является первым символом, интерпретируется как часть программы; все остальные строки являются комментарием.

Текст программы восстанавливается путем выбора только тех строк, которые начинаются с "`>`", и замены первого "`>`" на пробел. В полученном тексте размещение и комментарии применяются в точности как описано в главе 9.

Чтобы охватить некоторые случаи, где можно по ошибке пропустить "`>`", возникнет ошибка, если строка программы появится рядом с непробельной строкой комментария; строка рассматривается как пробельная, если она состоит только из пробельных символов.

Условно на стиль комментария указывает расширение файла: "`.hs`" указывает на обычный файл на Haskell, а "`.lhs`" указывает на файл с грамотным Haskell. С использованием этого стиля простая программа вычисления факториала могла бы выглядеть так:

```
    Эта грамотная программа запрашивает у пользователя число
    и выводит на экран факториал этого числа:

> main :: IO ()

> main = do putStr "Введите число: "
>           l <- readLine
>           putStr "n!= "
>           print (fact (read l))

  Это программа вычисления факториала.

> fact :: Integer -> Integer
> fact 0 = 1
> fact n = n * fact (n-1)
```

Альтернативный стиль грамотного программирования особенно подходит для использования вместе с системой обработки текста LaTeX. По этому соглашению только те части грамотной программы, которые полностью заключены между разделителями `\begin{code}`...`\end{code}`, рассматриваются как текст программы; все остальные строки — комментарии. Более точно:

- Код программы начинается на первой строке, следующей за строкой, которая начинает `\begin{code}`.

- Код программы заканчивается сразу перед последующей строкой, которая начинает `\end{code}` (конечно, игнорируя строковые литералы).

Нет необходимости вставлять дополнительные пустые строки до или после этих разделителей, хотя со стилистической точки зрения это может быть желательно. Например,

```
\documentstyle{article}

\begin{document}

\section{Introduction}

Это тривиальная программа, которая выводит первые 20 факториалов.

\begin{code}
main :: IO ()
main =  print [ (n, product [1..n]) | n <- [1..20]]
\end{code}

\end{document}
```

Этот стиль использует то же расширение файла. Нежелательно смешивать эти два стиля в одном файле.

9.5 Контекстно-свободный синтаксис

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

impdecls → $impdecl_1$ `;` ... `;` $impdecl_n$ $(n \geq 1)$

Перевод:
модуль →
`module` *идентификатор-модуля* [*список-экспорта*] `where` *тело*
| *тело*
тело →
`{` *список-объявлений-импорта* `;` *список-объявлений-верхнего-уровня* `}`
| `{` *список-объявлений-импорта* `}`
| `{` *список-объявлений-верхнего-уровня* `}`

список-объявлений-импорта →
$объявление\text{-}импорта_1$ `;` ... `;` $объявление\text{-}импорта_n$
$(n \geq 1)$

exports → `(` $export_1$ `,` ... `,` $export_n$ [`,`] `)` $(n \geq 0)$

export → *qvar*
| *qtycon* [`(..)` | `(` $cname_1$ `,` ... `,` $cname_n$ `)`] $(n \geq 0)$
| *qtycls* [`(..)` | `(` $qvar_1$ `,` ... `,` $qvar_n$ `)`] $(n \geq 0)$
| `module` *modid*

Перевод:
список-экспорта →
`(` $экспорт_1$ `,` ... `,` $экспорт_n$ [`,`] `)`
$(n \geq 0)$

экспорт →
квалифицированная-переменная
| *квалифицированный-конструктор-типа* [`(..)` | `(` $с\text{-}имя_1$ `,` ... `,` $с\text{-}имя_n$ `)`]
$(n \geq 0)$
| *квалифицированный-класс-типа* [`(..)` |

```
    ( квалифицированная-переменная_1 , ... , квалифицированная-переменная_n )]
    (n ≥ 0)
  | module идентификатор-модуля
```

```
impdecl   →  import [qualified] modid [as modid] [impspec]
          |                                          (пустое объявление)

impspec   →  ( import_1 , ... , import_n [ , ] )     (n ≥ 0)
          |  hiding ( import_1 , ... , import_n [ , ] )  (n ≥ 0)

import    →  var
          |  tycon [ (..) | ( cname_1 , ... , cname_n )]  (n ≥ 0)
          |  tycls [(..) | ( var_1 , ... , var_n )]       (n ≥ 0)
cname     →  var | con
```

Перевод:

объявление-импорта →
  `import` [`qualified`] *идентификатор-модуля* [`as` *идентификатор-модуля*]
    [*спецификатор-импорта*]
  |
    (пустое объявление)

спецификатор-импорта →
  `(` *импорт*$_1$ `,` ... `,` *импорт*$_n$ [`,`] `)`
    ($n \geq 0$)
  | `hiding (` *импорт*$_1$ `,` ... `,` *импорт*$_n$ [`,`] `)`
    ($n \geq 0$)

импорт →
  переменная
  | *конструктор-типа* [`(..)` | `(` *с-имя*$_1$ `,` ... `,` *с-имя*$_n$ `)`]
    ($n \geq 0$)
  | *класс-типа* [`(..)` | `(` *переменная*$_1$ `,` ... `,` *переменная*$_n$ `)`]
    ($n \geq 0$)

с-имя →
  переменная
  | *конструктор*

```
topdecls  →  topdecl_1 ; ... ; topdecl_n              (n ≥ 0)
topdecl   →  type simpletype = type
          |  data [context =>] simpletype = constrs [deriving]
          |  newtype [context =>] simpletype = newconstr [deriving]
          |  class [scontext =>] tycls tyvar [where cdecls]
```

	`instance` [*scontext* `=>`] *qtycls inst* [`where` *idecls*]	
	`default` (*type*$_1$, ... , *type*$_n$)	$(n \geq 0)$
	decl	

Перевод:
список-объявлений-верхнего-уровня →
  объявление-верхнего-уровня$_1$; ... ; *объявление-верхнего-уровня*$_n$
    $(n \geq 1)$
объявление-верхнего-уровня →
  `type` *простой-тип* `=` *тип*
  | `data` [*контекст* `=>`] *простой-тип* `=` *список-конструкций* [*deriving-инструкция*]
  | `newtype` [*контекст* `=>`] *простой-тип* `=` *новая-конструкция*
    [*deriving-инструкция*]
  | `class` [*простой-контекст* `=>`] *класс-типа переменная-типа*
    [`where` *список-объявлений-классов*]
  | `instance` [*простой-контекст* `=>`] *квалифицированный-класс-типа экземпляр*
    [`where` *список-объявлений-экземпляров*]
  | `default` (*тип*$_1$, ... , *тип*$_n$)
    $(n \geq 0)$
  | *объявление*

decls	→	`{` *decl*$_1$; ... ; *decl*$_n$ `}`	$(n \geq 0)$
decl	→	*gendecl*	
	|	(*funlhs* | *pat*^{0}) *rhs*	
cdecls	→	`{` *cdecl*$_1$; ... ; *cdecl*$_n$ `}`	$(n \geq 0)$
cdecl	→	*gendecl*	
	|	(*funlhs* | *var*) *rhs*	
idecls	→	`{` *idecl*$_1$; ... ; *idecl*$_n$ `}`	$(n \geq 0)$
idecl	→	(*funlhs* | *var*) *rhs*	
	|		(empty)
gendecl	→	*vars* `::` [*context* `=>`] *type*	(сигнатура типа)
	|	*fixity* [*integer*] *ops*	(infix-объявление)
	|		(пустое объявление)
ops	→	*op*$_1$, ... , *op*$_n$	$(n \geq 1)$
vars	→	*var*$_1$, ..., *var*$_n$	$(n \geq 1)$
fixity	→	`infixl` | `infixr` | `infix`	

Перевод:
список-объявлений →
  { *объявление*$_1$; ... ; *объявление*$_n$ }
    ($n \geq 0$)
объявление →
  общее-объявление
  | (*левая-часть-функции* | *образец*^{0}) *правая-часть*

список-объявлений-классов →
  { *объявление-класса*$_1$; ... ; *объявление-класса*$_n$ }
    ($n \geq 0$)
объявление-класса →
  общее-объявление
  | (*левая-часть-функции* | *переменная*) *правая-часть*

список-объявлений-экземпляров →
  { *объявление-экземпляра*$_1$; ... ; *объявление-экземпляра*$_n$ }
    ($n \geq 0$)
объявление-экземпляра →
  (*левая-часть-функции* | *переменная*) *правая-часть*
  |
    (пусто)

общее-объявление →
  список-переменных :: [*контекст* =>] *тип*
    (сигнатура типа)
  | *ассоциативность* [*целый-литерал*] *список-операторов*
    (infix-объявление)
  |
    (пустое объявление)

список-операторов →
  оператор$_1$, ... , *оператор*$_n$
    ($n \geq 1$)
список-переменных →
  переменная$_1$, ... , *переменная*$_n$
    ($n \geq 1$)
ассоциативность →
  infixl | infixr | infix

type	→	*btype* [-> *type*]	(тип функции)
btype	→	[*btype*] *atype*	(наложение типов)
atype	→	*gtycon*	

	| *tyvar*	
	| (*type*$_1$, ... , *type*$_k$)	(тип кортежа, $k \geq 2$)
	| [*type*]	(тип списка)
	| (*type*)	(конструктор в скобках)
gtycon	→ *qtycon*	
	| ()	(тип объединения)
	| []	(конструктор списка)
	| (->)	(конструктор функции)
	| (,{,})	(конструкторы кортежей)
context	→ *class*	
	| (*class*$_1$, ... , *class*$_n$)	($n \geq 0$)
class	→ *qtycls tyvar*	
	| *qtycls* (*tyvar atype*$_1$ ... *atype*$_n$)	($n \geq 1$)
scontext	→ *simpleclass*	
	| (*simpleclass*$_1$, ... , *simpleclass*$_n$)	($n \geq 0$)
simpleclass	→ *qtycls tyvar*	

Перевод:

тип →
  b-тип [-> *тип*]
    (тип функции)

b-тип →
  [*b-тип*] *a-тип*
    (наложение типов)

a-тип →
  общий-конструктор-типа
  | *переменная-типа*
  | (*тип*$_1$, ... , *тип*$_k$)
    (тип кортежа, $k \geq 2$)
  | [*тип*]
    (тип списка)
  | (*тип*)
    (конструктор в скобках)

общий-конструктор-типа →
  квалифицированный-конструктор-типа
  | ()
    (тип объединения)
  | []
    (конструктор списка)

| `(->)`
  (конструктор функции)
| `(,{,})`
  (конструкторы кортежей)

контекст →
  класс
  | `(` *класс*$_1$ `,` ... `,` *класс*$_n$ `)`
    ($n \geq 0$)
класс →
  квалифицированный-класс-типа переменная-типа
  | *квалифицированный-класс-типа* `(` *переменная-типа а-тип*$_1$ ... *а-тип*$_n$ `)`
    ($n \geq 1$)
простой-контекст →
  простой-класс
  | `(` *простой-класс*$_1$ `,` ... `,` *простой-класс*$_n$ `)`
    ($n \geq 0$)
простой-класс →
  квалифицированный-класс-типа переменная-типа

simpletype	→	*tycon tyvar*$_1$ ... *tyvar*$_k$	($k \geq 0$)
constrs	→	*constr*$_1$ `|` ... `|` *constr*$_n$	($n \geq 1$)
constr	→	*con* [`!`] *atype*$_1$ ... [`!`] *atype*$_k$	(число аргументов конструктора *con* $= k$, $k \geq 0$)
	|	(*btype* | `!` *atype*) *conop* (*btype* | `!` *atype*)	(инфиксный оператор *conop*)
	|	*con* `{` *fielddecl*$_1$ `,` ... `,` *fielddecl*$_n$ `}`	($n \geq 0$)
newconstr	→	*con atype*	
	|	*con* `{` *var* `::` *type* `}`	
fielddecl	→	*vars* `::` (*type* | `!` *atype*)	
deriving	→	`deriving` (*dclass* | `(`*dclass*$_1$ `,` ... `,` *dclass*$_n$`)`)	($n \geq 0$)
dclass	→	*qtycls*	

Перевод:
простой-тип →
  конструктор-типа переменная-типа$_1$ ... *переменная-типа*$_k$
    ($k \geq 0$)
список-конструкций →
  конструкция$_1$ `|` ... `|` *конструкция*$_n$
    ($n \geq 1$)
конструкция →

конструктор [`!`] *a-тип*$_1$ ... [`!`] *a-тип*$_k$
(число аргументов конструктора *con* $= k$, $k \geq 0$)
| (*b-тип* | `!` *a-тип*) *оператор-конструктора* (*b-тип* | `!` *a-тип*)
(инфиксный оператор *сопор*)
|*конструктор* `{` *объявление-поля*$_1$ `,` ... `,` *объявление-поля*$_n$ `}`
($n \geq 0$)

новая-конструкция →
конструктор a-тип
| *конструктор* `{` *переменная* `::` *тип* `}`

объявление-поля →
список-переменных `::` (*тип* | `!` *a-тип*)

deriving-инструкция →
`deriving` (*производный-класс* |
(*производный-класс*$_1$`,` ... `,` *производный-класс*$_n$))
($n \geq 0$)

производный-класс →
квалифицированный-класс-типа

inst	→	*gtycon*	
	|	`(` *gtycon tyvar*$_1$ ... *tyvar*$_k$ `)`	($k \geq 0$, все *tyvar* различны)
	|	`(` *tyvar*$_1$ `,` ... `,` *tyvar*$_k$ `)`	($k \geq 2$, все *tyvar* различны)
	|	`[` *tyvar* `]`	
	|	`(` *tyvar*$_1$ `->` *tyvar*$_2$ `)`	(*tyvar*$_1$ и *tyvar*$_2$ различны)

Перевод:

экземпляр →
общий-конструктор-типа
| `(` *общий-конструктор-типа переменная-типа*$_1$ ... *переменная-типа*$_k$ `)`
($k \geq 0$, все *переменные-типа* различны)
| `(` *переменная-типа*$_1$ `,` ... `,` *переменная-типа*$_k$ `)`
($k \geq 2$, все *переменные-типа* различны)
| `[` *переменная-типа* `]`
| `(` *переменная-типа*$_1$ `->` *переменная-типа*$_2$ `)`
(*переменная-типа*$_1$ и *переменная-типа*$_2$ различны)

funlhs	→	*var apat* { *apat* }
	|	pat^{i+1} $varop^{(a,i)}$ pat^{i+1}
	|	$lpat^{i}$ $varop^{(l,i)}$ pat^{i+1}
	|	pat^{i+1} $varop^{(r,i)}$ $rpat^{i}$
	|	`(` *funlhs* `)` *apat* { *apat* }

rhs → `=` *exp* [`where` *decls*]
| *gdrhs* [`where` *decls*]

gdrhs → *gd* `=` *exp* [*gdrhs*]

gd → `|` exp^0

Перевод:
левая-часть-функции →
переменная такой-как-образец { *такой-как-образец* }
| $образец^{i+1}$ $оператор\text{-}переменной^{(a,i)}$ $образец^{i+1}$
| $левый\text{-}образец^{i}$ $оператор\text{-}переменной^{(\mathrm{l},i)}$ $образец^{i+1}$
| $образец^{i+1}$ $оператор\text{-}переменной^{(\mathrm{r},i)}$ $правый\text{-}образец^{i}$
| `(` *левая-часть-функции* `)` *такой-как-образец* { *такой-как-образец* }

правая-часть →
`=` *выражение* [`where` *список-объявлений*]
| *правая-часть-со-стражами* [`where` *список-объявлений*]

правая-часть-со-стражами →
страж `=` *выражение* [*правая-часть-со-стражами*]

страж →
`|` $выражение^0$

exp	→	exp^0 `::` [*context* `=>`] *type*	(сигнатура типа выражения)
	|	exp^0	
exp^i	→	exp^{i+1} $[qop^{(\mathrm{n},i)}\ exp^{i+1}]$	
	|	$lexp^i$	
	|	$rexp^i$	
$lexp^i$	→	$(lexp^i \mid exp^{i+1})\ qop^{(\mathrm{l},i)}\ exp^{i+1}$	
$lexp^6$	→	`-` exp^7	
$rexp^i$	→	$exp^{i+1}\ qop^{(\mathrm{r},i)}\ (rexp^i \mid exp^{i+1})$	
exp^{10}	→	`\` $apat_1$ ... $apat_n$ `->` *exp*	(лямбда-абстракция, $n \geq 1$)
	|	`let` *decls* `in` *exp*	(let-выражение)
	|	`if` *exp* `then` *exp* `else` *exp*	(условное выражение)
	|	`case` *exp* `of` `{` *alts* `}`	(case-выражение)
	|	`do` `{` *stmts* `}`	(do-выражение)
	|	*fexp*	
fexp	→	[*fexp*] *aexp*	(применение функции)

Перевод:

выражение →
  выражение^{0} `::` [*контекст* `=>`] *тип*
    (сигнатура типа выражения)
  | *выражение*^{0}
выражение^{i} →
  выражение$^{i+1}$ [*квалифицированный-оператор*$^{(\mathrm{n},i)}$ *выражение*$^{i+1}$]
  | *левое-сечение-выражения*^{i}
  | *правое-сечение-выражения*^{i}
левое-сечение-выражения^{i} →
  (*левое-сечение-выражения*^{i} | *выражение*$^{i+1}$) *квалифицированный-оператор*$^{(\mathrm{l},i)}$
    выражение$^{i+1}$
левое-сечение-выражения^{6} →
  `-` *выражение*^{7}
правое-сечение-выражения^{i} →
  выражение$^{i+1}$ *квалифицированный-оператор*$^{(\mathrm{r},i)}$
    (*правое-сечение-выражения*^{i} | *выражение*$^{i+1}$)
выражение^{10} →
  `\` *такой-как-образец*$_1$ ... *такой-как-образец*$_n$ `->` *выражение*
    (лямбда-абстракция, $n \geq 1$)
  | `let` *списки-объявлений* `in` *выражение*
    (let-выражение)
  | `if` *выражение* `then` *выражение* `else` *выражение*
    (условное выражение)
  | `case` *выражение* `of` `{` *список-альтернатив* `}`
    (case-выражение)
  | `do` `{` *список-инструкций* `}`
    (do-выражение)
  | *функциональное-выражение*
функциональное-выражение →
  [*функциональное-выражение*] *выражение-аргумента*
    (применение функции)

aexp	→	*qvar*	(переменная)
	|	*gcon*	(общий конструктор)
	|	*literal*	
	|	`(` *exp* `)`	(выражение в скобках)
	|	`(` *exp*$_1$ `,` ... `,` *exp*$_k$ `)`	(кортеж, $k \geq 2$)
	|	`[` *exp*$_1$ `,` ... `,` *exp*$_k$ `]`	(список, $k \geq 1$)
	|	`[` *exp*$_1$ [`,` *exp*$_2$] `..` [*exp*$_3$] `]`	(арифметическая последовательность)
	|	`[` *exp* `|` *qual*$_1$ `,` ... `,` *qual*$_n$ `]`	(описание списка, $n \geq 1$)
	|	`(` *exp*$^{i+1}$ *qop*$^{(a,i)}$ `)`	(левое сечение)
	|	`(` *lexp*^{i} *qop*$^{(l,i)}$ `)`	(левое сечение)

	`(` $qop^{(a,i)}_{\langle \_ \rangle}$ exp^{i+1} `)`	(правое сечение)
|	`(` $qop^{(r,i)}_{\langle \_ \rangle}$ $rexp^{i}$ `)`	(правое сечение)
|	*qcon* `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(именованная конструкция, $n \geq 0$)
|	$aexp_{\langle qcon \rangle}$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(именованное обновление, $n \geq 1$)

Перевод:
выражение-аргумента →
  квалифицированная-переменная
    (переменная)
  | *общий-конструктор*
    (общий конструктор)
  | *литерал*
  | `(` *выражение* `)`
    (выражение в скобках)
  | `(` $выражение_1$ `,` ... `,` $выражение_k$ `)`
    (кортеж, $k \geq 2$)
  | `[` $выражение_1$ `,` ... `,` $выражение_k$ `]`
    (список, $k \geq 1$)
  | `[` $выражение_1$ `[,` $выражение_2$`]` `..` `[`$выражение_3$`]` `]`
    (арифметическая последовательность)
  | `[` *выражение* `|` $квалификатор_1$ `,` ... `,` $квалификатор_n$ `]`
    (описание списка, $n \geq 1$)
  | `(` $выражение^{i+1}$ $квалифицированный\text{-}оператор^{(a,i)}$ `)`
    (левое сечение)
  | `(` $левое\text{-}сечение\text{-}выражения^{i}$ $квалифицированный\text{-}оператор^{(l,i)}$ `)`
    (левое сечение)
  | `(` $квалифицированный\text{-}оператор^{(a,i)}_{\langle \_ \rangle}$ $выражение^{i+1}$ `)`
    (правое сечение)
  | `(` $квалифицированный\text{-}оператор^{(r,i)}_{\langle \_ \rangle}$ $правое\text{-}сечение\text{-}выражения^{i}$ `)`
    (правое сечение)
  | *квалифицированный-конструктор*
    `{` $связывание\text{-}имени\text{-}поля_1$ `,` ... `,` $связывание\text{-}имени\text{-}поля_n$ `}`
    (именованная конструкция, $n \geq 0$)
  | $выражение\text{-}аргумента_{\langle квалифицированный\text{-}конструктор \rangle}$
    `{` $связывание\text{-}имени\text{-}поля_1$ `,` ... `,` $связывание\text{-}имени\text{-}поля_n$ `}`
    (именованное обновление, $n \geq 1$)

qual	→	*pat* `<-` *exp*	(генератор)
	|	`let` *decls*	(локальное объявление)
	|	*exp*	(страж)

alts	→	alt_1 ; ... ; alt_n	$(n \geq 1)$
alt	→	*pat* `->` *exp* [`where` *decls*]	
	|	*pat gdpat* [`where` *decls*]	
	|		(*пустая альтернатива*)
gdpat	→	*gd* `->` *exp* [*gdpat*]	
stmts	→	$stmt_1$ ... $stmt_n$ *exp* [;]	$(n \geq 0)$
stmt	→	*exp* ;	
	|	*pat* `<-` *exp* ;	
	|	`let` *decls* ;	
	|	;	(*пустая инструкция*)
fbind	→	*qvar* `=` *exp*	

Перевод:

квалификатор →
  образец `<-` *выражение*
    (генератор)
  | `let` *списки-объявлений*
    (локальное объявление)
  | *выражение*
    (страж)

список-альтернатив →
  $альтернатива_1$; ... ; $альтернатива_n$
    $(n \geq 1)$
альтернатива →
  образец `->` *выражение* [`where` *список-объявлений*]
  | *образец образец-со-стражами* [`where` *список-объявлений*]
  |
    (*пустая альтернатива*)

образец-со-стражами →
  страж `->` *выражение* [*образец-со-стражами*]

список-инструкций →
  $инструкция_1$ ... $инструкция_n$ *выражение* [;]
    $(n \geq 0)$
инструкция →
  выражение ;
  | *образец* `<-` *выражение* ;

| `let` *список-объявлений* `;`
| `;`
(*пустая инструкция*)

связывание-имени-поля →
квалифицированная-переменная `=` *выражение*

pat	→	*var* `+` *integer*	(образец упорядочивания)
	|	pat^0	
pat^i	→	pat^{i+1} $[qconop^{(\mathrm{n},i)}\ pat^{i+1}]$	
	|	$lpat^i$	
	|	$rpat^i$	
$lpat^i$	→	$(lpat^i \mid pat^{i+1})\ qconop^{(\mathrm{l},i)}\ pat^{i+1}$	
$lpat^6$	→	`-` (*integer* | *float*)	(отрицательный литерал)
$rpat^i$	→	$pat^{i+1}\ qconop^{(\mathrm{r},i)}\ (rpat^i \mid pat^{i+1})$	
pat^{10}	→	*apat*	
	|	$gcon\ apat_1\ \ldots\ apat_k$	(число аргументов конструктора $gcon\ =\ k$, $k \geq 1$)

Перевод:
образец →
переменная `+` *целый-литерал*
(образец упорядочивания)
| $образец^0$
$образец^i$ →
$образец^{i+1}$ $[квалифицированный\text{-}оператор\text{-}конструктора^{(\mathrm{n},i)}\ образец^{i+1}]$
| $левый\text{-}образец^i$
| $правый\text{-}образец^i$
$левый\text{-}образец^i$ →
$(левый\text{-}образец^i \mid образец^{i+1})$ $квалифицированный\text{-}оператор\text{-}конструктора^{(\mathrm{l},i)}$
$образец^{i+1}$
$левый\text{-}образец^6$ →
`-` (*целый-литерал* | *литерал-с-плавающей-точкой*)
(отрицательный литерал)
$правый\text{-}образец^i$ →
$образец^{i+1}$ $квалифицированный\text{-}оператор\text{-}конструктора^{(\mathrm{r},i)}$
$(правый\text{-}образец^i \mid образец^{i+1})$
$образец^{10}$ →
такой-как-образец
| *общий-конструктор такой-как-образец*$_1$ … *такой-как-образец*$_k$
(число аргументов конструктора $gcon\ =\ k,\ k \geq 1$)

apat	→	*var* [@ *apat*]	(“такой как”-образец)
	|	*gcon*	(число аргументов конструктора $gcon = 0$)
	|	*qcon* { $fpat_1$, … , $fpat_k$ }	(именованный образец, $k \geq 0$)
	|	*literal*	
	|	_	(любые символы)
	|	(*pat*)	(образец в скобках)
	|	(pat_1 , … , pat_k)	(образец кортежа, $k \geq 2$)
	|	[pat_1 , … , pat_k]	(образец списка, $k \geq 1$)
	|	~ *apat*	(неопровержимый образец)
fpat	→	*qvar* = *pat*	

Перевод:
такой-как-образец →
  переменная [@ *такой-как-образец*]
    (“такой как”-образец)
  | *общий-конструктор*
    (число аргументов конструктора $gcon = 0$)
  | *квалифицированный-конструктор*
    { $образец\text{-}с\text{-}именем_1$, … , $образец\text{-}с\text{-}именем_k$ }
    (именованный образец, $k \geq 0$)
  | *литерал*
  | _
    (любые символы)
  | (*образец*)
    (образец в скобках)
  | ($образец_1$, … , $образец_k$)
    (образец кортежа, $k \geq 2$)
  | [$образец_1$, … , $образец_k$]
    (образец списка, $k \geq 1$)
  | ~ *такой-как-образец*
    (неопровержимый образец)

образец-с-именем →
  квалифицированная-переменная = *образец*

gcon	→	()	
	|	[]	
	|	(,{,})	
	|	*qcon*	

var	→	*varid* | (*varsym*)	(переменная)
qvar	→	*qvarid* | (*qvarsym*)	(квалифицированная переменная)
con	→	*conid* | (*consym*)	(конструктор)
qcon	→	*qconid* | (*gconsym*)	(квалифицированный конструктор)
varop	→	*varsym* | ` *varid* `	(оператор переменной)
qvarop	→	*qvarsym* | ` *qvarid* `	(квалифицированный оператор переменной)
conop	→	*consym* | ` *conid* `	(оператор конструктора)
qconop	→	*gconsym* | ` *qconid* `	(квалифицированный оператор конструктора)
op	→	*varop* | *conop*	(оператор)
qop	→	*qvarop* | *qconop*	(квалифицированный оператор)
gconsym	→	: | *qconsym*	

Перевод:

общий-конструктор →
  ()
  | []
  | (,{,})
  | *квалифицированный-конструктор*

переменная →
  идентификатор-переменной
  | (*символ-переменной*)
    (переменная)

квалифицированная-переменная →
  квалифицированный-идентификатор-переменной
  | (*квалифицированный-символ-переменной*)
    (квалифицированная переменная)

конструктор →
  идентификатор-конструктора
  | (*символ-конструктора*)
    (конструктор)

квалифицированный-конструктор →
  квалифицированный-идентификатор-конструктора
  | (*символ-общего-конструктора*)
    (квалифицированный конструктор)

оператор-переменной →
  символ-переменной
  | ` *идентификатор-переменной* `

(оператор переменной)
квалифицированный-оператор-переменной →
квалифицированный-символ-переменной
| ‘ *квалифицированный-идентификатор-переменной* ‘
(квалифицированный оператор переменной)
оператор-конструктора →
символ-конструктора
| ‘ *идентификатор-конструктора* ‘
(оператор конструктора)
квалифицированный-оператор-конструктора →
символ-общего-конструктора
| ‘ *квалифицированный-идентификатор-конструктора* ‘
(квалифицированный оператор конструктора)
оператор →
оператор-переменной
| *оператор-конструктора*
(оператор)
квалифицированный-оператор →
квалифицированный-оператор-переменной
| *квалифицированный-оператор-конструктора*
(квалифицированный оператор)
символ-общего-конструктора →
: | *квалифицированный-символ-конструктора*

Глава 10

Спецификация производных экземпляров

Производный экземпляр представляет собой объявление экземпляра, которое генерируется автоматически в связи с объявлением `data` или `newtype`. Тело объявления производного экземпляра получается синтаксически из определения связанного с ним типа. Производные экземпляры возможны только для классов, известных компилятору: тех, которые определены или в Prelude, или в стандартной библиотеке. В этой главе мы опишем выведение производных экземпляров классов, определенных в Prelude.

Если T — алгебраический тип данных, объявленный с помощью:

$$\texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \ldots\ u_k\ \ =\ \ K_1\ t_{11}\ \ldots\ t_{1k_1}\ \texttt{|}\ \cdots\ \texttt{|}\ K_n\ t_{n1}\ \ldots\ t_{nk_n}\ \ \texttt{deriving}\ (C_1, \ldots, C_m)$$

(где $m \geq 0$ и круглые скобки можно опустить, если $m = 1$), тогда объявление производного экземпляра возможно для класса C, если выполняются следующие условия:

1. C является одним из классов `Eq`, `Ord`, `Enum`, `Bounded`, `Show` или `Read`.
2. Есть контекст cx' такой, что $cx' \Rightarrow C\ t_{ij}$ выполняется для каждого из типов компонент t_{ij}.
3. Если C является классом `Bounded`, тип должен быть перечислимым (все конструкторы должны быть с нулевым числом аргументов) или иметь только один конструктор.
4. Если C является классом `Enum`, тип должен быть перечислимым.
5. Не должно быть никакого явного объявления экземпляра где-либо в другом месте в программе, которое делает $T\ u_1\ \ldots\ u_k$ экземпляром C.

С целью выведения производных экземпляров, объявление `newtype` обрабатывается как объявление `data` с одним конструктором.

Если присутствует инструкция `deriving`, объявление экземпляра автоматически генерируется для $T\ u_1\ \ldots\ u_k$ по каждому классу C_i. Если объявление производного экземпляра невозможно для какого-либо из C_i, то возникнет статическая ошибка. Если производные экземпляры не требуются, инструкцию `deriving` можно опустить или можно использовать инструкцию `deriving ()`.

Каждое объявление производного экземпляра будет иметь вид:

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

где d выводится автоматически в зависимости от C_i и объявления типа данных для T (как будет описано в оставшейся части этого раздела).

Контекст cx' является самым маленьким контекстом, удовлетворяющим приведенному выше пункту (2). Для взаимно рекурсивных типов данных компилятор может потребовать выполнения вычисления с фиксированной точкой, чтобы его вычислить.

Теперь дадим оставшиеся детали производных экземпляров для каждого из выводимых классов Prelude. Свободные переменные и конструкторы, используемые в этих трансляциях, всегда ссылаются на объекты, определенные в `Prelude`.

10.1 Производные экземпляры классов `Eq` и `Ord`

Производные экземпляры классов `Eq` и `Ord` автоматически вводят методы класса `(==)`, `(/=)`, `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max` и `min`. Последние семь операторов определены так, чтобы сравнивать свои аргументы лексикографически по отношению к заданному набору конструкторов: чем раньше стоит конструктор в объявлении типа данных, тем меньше он считается по сравнению с более поздними. Например, для типа данных `Bool` мы получим: `(True > False) == True`.

Выведенные сравнения всегда обходят конструкторы слева направо. Приведенные ниже примеры иллюстрируют это свойство:

```
(1,undefined) == (2,undefined) ⇒    False
(undefined,1) == (undefined,2) ⇒    ⊥
```

Все выведенные операции классов `Eq` и `Ord` являются строгими в отношении обоих аргументов. Например, `False <=` $\bot$ является $\bot$, даже если `False` является первым конструктором типа `Bool` .

10.2 Производные экземпляры класса Enum

Объявления производных экземпляров класса `Enum` возможны только для перечислений (типов данных с конструкторами, которые имеют только нулевое число аргументов).

Конструкторы с нулевым числом аргументов считаются пронумерованными слева направо индексами от 0 до $n-1$. Операторы `succ` и `pred` дают соответственно предыдущее и последующее значение в соответствии с этой схемой нумерации. Будет ошибкой применить `succ` к максимальному элементу или `pred` — к минимальному элементу.

Операторы `toEnum` и `fromEnum` отображают перечислимые значения в значения типа `Int` и обратно; `toEnum` вызывает ошибку времени выполнения программы, если аргумент `Int` не является индексом одного из конструкторов.

Определения оставшихся методов:

```
enumFrom x           = enumFromTo x lastCon
enumFromThen x y     = enumFromThenTo x y bound
                     where
                       bound | fromEnum y >= fromEnum x = lastCon
                             | otherwise                = firstCon
enumFromTo x y       = map toEnum [fromEnum x .. fromEnum y]
enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

где `firstCon` и `lastCon` — соответственно первый и последний конструкторы, перечисленные в объявлении `data`. Например, с учетом типа данных:

```
data  Color = Red | Orange | Yellow | Green  deriving (Enum)
```

мы имели бы:

```
[Orange ..]          ==  [Orange, Yellow, Green]
fromEnum Yellow      ==  2
```

10.3 Производные экземпляры класса Bounded

Класс `Bounded` вводит методы класса `minBound` и `maxBound`, которые определяют минимальный и максимальный элемент типа. Для перечисления границами являются первый и последний конструкторы, перечисленные в объявлении `data`. Для типа с одним конструктором конструктор применяется к границам типов компонент. Например, следующий тип данных:

```
data  Pair a b = Pair a b deriving Bounded
```

произвел бы следующий экземпляр класса `Bounded`:

```
instance (Bounded a,Bounded b) => Bounded (Pair a b) where
  minBound = Pair minBound minBound
  maxBound = Pair maxBound maxBound
```

10.4 Производные экземпляры классов `Read` и `Show`

Производные экземпляры классов `Read` и `Show` автоматически вводят методы класса `showsPrec`, `readsPrec`, `showList` и `readList`. Они используются для перевода значений в строки и перевода строк в значения.

Функция `showsPrec d x r` принимает в качестве аргументов уровень приоритета `d` (число от `0` до `11`), значение `x` и строку `r`. Она возвращает строковое представление `x`, соединенное с `r`. `showsPrec` удовлетворяет правилу:

```
showsPrec d x r ++ s  ==  showsPrec d x (r ++ s)
```

Представление будет заключено в круглые скобки, если приоритет конструктора верхнего уровня в `x` меньше чем `d`. Таким образом, если `d` равно `0`, то результат никогда не будет заключен в круглые скобки; если `d` равно `11`, то он всегда будет заключен в круглые скобки, если он не является атомарным выражением (вспомним, что применение функции имеет приоритет `10`). Дополнительный параметр `r` необходим, если древовидные структуры должны быть напечатаны за линейное, а не квадратичное время от размера дерева.

Функция `readsPrec d s` принимает в качестве аргументов уровень приоритета `d` (число от `0` до `10`) и строку `s` и пытается выполнить разбор значения в начале строки, возвращая список пар (разобранное значение, оставшаяся часть строки). Если нет успешного разбора, возвращаемый список пуст. Разбор инфиксного оператора, который не заключен в круглые скобки, завершится успешно, только если приоритет оператора больше чем или равен `d`.

Должно выполняться следующее:

`(x,"")` должен являться элементом `(readsPrec d (showsPrec d x ""))`

То есть `readsPrec` должен уметь выполнять разбор строки, полученной с помощью `showsPrec`, и должен передавать значение, с которого `showsPrec` начал работу.

`showList` и `readList` позволяют получить представление списков объектов, используя нестандартные обозначения. Это особенно полезно для строк (списков `Char`).

`readsPrec` выполняет разбор любого допустимого представления стандартных типов, кроме строк, для которых допустимы только строки, заключенные в кавычки, и других

списков, для которых допустим только вид в квадратных скобках `[...]`. См. главу 8 для получения исчерпывающих подробностей.

Результат `show` представляет собой синтаксически правильное выражение Haskell, содержащее только константы, с учетом находящихся в силе infix-объявлений в месте, где объявлен тип. Он содержит только имена конструкторов, определенных в типе данных, круглые скобки и пробелы. Когда используются именованные поля конструктора, также используются фигурные скобки, запятые, имена полей и знаки равенства. Круглые скобки добавляются только там, где это необходимо, *игнорируя ассоциативность*. Никакие разрывы строк не добавляются. Результат `show` может быть прочитан с помощью `read`, если все типы компонент могут быть прочитаны. (Это выполняется для экземпляров, определенных в Prelude, но может не выполняться для определяемых пользователем экземпляров.)

Производные экземпляры класса `Read` делают следующие предположения, которым подчиняются производные экземпляры класса `Show`:

- Если конструктор определен как инфиксный оператор, то выведенный экземпляр класса `Read` будет разбирать только инфиксные применения конструктора (не префиксную форму).
- Ассоциативность не используется для того, чтобы уменьшить появление круглых скобок, хотя приоритет может быть. Например, с учетом

  ```
  infixr 4 :$
  data T = Int :$ T  |  NT
  ```

  получим:

  - `show (1 :$ 2 :$ NT)` породит строку `"1 :$ (2 :$ NT)"`.
  - `read "1 :$ (2 :$ NT)"` завершится успешно с очевидным результатом.
  - `read "1 :$ 2 :$ NT"` завершится неуспешно.
- Если конструктор определен с использованием синтаксиса записей, выведенный экземпляр `Read` будет выполнять разбор только форм синтаксиса записей, и более того, поля должны быть заданы в том же порядке, что и в исходном объявлении.
- Производный экземпляр `Read` допускает произвольные пробельные символы Haskell между токенами входной строки. Дополнительные круглые скобки также разрешены.

Производные экземпляры `Read` и `Show` могут не подходить для некоторых использований. Среди таких проблем есть следующие:

- Круговые структуры не могут быть напечатаны или считаны с помощью этих экземпляров.

- При распечатывании структур теряется их общая основа; напечатанное представление объекта может оказаться намного больше, чем это необходимо.
- Методы разбора, используемые при считывании, очень неэффективны; считывание большой структуры может оказаться весьма медленным.
- Нет никакого пользовательского контроля над распечатыванием типов, определенных в Prelude. Например, нет никакого способа изменить форматирование чисел с плавающей точкой.

10.5 Пример

В качестве законченного примера рассмотрим тип данных дерево:

```
data Tree a = Leaf a | Tree a :^: Tree a
     deriving (Eq, Ord, Read, Show)
```

Автоматическое выведение объявлений экземпляров для `Bounded` и `Enum` невозможны, поскольку `Tree` не является перечислением и не является типом данных с одним конструктором. Полные объявления экземпляров для `Tree` приведены на рис. 10.1. Обратите внимание на неявное использование заданных по умолчанию определений методов классов — например, только `<=` определен для `Ord`, тогда как другие методы класса (`<`, `>`, `>=`, `max` и `min`), определенные по умолчанию, заданы в объявлении класса, приведенном на рис. 6.1 (стр. 112).

```
infixr 5 :^:
data Tree a =  Leaf a  |  Tree a :^: Tree a

instance (Eq a) => Eq (Tree a) where
        Leaf m == Leaf n  =  m==n
        u:^:v  == x:^:y   =  u==x && v==y
             _ == _       =  False

instance (Ord a) => Ord (Tree a) where
        Leaf m <= Leaf n  =  m<=n
        Leaf m <= x:^:y   =  True
        u:^:v  <= Leaf n  =  False
        u:^:v  <= x:^:y   =  u<x || u==x && v<=y

instance (Show a) => Show (Tree a) where

        showsPrec d (Leaf m) = showParen (d > app_prec) showStr
          where
             showStr = showString "Лист " . showsPrec (app_prec+1) m

        showsPrec d (u :^: v) = showParen (d > up_prec) showStr
          where
             showStr = showsPrec (up_prec+1) u .
                       showString " :^: "      .
                       showsPrec (up_prec+1) v
                -- Обратите внимание: правоассоциативность :^: игнорируется

instance (Read a) => Read (Tree a) where

        readsPrec d r =  readParen (d > up_prec)
                         (\r -> [(u:^:v,w) |
                                 (u,s) <- readsPrec (up_prec+1) r,
                                 (":^:",t) <- lex s,
                                 (v,w) <- readsPrec (up_prec+1) t]) r

                      ++ readParen (d > app_prec)
                         (\r -> [(Leaf m,t) |
                                 ("Лист",s) <- lex r,
                                 (m,t) <- readsPrec (app_prec+1) s]) r

up_prec  = 5    -- Приоритет :^:
app_prec = 10   -- Применение имеет приоритет на единицу больше чем
                -- наиболее сильно связанный оператор
```

Рис. 10.1: Пример производных экземпляров

Глава 11

Указания компилятору (псевдокомментарии)

Некоторые реализации компилятора поддерживают *указания компилятору* — *псевдокомментарии*, которые используются, чтобы передать дополнительные указания или подсказки компилятору, но не являются частью свойства языка Haskell и не меняют семантику программы. Эта глава резюмирует эту существующую практику. Не требуется, чтобы реализация соблюдала любой псевдокомментарий, но псевдокомментарий должен игнорироваться, если реализация не готова его обработать. С лексической точки зрения, псевдокомментарии выглядят как комментарии, за исключением того, что заключаются в `{-# #-}`.

11.1 Встраивание

decl → `{-# INLINE` *qvars* `#-}`
decl → `{-# NOINLINE` *qvars* `#-}`

Перевод:
объявление →
  `{-# INLINE` *список-квалифицированных-переменных* `#-}`
объявление →
  `{-# NOINLINE` *список-квалифицированных-переменных* `#-}`

Псевдокомментарий `INLINE` указывает компилятору генерировать указанные переменные на месте их использования. Компиляторы будут чаще автоматически генерировать (встраивать) простые выражения. Это можно предотвратить с помощью псевдокомментария `NOINLINE`.

11.2 Специализация

decl → `{-# SPECIALIZE` *spec*$_1$ `,` ... `,` *spec*$_k$ `#-}` $(k \geq 1)$
spec → *vars* :: *type*

Перевод:
объявление →
`{-# SPECIALIZE` *спецификатор*$_1$ `,` ... `,` *спецификатор*$_k$ `#-}`
$(k \geq 1)$
спецификатор →
список-переменных :: *тип*

Специализация используется, чтобы избежать неэффективности, связанной с диспетчированием перегруженных функций. Например,

```
factorial :: Num a => a -> a
factorial 0 = 0
factorial n = n * factorial (n-1)
{-# SPECIALIZE factorial :: Int -> Int,
               factorial :: Integer -> Integer #-}
```

при обращениях к `factorial` компилятор может обнаружить, что параметр имеет тип `Int` или `Integer`, но он будет использовать специализированную версию `factorial`, которая не затрагивает перегруженные числовые операции.

Часть II

Библиотеки Haskell 98

Глава 12

Рациональные числа

```
module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %
data  (Integral a)      => Ratio a = ...
type  Rational          =  Ratio Integer
(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational
instance  (Integral a)  => Eq         (Ratio a)  where ...
instance  (Integral a)  => Ord        (Ratio a)  where ...
instance  (Integral a)  => Num        (Ratio a)  where ...
instance  (Integral a)  => Real       (Ratio a)  where ...
instance  (Integral a)  => Fractional (Ratio a)  where ...
instance  (Integral a)  => RealFrac   (Ratio a)  where ...
instance  (Integral a)  => Enum       (Ratio a)  where ...
instance  (Read a,Integral a) => Read (Ratio a)  where ...
instance  (Integral a)  => Show       (Ratio a)  where ...
```

Для каждого типа `Integral` t есть тип `Ratio` t рациональных пар с компонентами типа t. Имя типа `Rational` является синонимом для `Ratio Integer`.

`Ratio` является экземпляром классов `Eq`, `Ord`, `Num`, `Real`, `Fractional`, `RealFrac`, `Enum`, `Read` и `Show`. В каждом случае экземпляр для `Ratio` t просто "повышает" соответствующие операции над t. Если t является ограниченным типом, результаты могут быть непредсказуемы; например, `Ratio Int` может вызвать переполнение целого числа даже для небольших по абсолютной величине рациональных чисел.

Оператор `(%)` составляет отношение двух целых чисел, сокращая дробь до членов без общего делителя и таких, что знаменатель является положительным. Функции `numerator` и `denominator` извлекают компоненты отношения (соответственно числитель

и знаменатель дроби); эти компоненты находятся в приведенном виде с положительным знаменателем. `Ratio` является абстрактным типом. Например, `12 % 8` сокращается до $3/2$, а `12 % (-8)` сокращается до $(-3)/2$.

Функция `approxRational`, будучи примененной к двум действительным дробным числам `x` и `epsilon`, возвращает простейшее рациональное число в пределах открытого интервала $(\texttt{x} - \texttt{epsilon},\ \texttt{x} + \texttt{epsilon})$. Говорят, что рациональное число n/d в приведенном виде является более простым, чем другое число n'/d', если $|n| \leq |n'|$ и $d \leq d'$. Обратите внимание, что можно доказать, что любой действительный интервал содержит единственное простейшее рациональное число.

12.1 Библиотека Ratio

```
-- Стандартные функции над рациональными числами
module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %

ratPrec = 7 :: Int

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

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

-- "reduce" --- это вспомогательная функция, которая используется только в этом
-- модуле. Она нормирует отношение путем деления числителя и знаменателя
-- на их наибольший общий делитель.
--
-- Например, 12 `reduce` 8    ==  3 :%   2
--           12 `reduce` (-8) ==  3 :% (-2)
reduce _ 0              =  error "Ratio.% : нулевой знаменатель"
reduce x y              =  (x `quot` d) :% (y `quot` d)
                           where d = gcd x y

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

numerator (x :% _)      =  x

denominator (_ :% y)    =  y

instance  (Integral a)  => Ord (Ratio a)  where
    (x:%y) <= (x':%y')  =  x * y' <= x' * y
    (x:%y) <  (x':%y')  =  x * y' <  x' * y

instance  (Integral a)  => Num (Ratio a)  where
    (x:%y) + (x':%y')   =  reduce (x*y' + x'*y) (y*y')
    (x:%y) * (x':%y')   =  reduce (x * x') (y * y')
    negate (x:%y)       =  (-x) :% y
    abs (x:%y)          =  abs x :% y
    signum (x:%y)       =  signum x :% 1
    fromInteger x       =  fromInteger x :% 1

instance  (Integral a)  => Real (Ratio a)  where
    toRational (x:%y)   =  toInteger x :% toInteger y

instance  (Integral a)  => Fractional (Ratio a)  where
    (x:%y) / (x':%y')   =  (x*y') % (y*x')
    recip (x:%y)        =  y % x
    fromRational (x:%y) =  fromInteger x :% fromInteger y

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

```
instance  (Integral a)  => Enum (Ratio a)  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate  -- Может вызвать переполнение
    enumFrom         =  numericEnumFrom         -- Эти функции вида numericEnumXXX
    enumFromThen     =  numericEnumFromThen     -- определены в Prelude.hs
    enumFromTo       =  numericEnumFromTo       -- но не экспортируются оттуда!
    enumFromThenTo   =  numericEnumFromThenTo

instance  (Read a, Integral a)  => Read (Ratio a)  where
    readsPrec p  =  readParen (p > ratPrec)
                              (\r -> [(x%y,u) | (x,s)   <- readsPrec (ratPrec+1) r,
                                                ("%",t) <- lex s,
                                                (y,u)   <- readsPrec (ratPrec+1) t ])

instance  (Integral a)  => Show (Ratio a)  where
    showsPrec p (x:%y)  =  showParen (p > ratPrec)
                               (showsPrec (ratPrec+1) x .
                                showString " % " .
                                showsPrec (ratPrec+1) y)

approxRational x eps    =  simplest (x-eps) (x+eps)
        where simplest x y | y < x      =  simplest y x
                           | x == y     =  xr
                           | x > 0      =  simplest' n d n' d'
                           | y < 0      =  - simplest' (-n') d' (-n) d
                           | otherwise  =  0 :% 1
                                       where xr@(n:%d) = toRational x
                                             (n':%d')  = toRational y

              simplest' n d n' d'       -- предполагает, что 0 < n%d < n'%d'
                        | r == 0     =  q :% 1
                        | q /= q'    =  (q+1) :% 1
                        | otherwise  =  (q*n''+d'') :% n''
                                    where (q,r)      =  quotRem n d
                                          (q',r')    =  quotRem n' d'
                                          (n'':%d'') =  simplest' d' r' d r
```

Глава 13

Комплексные числа

```
module  Complex (
    Complex((:+)), realPart, imagPart, conjugate,
    mkPolar, cis, polar, magnitude, phase ) where

infix  6  :+

data  (RealFloat a)     => Complex a = !a :+ !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

instance  (RealFloat a) => Eq         (Complex a)  where ...
instance  (RealFloat a) => Read       (Complex a)  where ...
instance  (RealFloat a) => Show       (Complex a)  where ...
instance  (RealFloat a) => Num        (Complex a)  where ...
instance  (RealFloat a) => Fractional (Complex a)  where ...
instance  (RealFloat a) => Floating   (Complex a)  where ...
```

Комплексные числа являются алгебраическим типом. Конструктор `(:+)` образует комплексное число из его действительной и мнимой прямоугольных компонент. Этот конструктор является строгим: если действительной или мнимой частью числа является $\perp$, все число является $\perp$. Комплексное число может быть также образовано из полярных компонент величины и фазы с помощью функции `mkPolar`. Функция `cis` генерирует комплексное число из угла t. С другой стороны, `cis` t возвращает комплексное значение с величиной 1 и фазой t (по модулю 2π).

Функция `polar` принимает в качестве аргумента комплексное число и возвращает пару (величина, фаза) в канонической форме: величина является неотрицательной, а фаза

находится в диапазоне $(-\pi, \pi]$; если величина равна нулю, то фаза также равна нулю.

Функции `realPart` и `imagPart` извлекают прямоугольные компоненты комплексного числа, а функции `magnitude` и `phase` извлекают полярные компоненты комплексного числа. Функция `conjugate` вычисляет сопряженное комплексное число обычным способом.

Величина и знак комплексного числа определены следующим образом:

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

То есть `abs` z — это число с величиной z, но ориентированное в направлении положительной действительной полуоси, тогда как `signum` z имеет фазу z, но единичную величину.

13.1 Библиотека `Complex`

```
module Complex(Complex((:+)), realPart, imagPart, conjugate, mkPolar,
               cis, polar, magnitude, phase)  where

infix  6  :+

data  (RealFloat a)     => Complex a = !a :+ !a  deriving (Eq,Read,Show)

realPart, imagPart :: (RealFloat a) => Complex a -> a
realPart (x:+y)  =  x
imagPart (x:+y)  =  y

conjugate        :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y) =  x :+ (-y)

mkPolar          :: (RealFloat a) => a -> a -> Complex a
mkPolar r theta  =  r * cos theta :+ r * sin theta

cis              :: (RealFloat a) => a -> Complex a
cis theta        =  cos theta :+ sin theta

polar            :: (RealFloat a) => Complex a -> (a,a)
polar z          =  (magnitude z, phase z)

magnitude :: (RealFloat a) => Complex a -> a
magnitude (x:+y) =  scaleFloat k
                     (sqrt ((scaleFloat mk x)^2 + (scaleFloat mk y)^2))
                    where k  = max (exponent x) (exponent y)
                          mk = - k
```

```
phase :: (RealFloat a) => Complex a -> a
phase (0 :+ 0) = 0
phase (x :+ y) = atan2 y x

instance  (RealFloat a) => Num (Complex a)  where
    (x:+y) + (x':+y')   =  (x+x') :+ (y+y')
    (x:+y) - (x':+y')   =  (x-x') :+ (y-y')
    (x:+y) * (x':+y')   =  (x*x'-y*y') :+ (x*y'+y*x')
    negate (x:+y)       =  negate x :+ negate y
    abs z               =  magnitude z :+ 0
    signum 0            =  0
    signum z@(x:+y)     =  x/r :+ y/r  where r = magnitude z
    fromInteger n       =  fromInteger n :+ 0

instance  (RealFloat a) => Fractional (Complex a)  where
    (x:+y) / (x':+y')   =  (x*x''+y*y'') / d :+ (y*x''-x*y'') / d
                           where x'' = scaleFloat k x'
                                 y'' = scaleFloat k y'
                                 k   = - max (exponent x') (exponent y')
                                 d   = x'*x'' + y'*y''
    fromRational a      =  fromRational a :+ 0
```

```
instance  (RealFloat a) => Floating (Complex a) where
    pi             =  pi :+ 0
    exp (x:+y)     =  expx * cos y :+ expx * sin y
                      where expx = exp x
    log z          =  log (magnitude z) :+ phase z

    sqrt 0         =  0
    sqrt z@(x:+y)  =  u :+ (if y < 0 then -v else v)
                      where (u,v) = if x < 0 then (v',u') else (u',v')
                            v'    = abs y / (u'*2)
                            u'    = sqrt ((magnitude z + abs x) / 2)

    sin (x:+y)     =  sin x * cosh y :+ cos x * sinh y
    cos (x:+y)     =  cos x * cosh y :+ (- sin x * sinh y)
    tan (x:+y)     =  (sinx*coshy:+cosx*sinhy)/(cosx*coshy:+(-sinx*sinhy))
                      where sinx  = sin x
                            cosx  = cos x
                            sinhy = sinh y
                            coshy = cosh y

    sinh (x:+y)    =  cos y * sinh x :+ sin  y * cosh x
    cosh (x:+y)    =  cos y * cosh x :+ sin y * sinh x
    tanh (x:+y)    =  (cosy*sinhx:+siny*coshx)/(cosy*coshx:+siny*sinhx)
                      where siny  = sin y
                            cosy  = cos y
                            sinhx = sinh x
                            coshx = cosh x

    asin z@(x:+y)  =  y':+(-x')
                      where  (x':+y') = log (((-y):+x) + sqrt (1 - z*z))
    acos z@(x:+y)  =  y'':+(-x'')
                      where (x'':+y'') = log (z + ((-y'):+x'))
                            (x':+y')   = sqrt (1 - z*z)
    atan z@(x:+y)  =  y':+(-x')
                      where (x':+y') = log (((1-y):+x) / sqrt (1+z*z))

    asinh z        =  log (z + sqrt (1+z*z))
    acosh z        =  log (z + (z+1) * sqrt ((z-1)/(z+1)))
    atanh z        =  log ((1+z) / sqrt (1-z*z))
```

Глава 14

Числовые функции

```
module Numeric(fromRat,
               showSigned, showIntAtBase,
               showInt, showOct, showHex,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where

fromRat        :: (RealFloat a) => Rational -> a

showSigned     :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS
showIntAtBase  :: Integral a => a -> (Int -> Char) -> a -> ShowS
showInt        :: Integral a => a -> ShowS
showOct        :: Integral a => a -> ShowS
showHex        :: Integral a => a -> ShowS

readSigned     :: (Real a) => ReadS a -> ReadS a
readInt        :: (Integral a) =>
                    a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readDec        :: (Integral a) => ReadS a
readOct        :: (Integral a) => ReadS a
readHex        :: (Integral a) => ReadS a

showEFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat      :: (RealFloat a) => a -> ShowS

floatToDigits  :: (RealFloat a) => Integer -> a -> ([Int], Int)

readFloat      :: (RealFrac a) => ReadS a
lexDigits      :: ReadS String
```

Эта библиотека содержит числовые функции разных сортов, многие из которых используются в стандартном Prelude.

Далее, напомним следующие определения типов из `Prelude`:

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

14.1 Функции преобразования величин в строки

- `showSigned :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS`
  преобразует возможно отрицательное значение `Real` типа `a` в строку. В вызове (`showSigned` *show prec val*) *val* является значением для отображения, *prec* — приоритетом внешнего контекста, а *show* — функцией, которая может отобразить (преобразовать в строку) значения без знака.

- `showIntAtBase :: Integral a => a -> (Int -> Char) -> a -> ShowS`
  отображает *неотрицательное* число `Integral`, используя основание, указанное в первом аргументе, и символьное представление, указанное во втором.

- `showInt, showOct, showHex :: Integral a => a -> ShowS`
  отображает *неотрицательные* числа `Integral` по основанию 10, 8 и 16 соответственно.

- `showFFloat, showEFloat, showGFloat`
  `:: (RealFloat a) => Maybe Int -> a -> ShowS`
  Эти три функции отображают значения `RealFloat` со знаком:

  – `showFFloat` использует стандартную десятичную запись (например, `245000`, `0.0015`).

  – `showEFloat` использует научную (показательную) запись (например, `2.45e2`, `1.5e-3`).

  – `showGFloat` использует стандартную десятичную запись для аргументов, чье абсолютное значение находится между `0.1` и `9,999,999`, и научную запись иначе.

  В вызове (`showEFloat` *digs val*), если *digs* является `Nothing`, значение отображается с полной точностью; если *digs* является `Just` d, то самое большее d цифр после десятичный точки будет отображено. В точности то же самое касается аргумента *digs* двух других функций.

- `floatToDigits :: (RealFloat a) => Integer -> a -> ([Int], Int)`
  преобразует основание и значение в величину в цифрах плюс показатель степени. Более точно, если

$$\texttt{floatToDigits}\ b\ r = (\texttt{[}d_1, d_2 ... d_n\texttt{]},\ e),$$

то выполняются следующие свойства:

- $r = 0.d_1 d_2 ..., d_n \ * \ b^e$
- $n \geq 0$
- $d_1 \neq 0$ (когда $n > 0$)
- $0 \leq d_i \leq b - 1$

14.2 Функции преобразования строк в другие величины

- `readSigned :: (Real a) => ReadS a -> ReadS a`
  считывает значение `Real` *со знаком*, учитывая наличие функции для считывания значений без знака.
- `readInt :: (Integral a) => a -> (Char->Bool) -> (Char->Int) -> ReadS a`
  считывает значение `Integral` *без знака* с произвольным основанием. В вызове (`readInt` *base isdig d2i*) *base* является основанием, *isdig* — предикатом, различающим допустимые цифры по этому основанию, а *d2i* преобразует символ допустимой цифры в `Int`.
- `readFloat :: (RealFrac a) => ReadS a`
  считывает значение `RealFrac` *без знака*, изображенное в десятичной научной записи.
- `readDec, readOct, readHex :: (Integral a) => ReadS a`
  считывают число без знака в десятичной, восьмиричной и шестнадцатиричной записи соответственно. В случае шестнадцатиричной записи допустимы буквы верхнего и нижнего регистра.
- `lexDigits :: ReadS String` считывает непустую строку десятичных цифр.

(NB: `readInt` является "двойственной" для `showIntAtBase`, а `readDec` является "двойственной" для `showInt`. Противоречивые имена этих функций сложились исторически.)

14.3 Прочие функции

- `fromRat :: (RealFloat a) => Rational -> a` преобразует значение `Rational` к любому типу в классе `RealFloat`.

14.4 Библиотека Numeric

```
module Numeric(fromRat,
               showSigned, showIntAtBase,
               showInt, showOct, showHex,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where

import Char   ( isDigit, isOctDigit, isHexDigit
              , digitToInt, intToDigit )
import Ratio  ( (%), numerator, denominator )
import Array  ( (!), Array, array )

-- Эта функция выполняет преобразование рационального числа в число с плавающей
-- точкой.
-- Ее следует использовать в экземплярах Fractional классов Float и Double.
fromRat :: (RealFloat a) => Rational -> a
fromRat x =
    if x == 0 then encodeFloat 0 0 -- Сначала обрабатывает исключительные ситуации
    else if x < 0 then - fromRat' (-x)
    else fromRat' x

-- Процесс преобразования:
-- Перевести (масштабировать) рациональное число в систему счисления с основанием
-- RealFloat, пока оно не будет лежать в диапазоне мантиссы (используется
-- функциями decodeFloat/encodeFloat).
-- Затем округлить рациональное число до Integer и закодировать его с помощью
-- экспоненты, полученной при переводе числа в систему счисления.
-- Для того чтобы ускорить процесс масштабирования, мы вычисляем log2 числа, чтобы
-- получить первое приближение экспоненты.
fromRat' :: (RealFloat a) => Rational -> a
fromRat' x = r
  where b = floatRadix r
        p = floatDigits r
        (minExp0, _) = floatRange r
        minExp = minExp0 - p            -- действительная минимальная экспонента
        xMin = toRational (expt b (p-1))
        xMax = toRational (expt b p)
        p0 = (integerLogBase b (numerator x) -
              integerLogBase b (denominator x) - p) `max` minExp
        f = if p0 < 0 then 1 % expt b (-p0) else expt b p0 % 1
        (x', p') = scaleRat (toRational b) minExp xMin xMax p0 (x / f)
        r = encodeFloat (round x') p'
```

```
-- Масштабировать x, пока не выполнится условие xMin <= x < xMax или
-- p (экспонета) <= minExp.
scaleRat :: Rational -> Int -> Rational -> Rational ->
             Int -> Rational -> (Rational, Int)
scaleRat b minExp xMin xMax p x =
    if p <= minExp then
        (x, p)
    else if x >= xMax then
        scaleRat b minExp xMin xMax (p+1) (x/b)
    else if x < xMin  then
        scaleRat b minExp xMin xMax (p-1) (x*b)
    else
        (x, p)

-- Возведение в степень с помощью кэша наиболее частых чисел.
minExpt = 0::Int
maxExpt = 1100::Int
expt :: Integer -> Int -> Integer
expt base n =
    if base == 2 && n >= minExpt && n <= maxExpt then
        expts!n
    else
        base^n

expts :: Array Int Integer
expts = array (minExpt,maxExpt) [(n,2^n) | n <- [minExpt .. maxExpt]]

-- Вычисляет (нижнюю границу) log i по основанию b.
-- Наиболее простой способ --- просто делить i на b, пока оно не станет меньше b,
-- но это было бы очень медленно!  Мы просто немного более сообразительны.
integerLogBase :: Integer -> Integer -> Int
integerLogBase b i =
     if i < b then
        0
     else
        -- Пытается сначала возвести в квадрат основание, чтобы  сократить число
        -- делений.
        let l = 2 * integerLogBase (b*b) i
            doDiv :: Integer -> Int -> Int
            doDiv i l = if i < b then l else doDiv (i `div` b) (l+1)
        in  doDiv (i `div` (b^l)) l

-- Разные утилиты для отображения целых чисел и чисел с плавающей точкой
showSigned :: Real a => (a -> ShowS) -> Int -> a -> ShowS
showSigned showPos p x
  | x < 0     = showParen (p > 6) (showChar '-' . showPos (-x))
  | otherwise = showPos x

-- showInt, showOct, showHex используются только для положительных чисел
showInt, showOct, showHex :: Integral a => a -> ShowS
showOct = showIntAtBase  8 intToDigit
showInt = showIntAtBase 10 intToDigit
showHex = showIntAtBase 16 intToDigit
```

```
showIntAtBase :: Integral a
              => a              -- основание
              -> (Int -> Char)  -- цифра для символа
              -> a              -- число для отображения
              -> ShowS
showIntAtBase base intToDig n rest
  | n < 0     = error "Numeric.showIntAtBase: не могу отображать отрицательные числа"
  | n' == 0   = rest'
  | otherwise = showIntAtBase base intToDig n' rest'
  where
    (n',d) = quotRem n base
    rest'  = intToDig (fromIntegral d) : rest

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]

-- readInt считывает строку цифр, используя произвольное основание.
-- Знаки минус в начале строки должны обрабатываться где-то в другом месте.
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 ]

-- Функции для считывания беззнаковых чисел с различными основаниями
readDec, readOct, readHex :: (Integral a) => ReadS a
readDec = readInt 10 isDigit    digitToInt
readOct = readInt  8 isOctDigit digitToInt
readHex = readInt 16 isHexDigit digitToInt

showEFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat      :: (RealFloat a) => a -> ShowS

showEFloat d x =  showString (formatRealFloat FFExponent d x)
showFFloat d x =  showString (formatRealFloat FFFixed d x)
showGFloat d x =  showString (formatRealFloat FFGeneric d x)
showFloat      =  showGFloat Nothing

-- Это типы форматов.  Этот тип не экспортируется.
data FFFormat = FFExponent | FFFixed | FFGeneric
```

```
formatRealFloat :: (RealFloat a) => FFFormat -> Maybe Int -> a -> String
formatRealFloat fmt decs x
  = s
  where
    base = 10
    s = if isNaN x then
            "NaN"
        else if isInfinite x then
            if x < 0 then "-Infinity" else "Infinity"
        else if x < 0 || isNegativeZero x then
            '-' : doFmt fmt (floatToDigits (toInteger base) (-x))
        else
            doFmt fmt (floatToDigits (toInteger base) x)
    doFmt fmt (is, e)
      = let
           ds = map intToDigit is
        in
        case fmt of
          FFGeneric ->
              doFmt (if e < 0 || e > 7 then FFExponent else FFFixed)
                    (is, e)
          FFExponent ->
            case decs of
              Nothing ->
                case ds of
                   []    -> "0.0e0"
                   [d]   -> d : ".0e" ++ show (e-1)
                   d:ds  -> d : '.' : ds ++ 'e':show (e-1)
              Just dec ->
                let dec' = max dec 1 in
                case is of
                  [] -> '0':'.':take dec' (repeat '0') ++ "e0"
                  _ ->
                    let (ei, is') = roundTo base (dec'+1) is
                        d:ds = map intToDigit
                                   (if ei > 0 then init is' else is')
                    in d:'.':ds  ++ "e" ++ show (e-1+ei)
          FFFixed ->
            case decs of
               Nothing  -- Всегда печатает десятичную точку
                 | e > 0     -> take e (ds ++ repeat '0')
```

```
                              ++ '.' : mk0 (drop e ds)
                | otherwise -> "0." ++ mk0 (replicate (-e) '0' ++ ds)

              Just dec ->  -- Печатает десятичную точку, если dec > 0
                let dec' = max dec 0 in
                if e >= 0 then
                  let (ei, is') = roundTo base (dec' + e) is
                      (ls, rs)  = splitAt (e+ei)
                                          (map intToDigit is')
                  in  mk0 ls ++ mkdot0 rs
                else
                  let (ei, is') = roundTo base dec'
                                          (replicate (-e) 0 ++ is)
                      d : ds = map intToDigit
                                   (if ei > 0 then is' else 0:is')
                  in  d : mkdot0 ds
            where
              mk0  = "0"        -- Печатает 0.34, а не .34
              mk0 s  = s
              mkdot0  =        -- Печатает 34, а не 34.
              mkdot0 s  = '.' : s  -- когда формат задает отсутствие
                                   -- цифр после десятичной точки

roundTo :: Int -> Int -> [Int] -> (Int, [Int])
roundTo base d is = case f d is of
                (0, is) -> (0, is)
                (1, is) -> (1, 1 : is)
  where b2 = base `div` 2
        f n [] = (0, replicate n 0)
        f 0 (i:_) = (if i >= b2 then 1 else 0, [])
        f d (i:is) =
            let (c, ds) = f (d-1) is
                i' = c + i
            in  if i' == base then (1, 0:ds) else (0, i':ds)

--
-- Базируется на "Быстрой и точной печати чисел с плавающей точкой"
-- Р.Г. Бургера и Р.К. Дайбвига
-- ("Printing Floating-Point Numbers Quickly and Accurately"
-- R.G. Burger и R. K. Dybvig)
-- в PLDI 96.
-- Версия, приведенная здесь, использует намного более медленную оценку алгоритма.
-- Ее следует усовершенствовать.
-- Эта функция возвращает непустой список цифр (целые числа в диапазоне
-- [0..base-1]) и экспоненту.  В общем случае, если
--      floatToDigits r = ([a, b, ... z], e)
-- то
--      r = 0.ab..z * base^e
--
floatToDigits :: (RealFloat a) => Integer -> a -> ([Int], Int)
```

```
floatToDigits _ 0 = ([], 0)
floatToDigits base x =
    let (f0, e0) = decodeFloat x
        (minExp0, _) = floatRange x
        p = floatDigits x
        b = floatRadix x
        minExp = minExp0 - p            -- действительная минимальная экспонента
        -- В Haskell требуется, чтобы f было скорректировано так, чтобы
        -- денормализационные числа имели невозможно низкую экспоненту.
        -- Для этого используется коррекция.
        f :: Integer
        e :: Int
        (f, e) = let n = minExp - e0
                 in  if n > 0 then (f0 `div` (b^n), e0+n) else (f0, e0)
        (r, s, mUp, mDn) =
           if e >= 0 then
               let be = b^e in
               if f == b^(p-1) then
                   (f*be*b*2, 2*b, be*b, b)
               else
                   (f*be*2, 2, be, be)
           else
               if e > minExp && f == b^(p-1) then
                   (f*b*2, b^(-e+1)*2, b, 1)
               else
                   (f*2, b^(-e)*2, 1, 1)
        k =
            let k0 =
                     if b==2 && base==10 then
                         -- logBase 10 2 немного больше, чем 3/10, поэтому
                         -- следующее вызовет ошибку на нижней стороне.
                         -- Игнорирование дроби создаст эту ошибку даже больше.
                         -- Haskell обещает, что p-1 <= logBase b f < p.
                         (p - 1 + e0) * 3 `div` 10
                     else
                         ceiling ((log (fromInteger (f+1)) +
                                   fromIntegral e * log (fromInteger b)) /
                                    log (fromInteger base))
                fixup n =
                     if n >= 0 then
                         if r + mUp <= expt base n * s then n else fixup (n+1)
                     else
                         if expt base (-n) * (r + mUp) <= s then n
```

```
                                                    else fixup (n+1)
            in  fixup k0

        gen ds rn sN mUpN mDnN =
            let (dn, rn') = (rn * base) `divMod` sN
                mUpN' = mUpN * base
                mDnN' = mDnN * base
            in  case (rn' < mDnN', rn' + mUpN' > sN) of
                (True,  False) -> dn : ds
                (False, True)  -> dn+1 : ds
                (True,  True)  -> if rn' * 2 < sN then dn : ds else dn+1 : ds
                (False, False) -> gen (dn:ds) rn' sN mUpN' mDnN'
        rds =
            if k >= 0 then
                gen [] r (s * expt base k) mUp mDn
            else
                let bk = expt base (-k)
                in  gen [] (r * bk) s (mUp * bk) (mDn * bk)
    in  (map fromIntegral (reverse rds), k)

-- Эта функция для считывания чисел с плавающей точкой использует менее
-- ограничивающий синтаксис для чисел с плавающей точкой, чем лексический
-- анализатор Haskell. `.' является необязательной.

readFloat     :: (RealFrac a) => ReadS a
readFloat r    = [(fromRational ((n%1)*10^^(k-d)),t) | (n,d,s) <- readFix r,
                                                       (k,t)   <- readExp s] ++
                  [ (0/0, t) | ("NaN",t)      <- lex r] ++
                  [ (1/0, t) | ("Infinity",t) <- lex r]
                 where
                  readFix r = [(read (ds++ds'), length ds', t)
                               | (ds,d) <- lexDigits r,
                                 (ds',t) <- lexFrac d ]

                  lexFrac ('.':ds) = lexDigits ds
                  lexFrac s        = [(,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

lexDigits        :: ReadS String
lexDigits        =  nonnull isDigit

nonnull          :: (Char -> Bool) -> ReadS String
nonnull p s      =  [(cs,t) | (cs@(_:_),t) <- [span p s]]
```

Глава 15

Операции индексации

```
module Ix ( Ix(range, index, inRange, rangeSize) ) where

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

instance                    Ix Char      where ...
instance                    Ix Int       where ...
instance                    Ix Integer   where ...
instance  (Ix a, Ix b)  => Ix (a,b)      where ...
-- и так далее
instance                    Ix Bool      where ...
instance                    Ix Ordering  where ...
```

Класс `Ix` используется для того, чтобы отобразить непрерывный отрезок значений на тип целых чисел. Это используется прежде всего для индексации массивов (см. главу 16). Класс `Ix` содержит методы `range`, `index` и `inRange`. Операция `index` отображает пару ограничений, которая определяет нижнюю и верхнюю границы диапазона, и индекс в целое число. Операция `range` перечисляет все индексы; операция `inRange` сообщает, находится ли конкретный индекс в диапазоне, заданном парой ограничений.

Реализация имеет право предполагать выполнение следующих правил относительно этих операций:

```
range (l,u) !! index (l,u) i == i   -- когда i находится в указанном
                                    -- диапазоне
inRange (l,u) i              == i `elem` range (l,u)
map index (range (l,u))      == [0..rangeSize (l,u)]
```

15.1 Выведение экземпляров `Ix`

Есть возможность вывести (произвести) экземпляр класса `Ix` автоматически, используя инструкцию `deriving` в объявлении `data` (раздел 4.3.3). Объявления таких производных экземпляров класса `Ix` возможны только для перечислений (т.е. типов данных, имеющих конструкторы без аргументов) и типов данных с одним конструктором, у которого компоненты имеют типы, являющиеся экземплярами класса `Ix`. Реализация Haskell должна обеспечить экземпляры класса `Ix` для кортежей по меньшей мере вплоть до 15 размера.

- Для *перечисления* предполагается, что конструкторы без аргументов нумеруются слева направо индексами от 0 до $n-1$ включительно. Это та же самая нумерация, которая определена в классе `Enum`. Например, при типе данных:

  ```
  data Colour = Red | Orange | Yellow | Green | Blue | Indigo | Violet
  ```

  мы получили бы:

  ```
  range   (Yellow,Blue)        ==  [Yellow,Green,Blue]
  index   (Yellow,Blue) Green  ==  1
  inRange (Yellow,Blue) Red    ==  False
  ```

- Для *типов данных с одним конструктором* объявления производных экземпляров являются такими, как те, что изображены на рис. 15.1 для кортежей.

```
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'
-- Экземпляры для остальных кортежей получены по этой схеме:
--
--  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
```

Рис. 15.1: Выведение экземпляров класса Ix

15.2 Библиотека Ix

```
module Ix ( Ix(range, index, inRange, rangeSize) ) where

class  Ord a => Ix a  where
    range     :: (a,a) -> [a]
    index     :: (a,a) -> a -> Int
    inRange   :: (a,a) -> a -> Bool
    rangeSize :: (a,a) -> Int
    rangeSize b@(l,h) | null (range b) = 0
                      | otherwise      = index b h + 1
        -- NB: замена "null (range b)" на "not (l <= h)"
        -- завершится неудачей, если границы являются кортежами. Например,
        --      (1,2) <= (2,1)
        -- но диапазон (range) тем не менее пуст:
        --      range ((1,2),(2,1)) = []

instance  Ix Char  where
    range (m,n)         = [m..n]
    index b@(c,c') ci
        | inRange b ci  =  fromEnum ci - fromEnum c
        | otherwise     =  error "Ix.index: Индекс находится за пределами диапазона."
    inRange (c,c') i    =  c <= i && i <= c'

instance  Ix Int  where
    range (m,n)         = [m..n]
    index b@(m,n) i
        | inRange b i   =  i - m
        | otherwise     =  error "Ix.index: Индекс находится за пределами диапазона."
    inRange (m,n) i     =  m <= i && i <= n

instance  Ix Integer  where
    range (m,n)         = [m..n]
    index b@(m,n) i
        | inRange b i   =  fromInteger (i - m)
        | otherwise     =  error "Ix.index: Индекс находится за пределами диапазона."
    inRange (m,n) i     =  m <= i && i <= n

instance (Ix a,Ix b) => Ix (a, b) -- является производным, для всех кортежей
instance Ix Bool                  -- является производным
instance Ix Ordering              -- является производным
instance Ix ()                    -- является производным
```

Глава 16

Массивы

```
module  Array (
        module Ix,  -- экспортирует весь Ix в целях удобства
        Array, array, listArray, (!), bounds, indices, elems, assocs,
        accumArray, (//), accum, ixmap ) where

import Ix

infixl 9  !, //

data  (Ix a)    => Array a b = ...      -- Абстрактный

array           :: (Ix a) => (a,a) -> [(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 -> [(a,b)]
accumArray      :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                             -> Array a b
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
accum           :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                             -> Array a b
ixmap           :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                             -> Array a c

instance                              Functor (Array a) where ...
instance  (Ix a, Eq b)           => Eq   (Array a b)  where ...
instance  (Ix a, Ord b)          => Ord  (Array a b)  where ...
instance  (Ix a, Show a, Show b) => Show (Array a b)  where ...
instance  (Ix a, Read a, Read b) => Read (Array a b)  where ...
```

Haskell обеспечивает индексируемые *массивы*, которые можно рассматривать как функции, чьи области определения изоморфны соприкасающимся подмножествам целых чисел. Функции, ограниченные таким образом, можно эффективно реализовать; в частности, программист может ожидать разумно быстрого доступа к компонентам. Чтобы гарантировать возможность такой реализации, массивы обрабатываются как данные, а не как обычные функции.

Так как большинство функций массива затрагивают класс `Ix`, этот модуль экспортируется из `Array`, чтобы не было необходимости модулям импортировать и `Array`, и `Ix`.

16.1 Создание массивов

Если `a` — тип индекса, а `b` — любой тип, тип массивов с индексами в `a` и элементами в `b` записывается так: `Array a b`. Массив может быть создан с помощью функции `array`. Первым аргументом `array` является пара *границ*, каждая из которых имеет тип индекса массива. Эти границы являются соответственно наименьшим и наибольшим индексами в массиве. Например, вектор с началом в 1 длины `10` имеет границы `(1,10)`, а матрица 10 на 10 с началом в 1 имеет границы `((1,1),(10,10))`.

Вторым аргументом `array` является список *ассоциаций* вида (*индекс*, *значение*). Обычно этот список выражен в виде описания элементов. Ассоциация `(i, x)` определяет, что значением массива по индексу `i` является `x`. Массив не определен (т.е. $\perp$), если какой-нибудь индекс в списке находится вне границ. Если какие-нибудь две ассоциации в списке имеют один и тот же индекс, значение по этому индексу не определено (т.е. $\perp$). Так как индексы должны быть проверены на наличие этих ошибок, `array` является строгим по аргументу границ и по индексам списка ассоциаций, но не является строгим по значениям. Таким образом, возможны такие рекуррентные отношения:

```
a = array (1,100) ((1,1) : [(i, i * a!(i-1)) | i <- [2..100]])
```

Не каждый индекс в пределах границ массива обязан появиться в списке ассоциаций, но значения, связанные с индексами, которых нет в списке, будут не определены (т.е. $\perp$). На рис. 16.1 изображены некоторые примеры, которые используют конструктор `array`.

Оператор `(!)` обозначает доступ к элементам массива по индексу (операция индексации массива). Функция `bounds`, будучи примененной к массиву, возвращает его границы. Функции `indices`, `elems` и `assocs`, будучи примененными к массиву, возвращают соответственно списки индексов, элементов или ассоциаций в порядке возрастания их индексов. Массив можно создать из пары границ и списка значений в порядке возрастания их индексов, используя функцию `listArray`.

Если в каком-либо измерении нижняя граница больше чем верхняя граница, то такой массив допустим, но он пуст. Индексация пустого массива всегда приводит к ошибке

```
-- Масштабирование массива чисел с помощью заданного числа:
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

-- Инвертирование массива, который содержит перестановку своих индексов
invPerm :: (Ix a) => Array a a -> Array a a
invPerm a = array b [(a!i, i) | i <- range b]
            where b = bounds a

-- Скалярное произведение двух векторов
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 "массивы не подходят для скалярного произведения"
            where b = bounds v
```

Рис. 16.1: Примеры массивов

выхода за границы массива, но `bounds` по-прежнему сообщает границы, с которыми массив был создан.

16.1.1 Накопленные массивы

Другая функция создания массива, `accumArray`, ослабляет ограничение, при котором данный индекс может появляться не более одного раза в списке ассоциаций, используя *функцию накопления*, которая объединяет значения ассоциаций с одним и тем же индексом. Первым аргументом `accumArray` является функция накопления; вторым — начальное значение; оставшиеся два аргумента являются соответственно парой границ и списком ассоциаций, как и для функции `array` . Например, при заданном списке значений некоторого типа индекса, `hist` создает гистограмму числа вхождений каждого индекса в пределах указанного диапазона:

```
hist :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | i<-is, inRange bnds i]
```

Если функция накопления является строгой, то `accumArray` является строгой в отношении значений, также как и в отношении индексов, в списке ассоциаций. Таким образом, в отличие от обычных массивов, накопленные массивы не должны быть в общем случае рекурсивными.

16.2 Добавочные обновления массивов

Оператор (`//`) принимает в качестве аргументов массив и список пар и возвращает массив, идентичный левому аргументу, за исключением того, что он обновлен

```
-- Прямоугольный подмассив
subArray :: (Ix a) => (a,a) -> Array a b -> Array a b
subArray bnds = ixmap bnds (\i->i)

-- Строка матрицы
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'),(_,u')) = bounds x

-- Диагональ матрицы (матрица предполагается квадратная)
diag :: (Ix a) => Array (a,a) b -> Array a b
diag x = ixmap (l,u) (\i->(i,i)) x
       where
         ((l,_),(u,_)) = bounds x

-- Проекция первых компонент массива пар
firstArray :: (Ix a) => Array a (b,c) -> Array a b
firstArray = fmap (\(x,y)->x)
```

Рис. 16.2: Примеры производных массивов

ассоциациями из правого аргумента. (Как и с функцией `array`, индексы в списке ассоциаций должна быть уникальны по отношению к обновляемым элементам, которые определены.) Например, если `m` — матрица `n` на `n` с началом в 1, то `m//[((i,i), 0) | i <- [1..n]]` — та же самая матрица, у которой диагональ заполнена нулями.

`accum` f принимает в качестве аргументов массив и список ассоциаций и накапливает пары из списка в массив с помощью функции накопления f. Таким образом, `accumArray` можно определить через `accum`:

```
accumArray f z b = accum f (array b [(i, z) | i <- range b])
```

16.3 Производные массивы

Функции `fmap` и `ixmap` получают новые массивы из существующих; их можно рассматривать как обеспечение композиции функций слева и справа соответственно, с отображением, которое реализует исходный массив. Функция `fmap` преобразовывает значения массива, в то время как `ixmap` позволяет выполнять преобразования на индексах массива. На рис. 16.2 изображены некоторые примеры.

16.4 Библиотека `Array`

```
module  Array (
    module Ix,  -- экспортировать весь Ix
    Array, array, listArray, (!), bounds, indices, elems, assocs,
    accumArray, (//), accum, ixmap ) where
```

```
import Ix
import List( (\\) )

infixl 9  !, //

data (Ix a) => Array a b = MkArray (a,a) (a -> b) deriving ()

array       :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
array b ivs =
    if and [inRange b i | (i,_) <- ivs]
        then MkArray b
                    (\j -> case [v | (i,v) <- ivs, i == j] of
                           [v]   -> v
                           []    -> error "Array.!: \
                                          \неопределенный элемент массива"
                           _     -> error "Array.!: \
                                          \множественно определенный элемент массива")
        else error "Array.array: ассоциация массива находится за пределами диапазона"

listArray             :: (Ix a) => (a,a) -> [b] -> Array a b
listArray b vs        =  array b (zipWith (\ a b -> (a,b)) (range b) vs)

(!)                   :: (Ix a) => Array a b -> a -> b
(!) (MkArray _ f)     =  f

bounds                :: (Ix a) => Array a b -> (a,a)
bounds (MkArray b _)  =  b

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

elems                 :: (Ix a) => Array a b -> [b]
elems a               =  [a!i | i <- indices a]

assocs                :: (Ix a) => Array a b -> [(a,b)]
assocs a              =  [(i, a!i) | i <- indices a]

(//)                  :: (Ix a) => Array a b -> [(a,b)] -> Array a b
a // new_ivs          = array (bounds a) (old_ivs ++ new_ivs)
                      where
                        old_ivs = [(i,a!i) | i <- indices a,
                                             i `notElem` new_is]
                        new_is  = [i | (i,_) <- new_ivs]

accum                 :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                                -> Array a b
accum f               =  foldl (\a (i,v) -> a // [(i,f (a!i) v)])

accumArray            :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                                -> Array a b
accumArray f z b      =  accum f (array b [(i,z) | i <- range b])

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

instance  (Ix a)          => Functor (Array a) where
    fmap fn (MkArray b f) =  MkArray b (fn . f)
```

```
instance  (Ix a, Eq b)  => Eq (Array a b)  where
    a == a' =  assocs a == assocs a'

instance  (Ix a, Ord b) => Ord (Array a b)  where
    a <= a' =  assocs a <= assocs a'

instance  (Ix a, Show a, Show b) => Show (Array a b)  where
    showsPrec p a = showParen (p > arrPrec) (
                    showString "array " .
                    showsPrec (arrPrec+1) (bounds a) . showChar ' ' .
                    showsPrec (arrPrec+1) (assocs a)                 )

instance  (Ix a, Read a, Read b) => Read (Array a b)  where
    readsPrec p = readParen (p > arrPrec)
           (\r -> [ (array b as, u)
                  | ("array",s) <- lex r,
                    (b,t)       <- readsPrec (arrPrec+1) s,
                    (as,u)      <- readsPrec (arrPrec+1) t ])

-- Приоритет функции 'array' --- тот же, что и приоритет самого применения функции
arrPrec = 10
```

Глава 17

Утилиты работы со списками

```
module List (
    elemIndex, elemIndices,
    find, findIndex, findIndices,
    nub, nubBy, delete, deleteBy, (\\), deleteFirstsBy,
    union, unionBy, intersect, intersectBy,
    intersperse, transpose, partition, group, groupBy,
    inits, tails, isPrefixOf, isSuffixOf,
    mapAccumL, mapAccumR,
    sort, sortBy, insert, insertBy, maximumBy, minimumBy,
    genericLength, genericTake, genericDrop,
    genericSplitAt, genericIndex, genericReplicate,
    zip4, zip5, zip6, zip7,
    zipWith4, zipWith5, zipWith6, zipWith7,
    unzip4, unzip5, unzip6, unzip7, unfoldr,
    -- ...и то, что экспортирует Prelude
    -- [](:), [],     -- Это встроенный синтаксис
    map, (++), concat, filter,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3
    ) where

infix 5 \\
```

```
elemIndex           :: Eq a => a -> [a] -> Maybe Int
elemIndices         :: Eq a => a -> [a] -> [Int]
find                :: (a -> Bool) -> [a] -> Maybe a
findIndex           :: (a -> Bool) -> [a] -> Maybe Int
findIndices         :: (a -> Bool) -> [a] -> [Int]
nub                 :: Eq a => [a] -> [a]
nubBy               :: (a -> a -> Bool) -> [a] -> [a]
delete              :: Eq a => a -> [a] -> [a]
deleteBy            :: (a -> a -> Bool) -> a -> [a] -> [a]
(\\)                :: Eq a => [a] -> [a] -> [a]
deleteFirstsBy      :: (a -> a -> Bool) -> [a] -> [a] -> [a]
union               :: Eq a => [a] -> [a] -> [a]
unionBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]
```

```
intersect        :: Eq a => [a] -> [a] -> [a]
intersectBy      :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersperse      :: a -> [a] -> [a]
transpose        :: [[a]] -> [[a]]
partition        :: (a -> Bool) -> [a] -> ([a],[a])
group            :: Eq a => [a] -> [[a]]
groupBy          :: (a -> a -> Bool) -> [a] -> [[a]]
inits            :: [a] -> [[a]]
tails            :: [a] -> [[a]]
isPrefixOf       :: Eq a => [a] -> [a] -> Bool
isSuffixOf       :: Eq a => [a] -> [a] -> Bool
mapAccumL        :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR        :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
unfoldr          :: (b -> Maybe (a,b)) -> b -> [a]
sort             :: Ord a => [a] -> [a]
sortBy           :: (a -> a -> Ordering) -> [a] -> [a]
insert           :: Ord a => a -> [a] -> [a]
insertBy         :: (a -> a -> Ordering) -> a -> [a] -> [a]
maximumBy        :: (a -> a -> Ordering) -> [a] -> a
minimumBy        :: (a -> a -> Ordering) -> [a] -> a
genericLength    :: Integral a => [b] -> a
genericTake      :: Integral a => a -> [b] -> [b]
genericDrop      :: Integral a => a -> [b] -> [b]
genericSplitAt   :: Integral a => a -> [b] -> ([b],[b])
genericIndex     :: Integral a => [b] -> a -> b
genericReplicate :: Integral a => a -> b -> [b]

zip4             :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip5             :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip6             :: [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)]
zipWith4         :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith5         :: (a->b->c->d->e->f) ->
                    [a]->[b]->[c]->[d]->[e]->[f]
zipWith6         :: (a->b->c->d->e->f->g) ->
                    [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7         :: (a->b->c->d->e->f->g->h) ->
                    [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
unzip4           :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip5           :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip6           :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip7           :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
```

В этой библиотеке определены некоторые редко используемые операции над списками.

17.1 Индексирование списков

- `elemIndex val list` возвращает индекс первого вхождения, если таковые имеются, `val` в `list` в виде `Just index`. `Nothing` возвращается, если выполняется `not (val ‘elem‘ list)`.
- `elemIndices val list` возвращает упорядоченный список индексов вхождений `val` в `list`.
- `find` возвращает первый элемент списка, который удовлетворяет предикату, или Nothing, если нет такого элемента. `findIndex` возвращает соответствующий индекс. `findIndices` возвращает список всех таких индексов.

17.2 Операции над “множествами”

Имеется ряд операций над “множествами”, определенные над типом `List`. `nub` (означает “сущность”) удаляет дублирующие элементы из списка. `delete`, `(\\)`, `union` и `intersect` (и их `By`-варианты) сохраняют инвариант: их результат не содержит дубликаты, при условии, что их первый аргумент не содержит дубликаты.

- `nub` удаляет дублирующие элементы из списка. Например:

```
nub [1,3,1,4,3,3] = [1,3,4]
```

- `delete x` удаляет первое вхождение `x` из указанного в его аргументе списка, например,

```
delete 'a' "banana" == "bnana"
```

- `(\\)` является разницей списков (неассоциативная операция). В результате `xs \\ ys` первое вхождение каждого элемента `ys` поочередно (если таковые имеются) удалены из `xs`. Таким образом, `(xs ++ ys) \\ xs == ys`.
- `union` является объединением списков, например,

```
"dog" `union` "cow" == "dogcw"
```

- `intersect` является пересечением списков, например,

```
[1,2,3,4] `intersect` [2,4,6,8] == [2,4]
```

17.3 Преобразования списков

- `intersperse sep` вставляет `sep` между элементами указанного в его аргументе списка, Например,

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

- `transpose` переставляет строки и столбцы своего аргумента, например,

```
transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]
```

- `partition` принимает в качестве аргументов предикат и список и возвращает пару списков: соответственно те элементы списка, которые удовлетворяют, и те, которые не удовлетворяют предикату, т.е.,

```
partition p xs == (filter p xs, filter (not . p) xs)
```

- `sort` реализует устойчивый алгоритм сортировки, заданной здесь в терминах функции `insertBy`, которая вставляет объекты в список согласно указанному отношению упорядочивания.

- `insert` помещает новый элемент в *упорядоченный* список (элементы размещаются по возрастанию).

- `group` разделяет указанный в его аргументе список на список списков одинаковых, соседних элементов. Например,

```
group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
```

- `inits` возвращает список начальных сегментов указанного в его аргументе списка, наиболее короткие — в начале списка.

```
inits "abc" == [,"a","ab","abc"]
```

- `tails` возвращает список всех конечных сегментов указанного в его аргументе списка, наиболее длинные — в начале списка.

```
tails "abc" == ["abc", "bc", "c",]
```

- `mapAccumL f s l` применяет `f` по отношению к накапливающему аргументу "состояния " `s` и к каждому элементу `l` по очереди.

- `mapAccumR` похожа на `mapAccumL` за исключением того, что список обрабатывается справа налево, а не слева направо.

17.4 unfoldr

Функция `unfoldr` является "двойственной" к `foldr`: тогда как `foldr` приводит список к суммарному значению, `unfoldr` строит список из случайного значения. Например:

```
iterate f == unfoldr (\x -> Just (x, f x))
```

В некоторых случаях `unfoldr` может аннулировать операцию `foldr`:

```
unfoldr f' (foldr f z xs) == xs
```

если выполняется следующее:

```
f' (f x y) = Just (x,y)
f' z       = Nothing
```

17.5 Предикаты

`isPrefixOf` и `isSuffixOf` проверяют, является ли первый аргумент соответственно приставкой или суффиксом второго аргумента.

17.6 "By"-операции

В соответствии с соглашением, перегруженные функции имеют неперегруженные копии, чьи имена имеют суффикс "`By`". Например, функция `nub` могла быть определена следующим образом:

```
nub              :: (Eq a) => [a] -> [a]
nub []           =  []
nub (x:xs)       =  x : nub (filter (\y -> not (x == y)) xs)
```

Тем не менее, метод сравнения на равенство не может подходить под все ситуации. Функция:

```
nubBy            :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq []      =  []
nubBy eq (x:xs)  =  x : nubBy eq (filter (\y -> not (eq x y)) xs)
```

позволяет программисту добавлять свою собственную проверку равенства. Когда "`By`"-функция заменяет контекст `Eq` бинарным предикатом, предполагается, что предикат определяет эквивалентность; когда "`By`"-функция заменяет контекст `Ord` бинарным предикатом, предполагается, что предикат определяет нестрогий порядок.

"By"-вариантами являются следующие: `nubBy`, `deleteBy`, `deleteFirstsBy` (By-вариант \\), `unionBy`, `intersectBy`, `groupBy`, `sortBy`, `insertBy`, `maximumBy`, `minimumBy`.

Библиотека не обеспечивает `elemBy`, потому что `any (eq x)` выполняет ту же работу, что выполняла бы `elemBy eq x`. Небольшое количество перегруженных функций (`elemIndex`, `elemIndices`, `isPrefixOf`, `isSuffixOf`) посчитали недостаточно важными для того, чтобы они имели "By"-варианты.

17.7 "generic"-операции

Приставка "`generic`" указывает на перегруженную функцию, которая является обобщенной версией функции `Prelude` . Например,

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

является обобщенной версией `length`.

"`generic`"-операциями являются следующие: `genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex` (обобщенная версия `!!`), `genericReplicate`.

17.8 Дополнительные "zip"-операции

Prelude обеспечивает `zip`, `zip3`, `unzip`, `unzip3`, `zipWith` и `zipWith3`. Библиотека List обеспечивает те же три операции для 4, 5, 6 и 7 аргументов.

17.9 Библиотека `List`

```
module List (
    elemIndex, elemIndices,
    find, findIndex, findIndices,
    nub, nubBy, delete, deleteBy, (\\), deleteFirstsBy,
    union, unionBy, intersect, intersectBy,
    intersperse, transpose, partition, group, groupBy,
    inits, tails, isPrefixOf, isSuffixOf,
    mapAccumL, mapAccumR,
    sort, sortBy, insert, insertBy, maximumBy, minimumBy,
    genericLength, genericTake, genericDrop,
    genericSplitAt, genericIndex, genericReplicate,
    zip4, zip5, zip6, zip7,
    zipWith4, zipWith5, zipWith6, zipWith7,
    unzip4, unzip5, unzip6, unzip7, unfoldr,
    -- ...и то, что экспортирует Prelude
    -- [](:), [],     -- Это встроенный синтаксис
    map, (++), concat, filter,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3
    ) where

import Maybe( listToMaybe )

infix 5 \\

elemIndex               :: Eq a => a -> [a] -> Maybe Int
elemIndex x             =  findIndex (x ==)

elemIndices             :: Eq a => a -> [a] -> [Int]
elemIndices x           =  findIndices (x ==)

find                    :: (a -> Bool) -> [a] -> Maybe a
find p                  =  listToMaybe . filter p

findIndex               :: (a -> Bool) -> [a] -> Maybe Int
findIndex p             =  listToMaybe . findIndices p

findIndices             :: (a -> Bool) -> [a] -> [Int]
findIndices p xs        =  [ i | (x,i) <- zip xs [0..], p x ]

nub                     :: Eq a => [a] -> [a]
nub                     =  nubBy (==)
```

```
nubBy                   :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq []             =  []
nubBy eq (x:xs)         =  x : nubBy eq (filter (\y -> not (eq x y)) xs)

delete                  :: Eq a => a -> [a] -> [a]
delete                  =  deleteBy (==)

deleteBy                :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy eq x []        = []
deleteBy eq x (y:ys)    = if x `eq` y then ys else y : deleteBy eq x ys

(\\)                    :: Eq a => [a] -> [a] -> [a]
(\\)                    =  foldl (flip delete)

deleteFirstsBy          :: (a -> a -> Bool) -> [a] -> [a] -> [a]
deleteFirstsBy eq       =  foldl (flip (deleteBy eq))

union                   :: Eq a => [a] -> [a] -> [a]
union                   =  unionBy (==)

unionBy                 :: (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys        =  xs ++ deleteFirstsBy eq (nubBy eq ys) xs

intersect               :: Eq a => [a] -> [a] -> [a]
intersect               =  intersectBy (==)

intersectBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy eq xs ys    =  [x | x <- xs, any (eq x) ys]

intersperse             :: a -> [a] -> [a]
intersperse sep []      =  []
intersperse sep [x]     =  [x]
intersperse sep (x:xs)  =  x : sep : intersperse sep xs

-- transpose является ленивой и в отношении строк, и в отношении столбцов,
--       и работает для непрямоугольных 'матриц'
-- Например, transpose [[1,2],[3,4,5],[]]  =  [[1,3],[2,4],[5]]
-- Обратите внимание, что [h | (h:t) <- xss] --- не то же самое, что
-- (map head xss) потому что первый отбрасывает пустые подсписки внутри xss
transpose               :: [[a]] -> [[a]]
transpose []            = []
transpose ([]     : xss) = transpose xss
transpose ((x:xs) : xss) = (x : [h | (h:t) <- xss]) :
                           transpose (xs : [t | (h:t) <- xss])

partition               :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs          =  (filter p xs, filter (not . p) xs)

-- group делит указанный в аргументе список на список списков одинаковых,
-- соседних элементов.  Например,
-- group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
group                   :: Eq a => [a] -> [[a]]
group                   =  groupBy (==)
```

```
groupBy                 :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy eq []           =  []
groupBy eq (x:xs)       =  (x:ys) : groupBy eq zs
                           where (ys,zs) = span (eq x) xs

-- inits xs возвращает список начальных сегментов xs, наиболее короткий ---
-- в начале списка.
-- Например, inits "abc" == [,"a","ab","abc"]
inits                   :: [a] -> [[a]]
inits []                =  [[]]
inits (x:xs)            =  [[]] ++ map (x:) (inits xs)

-- tails xs возвращает список всех конечных сегментов xs, наиболее длинный ---
-- в начале списка.
-- Например, tails "abc" == ["abc", "bc", "c", ""]
tails                   :: [a] -> [[a]]
tails []                =  [[]]
tails xxs@(_:xs)        =  xxs : tails xs

isPrefixOf               :: Eq a => [a] -> [a] -> Bool
isPrefixOf []     _      =  True
isPrefixOf _      []     =  False
isPrefixOf (x:xs) (y:ys) =  x == y && isPrefixOf xs ys

isSuffixOf              :: Eq a => [a] -> [a] -> Bool
isSuffixOf x y          =  reverse x `isPrefixOf` reverse y

mapAccumL               :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumL f s []        =  (s, [])
mapAccumL f s (x:xs)    =  (s'',y:ys)
                           where (s', y ) = f s x
                                 (s'',ys) = mapAccumL f s' xs

mapAccumR               :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR f s []        =  (s, [])
mapAccumR f s (x:xs)    =  (s'', y:ys)
                           where (s'',y ) = f s' x
                                 (s', ys) = mapAccumR f s xs

unfoldr                 :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b             = case f b of
                                Nothing    -> []
                                Just (a,b) -> a : unfoldr f b

sort                    :: (Ord a) => [a] -> [a]
sort                    =  sortBy compare

sortBy                  :: (a -> a -> Ordering) -> [a] -> [a]
sortBy cmp              =  foldr (insertBy cmp) []

insert                  :: (Ord a) => a -> [a] -> [a]
insert                  = insertBy compare
```

```
insertBy                :: (a -> a -> Ordering) -> a -> [a] -> [a]
insertBy cmp x []       =  [x]
insertBy cmp x ys@(y:ys')
                        =  case cmp x y of
                             GT -> y : insertBy cmp x ys'
                             _  -> x : ys

maximumBy               :: (a -> a -> Ordering) -> [a] -> a
maximumBy cmp []        =  error "List.maximumBy: пустой список"
maximumBy cmp xs        =  foldl1 max xs
                        where
                           max x y = case cmp x y of
                                        GT -> x
                                        _  -> y

minimumBy               :: (a -> a -> Ordering) -> [a] -> a
minimumBy cmp []        =  error "List.minimumBy: пустой список"
minimumBy cmp xs        =  foldl1 min xs
                        where
                           min x y = case cmp x y of
                                        GT -> y
                                        _  -> x

genericLength           :: (Integral a) => [b] -> a
genericLength []        =  0
genericLength (x:xs)    =  1 + genericLength xs

genericTake             :: (Integral a) => a -> [b] -> [b]
genericTake _ []        =  []
genericTake 0 _         =  []
genericTake n (x:xs)
   | n > 0              =  x : genericTake (n-1) xs
   | otherwise          =  error "List.genericTake: отрицательный аргумент"

genericDrop             :: (Integral a) => a -> [b] -> [b]
genericDrop 0 xs        =  xs
genericDrop _ []        =  []
genericDrop n (_:xs)
   | n > 0              =  genericDrop (n-1) xs
   | otherwise          =  error "List.genericDrop: отрицательный аргумент"

genericSplitAt          :: (Integral a) => a -> [b] -> ([b],[b])
genericSplitAt 0 xs     =  ([],xs)
genericSplitAt _ []     =  ([],[])
genericSplitAt n (x:xs)
   | n > 0              =  (x:xs',xs'')
   | otherwise          =  error "List.genericSplitAt: отрицательный аргумент"
      where (xs',xs'')  =  genericSplitAt (n-1) xs
```

```
genericIndex            :: (Integral a) => [b] -> a -> b
genericIndex (x:_)  0   =  x
genericIndex (_:xs) n
        | n > 0         =  genericIndex xs (n-1)
        | otherwise     =  error "List.genericIndex: отрицательный аргумент"
genericIndex _ _        =  error "List.genericIndex: слишком большой индекс"

genericReplicate        :: (Integral a) => a -> b -> [b]
genericReplicate n x    =  genericTake n (repeat x)

zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (,,,)

zip5                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip5                    =  zipWith5 (,,,,)

zip6                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] ->
                              [(a,b,c,d,e,f)]
zip6                    =  zipWith6 (,,,,,)

zip7                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] ->
                              [g] -> [(a,b,c,d,e,f,g)]
zip7                    =  zipWith7 (,,,,,,)

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 _ _ _ _ _ _ _ _ = []

unzip4                  :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip4                  =  foldr (\(a,b,c,d) ~(as,bs,cs,ds) ->
                                        (a:as,b:bs,c:cs,d:ds))
                                 ([],[],[],[])
```

```
unzip5                  :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip5                  =  foldr (\(a,b,c,d,e) ~(as,bs,cs,ds,es) ->
                                        (a:as,b:bs,c:cs,d:ds,e:es))
                                 ([],[],[],[],[])

unzip6                  :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip6                  =  foldr (\(a,b,c,d,e,f) ~(as,bs,cs,ds,es,fs) ->
                                        (a:as,b:bs,c:cs,d:ds,e:es,f:fs))
                                 ([],[],[],[],[],[])

unzip7          :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
unzip7          =  foldr (\(a,b,c,d,e,f,g) ~(as,bs,cs,ds,es,fs,gs) ->
                                (a:as,b:bs,c:cs,d:ds,e:es,f:fs,g:gs))
                         ([],[],[],[],[],[],[])
```

Глава 18

Утилиты Maybe

```
module Maybe(
    isJust, isNothing,
    fromJust, fromMaybe, listToMaybe, maybeToList,
    catMaybes, mapMaybe,
    -- ...и то, что экспортирует Prelude
    Maybe(Nothing, Just),
    maybe
  ) where

isJust, isNothing    :: Maybe a -> Bool
fromJust             :: Maybe a -> a
fromMaybe            :: a -> Maybe a -> a
listToMaybe          :: [a] -> Maybe a
maybeToList          :: Maybe a -> [a]
catMaybes            :: [Maybe a] -> [a]
mapMaybe             :: (a -> Maybe b) -> [a] -> [b]
```

Конструктор типа `Maybe` определен в `Prelude` следующим образом:

```
data Maybe a = Nothing | Just a
```

Назначение типа `Maybe` заключается в том, чтобы предоставить метод обработки неправильных или необязательных значений без завершения программы, что произошло бы, если бы использовалась функция `error`, и без использования функции `IOError` из монады `IO`, которая потребовала бы, чтобы выражение стало монадическим. Правильный результат инкапсулируется путем обертывания его в `Just`; неправильный результат возвращается в виде `Nothing`.

Другие операции над `Maybe` предусмотрены как часть монадических классов в Prelude.

18.1 Библиотека Maybe

```
module Maybe(
    isJust, isNothing,
    fromJust, fromMaybe, listToMaybe, maybeToList,
    catMaybes, mapMaybe,

    -- ... и то, что экспортирует Prelude
    Maybe(Nothing, Just),
    maybe
  ) where

isJust                  :: Maybe a -> Bool
isJust (Just a)         =  True
isJust Nothing          =  False

isNothing               :: Maybe a -> Bool
isNothing               =  not . isJust

fromJust                :: Maybe a -> a
fromJust (Just a)       =  a
fromJust Nothing        =  error "Maybe.fromJust: Nothing"

fromMaybe               :: a -> Maybe a -> a
fromMaybe d Nothing     =  d
fromMaybe d (Just a)    =  a

maybeToList             :: Maybe a -> [a]
maybeToList Nothing     =  []
maybeToList (Just a)    =  [a]

listToMaybe             :: [a] -> Maybe a
listToMaybe []          =  Nothing
listToMaybe (a:_)       =  Just a

catMaybes               :: [Maybe a] -> [a]
catMaybes ms            =  [ m | Just m <- ms ]

mapMaybe                :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f              =  catMaybes . map f
```

Глава 19

Утилиты работы с символами

```
module Char (
    isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
    isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum,
    digitToInt, intToDigit,
    toUpper, toLower,
    ord, chr,
    readLitChar, showLitChar, lexLitChar,

        -- ...и то, что экспортирует Prelude
    Char, String
    ) where

isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
 isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum :: Char -> Bool

toUpper, toLower        :: Char -> Char

digitToInt :: Char -> Int
intToDigit :: Int -> Char

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

lexLitChar  :: ReadS String
readLitChar :: ReadS Char
showLitChar :: Char -> ShowS
```

Эта библиотека предоставляет ограниченный набор операций над символами Unicode. Первые 128 элементов этого набора символов идентичны набору символов ASCII; следующие 128 элементов образуют остаток набора символов Latin 1. Этот модуль предлагает только ограниченное представление полного набора символов Unicode; полный набор атрибутов символов Unicode в этой библиотеке недоступен.

Символы Unicode можно разделить на пять общих категорий: непечатаемые символы, строчные алфавитные символы, остальные алфавитные символы, числовые цифры и остальные печатаемые символы. В Haskell любой алфавитный символ, который не является строчной буквой, рассматривается как заглавный символ (верхнего регистра) (Unicode на самом деле имеет три регистра: верхний, нижний и заглавный). Числовые цифры могут являться частью идентификаторов, но цифры вне диапазона ASCII не должны использоваться читателем для обозначения чисел.

Для каждого вида символов Unicode выполняются следующие предикаты, которые возвращают `True`:

Тип символов	Предикаты			
Строчные алфавитные символы	`isPrint`	`isAlphaNum`	`isAlpha`	`isLower`
Остальные алфавитные символы	`isPrint`	`isAlphaNum`	`isAlpha`	`isUpper`
Цифры	`isPrint`	`isAlphaNum`		
Остальные печатаемые символы	`isPrint`			
Непечатаемые символы				

Функции `isDigit`, `isOctDigit` и `isHexDigit` выбирают только символы ASCII. `intToDigit` и `digitToInt` осуществляют преобразование между одной цифрой `Char` и соответствующим `Int`. `digitToInt` завершается неуспешно, если ее аргумент не удовлетворяет условию `isHexDigit`, но она распознает шестнадцатиричные цифры как в верхнем, так и в нижнем регистрах (т.е. `'0' .. '9'`, `'a'.. 'f'`, `'A' .. 'F'`). `intToDigit` завершается неуспешно, если ее аргумент не находится в диапазоне `0.. 15`; она генерирует шестнадцатиричные цифры в нижнем регистре.

Функция `isSpace` распознает пробельные символы только в диапазоне Latin 1.

Функция `showLitChar` преобразует символ в строку, используя только печатаемые символы и соглашения исходного языка Haskell об эскейп-символах. Функция `lexLitChar` делает обратное, возвращая последовательность символов, которые кодируют символ. Функция `readLitChar` делает то же самое, но кроме того осуществляет преобразование к символу, который это кодирует. Например:

```
showLitChar '\n' s       =  "\\n" ++ s
lexLitChar  "\\nЗдравствуйте"   =  [("\\n", "Здравствуйте")]
readLitChar "\\nЗдравствуйте"   =  [('\n', "Здравствуйте")]
```

Функция `toUpper` преобразовывает букву в соответствующую заглавную букву, оставляя все остальные символы без изменений. Любая буква Unicode, которая имеет эквивалент в верхнем регистре, подвергается преобразованию. Аналогично, `toLower` преобразовывает букву в соответствующую строчную букву, оставляя все остальные символы без изменений.

Функции `ord` и `chr` являются функциями `fromEnum` и `toEnum`, ограниченными типом `Char`.

19.1 Библиотека Char

```
module Char (
    isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
    isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum,
    digitToInt, intToDigit,
    toUpper, toLower,
    ord, chr,
    readLitChar, showLitChar, lexLitChar,

        -- ...и то, что экспортирует Prelude
    Char, String
    ) where

import Array         -- Используется для таблицы имен символов.
import Numeric (readDec, readOct, lexDigits, readHex)
import UnicodePrims  -- Исходный код примитивных функций Unicode.

-- Операции проверки символов
isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
 isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum :: Char -> Bool

isAscii c               =  c < '\x80'

isLatin1 c              =  c <= '\xff'

isControl c             =  c < ' ' || c >= '\DEL' && c <= '\x9f'

isPrint                 =  primUnicodeIsPrint

isSpace c               =  c `elem` " \t\n\r\f\v\xA0"
        -- Распознаются только пробельные символы Latin-1

isUpper                 =  primUnicodeIsUpper  -- 'A'..'Z'

isLower                 =  primUnicodeIsLower  -- 'a'..'z'

isAlpha c               =  isUpper c || isLower c

isDigit c               =  c >= '0' && c <= '9'

isOctDigit c            =  c >= '0' && c <= '7'

isHexDigit c            =  isDigit c || c >= 'A' && c <= 'F' ||
                                        c >= 'a' && c <= 'f'

isAlphaNum              =  primUnicodeIsAlphaNum

-- Операции преобразования цифр
digitToInt :: Char -> Int
digitToInt c
  | isDigit c           =  fromEnum c - fromEnum '0'
  | c >= 'a' && c <= 'f' =  fromEnum c - fromEnum 'a' + 10
  | c >= 'A' && c <= 'F' =  fromEnum c - fromEnum 'A' + 10
  | otherwise           =  error "Char.digitToInt: не является цифрой"
```

```
intToDigit :: Int -> Char
intToDigit i
  | i >= 0  && i <=  9   =  toEnum (fromEnum '0' + i)
  | i >= 10 && i <= 15   =  toEnum (fromEnum 'a' + i - 10)
  | otherwise            =  error "Char.intToDigit: не является цифрой"

-- Операции изменения регистра букв
toUpper :: Char -> Char
toUpper =  primUnicodeToUpper

toLower :: Char -> Char
toLower =  primUnicodeToLower

-- Функции кодирования символов
ord  :: Char -> Int
ord  =  fromEnum

chr  :: Int  -> Char
chr  =  toEnum

-- Функции над текстом
readLitChar          :: ReadS Char
readLitChar ('\\':s) =  readEsc s
readLitChar (c:s)    =  [(c,s)]

readEsc          :: ReadS Char
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 _        = []

match                        :: (Eq a) => [a] -> [a] -> ([a],[a])
match (x:xs) (y:ys) | x == y  =  match xs ys
match xs     ys               =  (xs,ys)
```

```
showLitChar               :: Char -> ShowS
showLitChar c | c > '\DEL' =  showChar '\\' .
                              protectEsc isDigit (shows (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
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"]

lexLitChar          :: ReadS String
lexLitChar ('\\':s) =  map (prefix '\\') (lexEsc s)
        where
          lexEsc (c:s)     | c `elem` "abfnrtv\\\"'" = [([c],s)]
          lexEsc ('^':c:s) | c >= '@' && c <= '_'    = [(['^',c],s)]

          -- Числовые эскейп-символы
          lexEsc ('o':s)               = [prefix 'o' (span isOctDigit s)]
          lexEsc ('x':s)               = [prefix 'x' (span isHexDigit s)]
          lexEsc s@(d:_)   | isDigit d = [span isDigit s]

          -- Очень грубое приближение к \XYZ.
          lexEsc s@(c:_)   | isUpper c = [span isCharName s]
          lexEsc _                     = []

          isCharName c   = isUpper c || isDigit c
          prefix c (t,s) = (c:t, s)

lexLitChar (c:s)    =  [([c],s)]
lexLitChar ""       =  []
```

Глава 20

Утилиты работы с монадами

```
module Monad (
    MonadPlus(mzero, mplus),
    join, guard, when, unless, ap,
    msum,
    filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM,
    liftM, liftM2, liftM3, liftM4, liftM5,
    -- ...и то, что экспортирует Prelude
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    ) where

class  Monad m => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

join             :: Monad m => m (m a) -> m a
guard            :: MonadPlus m => Bool -> m ()
when             :: Monad m => Bool -> m () -> m ()
unless           :: Monad m => Bool -> m () -> m ()
ap               :: Monad m => m (a -> b) -> m a -> m b

mapAndUnzipM     :: Monad m => (a -> m (b,c)) -> [a] -> m ([b], [c])
zipWithM         :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM_        :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
foldM            :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
filterM          :: Monad m => (a -> m Bool) -> [a] -> m [a]

msum             :: MonadPlus m => [m a] -> m a
```

```
liftM           :: Monad m => (a -> b) -> (m a -> m b)
liftM2          :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3          :: Monad m => (a -> b -> c -> d) ->
                              (m a -> m b -> m c -> m d)
liftM4          :: Monad m => (a -> b -> c -> d -> e) ->
                              (m a -> m b -> m c -> m d -> m e)
liftM5          :: Monad m => (a -> b -> c -> d -> e -> f) ->
                              (m a -> m b -> m c -> m d -> m e -> m f)
```

Библиотека `Monad` определяет класс `MonadPlus` и обеспечивает некоторые полезные операции над монадами.

20.1 Соглашения об именах

Функции в этой библиотеке используют следующие соглашения об именах:

- Суффикс "`M`" всегда обозначает функцию в категории Клейсли (Kleisli): `m` добавляется к результатам функции (карринг по модулю) и больше нигде. Так, например,

```
filter  ::            (a ->   Bool) -> [a] ->   [a]
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

- Суффикс "`_`" меняет тип результата `(m a)` на `(m ())`. Таким образом (в `Prelude`):

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- Приставка "`m`" обобщает существующую функцию на монадическую форму. Таким образом, например:

```
sum  :: Num a       => [a]   -> a
msum :: MonadPlus m => [m a] -> m a
```

20.2 Класс `MonadPlus`

Класс `MonadPlus` определен следующим образом:

```
class  Monad m => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a
```

Методы класса `mzero` и `mplus` являются соответственно нулем и плюсом для монады.

Списки и тип `Maybe` являются экземплярами класса `MonadPlus`, таким образом:

```
instance  MonadPlus Maybe  where
    mzero                 = Nothing
    Nothing `mplus` ys    = ys
    xs      `mplus` ys    = xs

instance  MonadPlus []  where
    mzero = []
    mplus = (++)
```

20.3 Функции

Функция `join` является обычным оператором объединения монад. Он используется для того, чтобы убрать один уровень монадической структуры, проектируя его связанный аргумент во внешний уровень.

Функция `mapAndUnzipM` устанавливает соответствие (отображает) между своим первым аргументом и списком, возвращая результат в виде пары списков. Эта функция главным образом используется со сложными структурами данных или с монадой преобразований состояний.

Функция `zipWithM` обобщает `zipWith` на произвольные монады. Например, следующая функция выводит на экран файл, добавляя в начало каждой строки ее номер:

```
listFile :: String -> IO ()
listFile nm =
  do cts <- readFile nm
     zipWithM_ (\i line -> do putStr (show i); putStr ": "; putStrLn line)
               [1..]
               (lines cts)
```

Функция `foldM` аналогична `foldl`, за исключением того, что ее результат инкапсулируется в монаде. Обратите внимание, что `foldM` работает над перечисленными аргументами слева направо. При этом могла бы возникнуть проблема там, где `(>>)` и "сворачивающая функция" не являются коммутативными.

```
    foldM f a1 [x1, x2, ..., xm ]
==
    do
      a2 <- f a1 x1
      a3 <- f a2 x2
      ...
      f am xm
```

Если требуется вычисление справа налево, входной список следует обратить (поменять порядок элементов на обратный).

Функции `when` и `unless` обеспечивают условное выполнение монадических выражений. Например,

```
when debug (putStr "Отладка\n")
```

выведет строку `"Отладка\n"`, если булево значение `debug` равняется `True`, иначе не выведет ничего.

Монадическое повышение операторов повышает функцию до монады. Аргументы функции рассматриваются слева направо. Например,

```
liftM2 (+) [0,1] [0,2] = [0,2,1,3]
liftM2 (+) (Just 1) Nothing = Nothing
```

Во многих ситуациях операции `liftM` могут быть заменены на использование `ap`, которое повышает применение функции.

```
return f `ap` x1 `ap` ... `ap` xn
```

эквивалентно

```
liftMn f x1 x2 ... xn
```

20.4 Библиотека Monad

```
module Monad (
    MonadPlus(mzero, mplus),
    join, guard, when, unless, ap,
    msum,
    filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM,
    liftM, liftM2, liftM3, liftM4, liftM5,

    -- ...и то, что экспортирует Prelude
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    ) where

-- Определение класса MonadPlus

class  (Monad m) => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

-- Экземпляры класса MonadPlus

instance  MonadPlus Maybe  where
    mzero                 = Nothing

    Nothing `mplus` ys    =  ys
    xs      `mplus` ys    =  xs

instance  MonadPlus []  where
    mzero =  []
    mplus = (++)

-- Функции

msum             :: MonadPlus m => [m a] -> m a
msum xs          =  foldr mplus mzero xs

join             :: (Monad m) => m (m a) -> m a
join x           =  x >>= id

when             :: (Monad m) => Bool -> m () -> m ()
when p s         =  if p then s else return ()

unless           :: (Monad m) => Bool -> m () -> m ()
unless p s       =  when (not p) s

ap               :: (Monad m) => m (a -> b) -> m a -> m b
ap               =  liftM2 ($)

guard            :: MonadPlus m => Bool -> m ()
guard p          =  if p then return () else mzero

mapAndUnzipM     :: (Monad m) => (a -> m (b,c)) -> [a] -> m ([b], [c])
mapAndUnzipM f xs = sequence (map f xs) >>= return . unzip
```

```
zipWithM         :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys =  sequence (zipWith f xs ys)

zipWithM_         :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys =  sequence_ (zipWith f xs ys)

foldM            :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a []     =  return a
foldM f a (x:xs) =  f a x >>= \ y -> foldM f y xs

filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do { b  <- p x;
                        ys <- filterM p xs;
                        return (if b then (x:ys) else ys)
                      }

liftM            :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f          =  \a -> do { a' <- a; return (f a') }

liftM2           :: (Monad m) => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f         =  \a b -> do { a' <- a; b' <- b; return (f a' b') }

liftM3           :: (Monad m) => (a -> b -> c -> d) ->
                                 (m a -> m b -> m c -> m d)
liftM3 f         =  \a b c -> do { a' <- a; b' <- b; c' <- c;
                                   return (f a' b' c') }

liftM4           :: (Monad m) => (a -> b -> c -> d -> e) ->
                                 (m a -> m b -> m c -> m d -> m e)
liftM4 f         =  \a b c d -> do { a' <- a; b' <- b; c' <- c; d' <- d;
                                     return (f a' b' c' d') }

liftM5           :: (Monad m) => (a -> b -> c -> d -> e -> f) ->
                                 (m a -> m b -> m c -> m d -> m e -> m f)
liftM5 f         =  \a b c d e -> do { a' <- a; b' <- b; c' <- c; d' <- d;
                                       e' <- e; return (f a' b' c' d' e') }
```

Глава 21

Ввод - вывод

```
module IO (
    Handle, HandlePosn,
    IOMode(ReadMode,WriteMode,AppendMode,ReadWriteMode),
    BufferMode(NoBuffering,LineBuffering,BlockBuffering),
    SeekMode(AbsoluteSeek,RelativeSeek,SeekFromEnd),
    stdin, stdout, stderr,
    openFile, hClose, hFileSize, hIsEOF, isEOF,
    hSetBuffering, hGetBuffering, hFlush,
    hGetPosn, hSetPosn, hSeek,
    hWaitForInput, hReady, hGetChar, hGetLine, hLookAhead,
    hGetContents, hPutChar, hPutStr, hPutStrLn, hPrint,
    hIsOpen, hIsClosed, hIsReadable, hIsWritable, hIsSeekable,
    isAlreadyExistsError, isDoesNotExistError, isAlreadyInUseError,
    isFullError, isEOFError,
    isIllegalOperation, isPermissionError, isUserError,
    ioeGetErrorString, ioeGetHandle, ioeGetFileName,
    try, bracket, bracket_,
    -- ...и то, что экспортирует Prelude
    IO, FilePath, IOError, ioError, userError, catch, interact,
    putChar, putStr, putStrLn, print, getChar, getLine, getContents,
    readFile, writeFile, appendFile, readIO, readLn
    ) where

import Ix(Ix)

data Handle = ...                       -- зависит от реализации
instance Eq Handle where ...
instance Show Handle where ..           -- зависит от реализации

data HandlePosn = ...                   -- зависит от реализации
instance Eq HandlePosn where ...
instance Show HandlePosn where ---      -- зависит от реализации
```

```
data IOMode      =  ReadMode | WriteMode | AppendMode | ReadWriteMode
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
data BufferMode  =  NoBuffering | LineBuffering
                 |  BlockBuffering (Maybe Int)
                    deriving (Eq, Ord, Read, Show)
data SeekMode    =  AbsoluteSeek | RelativeSeek | SeekFromEnd
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)

stdin, stdout, stderr :: Handle

openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()
```

```
hFileSize             :: Handle -> IO Integer
hIsEOF                :: Handle -> IO Bool
isEOF                 :: IO Bool
isEOF                 =  hIsEOF stdin

hSetBuffering         :: Handle  -> BufferMode -> IO ()
hGetBuffering         :: Handle  -> IO BufferMode
hFlush                :: Handle -> IO ()
hGetPosn              :: Handle -> IO HandlePosn
hSetPosn              :: HandlePosn -> IO ()
hSeek                 :: Handle -> SeekMode -> Integer -> IO ()

hWaitForInput         :: Handle -> Int -> IO Bool
hReady                :: Handle -> IO Bool
hReady h              =  hWaitForInput h 0
hGetChar              :: Handle -> IO Char
hGetLine              :: Handle -> IO String
hLookAhead            :: Handle -> IO Char
hGetContents          :: Handle -> IO String
hPutChar              :: Handle -> Char -> IO ()
hPutStr               :: Handle -> String -> IO ()
hPutStrLn             :: Handle -> String -> IO ()
hPrint                :: Show a => Handle -> a -> IO ()

hIsOpen               :: Handle -> IO Bool
hIsClosed             :: Handle -> IO Bool
hIsReadable           :: Handle -> IO Bool
hIsWritable           :: Handle -> IO Bool
hIsSeekable           :: Handle -> IO Bool

isAlreadyExistsError  :: IOError -> Bool
isDoesNotExistError   :: IOError -> Bool
isAlreadyInUseError   :: IOError -> Bool
isFullError           :: IOError -> Bool
isEOFError            :: IOError -> Bool
isIllegalOperation    :: IOError -> Bool
isPermissionError     :: IOError -> Bool
isUserError           :: IOError -> Bool

ioeGetErrorString     :: IOError -> String
ioeGetHandle          :: IOError -> Maybe Handle
ioeGetFileName        :: IOError -> Maybe FilePath

try                   :: IO a -> IO (Either IOError a)
bracket               :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket_              :: IO a -> (a -> IO b) -> IO c -> IO c
```

Монадическая система ввода - вывода, используемая в Haskell, описана в описании языка Haskell. Общеупотребительные функции ввода - вывода, такие как `print`, являются частью стандартного начала (Prelude) и нет необходимости их явно

импортировать. Эта библиотека содержит более продвинутые средства ввода - вывода. Некоторые относящиеся к ним операции над файловыми системами содержатся в библиотеке `Directory`.

21.1 Ошибки ввода - вывода

Ошибки типа `IOError` используются монадой ввода - вывода. Это абстрактный тип; библиотека обеспечивает функции для опроса и конструирования значений в `IOError`:

- `isAlreadyExistsError` — операция завершилась неуспешно, потому что один из ее аргументов уже существует.
- `isDoesNotExistError` — операция завершилась неуспешно, потому что один из ее аргументов не существует.
- `isAlreadyInUseError` — операция завершилась неуспешно, потому что один из ее аргументов является однопользовательским ресурсом, который уже используется (например, открытие одного и того же файла дважды для записи может вызвать эту ошибку).
- `isFullError` — операция завершилась неуспешно, потому что устройство заполнено.
- `isEOFError` — операция завершилась неуспешно, потому что был достигнут конец файла.
- `isIllegalOperation` — операция невозможна.
- `isPermissionError` — операция завершилась неуспешно, потому что пользователь не имеет достаточно привилегий операционной системы на выполнение этой операции.
- `isUserError` — определенное программистом значение ошибки вызвано использованием `fail`.

Все эти функции возвращают значение типа `Bool`, которое равно `True`, если ее аргументом является соответствующий вид ошибки, и `False` иначе.

Любая функция, которая возвращает результат `IO`, может завершиться с ошибкой `isIllegalOperation`. Дополнительные ошибки, которые могут быть вызваны реализацией, перечислены после соответствующей операции. В некоторых случаях реализация не способна различить возможные причины ошибки. В этом случае она должна вернуть `isIllegalOperation`.

Имеются три дополнительные функции для того, чтобы получить информацию о значении ошибки, — это `ioeGetHandle`, которая возвращает `Just` *hdl*, если значение

ошибки относится к дескриптору *hdl*, и `Nothing` иначе; `ioeGetFileName`, которая возвращает `Just` *имя*, если значение ошибки относится к файлу *имя*, и `Nothing` иначе; и `ioeGetErrorString`, которая возвращает строку. Для "пользовательских" ошибок (которые вызваны использованием `fail`), строка, возвращенная `ioeGetErrorString`, является аргументом, который был передан в `fail`; для всех остальных ошибок строка зависит от реализации.

Функция `try` возвращает ошибку в вычислении, явно использующем тип `Either` .

Функция `bracket` охватывает обычный способ выделения памяти, вычисления и освобождения памяти, в котором шаг освобождения должен произойти даже в случае ошибки во время вычисления. Это аналогично try-catch-finally в Java.

21.2 Файлы и дескрипторы

Haskell взаимодействует с внешнем миром через абстрактную *файловую систему*. Эта файловая система представляет собой совокупность именованных *объектов файловой системы*, которые можно организовать в *каталоги (директории)* (см. `Directory`). В некоторых реализациях каталоги могут сами являться объектами файловой системы и могут являться элементами в других каталогах. Для простоты любой объект файловой системы, который не является каталогом, называется *файлом*, хотя на самом деле это может быть канал связи или любой другой объект, распознаваемый операционной системой. *Физические файлы* — это постоянные, упорядоченные файлы, которые обычно находятся на диске.

Имена файлов и каталогов являются значениями типа `String`, их точный смысл зависит от операционной системы. Файлы могут быть открыты; результатом открытия файла является дескриптор, который можно затем использовать для работы с содержимом этого файла.

Haskell определяет операции для чтения и записи символов соответственно из файла и в файл, они представлены значениями типа `Handle`. Каждое значение этого типа является *дескриптором*: записью, используемой системой поддержки выполнения программ Haskell для *управления* вводом - выводом объектов файловой системы. Дескриптор имеет по крайней мере следующие признаки:

- управляет он вводом или выводом или ими обоими;
- является он *открытым*, *закрытым* или *полузакрытым*;
- позволяет ли объект изменять текущую позицию ввода - вывода;
- отключена ли буферизация, а если включена — то какая: буферизация блоков или строк;
- буфер (его длина может быть равна нулю).

Большинство дескрипторов будет также иметь текущую позицию ввода - вывода, указывающую где произойдет следующая операция ввода или вывода. Дескриптор является *читаемым*, если он управляет только вводом или и вводом, и выводом; аналогично, он является *записываемым*, если он управляет только выводом или и вводом, и выводом. Дескриптор является *открытым*, когда он впервые назначается. Как только он закрывается, его больше нельзя использовать ни для ввода, ни для вывода, хотя реализация не может повторно использовать его память, пока остаются ссылки на него. Дескрипторы находятся в классах `Show` и `Eq`. Строка, полученная в результате вывода дескриптора, зависит от системы; она должна включать достаточно информации, чтобы идентифицировать дескриптор для отладки. В соответствии с `==` дескриптор равен только самому себе; не делается никаких попыток сравнить внутреннее состояние различных дескрипторов на равенство.

21.2.1 Стандартные дескрипторы

Во время инициализации программы назначаются три дескриптора. Первые два (`stdin` и `stdout`) управляют вводом или выводом из стандартных каналов ввода или вывода программы на Haskell соответственно. Третий (`stderr`) управляет выводом в стандартный канал вывода ошибок. Эти дескрипторы первоначально открыты.

21.2.2 Полузакрытые дескрипторы

Операция `hGetContents` *hdl* (раздел 21.9.4) помещает дескриптор *hdl* в промежуточное состояние — *полузакрытое*. В этом состоянии *hdl* фактически закрыт, но элементы читаются из *hdl* по требованию и накапливаются в специальном списке, возвращаемом `hGetContents` *hdl*.

Любая операция, которая завершается неуспешно из-за того, что дескриптор закрыт, также завершится неуспешно, если дескриптор полузакрыт. Единственное исключение — `hClose`. Полузакрытый дескриптор становится закрытым, если:

- по отношению к нему применен `hClose`;
- при чтении элемента из дескриптора возникла ошибка ввода - вывода;
- или как только все содержимое дескриптора будет прочитано.

Как только полузакрытый дескриптор становится закрытым, содержимое связанного с ним списка становится постоянным. Содержимое этого окончательного списка определено только частично: список будет содержать по крайней мере все элементы потока, которые были вычислены до того, как дескриптор стал закрытым.

Любые ошибки ввода - вывода, которые возникли в то время, когда дескриптор был полузакрыт, просто игнорируются.

21.2.3 Блокировка файлов

Реализации должны по возможности вызывать по крайней мере локально по отношению к процессу Haskell блокировку файлов со множественным чтением и единственной записью. То есть *может быть или много дескрипторов одного и того же файла, которые управляют вводом, или только один дескриптор файла, который управляет выводом.* Если какой-либо открытый или полузакрытый дескриптор управляет файлом для вывода, никакой новый дескриптор не может быть назначен для этого файла. Если какой-либо открытый или полузакрытый дескриптор управляет файлом для ввода, новые дескрипторы могут быть назначены, только если они не управляют выводом. Хотя совпадение двух файлов зависит от реализации, но они обычно должны быть одинаковыми, если они имеют одинаковый абсолютный путь и ни один из них не был переименован, например.

Предупреждение: операция `readFile` (раздел 7.1) хранит полузакрытый дескриптор файла до тех пор, пока все содержимое файла не будет считано. Из этого следует, что попытка записать в файл (используя `writeFile`, например), который было ранее открыт с помощью `readFile`, обычно завершается с ошибкой `isAlreadyInUseError`.

21.3 Открытие и закрытие файлов

21.3.1 Открытие файлов

Функция `openFile` *файл режим* назначает и возвращает новый, открытый дескриптор для управления файлом *файл*. Он управляет вводом, если *режим* равен `ReadMode`, выводом, если *режим* равен `WriteMode` или `AppendMode`, и вводом и выводом, если режим равен `ReadWriteMode`.

Если файл не существует и открывается для вывода, должен быть создан новый файл. Если *режим* равен `WriteMode` и файл уже существует, то файл должен быть усечен до нулевой длины. Некоторые операционные системы удаляют пустые файлы, поэтому нет гарантии, что файл будет существовать после `openFile` с *режимом* `WriteMode`, если он не будет впоследствии успешно записан. Дескриптор устанавливается в конец файла, если *режим* равен `AppendMode`, иначе — в начало файла (в этом случае его внутренняя позиция ввода - вывода равна 0). Начальный режим буферизации зависит от реализации.

Если `openFile` завершается неуспешно для файла, открываемого для вывода, файл тем не менее может быть создан, если он уже не существует.

Сообщения об ошибках: функция `openFile` может завершиться с ошибкой `isAlreadyInUseError`, если файл уже открыт и не может быть повторно открыт; `isDoesNotExistError`, если файл не существует, или `isPermissionError`, если пользователь не имеет прав на открытие файла.

21.3.2 Закрытие файлов

Функция `hClose` *hdl* делает дескриптор *hdl* закрытым. До того как завершится вычисление функции, если *hdl* является записываемым, то его буфер сбрасывается на диск как при использовании `hFlush`. Выполнение `hClose` на дескрипторе, который уже был закрыт, не влечет никаких действий; такое выполнение не является ошибкой. Все остальные операции на закрытом дескрипторе будут завершаться с ошибкой. Если `hClose` завершается неуспешно по какой-либо причине, все дальнейшие операции (кроме `hClose`) на дескрипторе будут тем не менее завершаться неуспешно, как будто *hdl* был успешно закрыт.

21.4 Определение размера файла

Для дескриптора *hdl*, который прикреплен к физическому файлу, `hFileSize` *hdl* возвращает размер этого файла в 8-битных байтах (≥ 0).

21.5 Обнаружение конца ввода

Для читаемого дескриптора *hdl* функция `hIsEOF` *hdl* возвращает `True`, если никакой дальнейший ввод не может быть получен из *hdl*; для дескриптора, прикрепленного к физическому файлу, это означает, что текущая позиция ввода - вывода равна длине файла. В противном случае функция возвращает `False`. Функция `isEOF` идентична описанной функции, за исключением того, что она работает только с `stdin`.

21.6 Операции буферизации

Поддерживаются три вида буферизации: буферизация строк, буферизация блоков или отсутствие буферизации. Эти режимы имеют следующие результаты. При выводе элементы записываются или *сбрасываются на диск* из внутреннего буфера в соответствии с режимом буферизации:

- **буферизация строк:** весь буфер сбрасывается на диск всякий раз, когда выводится символ новой строки, переполняется буфер, вызывается `hFlush` или закрывается дескриптор.
- **буферизация блоков:** весь буфер записывается всякий раз, когда он переполняется, вызывается `hFlush` или закрывается дескриптор.
- **отсутствие буферизации:** вывод записывается сразу и никогда не сохраняется в буфере.

В реализации содержимое буфера может сбрасываться на диск более часто, но не менее часто, чем определено выше. Буфер опусташается, как только он записывается.

Аналогично, в соответствии с режимом буферизации для дескриптора *hdl* выполняется ввод:

- **буферизация строк:** когда буфер для *hdl* не является пустым, следующий элемент получается из буфера; иначе, когда буфер пуст, символы считываются в буфер до тех пор, пока не встретится символ новой строки или буфер станет полон. Символы недоступны, пока не появится символ новой строки или буфер не станет полон.
- **буферизация блоков:** когда буфер для *hdl* становится пустым, следующий блок данных считывается в буфер.
- **отсутствие буферизации:** следующий элемент ввода считывается и возвращается. Операция `hLookAhead` (раздел 21.9.3) предполагает, что даже дескриптор в режиме отсутствия буферизации может потребовать буфер в один символ.

В большинстве реализаций для физических файлов обычно применяется буферизация блоков, а для терминалов обычно применяется буферизация строк.

Функция `hSetBuffering` *hdl режим* устанавливает режим буферизации для дескриптора *hdl* для последующих чтений и записей.

- Если *режим* равен `LineBuffering`, включается режим буферизации строк, если это возможно.
- Если *режим* равен `BlockBuffering` *размер*, включается режим буферизации блоков, если это возможно. Размер буфера равен n элементов, если *размер* равен `Just` n, а иначе зависит от реализации.
- Если *режим* равен `NoBuffering`, то режим буферизации отключается, если это возможно.

Если режим буферизации `BlockBuffering` или `LineBuffering` изменен на `NoBuffering`, тогда

- если *hdl* является записываемым, то буфер сбрасывается на диск как при `hFlush`;
- если *hdl* не является записываемым, то содержимое буфера игнорируется.

Сообщения об ошибках: функция `hSetBuffering` может завершиться с ошибкой `isPermissionError`, если дескриптор уже использовался для чтения или записи и реализация не позволяет изменить режим буферизации.

Функция `hGetBuffering` *hdl* возвращает текущий режим буферизации для *hdl*.

Режим буферизации по умолчанию при открытии дескриптора зависит от реализации и может зависеть от объекта файловой системы, к которому прикреплен дескриптор.

21.6.1 Сбрасывание буферов на диск

Функция `hFlush` *hdl* заставляет любые элементы, буферизированные для вывода в дескрипторе *hdl*, тотчас же передать операционной системе.

Сообщения об ошибках: функция `hFlush` может завершиться с ошибкой `isFullError`, если устройство заполнено; `isPermissionError`, если превышены пределы системных ресурсов. При этих обстоятельствах не определено, будут ли символы в буфере проигнорированы или останутся в буфере.

21.7 Позиционирование дескрипторов

21.7.1 Повторное использование позиции ввода - вывода

Функция `hGetPosn` *hdl* возвращает текущую позицию ввода - вывода *hdl* в виде значения абстрактного типа `HandlePosn`. Если вызов `hGetPosn` h возвращает позицию p, тогда функция `hSetPosn` p устанавливает позицию h в позицию, которую она содержала во время вызова `hGetPosn`.

Сообщения об ошибках: функция `hSetPosn` может завершиться с ошибкой `isPermissionError`, если были превышены пределы ресурсов системы.

21.7.2 Установка новой позиции

Функция `hSeek` *hdl режим i* устанавливает позицию дескриптора *hdl* в зависимости от *режима*. Если *режим* равен:

- `AbsoluteSeek`: позиция *hdl* устанавливается в i.
- `RelativeSeek`: позиция *hdl* смещается на i от текущей позиции.
- `SeekFromEnd`: позиция *hdl* смещается на i от конца файла.

Смещение задается в 8-битных байтах.

Если для *hdl* используется режим буферизации строк или блоков, то установка позиции, которая не находится в текущем буфере, приведет к тому, что сначала будут

записаны в устройство все элементы в выходном буфере, а затем входной буфер будет проигнорирован. Некоторые дескрипторы не могут использоваться для установки позиции (см. `hIsSeekable`) или могут поддерживать только подмножество возможных операций позиционирования (например, только возможность установить дескриптор в конец ленты или возможность сместить дескриптор в положительном направлении от начала файла или от текущей позиции). Невозможно установить отрицательную позицию ввода - вывода или для физического файла установить позицию ввода - вывода за пределы текущего конца файла.

Сообщения об ошибках: функция `hSeek` может завершиться с ошибкой `isPermissionError`, если были превышены пределы системных ресурсов.

21.8 Свойства дескрипторов

Функции `hIsOpen`, `hIsClosed`, `hIsReadable`, `hIsWritable` и `hIsSeekable` возвращают информацию о свойствах дескриптора. Каждый из возвращаемых результатов равен `True`, если дескриптор обладает указанным свойством, и `False` иначе.

21.9 Ввод и вывод текста

Здесь мы дадим определения стандартного набора операций ввода для чтения символов и строк из текстовых файлов, использующих дескрипторы. Многие из этих функций являются обобщениями функций Prelude. Ввод - вывод в Prelude как правило использует `stdin` и `stdout`; здесь дескрипторы явно определены с помощью операцией ввода - вывода.

21.9.1 Проверка ввода

Функция `hWaitForInput` *hdl t* ждет, пока не станет доступным ввод по дескриптору *hdl*. Она возвращает `True`, как только станет доступным ввод для *hdl*, или `False`, если ввод не доступен в пределах *t* миллисекунд.

Функция `hReady` *hdl* указывает, есть ли по крайней мере один элемент, доступный для ввода из дескриптора *hdl*.

Сообщения об ошибках: функции `hWaitForInput` и `hReady` завершаются с ошибкой `isEOFError`, если был достигнут конец файла.

21.9.2 Чтение ввода

Функция `hGetChar` *hdl* считывает символ из файла или канала, управляемого *hdl*.

Функция `hGetLine` *hdl* считывает строку из файла или канала, управляемого *hdl*. `getLine` из Prelude является краткой записью для `hGetLine stdin`.

Сообщения об ошибках: функция `hGetChar` завершается с ошибкой `isEOFError`, если был достигнут конец файла. Функция `hGetLine` завершается с ошибкой `isEOFError`, если конец файла был обнаружен при считывании *первого* символа строки. Если `hGetLine` обнаружит конец файла в любом другом месте при считывании строки, он будет обработан как признак конца строки и будет возвращена (неполная) строка.

21.9.3 Считывание вперед

Функция `hLookAhead` *hdl* возвращает следующий символ из дескриптора *hdl*, не удаляя его из входного буфера; она блокируется до тех пор, пока символ не станет доступен.

Сообщения об ошибках: функция `hLookAhead` может завершиться с ошибкой `isEOFError`, если был достигнут конец файла.

21.9.4 Считывание всего ввода

Функция `hGetContents` *hdl* возвращает список символов, соответствующих непрочитанной части канала или файла, управляемого *hdl*, который сделан полузакрытым.

Сообщения об ошибках: функция `hGetContents` может завершиться с ошибкой `isEOFError`, если был достигнут конец файла.

21.9.5 Вывод текста

Функция `hPutChar` *hdl c* записывает символ *c* в файл или канал, управляемый *hdl*. Символы могут быть буферизированы, если включена буферизация для *hdl*.

Функция `hPutStr` *hdl s* записывает строку *s* в файл или канал, управляемый *hdl*.

Функция `hPrint` *hdl t* записывает строковое представление *t*, полученное функцией `shows`, в файл или канал, управляемый *hdl*, и добавляет в конец символ новой строки.

Сообщения об ошибках: функции `hPutChar`, `hPutStr` и `hPrint` могут завершиться с ошибкой `isFullError`, если устройство заполнено, или `isPermissionError`, если превышены пределы других системных ресурсов.

21.10 Примеры

Рассмотрим некоторые простые примеры, иллюстрирующие ввод - вывод в Haskell.

21.10.1 Суммирование двух чисел

Эта программа считывает и суммирует два числа типа `Integer`.

```
import IO

main = do
         hSetBuffering stdout NoBuffering
         putStr   "Введите целое число: "
         x1 <- readNum
         putStr   "Введите другое целое число: "
         x2 <- readNum
         putStr  ("Их сумма равна " ++ show (x1+x2) ++ "\n")
       where readNum :: IO Integer
                -- Указание сигнатуры типа позволяет избежать
                -- исправления типов x1,x2 правилом по умолчанию
             readNum = readLn
```

21.10.2 Копирование файлов

Простая программа для создания копии файла, в которой все символы в нижнем регистре заменены на символы в верхнем регистре. Эта программа не разрешает копировать файл в самого себя. Эта версия использует ввод - вывод символьного уровня. Обратите внимание, что ровно два аргумента должны быть переданы программе.

```
import IO
import System
import Char( toUpper )

main = do
         [f1,f2] <- getArgs
         h1 <- openFile f1 ReadMode
         h2 <- openFile f2 WriteMode
         copyFile h1 h2
         hClose h1
         hClose h2

copyFile h1 h2 = do
                   eof <- hIsEOF h1
                   if eof then return () else
                      do
                        c <- hGetChar h1
                        hPutChar h2 (toUpper c)
                        copyFile h1 h2
```

Эквивалентная, но значительно более короткая версия, использующая ввод - вывод строк:

```
import System
import Char( toUpper )

main = do
          [f1,f2] <- getArgs
          s <- readFile f1
          writeFile f2 (map toUpper s)
```

21.11 Библиотека IO

```
module IO {- список экспорта пропущен -} where

-- Только обеспечивает реализацию не зависящих от системы
-- действий, которые экспортирует IO.
try            :: IO a -> IO (Either IOError a)
try f          =  catch (do r <- f
                            return (Right r))
                        (return . Left)

bracket        :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after m = do
        x  <- before
        rs <- try (m x)
        after x
        case rs of
           Right r -> return r
           Left  e -> ioError e

-- вариант приведенного выше, где средняя функция не требует x
bracket_        :: IO a -> (a -> IO b) -> IO c -> IO c
bracket_ before after m = do
         x  <- before
         rs <- try m
         after x
         case rs of
            Right r -> return r
            Left  e -> ioError e
```

Глава 22

Функции с каталогами

```
module Directory (
    Permissions( Permissions, readable, writable,
                              executable, searchable ),
    createDirectory, removeDirectory, removeFile,
    renameDirectory, renameFile, getDirectoryContents,
    getCurrentDirectory, setCurrentDirectory,
    doesFileExist, doesDirectoryExist,
    getPermissions, setPermissions,
    getModificationTime ) where

import Time ( ClockTime )

data Permissions = Permissions {
                         readable,   writable,
                         executable, searchable :: Bool
                   }

instance Eq   Permissions where ...
instance Ord  Permissions where ...
instance Read Permissions where ...
instance Show Permissions where ...

createDirectory         :: FilePath -> IO ()
removeDirectory         :: FilePath -> IO ()
removeFile              :: FilePath -> IO ()
renameDirectory         :: FilePath -> FilePath -> IO ()
renameFile              :: FilePath -> FilePath -> IO ()

getDirectoryContents    :: FilePath -> IO [FilePath]
getCurrentDirectory     :: IO FilePath
setCurrentDirectory     :: FilePath -> IO ()

doesFileExist           :: FilePath -> IO Bool
doesDirectoryExist      :: FilePath -> IO Bool
```

```
getPermissions          :: FilePath -> IO Permissions
setPermissions          :: FilePath -> Permissions -> IO ()

getModificationTime     :: FilePath -> IO ClockTime
```

Эти функции работают с каталогами (директориями) в файловой системе.

Любая операция `Directory` может вызвать `isIllegalOperation`, как описано в разделе 21.1; все другие допустимые ошибки описаны ниже. Обратите внимание, в частности, на то, что если реализация не поддерживает операцию, она должно вызвать `isIllegalOperation`. Каталог содержит ряд записей, каждая из которых является именованной ссылкой на объект файловой системы (файл, каталог и т.д.). Некоторые записи могут быть скрыты, недоступны или иметь некоторую административную функцию (например, "." или ".." в POSIX), но считается, что все такие записи формируют часть содержимого каталога. Записи в подкаталогах, однако, не считаются частями содержимого каталога. Тем не менее могут быть объекты файловой системы, отличные от файлов и каталогов, эта библиотека не различает физические файлы и другие объекты, не являющиеся каталогами. Все такие объекты поэтому должны обрабатываться как если бы они были файлами.

На каждый объект файловой системы ссылается *путь*. Обычно есть по крайней мере один абсолютный путь к каждому объекту файловой системы. В некоторых операционных системах возможно также иметь пути, которые относятся к текущему каталогу.

Функция `createDirectory` *dir* создает новый каталог *dir*, который первоначально пуст, или настолько пуст, насколько позволяет операционная система.

Сообщения об ошибках. Функция `createDirectory` может завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на создание каталога; `isAlreadyExistsError`, если каталог уже существует; `isDoesNotExistError`, если родительский каталог для нового каталога не существует.

Функция `removeDirectory` *dir* удаляет существующий каталог *dir*. Реализация может установить дополнительные ограничения, которые должны быть выполнены до того, как каталог будет удален (например, каталог должен быть пуст или не может использоваться другими процессами). Реализация не должна частично удалять каталог, если весь каталог не удален. Соответствующая реализация не обязана поддерживать удаление каталогов во всех ситуациях (например, удаление корневого каталога).

Функция `removeFile` *file* удаляет запись в каталоге для существующего файла *file*, где *file* сам не является каталогом. Реализация может установить дополнительные ограничения, которые должны быть выполнены до того, как файл будет удален (например, файл не может использоваться другими процессами).

Сообщения об ошибках. Функции `removeDirectory` и `removeFile` могут завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на удаление файла/каталога; `isDoesNotExistError`, если файл/каталог не существует.

Функция `renameDirectory` *old new* изменяет имя существующего каталога *old* на *new*. Если каталог *new* уже существует, он атомарно замещается каталогом *old*. Если каталог *new* не является ни каталогом *old*, ни псевдонимом каталогом *old*, он удаляется как с помощью `removeDirectory`. Соответствующая реализация не обязана поддерживать переименование каталогов во всех ситуациях (например, переименование существующего каталога или переименование между различными физическими устройствами), но ограничения должны быть задокументированы

Функция `renameFile` *old new* изменяет имя существующего объекта файловой системы *old* на *new*. Если объект *new* уже существует, он атомарно замещается объектом *old*. Также путь не может ссылаться на существующий каталог. Соответствующая реализация не обязана поддерживать переименование файлов во всех ситуациях (например, переименовывание между различными физическими устройствами), но ограничения должны быть задокументированы.

Сообщения об ошибках. Функции `renameDirectory` и `renameFile` могут завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на переименование файла/каталога или если один из двух аргументов `renameFile` является каталогом; `isDoesNotExistError`, если файл/каталог не существует.

Функция `getDirectoryContents` *dir* возвращает список *всех* записей в *dir*. Каждая запись в возвращаемом списке является именем относительно каталога *dir*, а не абсолютным путем.

Если операционная система имеет понятие о текущих каталогах, `getCurrentDirectory` возвращает абсолютный путь к текущему каталогу вызывающего процесса.

Сообщения об ошибках. Функции `getDirectoryContents` и `getCurrentDirectory` могут завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на доступ к директории; `isDoesNotExistError`, если каталог не существует.

Если операционная система имеет понятие о текущих каталогах, `setCurrentDirectory` *dir* меняет текущий каталог вызывающего процесса на *dir*.

Сообщения об ошибках. `setCurrentDirectory` может завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на смену каталога на указанный; `isDoesNotExistError`, если каталог не существует.

Тип `Permissions` используется для регистрации того, являются ли определенные операции допустимыми над файлом/каталогом. `getPermissions` и `setPermissions` соответственно получают и устанавливают эти права. Права применяются и к файлам, и каталогам. Для каталогов значение поля `executable` будет равно `False`, а для файлов значение поля `searchable` будет равно `False`. Обратите внимание, что каталоги могут

быть доступны для поиска, даже будучи не доступными для чтения, если имеется право на то, чтобы использовать их как часть пути, но не исследовать содержание каталога.

Обратите внимание, чтобы изменить некоторые, но не все права, следует использовать следующую конструкцию:

```
makeReadable f = do
                    p <- getPermissions f
                    setPermissions f (p {readable = True})
```

Операция `doesDirectoryExist` возвращает `True`, если указанный в аргументе файл существует и является каталогом, и `False` иначе. Операция `doesFileExist` возвращает `True`, если указанный в аргументе файл существует и не является каталогом, и `False` иначе.

Операция `getModificationTime` возвращает время системных часов, в которое файл/каталог был изменен в последний раз.

Сообщения об ошибках. `get(set)Permissions`, `doesFile(Directory)Exist` и `getModificationTime` могут завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на доступ к соответствующей информации; `isDoesNotExistError`, если файл/каталог не существует. Функция `setPermissions` может также завершиться с ошибкой `isPermissionError`, если пользователь не имеет прав на изменение прав для указанного файла или каталога; `isDoesNotExistError`, если файл/каталог не существует.

Глава 23

Системные функции

```
module System (
    ExitCode(ExitSuccess,ExitFailure),
    getArgs, getProgName, getEnv, system, exitWith, exitFailure
  ) where

data ExitCode = ExitSuccess | ExitFailure Int
                deriving (Eq, Ord, Read, Show)

getArgs                 :: IO [String]
getProgName             :: IO String
getEnv                  :: String -> IO String
system                  :: String -> IO ExitCode
exitWith                :: ExitCode -> IO a
exitFailure             :: IO a
```

Эта библиотека описывает взаимодействие программы с операционной системой.

Любая операция из `System` может вызвать `isIllegalOperation`, как описано в разделе 21.1; все остальные допустимые ошибки описаны ниже. Обратите внимание, в частности, на то, что если реализация не поддерживает операцию, она должна вызвать `isIllegalOperation`.

Тип `ExitCode` задает коды завершения, которые программа может вернуть. `ExitSuccess` указывает на успешное завершение, а `ExitFailure` *код* указывает на неуспешное завершение программы со значением *код*. Точная интерпретация *кода* зависит от операционной системы. В частности, некоторые значения *кода* могут быть запрещены (например, 0 в POSIX-системах).

Функция `getArgs` возвращает список аргументов командной строки (исключая имя программы). Функция `getProgName` возвращает имя программы, посредством которого

она была вызвана. Функция `getEnv` *var* возвращает значение переменной среды *var*. Если переменная *var* не определена, будет вызвано исключение `isDoesNotExistError`.

Функция `system` *cmd* возвращает код завершения, сгенерированный операционной системой в результате обработки команды *cmd*.

Функция `exitWith` *код* завершает программу, возвращая *код* процессу, вызвавшему программу. Перед завершением программы сначала будут закрыты все открытые или полузакрытые дескрипторы. Вызвавший процесс может интерпретировать код возврата как пожелает, но программа должна возвратить `ExitSuccess` для обозначения нормального завершения и `ExitFailure` *n* для обозначения ситуации, когда программа столкнулась с проблемой, из-за которой она не может восстановиться. Значение `exitFailure` равно `exitWith (ExitFailure` *exitfail*`)`, где *exitfail* зависит от реализации. `exitWith` игнорирует обработку ошибок в монаде ввода - вывода и не может быть перехвачена с помощью `catch`.

Если программа завершается в результате вызова функции `error` или потому что установлено, что ее значением является $\perp$, тогда она обрабатывается так же, как и функция `exitFailure`. Иначе, если какая-нибудь программа *p* завершается без явного вызова `exitWith`, то она обрабатывается так же, как и выражение

```
(p >> exitWith ExitSuccess) `catch` \ _ -> exitFailure
```

Глава 24

Дата и время

```
module Time (
        ClockTime,
        Month(January,February,March,April,May,June,
              July,August,September,October,November,December),
        Day(Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday),
        CalendarTime(CalendarTime, ctYear, ctMonth, ctDay, ctHour, ctMin,
                     ctSec, ctPicosec, ctWDay, ctYDay,
                     ctTZName, ctTZ, ctIsDST),
        TimeDiff(TimeDiff, tdYear, tdMonth, tdDay, tdHour,
                 tdMin, tdSec, tdPicosec),
        getClockTime, addToClockTime, diffClockTimes,
        toCalendarTime, toUTCTime, toClockTime,
        calendarTimeToString, formatCalendarTime ) where

import Ix(Ix)

data ClockTime = ...                    -- Зависит от реализации
instance Ord  ClockTime where ...
instance Eq   ClockTime where ...

data Month =  January   | February | March    | April
           |  May       | June     | July     | August
           |  September | October  | November | December
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)

data Day   =  Sunday | Monday  | Tuesday  | Wednesday | Thursday
           |  Friday | Saturday
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)
```

```
data CalendarTime = CalendarTime {
                ctYear                          :: Int,
                ctMonth                         :: Month,
                ctDay, ctHour, ctMin, ctSec     :: Int,
                ctPicosec                       :: Integer,
                ctWDay                          :: Day,
                ctYDay                          :: Int,
                ctTZName                        :: String,
                ctTZ                            :: Int,
                ctIsDST                         :: Bool
        } deriving (Eq, Ord, Read, Show)

data TimeDiff = TimeDiff {
                tdYear, tdMonth, tdDay, tdHour, tdMin, tdSec :: Int,
                tdPicosec                                    :: Integer
        } deriving (Eq, Ord, Read, Show)
```

```
-- Функции со временем
getClockTime         :: IO ClockTime

addToClockTime       :: TimeDiff  -> ClockTime -> ClockTime
diffClockTimes       :: ClockTime -> ClockTime -> TimeDiff

toCalendarTime       :: ClockTime    -> IO CalendarTime
toUTCTime            :: ClockTime    -> CalendarTime
toClockTime          :: CalendarTime -> ClockTime
calendarTimeToString :: CalendarTime -> String
formatCalendarTime   :: TimeLocale -> String -> CalendarTime -> String
```

Библиотека `Time` обеспечивает стандартные функциональные возможности работы со временем системных часов, включая информацию о часовых поясах. Она соответствует RFC 1129 в его использовании всеобщего скоординированного времени (UTC — Coordinated Universal Time).

`ClockTime` является абстрактным типом, который используется для системных внутренних часов (системного внутреннего генератора тактовых импульсов). Такты системных часов можно сравнивать непосредственно или после преобразования их в календарное время `CalendarTime` для ввода - вывода или других манипуляций. `CalendarTime` является удобным для использования и манипулирования представлением внутреннего типа `ClockTime`. Числовые поля имеют следующие диапазоны:

Значение	Диапазон	Комментарии
ctYear	-maxInt ... maxInt	Даты до григорианских являются ошибочными
ctDay	1 ... 31	
ctHour	0 ... 23	
ctMin	0 ... 59	
ctSec	0 ... 61	Допускает до двух скачков секунд
ctPicosec	0 ... $(10^{12}) - 1$	
ctYDay	0 ... 365	364 в невисокосные годы
ctTZ	-89999 ... 89999	Отклонение от UTC в секундах

Поле *ctTZName* — это название часового пояса. Значение поля *ctIsDST* равно `True`, если имеет место летнее время, и `False` иначе. Тип `TimeDiff` регистрирует различия между двумя тактами системных часов в удобной для использования форме.

Функция `getClockTime` возвращает текущее время в его внутреннем представлении. Выражение `addToClockTime` *d t* складывает разницу во времени *d* и время системных часов *t*, чтобы получить новое время системных часов. Разница *d* может быть положительной или отрицательной. Выражение `diffClockTimes` *t1 t2* возвращает разницу между двумя значениями системных часов *t1* и *t2*, как и `TimeDiff`.

Функция `toCalendarTime` *t* преобразовывает *t* в местное время в соответствии с часовым поясом и установками летнего времени. Из-за этой зависимости от местной среды `toCalendarTime` находится в монаде `IO`.

Функция `toUTCTime` *t* преобразовывает *t* в `CalendarTime` в стандартном формате UTC. `toClockTime` *l* преобразовывает *l* в соответствующее внутреннее значение `ClockTime`, игнорируя содержимое полей *ctWDay*, *ctYDay*, *ctTZName* и *ctIsDST*.

Функция `calendarTimeToString` форматирует значения календарного времени, используя национальные особенности и строку форматирования.

24.1 Библиотека Time

```
module Time (
        ClockTime,
        Month(January,February,March,April,May,June,
              July,August,September,October,November,December),
        Day(Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday),
        CalendarTime(CalendarTime, ctYear, ctMonth, ctDay, ctHour, ctMin,
                     ctSec, ctPicosec, ctWDay, ctYDay,
                     ctTZName, ctTZ, ctIsDST),
        TimeDiff(TimeDiff, tdYear, tdMonth, tdDay,
                 tdHour, tdMin, tdSec, tdPicosec),
        getClockTime, addToClockTime, diffClockTimes,
        toCalendarTime, toUTCTime, toClockTime,
        calendarTimeToString, formatCalendarTime ) where

import Ix(Ix)
import Locale(TimeLocale(..),defaultTimeLocale)
import Char ( intToDigit )

data ClockTime = ...                    - Зависит от реализации
instance Ord  ClockTime where ...
instance Eq   ClockTime where ...

data Month =  January   | February | March    | April
           |  May       | June     | July     | August
           |  September | October  | November | December
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)

data Day   =  Sunday | Monday  | Tuesday  | Wednesday | Thursday
           |  Friday | Saturday
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)

data CalendarTime = CalendarTime {
                ctYear                          :: Int,
                ctMonth                         :: Month,
                ctDay, ctHour, ctMin, ctSec     :: Int,
                ctPicosec                       :: Integer,
                ctWDay                          :: Day,
                ctYDay                          :: Int,
                ctTZName                        :: String,
                ctTZ                            :: Int,
                ctIsDST                         :: Bool
        } deriving (Eq, Ord, Read, Show)

data TimeDiff = TimeDiff {
                tdYear, tdMonth, tdDay, tdHour, tdMin, tdSec :: Int,
                tdPicosec                                    :: Integer
        } deriving (Eq, Ord, Read, Show)

getClockTime            :: IO ClockTime
getClockTime            = ...           -- Зависит от реализации
```

```
addToClockTime          :: TimeDiff     -> ClockTime -> ClockTime
addToClockTime td ct    =  ...          -- Зависит от реализации

diffClockTimes          :: ClockTime    -> ClockTime -> TimeDiff
diffClockTimes ct1 ct2  =  ...          -- Зависит от реализации

toCalendarTime          :: ClockTime    -> IO CalendarTime
toCalendarTime ct       =  ...          -- Зависит от реализации

toUTCTime               :: ClockTime    -> CalendarTime
toUTCTime ct            =  ...          -- Зависит от реализации

toClockTime             :: CalendarTime -> ClockTime
toClockTime cal         =  ...          -- Зависит от реализации

calendarTimeToString    :: CalendarTime -> String
calendarTimeToString    =  formatCalendarTime defaultTimeLocale "%c"
```

```
formatCalendarTime :: TimeLocale -> String -> CalendarTime -> String
formatCalendarTime l fmt ct@(CalendarTime year mon day hour min sec sdec
                                          wday yday tzname _ _) =
       doFmt fmt
  where doFmt ('%':c:cs) = decode c ++ doFmt cs
       doFmt (c:cs) = c : doFmt cs
       doFmt "" = ""

       to12 :: Int -> Int
       to12 h = let h' = h `mod` 12 in if h' == 0 then 12 else h'

       decode 'A' = fst (wDays l  !! fromEnum wday)
       decode 'a' = snd (wDays l  !! fromEnum wday)
       decode 'B' = fst (months l !! fromEnum mon)
       decode 'b' = snd (months l !! fromEnum mon)
       decode 'h' = snd (months l !! fromEnum mon)
       decode 'C' = show2 (year `quot` 100)
       decode 'c' = doFmt (dateTimeFmt l)
       decode 'D' = doFmt "%m/%d/%y"
       decode 'd' = show2 day
       decode 'e' = show2' day
       decode 'H' = show2 hour
       decode 'I' = show2 (to12 hour)
       decode 'j' = show3 yday
       decode 'k' = show2' hour
       decode 'l' = show2' (to12 hour)
       decode 'M' = show2 min
       decode 'm' = show2 (fromEnum mon+1)
       decode 'n' = "\n"
       decode 'p' = (if hour < 12 then fst else snd) (amPm l)
       decode 'R' = doFmt "%H:%M"
       decode 'r' = doFmt (time12Fmt l)
       decode 'T' = doFmt "%H:%M:%S"
       decode 't' = "\t"
       decode 'S' = show2 sec
       decode 's' = ...                -- Зависит от реализации
       decode 'U' = show2 ((yday + 7 - fromEnum wday) `div` 7)
       decode 'u' = show (let n = fromEnum wday in
                          if n == 0 then 7 else n)
       decode 'V' =
           let (week, days) =
                  (yday + 7 - if fromEnum wday > 0 then
                              fromEnum wday - 1 else 6) `divMod` 7
           in  show2 (if days >= 4 then
```

```
                        week+1
                    else if week == 0 then 53 else week)
        decode 'W' =
            show2 ((yday + 7 - if fromEnum wday > 0 then
                               fromEnum wday - 1 else 6) `div` 7)
        decode 'w' = show (fromEnum wday)
        decode 'X' = doFmt (timeFmt l)
        decode 'x' = doFmt (dateFmt l)
        decode 'Y' = show year
        decode 'y' = show2 (year `rem` 100)
        decode 'Z' = tzname
        decode '%' = "%"
        decode c   = [c]

show2, show2', show3 :: Int -> String
show2 x = [intToDigit (x `quot` 10), intToDigit (x `rem` 10)]

show2' x = if x < 10 then [ ' ', intToDigit x] else show2 x

show3 x = intToDigit (x `quot` 100) : show2 (x `rem` 100)
```

Глава 25

Локализация

```
module Locale(TimeLocale(..), defaultTimeLocale) where

data TimeLocale = TimeLocale {
        wDays  :: [(String, String)],   -- полные и сокращенные названия дней
                                        -- недели
        months :: [(String, String)],   -- полные и сокращенные названия
                                        -- месяцев
        amPm   :: (String, String),     -- символы AM/PM (a.m. --- до полудня,
                                        -- p.m. --- после полудня)
        dateTimeFmt, dateFmt,           -- строки форматирования
          timeFmt, time12Fmt :: String
        } deriving (Eq, Ord, Show)

defaultTimeLocale :: TimeLocale
```

Библиотека `Locale` предоставляет возможность адаптировать программу к национальным особенностям. В настоящее время она поддерживает только информацию о дате и времени, которая используется функцией `calendarTimeToString` из библиотеки `Time`.

25.1 Библиотека Locale

```
module Locale(TimeLocale(..), defaultTimeLocale) where

data TimeLocale = TimeLocale {
        wDays  :: [(String, String)],   -- полные и сокращенные названия дней
                                        -- недели
        months :: [(String, String)],   -- полные и сокращенные названия месяцев
        amPm   :: (String, String),     -- символы AM/PM (a.m. --- до полудня,
                                        -- p.m. --- после полудня)
        dateTimeFmt, dateFmt,           -- строки форматирования
          timeFmt, time12Fmt :: String
        } deriving (Eq, Ord, Show)

defaultTimeLocale :: TimeLocale
defaultTimeLocale =  TimeLocale {
        wDays  = [("Sunday",   "Sun"),  ("Monday",    "Mon"),
                  ("Tuesday",  "Tue"),  ("Wednesday", "Wed"),
                  ("Thursday", "Thu"),  ("Friday",    "Fri"),
                  ("Saturday", "Sat")],

        months = [("January",   "Jan"), ("February",  "Feb"),
                  ("March",     "Mar"), ("April",     "Apr"),
                  ("May",       "May"), ("June",      "Jun"),
                  ("July",      "Jul"), ("August",    "Aug"),
                  ("September", "Sep"), ("October",   "Oct"),
                  ("November",  "Nov"), ("December",  "Dec")],

        amPm = ("AM", "PM"),
        dateTimeFmt = "%a %b %e %H:%M:%S %Z %Y",
        dateFmt = "%m/%d/%y",
        timeFmt = "%H:%M:%S",
        time12Fmt = "%I:%M:%S %p"
        }
```

Глава 26

Время CPU

```
module CPUTime ( getCPUTime, cpuTimePrecision ) where

getCPUTime        :: IO Integer
cpuTimePrecision  :: Integer
```

Функция `getCPUTime` возвращает время CPU (центрального процессора) в пикосекундах (пикосекунда — одна триллионная доля секунды), используемое данной программой. Точность этого результата задается с помощью `cpuTimePrecision`. `cpuTimePrecision` — это наименьшая измеримая разница во времени CPU, которую реализация может зарегистрировать; она задается в виде целого числа пикосекунд.

Глава 27

Случайные числа

```
module Random (
        RandomGen(next, split, genRange),
        StdGen, mkStdGen,
        Random( random,   randomR,
                randoms,  randomRs,
                randomIO, randomRIO ),
        getStdRandom, getStdGen, setStdGen, newStdGen
  ) where

-------- Класс RandomGen ------------

class RandomGen g where
  genRange :: g -> (Int, Int)
  next     :: g -> (Int, g)
  split    :: g -> (g, g)

-------- Стандартный экземпляр класса RandomGen ------
data StdGen = ...       -- Абстрактный

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen

-------- Класс Random --------------
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)

   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]

   randomRIO :: (a,a) -> IO a
   randomIO  :: IO a
```

```
instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...

-------- Глобальный генератор случайных чисел --------
newStdGen    :: IO StdGen
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

Библиотека `Random` имеет дело со стандартной задачей генерации псевдослучайных чисел. Библиотека делает возможным генерацию повторных результатов, посредством старта с указанного начального случайного числа генератора, или получения различных результатов при каждом выполнении, посредством использования инициализируемого системой генератора или предоставления случайного значения из некоторого другого источника.

Библиотека делится на два уровня:

- Ядро *генератора случайных чисел* обеспечивает поставку битов. Класс `RandomGen` обеспечивает общий интерфейс к таким генераторам.
- Класс `Random` обеспечивает способ извлечь конкретные значения из генератора случайных чисел. Например, экземпляр `Float` класса `Random` позволяет генерировать случайные значения типа `Float`.

27.1 Класс `RandomGen` и генератор `StdGen`

Класс `RandomGen` обеспечивает общий интерфейс к генераторам случайных чисел.

```
class RandomGen g where
  genRange :: g -> (Int,Int)
  next     :: g  -> (Int, g)
  split    :: g -> (g, g)

  -- Метод по умолчанию
  genRange g = (minBound,maxBound)
```

- Операция `genRange` дает диапазон значений, возвращенный генератором.

  Требуется, чтобы:

  - Если $(a, b) = \texttt{genRange}\ g$, то $a < b$.
  - $\texttt{genRange}\ \bot \neq \bot$.

  Второе условие гарантирует, что `genRange` не может проверять свой аргумент, и следовательно значение, которое он возвращает, можно определить только посредством экземпляра класса `RandomGen`. Это, в свою очередь, позволяет реализации выполнять один вызов `genRange`, чтобы установить диапазон генератора, не касаясь того, что генератор, возвращенный (скажем) `next`, мог бы иметь диапазон, отличный от диапазона генератора, переданного `next`.

- Операция `next` возвращает `Int`, который равномерно распределен в диапазоне, возвращенном `genRange` (включая обе конечные точки), и новый генератор.

- Операция `split` позволяет получить два независимых генератора случайных чисел. Это очень полезно в функциональных программах (например, при передаче генератора случайных чисел в рекурсивных вызовах), но очень мало работы было сделано над статистически надежными реализациями `split` ([1,4] — единственные примеры, о которых мы знаем).

Библиотека `Random` предоставляет один экземпляр `RandomGen` — абстрактный тип данных `StdGen`:

```
data StdGen = ...     -- Абстрактный

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen
```

Экземпляр `StgGen` класса `RandomGen` имеет `genRange` по крайней мере 30 битов.

Результат повторного использования `next` должен быть по крайней мере таким же статистически надежным, как “Минимальный стандартный генератор случайных чисел”, описанный [2,3]. До тех пор, пока нам больше ничего неизвестно о реализациях `split`, все, чего мы требуем, — чтобы `split` производил генераторы, которые являются: (a) не идентичными и (b) независимо надежными в только что данном смысле.

Экземпляры `Show`/`Read` класса `StdGen` обеспечивают примитивный способ сохранить состояние генератора случайных чисел. Требуется, чтобы `read (show g) == g`.

Кроме того, `read` можно использовать для отображения произвольной строки (необязательно порожденной `show`) на значение типа `StdGen`. Вообще, экземпляр `read` класса `StdGen` обладает следующими свойствами:

- Он гарантирует, что завершится успешно для любой строки.

- Он гарантирует, что потребит только конечную часть строки.

- Различные строки аргумента, вероятно, приведут к различным результатам.

Функция `mkStdGen` обеспечивает альтернативный способ создания начального генератора, посредством отображения `Int` в генератор. Опять, различные аргументы, вероятно, должны породить различные генераторы.

Программисты могут, конечно, поставлять свои собственные экземпляры класса `RandomGen`.

Предупреждение о реализациях. Внешне привлекательная реализация `split` выглядит так:

```
instance RandomGen MyGen where
  ...
  split g = (g, variantOf g)
```

Здесь `split` возвращает сам `g` и новый генератор, полученный из `g`. Но теперь рассмотрим эти два предположительно независимых генератора:

```
g1 = snd (split g)
g2 = snd (split (fst (split g)))
```

Если `split` искренне поставляет независимые генераторы (как указано), то `g1` и `g2` должны быть независимы, но на самом деле они оба равны `variantOf g`. Реализации вышеупомянутого вида не отвечают требованиям спецификации.

27.2 Класс `Random`

Имея в собственном распоряжении источник поставки случайных чисел, класс `Random` позволяет программисту извлекать случайные значения разнообразных типов:

```
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)

   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]

   randomRIO :: (a,a) -> IO a
   randomIO :: IO a

     -- Методы по умолчанию
   randoms g = x : randoms g'
                  where
                    (x,g') = random g
   randomRs = ...аналогично...

   randomIO        = getStdRandom random
   randomRIO range = getStdRandom (randomR range)

instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...
```

- `randomR` принимает в качестве аргументов диапазон (lo, hi) и генератор случайных чисел g и возвращает случайное значение, равномерно распределенное в закрытом интервале $[lo, hi]$ вместе с новым генератором. Если $lo > hi$, то поведение функции в этом случае не определено. Для непрерывных типов нет требования, чтобы значения lo и hi были когда-либо воспроизведены в качестве случайных значений, но они могут быть использованы в этом качестве, это зависит от реализации и интервала.

- `random` выполняет то же самое, что и `randomR`, но не использует диапазон.

  - Для ограниченных типов (экземпляров класса `Bounded`, например, `Char`), диапазон обычно является целым типом.
  - Для дробных типов диапазон обычно является полузакрытым интервалом $[0, 1)$.
  - Для `Integer` диапазон является (произвольно) диапазон `Int`.

- Множественные версии, `randomRs` и `randoms`, порождают бесконечный список случайных значений и не возвращают новый генератор.

- Версии `IO`, `randomRIO` и `randomIO`, используют глобальный генератор случайных чисел (см. раздел 27.3).

27.3 Глобальный генератор случайных чисел

Есть единственный, неявный, глобальный генератор случайных чисел типа `StdGen`, который хранится в некоторой глобальной переменной, поддерживаемой монадой `IO`. Он инициализируется автоматически некоторым зависящим от системы способом, например, посредством использования времени дня или генератора случайных чисел в ядре Linux. Для того чтобы получить детерминированное поведение, используйте `setStdGen`.

```
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
newStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

- `getStdGen` и `setStdGen` соответственно возвращают и устанавливают глобальный генератор случайных чисел.
- `newStdGen` применяет `split` по отношению к текущему глобальному генератору случайных чисел, обновляет его одним из результатов и возвращает другой.
- `getStdRandom` использует указанную функцию, чтобы получить значение из текущего глобального генератора случайных чисел, и обновляет глобальный генератор с помощью нового генератора, возвращенного функцией. Например, `rollDice` возвращает случайное целое число между 1 и 6:

```
rollDice :: IO Int
rollDice = getStdRandom (randomR (1,6))
```

Ссылки

[1] FW Burton and RL Page, “Distributed random number generation”, Journal of Functional Programming, 2(2):203-212, April 1992.

Ф.У. Бертон и Р.Л. Пейдж, “Генерация распределенных случайных чисел”, Журнал “Функциональное программирование”, 2 (2):203-212, апрель 1992.

[2] SK Park, and KW Miller, “Random number generators - good ones are hard to find”, Comm ACM 31(10), Oct 1988, pp1192-1201.

С.К. Парк и К.У. Миллер, “Генераторы случайных чисел — трудно найти хорошие”, Comm ACM 31 (10), октябрь 1988, стр.1192-1201.

[3] DG Carta, “Two fast implementations of the minimal standard random number generator”, Comm ACM, 33(1), Jan 1990, pp87-88.

Д.Г. Карта, “Две быстрые реализации минимального стандартного генератора случайных чисел”, Comm ACM, 33 (1), январь 1990, стр.87-88.

[4] P Hellekalek, "Don't trust parallel Monte Carlo", ACM SIGSIM Simulation Digest 28(1), pp82-89, July 1998.

П. Хеллекалек, "Не доверяйте параллельному методу Монте-Карло ", ACM SIGSIM Simulation Digest 28 (1), стр.82-89, июль 1998.

Web-сайт `http://random.mat.sbg.ac.at/` является большим источником информации.

Литература

[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.

Дж. Бэкус. Может ли программирование быть освобождено от стиля фон Неймана? Функциональный стиль и его алгебра программ. *CACM*, 21(8):613–641, август 1978.

[2] H.B. Curry and R. Feys. *Combinatory Logic.* North-Holland Pub. Co., Amsterdam, 1958.

Х.Б. Карри и Р. Фейс. *Комбинаторная логика.* North-Holland Pub. Co., Амстердам, 1958.

[3] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, N.M., January 1982.

Л. Дамас и Р. Милнер. Основные схемы типов для функциональных программ. В *трудах 9-го симпозиума ACM по принципам языков программирования*, стр. 207–212, Альбукерк, N.M., январь 1982.

[4] K-F. Faxén A static semantics for Haskell *Journal of Functional Programming*, 2002.

К-Ф. Факсен. Статическая семантика для Haskell. *Журнал "Функциональное программирование"*, 2002.

[5] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

Дж.Р. Хиндли. Основные схемы типов объектов в комбинаторной логике. *Переговоры Американского математического общества*, 146:29–60, декабрь 1969.

[6] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical Report YALEU/DCS/RR-901, Yale University, May 1996.

П. Худак, Дж. Фасел и Дж. Петерсон. Краткий вводный курс в Haskell. Техническое описание YALEU/DCS/RR-901, Йельский университет, май 1996.

[7] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

Марк П. Джонс. Система классов конструкторов: перегрузка и неявный полиморфизм высокого порядка. *Журнал "Функциональное программирование"*, 5(1), январь 1995.

[8] Mark P. Jones. Typing Haskell in Haskell. *Haskell Workshop*, Paris, October 1999.

Марк П. Джонс. Типизация Haskell в Haskell. *Семинар по Haskell*, Париж, октябрь 1999.

[9] P. Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings*, pages 248–256, San Francisco, September 1981.

П. Пенфилд, младший. Главные значения и ветви в сложном APL. В *трудах конференции по APL '81*, стр. 248–256, Сан-Франциско, сентябрь 1981.

[10] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1987

С.Л. Пейтон Джонс. *Реализация языков функционального программирования.* Prentice-Hall International, Englewood Cliffs, Нью-Джерси, 1987.

[11] Unicode Consortium. *The Unicode Standard, Version 3.0.* Addison Wesley, Reading, MA, 2000.

Консорциум Unicode. *Стандарт Unicode, версия 3.0.* Addison Wesley, Reading, MA, 2000.

[12] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.

П. Уодлер и С. Блотт. Как сделать *специальный* полиморфизм менее *специальным*. В *трудах 16-го симпозиума ACM по принципам языков программирования*, стр. 60–76, Austin, Техас, январь 1989.

Index

Элементы предметного указателя, которые относятся к нетерминалам в синтаксисе Haskell, изображены в *курсивном* шрифте. Сущности кода изображены в шрифте `пишущей машинки`. Обычные элементы предметного указателя изображены в прямом шрифте.