Содержание

Вступление

Почему именно OCaml?

Выбор языка программирования играет важную роль. Он влияет на надежность, безопасность и эффективность программ, а также простоту чтения кода, его рефакторинга и расширения. Языки способны также влиять на образ мышления программиста и приемы проектирования программ, даже когда они не используются.

Мы решили написать эту книгу, потому что верим в важность выбора языка программирования и особенно в важность изучения OCaml. Каждый из нас имеет более чем 15-летний опыт использования OCaml в академической практике и профессиональной карьере, и к настоящему времени все мы уверены, что этот язык действительно является мощным инструментом для создания сложных программных систем. Наша цель – сделать этот инструмент более доступным для широкого круга программистов, рассказав о нем все, что необходимо для эффективного использования OCaml в повседневной практике.

Что делает OCaml особенным, так это то, что он занимает золотую середину в пространстве языков программирования. Он обладает уникальной комбинацией эффективности, выразительности и практичности, которую вы не найдете ни в каком другом языке. В значительной степени это обусловлено превосходным сочетанием некоторых ключевых особенностей OCaml, перечисленных ниже, которые продолжают развиваться уже более 40 лет.

- *Механизм сборки мусора*, обеспечивающий автоматическое управление памятью. В наши дни этой особенностью обладают многие современные языки высокого уровня.
- *Функции первого порядка (first-class functions)*[1], которые можно передавать как самые обычные значения. Аналогичной особенностью обладают, например, JavaScript, Common Lisp и C#.
- *Статический контроль типов* для увеличения производительности и снижения числа ошибок, проявляющихся во время выполнения, как в Java и C#.
- *Параметрический полиморфизм*, позволяющий конструировать абстракции, применимые к данным разных типов. Похожая особенность существует в Java и C# в виде поддержки обобщенных типов (generics) и в C++? в виде шаблонов.
- Великолепная поддержка *неизменяемых (immutable) данных*, обеспечивающих возможность создания программ, не вносящих подчас разрушительных

[1] Термин «first-class functions» не имеет устоявшегося перевода на русский язык. В Интернете часто можно встретить такие толкования, как «функции первого рода», «функции первого класса» и даже «первоклассные функции». Однако, чтобы не вносить путаницу и подчеркнуть, что речь идет не о типах, а о свойстве функциональных языков, по согласованию с практиками, имеющими многолетний опыт функционального программирования, было решено использовать толкование «функции первого порядка». – *Прим. перев.*

изменений в структуры данных. Эта особенность является традиционным свойством функциональных языков, таких как Scheme, и поддерживается крупными фреймворками распределенных вычислений, такими как Hadoop.

- Механизм *автоматического вывода типов*, позволяющий избежать необходимости кропотливо объявлять тип каждой переменной в программе и автоматически определяющий типы на основе используемых значений. Эта особенность в ограниченном виде присутствует в C# в виде поддержки неявно типизированных локальных переменных (implicitly typed local variables) и в C++11 в виде ключевого слова `auto`.
- *Алгебраические типы данных* и механизм *сопоставления с образцом*, дающие возможность манипулировать сложными структурами данных. Подобные особенности доступны также в Scala и F#.

Некоторые из вас уже знают и с удовольствием используют эти особенности, для других они станут настоящим открытием, но большинство из вас будет встречать *некоторые* из них и в других языках. Как будет демонстрироваться на протяжении всей книги, наличие всех этих особенностей в одном языке придает ему особую силу. Несмотря на их важность, эти идеи нашли лишь ограниченное применение в господствующих языках, но, даже проникнув в эти языки, например функции первого порядка в C# или параметрический полиморфизм в Java, они обычно принимают ограниченную и нелепую форму. Единственными языками, воплотившими эти идеи в полном объеме, являются *функциональные языки со статическим контролем типов*, такие как OCaml, F#, Haskell, Scala и Standard ML.

В ряду этих достойных языков OCaml стоит особняком, потому что ему удается обеспечить большую власть, оставаясь при этом весьма практичным языком. Компилятор Ocaml использует довольно простую стратегию компиляции и производит высокоэффективный код, не требующий сложных оптимизаций и применения дорогостоящей динамической компиляции (Just-in-Time, JIT). Это, наряду со строгой моделью вычислений в языке OCaml, делает поведение программ легко предсказуемым. Сборщик мусора использует *инкрементальный* алгоритм, исключая возможность появления длительных пауз на сборку мусора, и обеспечивает высокую точность, в том смысле, что надежно соберет все неиспользуемые данные (в отличие от сборщиков мусора, использующих алгоритмы на основе подсчета ссылок).

Все вышеперечисленное делает OCaml превосходным выбором для программистов, желающих освоить лучший язык программирования и в то же время продолжать решать практические задачи.

Краткая история развития

OCaml был написан группой разработчиков, в состав которой вошли Ксавье Леруа (Xavier Leroy), Джером Вуийон (Jérôme Vouillon), Дамиан Долигес (Damien Doligez) и Дидье Реми (Didier Rémy), в 1996 году в институте INRIA (Франция). Он явился результатом многолетних исследований языков семейства ML, разработка которых началась в 60-х годах и которые имеют глубокие связи с академическим сообществом.

Язык ML (Meta Language) был создан в 1972 году Робином Милнером (Robin Milner) (работавшим сначала в Стэнфордском, а потом в Кембриджском университете) как метаязык логики вычислимых функций (Logic for Computable Functions, LCF) для построения формальных доказательств. Со временем ML был преобразован в компилятор с целью упростить использование LCF на компьютерах с разной архитектурой и к началу 80-х постепенно превратился в полноценную, развитую систему.

Первая реализация языка Caml появилась в 1987 году. Она была создана Аскандером Суаресом (Ascánder Suárez) и продолжена Пьером Вейссом (Pierre Weis) и Мишелем Мони (Michel Mauny). В 1990 году Ксавье Леруа и Дамиан Долигес создали новую реализацию под названием Caml Light, выполненную в виде интерпретатора байт-кода с быстрым, последовательным сборщиком мусора. В течение следующих нескольких лет были разработаны практичные библиотеки, такие как инструменты управления синтаксисом, написанные Мишелем Мони, что способствовало продвижению Caml в академические и исследовательские круги.

Ксавье Леруа продолжил работу над языком Caml Light, дополнив его новыми возможностями, что привело к выходу в 1995 году версии Caml Special Light. Она обеспечивала существенно более высокую производительность за счет собственного компилятора, сопоставимую с такими языками, как C++. Система модулей, реализованная в духе Standard ML, также обладала мощными возможностями и упрощала создание крупномасштабных программных продуктов.

Современный язык OCaml появился в 1996 году, когда Дидье Реми и Джером Вуйион добавили в него мощную и изящную объектную систему. Примечательной особенностью этой объектной системы была поддержка многих типичных объектно-ориентированных идиом в сочетании со статическим контролем типов, тогда как поддержка тех же идиом в других языках, таких как C++ и Java, требовала дополнительных проверок во время выполнения. В 2000 году Жак Гарриг (Jacques Garrigue) добавил в OCaml еще несколько новых особенностей, таких как полиморфные методы (polymorphic methods), варианты, а также аргументы с метками (labeled arguments) и необязательные аргументы.

В последнее десятилетие отмечался значительный рост популярности OCaml и постоянное его совершенствование с целью поддержать растущий коммерческий и академический интерес. Модули первого порядка, обобщённые алгебраические типы данных (Generalized Algebraic Data Types, GADT) и динамическое связывание повысили гибкость языка. Появилась также поддержка аппаратных архитектур x86_64, ARM, PowerPC и Sparc, что превратило OCaml в отличный выбор для систем, где потребление ресурсов, предсказуемость и производительность являются важными факторами.

Стандартная библиотека Core

Одного языка недостаточно для практического его применения. Для разработки прикладных программ необходим также богатый набор библиотек. Ограниченный объем возможностей стандартной библиотеки OCaml, распространяемой вместе

с компилятором, часто является причиной разочарований при изучении OCaml. Именно поэтому стандартную библиотеку не следует рассматривать как универсальный инструмент – она создавалась только для поддержки компилятора и потому целенаправленно сохранялась небольшой, с маленьким количеством функций.

К счастью, в мире открытого программного обеспечения ничто не мешает созданию альтернативных библиотек в дополнение к стандартной, и именно они включаются в состав базового дистрибутива.

В недрах Jane Street – компании, использующей OCaml уже более десяти лет, – для внутреннего применения была разработана стандартная библиотека Core. Она изначально проектировалась с прицелом на звание универсальной стандартной библиотеки. Как и сам язык OCaml, библиотека Core создавалась с учетом требований к безошибочности, надежности и производительности.

Библиотека Core распространяется вместе с дополнительными синтаксическими расширениями, добавляющими новые возможности в язык OCaml, и другими библиотеками, такими как библиотека Async поддержки асинхронных сетевых взаимодействий, обеспечивающая возможность создания сложных распределенных систем только средствами библиотеки Core. Все эти библиотеки распространяются под свободной лицензией Apache 2, что дает возможность беспрепятственно использовать ее в любительских, академических и коммерческих разработках.

Платформа OCaml

Core – всеобъемлющая и эффективная стандартная библиотека, но, кроме нее, существует еще масса программного обеспечения на OCaml. С момента выхода первой версии OCaml в 1996 году сформировалось обширное сообщество программистов, которым было создано множество полезных библиотек и инструментов. Мы представим некоторые из этих библиотек при рассмотрении примеров в данной книге.

Установка этих сторонних библиотек выполняется легко и просто, с помощью инструмента управления пакетами, известного как OPAM. Мы подробнее расскажем об этом инструменте далее в книге, а сейчас просто отметим, что он составляет основу платформы, образуемую инструментами и библиотеками, наряду с компилятором OCaml, которая позволяет быстро и эффективно создавать прикладные программы.

Мы также воспользуемся этим инструментом для установки *utop* – интерфейса командной строки. Это современная интерактивная оболочка, поддерживающая историю команд, интерпретацию макросов, дополнение имен модулей и другие приятные мелочи, которые делают работу с языком намного удобнее. Мы будем использовать *utop* на протяжении всей книги для опробования примеров.

Об этой книге

Данная книга предназначена для программистов, имеющих некоторый опыт использования обычных языков программирования, причем необязательно функциональных. В зависимости от уровня подготовки многие понятия, рассматриваe-

мые здесь, окажутся для вас незнакомыми, включая названия таких традиционных инструментов функционального программирования, как функции высшего порядка и неизменяемые типы данных, а также некоторые особенности мощной системы типов OCaml и системы модулей.

Если вы уже знакомы с языком OCaml, эта книга может преподнести вам немало сюрпризов. Библиотека Core переопределяет многие стандартные пространства имен с целью наделить систему модулей OCaml новыми особенностями и сделать доступными по умолчанию некоторые мощные структуры данных. Старый код на OCaml все еще сможет взаимодействовать с библиотекой Core, но вам может потребоваться изменить его, чтобы получить максимальную выгоду. Весь новый код, который еще только предстоит написать, изначально будет использовать Core, и мы уверены, что время на изучение модели Core не будет потрачено впустую – она с успехом используется многими программами и помогает устранить препятствия, возникающие при создании сложных приложений на OCaml.

Код, использующий традиционную стандартную библиотеку компилятора, всегда будет существовать, однако для изучения ее особенностей существует масса ресурсов в Интернете. Книга «Практический OCaml», напротив, описывает приемы, используемые авторами в своей работе, для создания масштабируемых, надежных программных систем.

План книги

Данная книга делится на три части:

- первая часть описывает сам язык, начиная с общего обзора OCaml. Не отчаивайтесь, если при чтении этой части что-то останется для вас непонятным. Основная задача данной части – дать вам представление о различных аспектах языка, а идеи, представленные здесь, более подробно будут описаны в последующих главах.

  После начального знакомства с языком мы перейдем к более сложным особенностям, таким как модули, функторы и объекты, обзор которых займет достаточно много времени. Знание этих понятий играет важную роль. Эти идеи послужат вам хорошим фундаментом, даже при использовании других новейших языков, отличных от OCaml, многие из которых основаны на идеях, заложенных в ML;
- во второй части закладываются основы разработки типичных приложений на примерах использования инструментов и приемов программирования, от анализа аргументов командной строки до реализации асинхронных сетевых взаимодействий. Попутно вы увидите, как объединяются некоторые понятия, представленные в первой части, в действующие библиотеки и инструменты, сочетающие в себе различные возможности языка;
- в третьей части обсуждаются система времени выполнения OCaml и набор инструментов компилятора. Эта тема достаточно проста, если сравнивать с реализациями других языков (такими как виртуальная машина Java или .NET CLR). В этой части вы получите все знания, необходимые для соз-

дания высокопроизводительных систем и интерфейсов с библиотеками на языке C. Здесь мы также поговорим о приемах профилирования и отладки с применением таких инструментов, как GNU *gdb*.

Инструкции по установке

Далее мы будем использовать некоторые инструменты, созданные нами в процессе работы над этой книгой. Некоторые из них направлены на улучшение компилятора OCaml, а это означает, что вам нужно будет обновить свою среду разработки (использовать компилятор версии 4.01). Процесс установки протекает преимущественно в автоматическом режиме, при использовании диспетчера пакетов OPAM. Инструкции по установке и перечень устанавливаемых пакетов можно найти по адресу: https://github.com/realworldocaml/book/wiki/Installation-Instructions.

К моменту публикации книги библиотека Core не поддерживала операционную систему Windows, поэтому ее работоспособность может быть гарантирована только в Mac OS X, Linux, FreeBSD и OpenBSD. Если вы пользуетесь ОС Windows, обращайтесь на страницу с инструкциями по установке, указанную выше (возможно, когда вы будете читать эту книгу, что-то изменится), или установите Linux в виртуальную машину, чтобы следовать за примерами в книге, поскольку они стоят того.

Эта книга не является справочным пособием. Ее цель – познакомить вас с языком, библиотеками, инструментами и приемами программирования, которые помогут вам стать эффективным программистом на OCaml. Но она не может служить заменой документации с описанием API и справочных руководств по языку OCaml. Ссылки на документацию с описанием всех библиотек и инструментов, упоминаемых в книге, вы найдете по адресу: https://ocaml.janestreet.com/ocaml-core/latest/doc/.

Примеры программного кода

Все примеры в книге свободно доступны на условиях общественной лицензии. Вы можете копировать и использовать любые фрагменты примеров в своих разработках, без каких-либо ограничений и упоминания авторства.

Репозиторий с программным кодом доступен по адресу: https://github.com/realworldocaml/examples. Все фрагменты кода в книге снабжены заголовками, из которых вы сможете узнать имена файлов с исходным кодом, сценариями командной оболочки или вспомогательными данными.

Если вам покажется, что использование вами примеров нарушит условия, описанные выше, обращайтесь с вопросами к нам по адресу: *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (www.safaribooksonline.com) – это виртуальная библиотека, содержащая авторитетную информацию в виде книг и видеоматериалов, созданных ведущими специалистами в области технологий и бизнеса. Профессионалы

в области технологии, разработчики программного обеспечения, веб-дизайнеры, а также бизнесмены и творческие работники используют Safari Books Online как основной источник информации для проведения исследований, решения проблем, обучения и подготовки к сертификационным испытаниям.

Библиотека Safari Books Online предлагает широкий выбор продуктов и тарифов для организаций, правительственных учреждений и физических лиц. Подписчики имеют доступ к поисковой базе данных, содержащей информацию о тысячах книг, видеоматериалов и рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesly Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и десятков других. За подробной информацией о Safari Books Online обращайтесь по адресу: http://www.safaribooksonline.com/.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (в Соединенных Штатах Америки или в Канаде)
(707) 829-0515 (международный)
(707) 829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

http://oreil.ly/realworldOCaml.

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com.

Дополнительную информацию о книгах, обсуждения, конференции и новости вы найдете на веб-сайте издательства: http://www.oreilly.com.

Ищите нас на Facebook: http://facebook.com/oreilly.

Следуйте за нами в Твиттере: http://twitter.com/oreillymedia.

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Нам хотелось бы выразить слова благодарности всем, кто помогал улучшать книгу «Практический OCaml»:

- Лео Уайт (Leo White) участвовал в работе над текстом глав 11 и 12 и предоставил примеры для них;

- Джереми Яллоп (Jeremy Yallop) создал и описал библиотеку Ctypes, представленную в главе 19;
- Стивен Уикс (Stephen Weeks) занимался разработкой модульной архитектуры библиотеки Core, и его многочисленные примечания послужили основой для глав 20 и 21;
- Джереми Димино (Jeremie Dimino) является автором *utop* – интерактивного интерфейса командной строки, используемого на протяжении всей книги. Мы особенно благодарны ему за изменения, которые он внес, чтобы улучшить работу *utop* в контексте этой книги;
- члены сообщества пользователей OCaml прислали более 2400 комментариев к рукописи книги, выложенной в Интернете для обсуждения. Благодаря им было выявлено и устранено огромное число ошибок и неточностей.

Часть I

Основы языка

Первая часть охватывает основные понятия языка, которые необходимо знать, чтобы создавать программы на языке OCaml. Она начинается обзором использования интерактивной оболочки. Последующие главы содержат более подробное описание сведений, представленных в первой главе, включая детальное изложение подходов к императивному программированию на OCaml.

В последних нескольких главах будут представлены мощные средства абстракции, имеющиеся в языке OCaml. Сначала мы познакомимся с функторами и используем их для создания библиотеки поддержки интервалов, а затем исследуем особенности применения модулей для построения системы плагинов с поддержкой контроля типов. Язык OCaml поддерживает также объектно-ориентированный стиль программирования, и мы закончим первую часть двумя главами, охватывающими объектную систему: сначала мы покажем, как напрямую использовать объекты OCaml, а затем продемонстрируем особенности использования системы классов и расскажем о некоторых дополнительных возможностях, таких как наследование. В этих же главах мы обсудим проект простой объектно-ориентированной графической библиотеки.

Глава 1

Введение

В этой главе дается обзор языка OCaml посредством знакомства с серией небольших примеров, демонстрирующих основные особенности языка. Ее цель – дать вам представление о возможностях OCaml без глубокого погружения в каждую отдельно взятую тему.

На протяжении всей книги мы будем использовать библиотеку Core, более богатую возможностями замену стандартной библиотеки OCaml. Также мы будем использовать *utop*, интерактивную оболочку, которая позволяет вводить выражения и получать результаты в диалоговом режиме. *utop* – это более простая в использовании версия стандартной интерактивной оболочки OCaml (которую, кстати, можно запустить командой *ocaml*). Следующие далее инструкции предполагают использование именно инструмента *utop*.

Прежде чем продолжить, убедитесь, что у вас имеется действующая версия OCaml, чтобы вы могли опробовать примеры в процессе чтения данной главы.

OCaml как калькулятор

Первое, что необходимо предпринять, чтобы включить в работу библиотеку Core, – открыть `Core.Std`:

OCaml utop
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
$ utop

# open Core.Std;;
```

Эта инструкция открывает доступ к определениям в библиотеке Core, необходимым во многих примерах в этой и в последующих главах.

Теперь попробуем выполнять простейшие арифметические операции:

OCaml utop (part 1)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# 3 + 4;;
- : int = 7
# 8 / 3;;
- : int = 2
# 3.5 +. 6.;;
3
- : float = 9.5
# 30_000_000 / 300_000;;
```

```
- : int = 100
# sqrt 9.;;
- : float = 3.
```

По большому счету, многое из того, что можно наблюдать здесь, имеется в любых других языках программирования, тем не менее сделаем сразу несколько замечаний:

- чтобы сообщить интерактивной оболочке, что она должна вычислить выражение, следует завершить инструкцию двумя символами `;;`. Это – особенность самой интерактивной оболочки. В программах так делать не следует (хотя иногда полезно включить `;;`, чтобы получить более подробное сообщение об ошибке, явно обозначив конец объявления верхнего уровня);
- после вычисления выражения интерактивная оболочка сначала печатает тип результата, а затем сам результат;
- аргументы отделяются от имен вызываемых функций пробелами, а не скобками или запятыми, что делает OCaml больше похожим на командную оболочку UNIX, чем на традиционные языки программирования, такие как C или Java;
- OCaml позволяет вставлять символы подчеркивания в числовые литералы для улучшения читаемости. Имейте в виду, что подчеркивания могут находиться в любом месте числового литерала, а не только через каждые три разряда;
- OCaml различает типы `float` (тип представления вещественных чисел) и `int` (тип представления целых чисел). Литералы разных типов отличаются (`6.` вместо `6`), так же отличаются инфиксные операторы, предназначенные для обработки разных типов (`+.` вместо `+`), а кроме того, OCaml не выполняет автоматического приведения между этими типами. Возможно, это не очень удобно, но в этом есть и свои преимущества, поскольку данная особенность препятствует появлению некоторых видов ошибок, часто возникающих в других языках из-за различий между целочисленным и вещественным типами. Например, во многих языках выражение `1 / 3` вернет нуль, а выражение `1 / 3.0` вернет одну третью. OCaml требует явно указывать, какая операция должна быть выполнена.

Можно также создать переменную, чтобы присвоить имя результату выражения. Делается это с помощью ключевого слова `let`. Эта операция называется *let-связывание* (let binding):

OCaml utop (part 2)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
val y : int = 14
```

После создания новой переменной интерактивная оболочка сообщает ее имя (`x` или `y`), тип (`int`) и значение (`7` или `14`).

Отметьте, что не каждый идентификатор может играть роль имени переменной. Имена переменных не могут включать знаки пунктуации, кроме символа подчеркивания (_) и одиночной кавычки ('), и должны начинаться с буквы в нижнем регистре или с символа подчеркивания. То есть следующие имена являются допустимыми:

OCaml utop (part 3)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let x7 = 3 + 4;;
val x7 : int = 7
# let x_plus_y = x + y;;
val x_plus_y : int = 21
# let x' = x + 1;;
val x' : int = 8
# let _x' = x' + x';;
# _x';;
- : int = 16
```

Обратите внимание, что по умолчанию *utop* не заботится о выводе имен переменных, начинающихся с символа подчеркивания.

Следующие имена являются недопустимыми:

OCaml utop (part 4)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let Seven = 3 + 4;;
Characters 4-9:
Error: Unbound constructor Seven
(Ошибка: Несвязанный конструктор Seven)
# let 7x = 7;;
Characters 5-10:
Error: This expression should not be a function, the expected type is
int
(Ошибка: Это выражение не должно быть функцией, ожидается тип int)
# let x-plus-y = x + y;;
Characters 4-5:
Error: Parse error: [fun_binding] expected after [ipatt] (in [let_binding])
(Ошибка: Синтаксическая ошибка: [fun_binding] ожидается после [ipatt]
(в [let_binding]))
```

Эти сообщения об ошибках немного сбивают с толку, но они станут более понятны, когда вы поближе познакомитесь с языком.

Функции и автоматический вывод типов

Ключевое слово `let` можно также использовать для определения функций:

OCaml utop (part 5)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let square x = x * x ;;
val square : int -> int = <fun>
```

```
# square 2;;
- : int = 4
# square (square 2);;
- : int = 16
```

Функции в языке OCaml являются самыми обычными значениями, как любые другие, именно поэтому связывание имени переменной с функцией выполняется с помощью ключевого слова `let`, как если бы мы использовали `let` для связывания имени переменной с простым значением, например целым числом. Когда `let` применяется для определения функции, первый идентификатор, следующий за `let`, интерпретируется как имя функции, а каждый последующий – как отдельный аргумент функции. То есть `square` в данном примере – это имя функции, принимающей единственный аргумент.

Теперь, после создания такого интересного значения, как функция, типы, что выводит интерактивная оболочка, также начинают становиться все интереснее и интереснее. Здесь `int -> int` представляет тип функции. В данном случае он сообщает, что мы имеем дело с функцией, принимающей аргумент типа `int` и возвращающей значение типа `int`. Имеется также возможность определять функции, принимающие несколько аргументов. (Обратите внимание, что следующий пример не будет работать, если не открыть `Core.Std`, как предлагалось выше.)

OCaml utop (part 6)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let ratio x y =
    Float.of_int x /. Float.of_int y
  ;;
val ratio : int -> int -> float = <fun>
# ratio 4 7;;
- : float = 0.571428571429
```

Так получилось, что этот пример одновременно является нашим первым модулем. Конструкция `Float.of_int` здесь – это ссылка на функцию `of_int` в модуле `Float`. Такой прием несколько отличается от используемого в объектно-ориентированных языках, где форма записи с точкой обычно применяется для обращения к методу объекта. Отметьте, что имена модулей всегда начинаются с большой буквы.

Первое время форма представления типов функций с несколькими аргументами может казаться непривычной, но мы объясним, чем это обусловлено, когда доберемся до карринга функций в разделе «Функции с несколькими аргументами» в главе 2. А пока просто считайте стрелки простыми разделителями в списке типов аргументов, где последняя стрелка отделяет тип возвращаемого значения. То есть конструкция `int -> int -> float` описывает функцию, принимающую два аргумента типа `int` и возвращающую значение типа `float`.

Можно также написать функцию, принимающую другие функции в аргументах. Ниже приводится пример функции с тремя аргументами: функцией `test` и двумя целочисленными аргументами. Функция возвращает сумму целых чисел, прошедших проверку с помощью функции `test`:

OCaml utop (part 7)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let sum_if_true test first second =
    (if test first then first else 0)
    + (if test second then second else 0)
  ;;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

Если внимательнее взглянуть на сигнатуру, выведенную компилятором, можно заметить, что первый аргумент является функцией, принимающей целое число и возвращающей логическое значение, а остальные два аргумента – целые числа. Ниже приводится пример использования этой функции:

OCaml utop (part 8)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let even x =
    x mod 2 = 0 ;;
val even : int -> bool = <fun>
# sum_if_true even 3 4;;
- : int = 4
# sum_if_true even 2 4;;
- : int = 6
```

Обратите внимание, что в определении функции `even` мы дважды использовали знак равно (`=`): первый из них является частью `let`-привязки и отделяет определяемый объект от самого определения, а второй выполняет проверку на равенство, сравнивая результат выражения `x mod 2` с нулем. Это совершенно разные операции, несмотря на то что в коде они выглядят одинаково.

Автоматический вывод типов

По мере усложнения типов, которые будут встречаться нам на пути, у многих из вас появится вопрос: «Как OCaml распознает их, учитывая, что мы никак явно не обозначаем типы?»

OCaml определяет тип выражения, используя прием под названием *автоматический вывод типа* (type inference), когда тип выражения определяется (выводится) по имеющейся информации о типах компонентов, составляющих выражение.

Для примера рассмотрим процесс вывода типа функции `sum_if_true`:

1. В языке OCaml требуется, чтобы обе ветви инструкции `if` имели одинаковый тип, соответственно для выражения `if test first then first else 0` требуется, чтобы значение `first` имело тот же тип, что и число `0`, то есть значение `first` должно иметь тип `int`. Аналогично из выражения `if test second then second else 0` можно заключить, что значение `second` имеет тип `int`.
2. Функции `test` передается значение `first` в качестве аргумента. Так как `first` имеет тип `int`, тип входного аргумента функции `test` также должен иметь тип `int`.
3. `test first` используется как условие в инструкции `if`, соответственно, `test` должна возвращать значение типа `bool`.

4. Из того факта, что + возвращает значение типа int, следует, что sum_if_true так же должна возвращать значение типа int.

Эта цепочка рассуждений закрепляет типы за всеми переменными и определяет общий тип функции sum_if_true.

Со временем вы получите более полное понимание, как действует механизм автоматического вывода типов в OCaml, что позволит вам проще тянуть нить рассуждений через свои программы. Впрочем, вы можете упростить себе эту задачу, добавляя явные аннотации типов. Эти аннотации не влияют на поведение программ на языке OCaml, но они могут играть роль дополнительного документирующего фактора, а также отлавливать непредвиденные изменения типов. Они могут также пригодиться в выяснении причин, вызывающих ошибки компиляции.

Ниже приводится аннотированная версия функции sum_if_true:

OCaml utop (part 9)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let sum_if_true (test : int -> bool) (x : int) (y : int) : int =
    (if test x then x else 0)
    + (if test y then y else 0)
  ;;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

Здесь каждый аргумент функции сопровождается аннотацией типа, а заключительная аннотация сообщает тип значения, возвращаемого функцией. Такие аннотации типов можно включать в любые выражения на языке OCaml.

Автоматический вывод обобщенных типов

Иногда информации оказывается недостаточно, чтобы точно определить конкретный тип некоторого значения. Взгляните на следующую функцию.

OCaml utop (part 10)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let first_if_true test x y =
    if test x then x else y
  ;;
val first_if_true : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

Функция first_if_true принимает в виде аргументов функцию test и два значения, x и y. Если условие if test x выполняется, она возвращает x, иначе возвращается y. Сможете ли вы определить тип first_if_true? Здесь нет никаких очевидных подсказок, таких как арифметические операторы или литералы, на основе которых можно было бы вывести типы x и y. Исходя из такого объявления, можно заключить, что функция first_if_true способна принимать значения любых типов.

И действительно, если взглянуть на тип, который вернула интерактивная оболочка, можно увидеть, что вместо конкретного типа OCaml ввел *переменный тип* (type variable) 'a, указывающий, что выражение имеет обобщенный тип. (Тот факт, что это переменный тип, отмечает начальная одиночная кавычка.) В частно-

сти, аргумент test имеет тип ('a -> bool), означающий, что test является функцией с одним аргументом, возвращающей значение типа bool и принимающей аргумент любого типа 'a. Но независимо от того, каким будет тип 'a, он должен совпадать с типами двух других аргументов, x и y, а также с типом возвращаемого значения функции first_if_true. Такого рода обобщенность называется *параметрическим полиморфизмом* (parametric polymorphism), потому что суть заключается в параметризации типа с помощью переменной типа. Это очень похоже на обобщенные типы (generics) в C# и Java.

Обобщенность функции first_if_true позволяет писать, к примеру, такой код:

OCaml utop (part 11)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let long_string s = String.length s > 6;;
val long_string : string -> bool = <fun>
# first_if_true long_string "short" "looooooong";;
- : string = "looooooong"
```

Или такой:

OCaml utop (part 12)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# let big_number x = x > 3;;
val big_number : int -> bool = <fun>
# first_if_true big_number 4 3;;
- : int = 4
```

Здесь long_string и big_number – это функции, и каждая из них пересылается в вызов функции first_if_true вместе с двумя аргументами соответствующего типа (строки в первом примере и целые числа во втором). Но мы не сможем смешать два разных конкретных типа для 'a в одном вызове first_if_true:

OCaml utop (part 13)
https://github.com/realworldocaml/examples/blob/v1/code/guided-tour/main.topscript

```
# first_if_true big_number "short" "looooooong";;
Characters 25-32:
Error: This expression has type string but an expression was expected of type
         int
(Ошибка: Это выражение имеет тип string, тогда как ожидалось выражение типа
         int)
```

В данном примере функция big_number требует, чтобы параметр 'a конкретизировался как int, тогда как аргументы "short" и "looooooong" требуют, чтобы параметр 'a конкретизировался как string, что невозможно получить одновременно.

Ошибки и исключения

В OCaml (как и в любом другом компилирующем языке) ошибки времени компиляции и времени выполнения имеют существенные отличия. Как показывает практика разработки программного обеспечения, чем раньше обнаруживается ошибка в процессе разработки, тем лучше, соответственно, этап компиляции является самым лучшим в этом смысле.