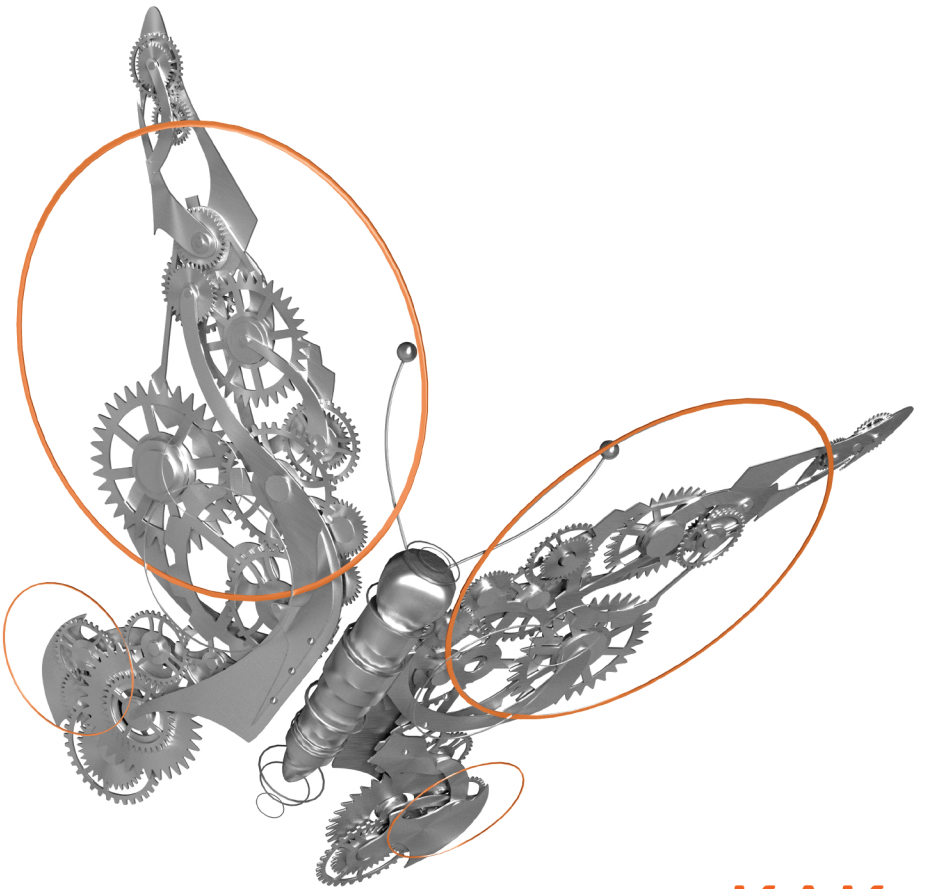
{Скандалы, интриги, расследования.
Что скрывает JavaScript}



КАК УСТРОЕН JavaScript

Дуглас Крокфорд

How JavaScript Works

Douglas Crockford

virgule solidus

КАК УСТРОЕН JavaScript

Дуглас Крокфорд

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

Дуглас Крокфорд

Как устроен JavaScript

Серия «Для профессионалов»

Перевел с английского *Н. Вильчинский*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Е. Павлович*
Верстка	*Г. Блинов*

ББК 32.988.02-018
УДК 004.738.5

Крокфорд Дуглас

К83 Как устроен JavaScript. — СПб.: Питер, 2019. — 304 с. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1260-9

Большинство языков программирования выросли из древней парадигмы, порожденной еще во времена Фортрана. Гуру JavaScript Дуглас Крокфорд выкорчевывает эти засохшие корни, позволяя нам задуматься над будущим программирования, перейдя на новый уровень понимания требований к Следующему Языку (The Next Language).

Автор начинает с основ: имен, чисел, логических значений, символов и другой базовой информации. Вы узнаете не только о проблемах и трудностях работы с типами в JavaScript, но и о том, как их можно обойти. Затем вы приступите к знакомству со структурами данных и функции, чтобы разобраться с механизмами, лежащими в их основе, и научитесь использовать функции высшего порядка и объектно-ориентированный стиль программирования без классов.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1949815009 англ.
ISBN 978-5-4461-1260-9

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.04.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1500. Заказ 0000.

Список глав

Капли дождя на розах и усы у котят.

Не Мария Августа фон Трапп
(Maria Augusta von Trapp). Не «Звуки музыки»

```
[
    {"number": 0, "chapter": "Сначала прочитайте меня!"},
    {"number": 1, "chapter": "Как работают имена"},
    {"number": 2, "chapter": "Как работают числа"},
    {"number": 3, "chapter": "Как работают большие целые числа"},
    {"number": 4, "chapter": "Как работают большие числа с плавающей точкой"},
    {"number": 5, "chapter": "Как работают большие рациональные числа"},
    {"number": 6, "chapter": "Как работают булевы значения"},
    {"number": 7, "chapter": "Как работают массивы"},
    {"number": 8, "chapter": "Как работают объекты"},
    {"number": 9, "chapter": "Как работают строки"},
    {"number": 10, "chapter": "Как работают ничтожно малые значения"},
    {"number": 11, "chapter": "Как работают инструкции"},
    {"number": 12, "chapter": "Как работают функции"},
    {"number": 13, "chapter": "Как работают генераторы"},
    {"number": 14, "chapter": "Как работают исключения"},
    {"number": 15, "chapter": "Как работают программы"},
    {"number": 16, "chapter": "Как работает this"},
    {"number": 17, "chapter": "Как работает код без классов"},
    {"number": 18, "chapter": "Как работают концевые вызовы"},
    {"number": 19, "chapter": "Как работает чистота"},
    {"number": 20, "chapter": "Как работает событийное программирование"},
    {"number": 21, "chapter": "Как работает Date"},
    {"number": 22, "chapter": "Как работает JSON"},
    {"number": 23, "chapter": "Как работает тестирование"},
    {"number": 24, "chapter": "Как работает оптимизация"},
    {"number": 25, "chapter": "Как работает транспиляция"},
    {"number": 26, "chapter": "Как работает разбиение на лексемы"},
    {"number": 27, "chapter": "Как работает парсер"},
    {"number": 28, "chapter": "Как работает генерация кода"},
    {"number": 29, "chapter": "Как работает среда выполнения"},
    {"number": 30, "chapter": "Как работают нелепости, или Что такое Wat!"},
    {"number": 31, "chapter": "Как устроена эта книга"}
]
```

Глава 0

Сначала прочитайте меня!

○ ○ ○ ○ ○

> Некоторые образы кажутся таинственными и не оставляют никаких сомнений в том, что являются результатом последовательности случайных событий, как будто созданы пресловутой обезьяной за пишущей машинкой.
>
> *Джордж Марсалья (George Marsaglia)*

Особой красотой JavaScript не блещет, но он работает.

Эта книга написана для людей, имеющих определенный опыт работы с JavaScript и желающих приобрести более четкое и глубокое понимание того, как этот язык работает и как добиться от него наибольшей отдачи. Она также подойдет опытным программистам, желающим освоить еще один язык.

Издание не для начинающих. Надеюсь, что когда-нибудь напишу книгу и для них. Но эта им не подойдет. Ее не назовешь легким чтивом. Беглый просмотр вам ничего не даст.

Здесь не рассматриваются механизмы обработки кода JavaScript или виртуальные машины. Книга — о самом языке и о том, что должен знать каждый программист. В ней я попробую сделать радикальную переоценку JavaScript, того, как он работает, как его можно усовершенствовать и как лучше использовать. Речь идет о том, как думать о JavaScript и как думать в JavaScript. Я планирую притвориться, что текущая версия языка — единственная, и не собираюсь тратить ваше время на демонстрацию того, как все работает в ES1, ES3 или ES5. Это не имеет никакого значения. Основное внимание будет уделено тому, как JavaScript работает для нас именно сейчас.

Эта книга не исчерпывающее руководство. В ней без какого-либо специального упоминания будут проигнорированы довольно большие и сложные части языка. Если не упомянута ваша самая любимая функция, то это, скорее всего, потому, что она не представляет ни малейшей ценности. Синтаксис также не будет удостоен особого внимания. Предполагается, что вы уже знаете, как написать инструкцию `if`. Если же вам такие тонкости неизвестны, обратитесь к информационному ресурсу JSLint по адресу jslint.com.

Некоторым весьма полезным составляющим языка, например большинству методов в базовых прототипах, также будет отведено не слишком много времени. Для их изучения есть отличные справочные материалы в Интернете. Мой любимый ресурс — Mozilla Foundation — находится по адресу developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

Важная цель при разработке языка программирования — сделать так, чтобы он был понятен, логичен и хорошо сформулирован, не приводил к возникновению странных тупиковых ситуаций. Но JavaScript даже близко не подвели к достижению этой цели. С каждым выпуском языка его странности растут как снежный ком, неизменно усугубляя ситуацию. В нем появляется множество тупиковых и критических проблем. В книге рассматриваются только некоторые из этих странностей, просто чтобы показать наличие подобных безобразий. Держитесь подальше от всего, что приводит к таким тупиковым и крайним ситуациям. Не углубляйтесь в этот мрак. Оставайтесь в той части языка, где все просто и понятно. Там есть все, что вам нужно для написания хороших программ.

Десять лет назад я написал небольшой памфлет о JavaScript с необычным посылом о творящемся в нем явном бардаке, намекая на то, что глубоко внутри него прячется очень хороший язык. А избегая проблемных функций, можно создавать вполне приличные программы.

С моим мнением не согласилось немало маститых программистов, утверждавших, что мастерство реально продемонстрировать, только используя все возможности языка. Они были абсолютно убеждены, что странности для того и существуют, чтобы демонстрировать мастерство программиста, поэтому плохих функций просто не бывает.

Похоже, что это в корне неверное мнение доминирует до сих пор. Истинное мастерство проявляется в создании хороших программ, код которых легко читается, сопровождается и не содержит ошибок. Если возникнет потребность показать себя с лучшей стороны, попробуйте работать в таком ключе. Будучи скромным программистом, я всегда критически отношусь к себе и своей работе, стремясь к совершенствованию мастерства. Я усвоил трудный урок: оптимизация с целью использования характерных особенностей приводит к обратным результатам.

Вот мой самый эффективный инструмент для наилучшего использования языка программирования:

> если функция в одних случаях полезна, а в других — опасна и есть более подходящий вариант, нужно именно им и воспользоваться.

Взяв этот принцип на вооружение, я всегда стараюсь свести язык к его меньшей и лучшей части, чтобы по возможности избежать тех функций, которые с большой долей вероятности могут привести к возникновению ошибок. Я постоянно пересматриваю свои взгляды на то, что считать хорошей практикой, а что — нет. В книге изложены самые последние размышления о JavaScript. Возможность рассматривать

самые удачные составляющие этого языка представилась мне именно потому, что они в нем есть. Если сравнивать мои нынешние представления с теми, что были десять лет назад, сейчас я полагаю: в использовании языка нужно исходить из принципа «лучше меньше, да лучше».

JavaScript стал наиболее востребованным языком программирования в мире. Извините, но отчасти в этом есть и моя вина. Выпуски новых редакций стандарта ECMAScript не приводят к устранению серьезных проблем JavaScript, а иногда и создают новые проблемы. Комитет по стандартам имеет ограниченные полномочия по коррекции языка. Его представители обладают практически безграничной властью над развитием языка, делая его все более сложным и странным. У них есть достаточные полномочия, чтобы не усугублять ситуацию, но заинтересованы ли они в этом?

С каждым годом языки программирования все больше страдают от пристрастия их разработчиков к некачественной пластической хирургии. В них лихорадочно вводят новые функции в отчаянной надежде сохранить их популярность или по крайней мере моду на их применение. Раздувание функциональных возможностей — такая же большая и глубокая проблема, как и раздувание кода. Полагаю, что вместо этого нужно восхвалять внутреннюю красоту JavaScript.

Я рекомендую вам обратиться к стандарту ECMAScript. Читать его нелегко, но он есть в свободном доступе по адресу ecma-international.org/publications/standards/Ecma-262.htm.

Чтение стандарта ECMAScript буквально изменило мою жизнь. Как и многие другие, я начал писать код на JavaScript, не потрудившись хорошенько изучить язык. И пришел к выводу, что он недоработан, запутан и сильно раздражает меня. Лишь когда нашлось время на изучение стандарта ECMAScript, раскрылось все великолепие JavaScript.

Ересь

Возможно, у кого-то эта книга о языке программирования вызовет раздражение. Я выступаю предвестником следующей парадигмы, и это угрожает хранителям старой парадигмы. Я привык к этому. Нападки на меня начались, когда я обнаружил достоинства JavaScript, что стало первым важным открытием XXI века. Меня атаковали за разработку JSON — на текущий момент самый популярный формат обмена данными.

Сообщества формируются вокруг общих убеждений и способны приносить пользу тем, кто в них входит, даже если убеждения ошибочны. Члены сообщества могут чувствовать угрозу, когда эти убеждения подвергаются сомнению. Я еретик. Я ценю стремление к истине, а не пользу от принадлежности к сообществу. Кто-то может счесть это оскорблением.

Я всего лишь программист, который пытается найти лучший способ создания программ. Вероятно, в чем-то я ошибаюсь, но очень стараюсь все исправить. Во многом образ мышления в нашей профессии сложился еще в эпоху Фортрана. Полагаю,

настало время выйти за рамки этого образа. Перемены же даются нелегко даже в самых инновационных профессиях.

Если вас смущает такая ересь, поставьте эту книгу обратно на полку и уходите.

Код

Весь код, приведенный в этой книге, находится в свободном доступе. Его можно применять для любых целей, но прошу не использовать его кому-нибудь во вред. Попробуйте сделать из него что-нибудь хорошее, если, конечно, представится такая возможность.

Я настоятельно рекомендую не заниматься копированием и вставкой непонятного вам кода, даже если он получен от меня. Похоже, при всей своей опасности и безрассудности подобные действия уже стали вполне обычной практикой. Это ничуть не глупее установки пакетов, на которые вы даже не смотрели, но все равно здесь нет ничего хорошего. Учитывая современное состояние дел, самым важным фильтром безопасности является ваше собственное осознанное поведение. Воспользуйтесь им. Это важно.

Не берусь утверждать, что представленные в книге программы близки к идеалу. Но уверен, что программы, которые я пишу сейчас, лучше, чем те, которые создавал десять лет назад. Я усердно работаю над повышением своей квалификации и надеюсь, что мой солидный стаж позволит наконец получить желаемый результат. Верю, что удача будет сопутствовать и вам. Тем не менее никто не застрахован от *ошибок*. На латыни множественное число от слова «ошибка» (erratum) — *errata*, поскольку это существительное среднего рода второго склонения в именительном падеже. Но я пишу на современном английском, где существительные множественного числа формируются добавлением *-s* или, при избыточном количестве шипящих, *-es*. Так что да, ошибки я обозначаю словом *erratums*. Выбирая между прогрессом и традицией, я отдаю предпочтение прогрессу, основанному на уроках истории. Так будет лучше. Поэтому перечень ошибок можно найти по адресу howjavascriptworks.com/erratums.

Пожалуйста, сообщайте обо всех допущенных мною грубых ошибках по адресу erratum@howjavascriptworks.com.

Следующий язык

Эта книга о JavaScript, но порой я завожу речь о следующем языке, который придет на смену JavaScript. Я должен верить, что у JavaScript будет последователь, поскольку даже сама мысль о том, что его никогда не будет, глубоко печалит. Нам нужно найти путь к следующему языку хотя бы ради наших детей. Они заслуживают лучшего наследия, чем JavaScript.

Я верю, что наше будущее — дети. А также роботы.

Следующей парадигмой станет глобально распределенное безопасное программирование, определяемое конечными событиями. Интернет требует этого. Почти

все современные языки программирования, включая JavaScript, все еще прочно прикованы к старой парадигме локального, небезопасного, последовательного программирования. Я рассматриваю JavaScript как переходный язык. И внедрение лучших приемов программирования в JavaScript станет хорошей подготовкой для понимания следующей парадигмы.

Английский язык

Слово для обозначения цифры 1 пишется неправильно. Я использую правильное написание *wun*. Произношение слова *one* не соответствует ни одному из стандартных или специальных правил английского произношения. И использовать слово, обозначающее цифру 1, которое начинается с буквы, похожей на цифру 0, — ошибка.

Написание *wun* вам незнакомо, поэтому может показаться ошибочным. Я же применяю его намеренно, чтобы приучить вас к мысли, что восприятие чего-то незнакомого как странного еще не доказательство неправильности.

Именно так и происходят реформы правописания. Например, кто-то решает, что было бы лучше, если бы слово *through* превратилось в *thru*, поскольку нет смысла в том, что половина букв в популярном слове не произносится, это крайне неэффективно и ложится ненужным бременем на тех, кто изучает язык. Реформы правописания — это борьба между традицией и разумом, и иногда разум побеждает. Я испытываю сходное чувство и в отношении языков программирования. Так что, если *wun* имеет для вас больше смысла, чем *one*, пожалуйста, станьте сторонником моих нововведений.

Когда обычные люди говорят о диапазоне, например, от 1 до 10 (1 to 10), считается, что диапазон заканчивается на 10. Но программисты зачастую склонны к исключению 10. Эта путаница связана с принятой в программировании практикой нумерации с 0 вместо 1. Поэтому я использую *to* для обозначения того, что программисты обычно имеют в виду под словом *«до»*, и *thru* для обозначения того, что подразумевают под этим словом обычные люди. Таким образом, запись «от 0 до 3» в виде *0 to 3* означает диапазон, включающий 0, 1, 2, а запись «от 0 до 3» в виде *0 thru 3* означает диапазон, включающий 0, 1, 2, 3. Под *to* подразумевается $<$, то есть «меньше чем», а под *thru* подразумевается $\leq$, то есть «меньше или равно».

Теперь что касается слова *«пока» (whilst)* в виде термина *whilst*: в этой книге по программированию сходный по смыслу термин *while* используется, когда речь заходит об итерации. Говоря о параллелизме или одновременности, я применяю термин *whilst*, под которым подразумеваю *«пока, в то же самое время»*.

Правила допускают написание слова *«невозможно»* как в форме *cannot*, так и в форме *can not*, но *cannot* намного популярнее. Форма *hasnot («не имеет»)*, в отличие от раздельного написания *has not*, не допускается. Форма *willnot («не будет»)* также недопустима, в отличие от *will not*. Именно поэтому я не могу вместо *can not* использовать форму *cannot*. Но в запарке все же способен воспользоваться формой *can't*.

Я говорю «Добро пожаловать!» моим многочисленным друзьям, для которых английский язык не родной. Спасибо, что читаете мою книгу. Быть может, не вполне справедливо, что английский стал языком Интернета и разработки программных средств. Но ваше умение читать в оригинале первичную документацию — важный и ценный навык. И я вас за это глубоко уважаю.

Примеры

Я использую регулярные выражения. К сожалению, разобраться в них, не запутавшись, весьма непросто. Я попытаюсь немного упростить ситуацию за счет вставки в них многочисленных пробелов. В JavaScript эти пробелы не допускаются, поэтому, когда вы увидите следующий код:

```
const number_pattern = /
    ^
     ( -? \d+ )
     (?: \. ( \d* ) )?
     (?:
         [ e E ]
         ( [ + \- ]? \d+ )
    )?
    $
/;
```

знайте, что он должен быть записан в таком виде:

```
const number_pattern = /^(-?\d+)(?:\.(\d*))?(?:[eE]([+\-]?\d+))?$/;
```

Я не хочу заставлять вас разбираться в таком нечитаемом коде, поэтому вставляю пробелы.

Во многих главах будут показаны выражения JavaScript. Для этого я использую специальный оператор выражения, который заканчивается не точкой с запятой (`;`), а двойной косой чертой (двойным слешем; `//`), за которой следует результат выражения:

```
// Примеры

3 + 4 === 7                                     // true
NaN === NaN                                     // false
typeof NaN                                      // "number"
typeof null                                     // "object"
0.1 + 0.2 === 0.3                               // false
3472073 ** 7 + 4627011 ** 7 === 4710868 ** 7    // true
```

Все остальное станет понятно в процессе чтения книги.

Глава 1

Как работают имена

○ ○ ○ ○ ●

You know my name. («Я знаю, как меня зовут».)

Джон Леннон (John Lennon)
и Пол Маккартни (Paul McCartney)

В JavaScript требуется, чтобы вы дали имена (или идентификаторы) своим переменным, свойствам и иногда функциям. Ограничений на длину имен переменных в JavaScript не накладывается, поэтому экономить на них не стоит. Позвольте программам как можно шире раскрыть в именах собственные истории. Не используйте загадочных имен.

Сначала меня учили программированию математических вычислений, а затем я пошел работать в компанию, производящую компьютеры, работающие на Бейсике. В то время имена переменных Бейсика состояли из одной прописной буквы и необязательной цифры, например `A1`. У меня выработалась очень плохая привычка использовать однобуквенные имена переменных. Прошел не один десяток лет, а я все еще не могу от нее избавиться. Если в голове застревает что-то не то, исправить это порой нелегко. Но собственный интеллект следует совершенствовать. Математикам нравятся загадочные и краткие обозначения. Ну а нам пришлось получить нелегкий урок: в программировании все должно быть выражено буквально и понятно само по себе. Программирование не математика. Это совершенно иной вид искусства.

Начинайте все свои имена с буквы и заканчивайте их буквой. JavaScript допускает, чтобы имена начинались с символа подчеркивания (`_`) или доллара (`$`) или же заканчивались этими же символами или цифрой. JavaScript позволяет многое, чего не следует делать. Подобные имена нужно оставлять для выбора генераторам кода и макропроцессорам. Люди же должны давать более подходящие имена.

Символ подчеркивания (`_`) в начале или в конце имени иногда предназначается для указания общедоступного свойства или глобальной переменной, которая была бы закрытой, если бы программа была написана правильно. Следовательно, использование символа подчеркивания — признак некомпетентности программиста.

Знак `$` был добавлен в язык для того, чтобы генераторы кода, компиляторы и макропроцессоры могли генерировать имена, гарантированно не конфликтующие

с именами, придуманными вами. Если вы человек, а не программа, не пользуйтесь знаком `$`.

Завершающая цифра в имени обычно указывает на то, что программист не смог придумать имя.

Своим *порядковым* переменным я даю имена вида *`предмет_номер`*, *количественным* переменным — *`количество_предметов`*.

Вполне резонно использовать имена, состоящие из нескольких слов, но нужно выработать некое соглашение по их оформлению, поскольку пробелы внутри имен запрещены. Одна школа настаивает на применении смешанного регистра букв, где первая буква слова прописная, чтобы показать границу между словами. Другая школа настаивает на том, чтобы ставить на месте предполагаемых пробелов на границе слов символ подчеркивания. Есть и третья школа, последователи которой все слова пишут слитно, без обозначения границ. И эти школы не могут прийти к согласию в вопросе о том, что лучше. Они спорили годами, и непохоже, чтобы дело близилось к консенсусу. А все дело в том, что не права ни одна из этих школ.

Правильный ответ таков: нужно использовать пробелы. Пока что языки программирования этого не допускают, поскольку компиляторам 1950-х годов приходилось запускаться в тесном пространстве из весьма скромного количества килослов и пробелы в именах считались непозволительной роскошью. Фортран фактически снял эти ограничения, разрешив именам содержать пробелы, но появившиеся позже языки, включая JavaScript, не последовали хорошему примеру, даже притом, что переняли у Фортрана весьма неудачный способ применения знака равенства (`=`) в качестве оператора присваивания и потребность в левой (`(`) и правой (`)`) скобках вокруг условий оператора `IF`, вместо того чтобы требовать заключения последовательности в левую (`{`) и правую (`}`) фигурные скобки.

Я надеюсь, что следующий язык будет правильнее и позволит именам содержать пробелы, чтобы их стало легче читать. Объем памяти мы измеряем в гигабайтах, следовательно, разработчики языка имеют все возможности для создания более качественного продукта. А пока в именах из нескольких слов нужно использовать символы подчеркивания (`_`). Это предопределит самый простой переход на новый язык.

Все имена в JavaScript должны начинаться с буквы в нижнем регистре. Это связано с проблемой имеющегося в нем оператора `new`. Если перед вызовом функции стоит `new`, она вызывается как конструктор, в противном случае — как обычная функция. Возможности конструкторов и функций могут радикально различаться. Ошибки бывают следствием неправильного вызова конструктора. Еще больше все запутывает то, что конструкторы и функции выглядят совершенно одинаково, поэтому не существует автоматического способа обнаружить проблему, вызванную отсутствием или ненужным присутствием ключевого слова `new`. Так что действует следующее соглашение: *все имена функций-конструкторов должны начинаться с прописной буквы и никакое другое имя не должно начинаться с буквы в верхнем регистре*. Тогда мы получим визуальный признак, помогающий выявить ошибку.

Мое решение этой проблемы еще надежнее: никогда не использовать `new`. Тогда можно будет избавиться от необходимости имен, начинающихся с прописной буквы. Я по-прежнему рекомендую не применять начальных прописных букв в этом языке, поскольку `new` задействовано в пугающем множестве ужасных программ и их количество растет с каждым днем.

Зарезервированные слова

Список зарезервированных слов JavaScript выглядит следующим образом:

```
arguments await break case catch class const continue debugger default delete
do else enum eval export extends false finally for function if implements
import in Infinity instanceof interface let NaN new null package private
protected public return static super switch this throw true try typeof
undefined var void while with yield
```

Запомните его. Это важно. Ни одно из этих слов не должно использоваться в качестве имен переменных или имен параметров. Правила JavaScript относительно зарезервированных слов удивительно сложны, поэтому предусмотрены исключения, в каких случаях некоторые из этих слов можно применять. Но даже в этих странных случаях так делать не нужно.

Работа с зарезервированными словами — еще одна нежелательная особенность, уходящая корнями в нехватку памяти 1950–1960-х годов. Наличие в языке зарезервированных слов способно упростить работу компилятора, позволяя ему сэкономить несколько байтов. Согласно закону Мура нехватка памяти ушла в прошлое, но ограниченность в образе мыслей сохраняется. Вполне очевидно, что зарезервированные слова не приносят программистам ничего хорошего. Вы уже смогли запомнить весь список зарезервированных слов? Вполне вероятно, среди них попадется слово, весьма удачно описывающее вашу переменную, но оно уже было выделено какой-то паршивой функции, которой вы никогда не воспользуетесь или которая, может быть, никогда не будет реализована. Они вредят и разработчикам языков, поскольку стратегия использования ненадежных зарезервированных слов существенно затрудняет добавление новых функциональных возможностей популярным языкам. Я надеюсь, что следующий язык не опустится до применения зарезервированных слов.

Глава 2

Как работают числа

○ ○ ○ ● ○

Look up the number. («Посмотри на номер».)

Джон Леннон и Пол Маккартни

Компьютеры — машины, работающие с числами. В принципе, это все, что могут делать компьютеры. И делают они это очень хорошо. В числах может быть отображена информация разного рода. В наши дни компьютеры выступают посредниками практически во всех видах человеческой деятельности.

Числа в JavaScript основаны на вещественных числах, но таковыми не являются. Мы можем применить наши математические представления к числам JavaScript, но не полностью и не всегда. Чтобы создавать на JavaScript качественные программы, нужно понимать, как работают его числа.

В JavaScript имеется только один числовой тип — `number`. Он заимствован из *стандарта IEEE для представления чисел с плавающей точкой (IEEE 754)*, который изначально разрабатывался для процессора Intel iAPX-432. В этот процессор заложено много блестящих идей, слишком много. Архитектура 432 усложнилась настолько, что не смогла достичь поставленных целей. Был утрачен главный принцип — простота. С процессором 432 канули в Лету многие хорошие идеи. Но модуль процессора 432 с плавающей запятой был спасен и продан как 8087 — математический сопроцессор для процессора 8086. Он превратился в стандартное оборудование на процессорах Pentium и AMD64.

JavaScript часто критикуют за наличие только одного числового типа, но фактически это одна из самых сильных его сторон. Продуктивность программистов повышается, если им не нужно тратить время на выбор из мешанины схожих типов с риском допустить скрытые ошибки, выбрав неверный тип. К тому же это позволяет избегать ошибок преобразования типов. Исключаются также ошибки переполнения, связанные с применением целочисленных типов `int`. Используемые в JavaScript целые числа намного надежнее целочисленных значений Java, поскольку переполнение им не грозит:

```
JavaScript: 2147483647 + 1     //  2147483648 абсолютно верно
Java:       2147483647 + 1     // -2147483648 совершенно неверно
```

Как можно быть уверенными в правильности программ, построенных на системе счисления, которая способна в любой момент абсолютно бесконтрольно и без всякого предупреждения неверно сработать? Типы `int` не предотвращают ошибок, они их вызывают.

Идея, положенная в основу чисел с плавающей точкой, проста: представить число в виде двух чисел. Первое число (иногда называемое коэффициентом, значимой частью числа, дробной частью или мантиссой) содержит цифры. Второе число (экспонента) указывает, где десятичная точка (или двоичная точка) должна быть вставлена в первое число. Реализация чисел с плавающей точкой может быть довольно сложной. Это обусловливается необходимостью наиболее рационального использования ограниченного количества разрядов в фиксированном формате.

В JavaScript стандарт IEEE 754 применяется частично. В нем задействуется поднабор того поднабора, который используется в Java. Имеющийся в JavaScript тип `number` очень тесно связан с типом `double` языка Java. Это 64-разрядный двоичный тип с плавающей точкой. Число содержит разряд знака, 11 разрядов экспоненты и 53 разряда значимой части числа. Некоторые рациональные способы кодирования позволяют упаковывать эти 65 разрядов в 64-разрядное слово.

Для IEEE 754, как и для многих предшествующих систем представления чисел с плавающей точкой, была выбрана система с основанием 2. Первое число разбито на две части — знак и значимую часть числа. Знак находится в самом старшем разряде из 64. Если число отрицательное, он обозначается цифрой 1. Значимая часть числа находится в самых младших разрядах. Обычно она представлена двоичной дробью в диапазоне:

```
0.5 <= значимая часть числа < 1.0
```

В этой форме представления самый значащий разряд всегда установлен в 1. Поскольку этот разряд всегда имеет значение 1, сохранять его в числе не имеет смысла. Это позволяет сэкономить, то есть получить бонусный разряд.

Второе число — экспонента. Оно заполняет пространство между знаком и значимой частью числа. Значением числа будет:

```
знак * значимая часть числа * (2 ** экспонента)
```

Здесь возникают другие сложности. Экспонента представлена в виде смещенного целочисленного значения со знаком, что позволяет выполнять сравнения, как будто это 64-разрядное целое число. Это может существенно повысить производительность, что играло важную роль 50 лет назад. В экспоненте также могут быть закодированы `NaN` и `Infinity`, субнормали и специальные формы для представления очень малых чисел и нуля.

Нуль

Нет никакого другого нуля, кроме нуля. В правильной системе только один нуль. Стандарт IEEE 754 содержит два нуля: `0` и `-0`. JavaScript берет на себя смелость скрыть от вас это извращение и практически всегда успешно с этим справляется.

Можно без всякого опасения игнорировать существование `-0`, за исключением таких случаев:

```
(1 / 0) === (1 / -0)                // false
Object.is(0, -0)                    // false
```

Я не рекомендую когда-либо делить что-либо на нуль. И вообще не рекомендую пользоваться методом `Object.is()`.

Числовые литералы

В JavaScript имеется 18 437 736 874 454 810 627 встроенных неизменяемых числовых объектов, каждый из которых уникальным образом представляет число. Числовой литерал создает ссылку на числовой объект, который наиболее точно соответствует значению литерала. Иногда будет достигаться абсолютная точность. А в некоторых случаях может возникать погрешность вплоть до следующей: 9,979201547673599058281863565184 2e+291.

Числовой литерал для целого числа — это просто последовательность десятичных цифр. С помощью префикса основания можно формировать литералы для целых чисел с разными основаниями. Все следующие литералы дают ссылки на число 2018:

```
binary:      0b11111100010
octal:       0o3742
decimal:     2018.00
hexadecimal: 0x7E2
```

JavaScript позволяет указывать букву основания в верхнем регистре, но вставка в числовой литерал прописной буквы «O» наверняка вызовет путаницу.

Литералы десятичных чисел могут содержать десятичную точку. Очень большие или очень маленькие числа можно компактно записать с помощью обозначения `e`, которое служит признаком умножения числа на степень 10. Таким образом, `6.022140857747475e23` — это сокращенная запись для `(6.022140857747475 * (10 ** 23))`, а `6.626070040818182e-34` — для `(6.626070040818182 * (10 ** -34))`.

`Infinity` (бесконечно большое число) — это значение, представляющее все слишком большие и не поддающиеся обычному представлению числа. Не путайте `Infinity` с ∞ (бесконечность). В математике ∞ не является значением — это образное выражение.

`NaN` — это специальное значение для числовых представлений, не являющихся числами. `NaN` означает *Not a Number (не число)*, и смысл этого сокращения сбивает с толку, поскольку оператор `typeof` правильно сообщает, что типом значения `NaN` является `number`.

`NaN` может быть результатом неудачного преобразования строки в число. Вместо выдачи исключения или остановки выполнения программы выдается значение `NaN`. Если `NaN` — одно из входных данных арифметического оператора, результатом выполнения выражения будет `NaN`.

Самое худшее, что можно сказать о NaN, — то, что значение NaN не равно самому себе. Это извращение стандарта IEEE 754, которое не скрывается в JavaScript. Тестирование значения NaN отличается от тестирования на равенство всех остальных числовых значений. Это может помешать написанию тестов. Если ожидается значение NaN, тест всегда заканчивается неудачей, даже если по факту получается значение NaN. Чтобы протестировать, равно значение NaN или не равно, следует воспользоваться методом Number.isNaN(*значение*). А метод Number.isFinite(*значение*) возвращает false, если *значение* равно NaN, или Infinity, или -Infinity.

Number

Number (но не number, начальная прописная N играет важную роль) — это функция, способная создавать числа. Числа в JavaScript — неизменяемые объекты. Когда оператору typeof передается число, он возвращает number (здесь значимую роль играет начальная n в нижнем регистре). Префикс new с функцией Number ни в коем случае нельзя применять.

```
const good_example = Number("432");
const bad_example = new Number("432");
typeof good_example                         // "number"
typeof bad_example                          // "object"
good_example === bad_example                // false
```

Number также служит контейнером для некоторых констант. Они могут дать представление о работе чисел.

Значение константы Number.EPSILON в языке JavaScript в точности равно 2.220446049250313080847263336181640625e-16. Это наименьшее положительное число, которое при прибавлении к нему единицы дает сумму больше единицы. Прибавление к единице положительного числа, меньшего Number.EPSILON, дает сумму, в точности равную единице. Может показаться абсурдом, что прибавление к единице числа, не равного нулю, дает в результате единицу. Это не связано с ошибкой разработки JavaScript или с каким-то сбоем. Такая странность присуща всем системам отображения чисел с плавающей точкой фиксированного размера, включая IEEE 754. Это разумный компромисс.

Значение константы Number.MAX_SAFE_INTEGER в точности равно 9007199254740991, или же примерно 9 квадриллионам. В JavaScript нет целочисленного типа, и язык в нем не нуждается, поскольку его тип number способен в точности представить все целые числа вплоть до значения Number.MAX_SAFE_INTEGER. В JavaScript-типе number имеются 54-разрядные целые числа со знаком.

Прибавление единицы к целочисленному значению, превышающему значение константы Number.MAX_SAFE_INTEGER, имеет такой же эффект, как и прибавление к нему нуля. JavaScript может выполнять точные целочисленные арифметические вычисления до тех пор, пока все значения, результаты и промежуточные результаты представлены целыми числами, которые вписываются в диапазон от -Number.MAX_SAFE_INTEGER до Number.MAX_SAFE_INTEGER. В этом диапазоне могут су-

ществовать обычные математические умозаключения. В нем работают сочетательный и распределительный законы. Вне этого диапазона все становится более хаотичным. Например, порядок, в котором происходило сложение последовательности чисел, может изменить сумму. Так, результат сложения `((0.1 + 0.2) + 0.3)` превышает результат сложения `(0.1 + (0.2 + 0.3))`. Метод `Number.isSafeInteger(`*`число`*`)` возвращает `true`, если *`число`* относится к безопасному диапазону.

Метод `Number.isInteger(`*`число`*`)` возвращает `true`, если *`число`* целое и из безопасного диапазона или оно выше этого диапазона. Все числа, превышающие значение константы `Number.MAX_SAFE_INTEGER`, рассматриваются как целые. Для некоторых из них это абсолютно верно. Но для большинства — нет.

В `Number.MAX_VALUE` содержится наибольшее число, которое может быть представлено в JavaScript. Его точное значение — `Number.MAX_SAFE_INTEGER * 2 ** 971`, или:

```
17976931348623157081452742373170435679807056752584499659891747680315726078002853
87605895586327668781715404589535143824642343213268894641827684675467035375169860
49910576551282076245490090389328944075868508455133942304583236903222948165808559
33212334827479782620414472316873817718091929988125040402618412485836 8
```

Это цифра 1, за которой следуют еще 308 цифр. Подавляющую часть этой конструкции составляет фиктивное значение. Эти числа могут дать 15,9 значимого десятичного разряда. Завершающие 292 цифры — это иллюзия, вызванная большей раздробленностью чисел с основанием 2 по сравнению с числами с основанием 10.

Прибавление к `Number.MAX_VALUE` любого безопасного целого числа дает сумму, также равную `Number.MAX_VALUE`. Это похоже на то, что программа в состоянии ошибки выдает в качестве результата значение `Number.MAX_VALUE`. Любой результат, превышающий `Number.MAX_SAFE_INTEGER`, вызывает подозрение. Стандарт IEEE 754 обещает возможности применения огромного диапазона, но если не действовать чрезвычайно осторожно, это, скорее всего, приведет к ошибкам.

Константа `Number.MIN_VALUE` содержит наименьшее представляемое число больше нуля. Его точное значение — `2 ** -1074`, или:

```
4.94065645841246544176568792868221372365059802614324764425585682500675507270208 7
51865299836361635992379796564695445717730926656710355939796398774796010781878126
30071319031140452784581716784898210368871863605699873072305000638740915356498438
73124733972731696151400317153853980741262385655911710266585566867681870395603106
24931945271591492455329305456544401127480129709999541931989409080416563324524757
14786901472678015935523861155013480352649347201937902681071074917033322684475 33
35720832431936092382893458368060106011506169809753078342277318329247904982524730
77637592724787465608477820373446969953364701797267771758512566055119913150489110
14510378627381672509558373897335989936648099411642057026370902792427675445652290
875386825064197182655334472656 25e-324
```

Все положительные числа меньше `Number.MIN_VALUE` неотличимы от нуля. Заметьте, что значимая часть числа `Number.MIN_VALUE` состоит только из одного бита в младшем значащем разряде. Этот одинокий бит создает большое фиктивное значение.

`Number.prototype` — это объект, из которого наследуются все числа. `Number.prototype` содержит набор методов, к сожалению не приносящий особой пользы.

Операторы

Префиксные операторы		
+	В число	Префиксный оператор в виде знака «плюс» преобразует свой операнд в число. Если преобразование терпит неудачу, получается значение NaN. Следует отдавать предпочтение применению функции Number, поскольку она более явно выражает намерения
–	Инвертирование	Префиксный оператор в виде знака «минус» изменяет знак своего операнда. Числовые литералы в JavaScript не имеют знаков. В выражении вида (–1) знак «минус» — это оператор, а не часть числового литерала
typeof	Тип операнда	Если операндом является число, он выдает строковое значение number, даже если операнд имеет значение NaN
Инфиксные операторы		
+	Сложение	К сожалению, оператор «плюс» также используется строками для объединения. Это переопределение создает фактор риска. Если один из операторов является строкой, он преобразует другой оператор в строку и объединяет строки. К сожалению, в языке нет других способов выполнения сложения, поэтому нужно проявлять осмотрительность. Функция Number может пригодиться, чтобы гарантировать, что операнды инфиксного оператора + фактически являются числами и что сложение будет выполнено
–	Вычитание	
*	Умножение	
/ (слеш)	Деление	Это не целочисленное деление. Если разделить одно целое число на другое целое число, можно получить дробный результат: 5 / 2 дает 2,5, а не 2
%	Остаток от деления	В JavaScript нет оператора модуля. Вместо него используется оператор остатка от деления. Я полагаю, что оператор модуля более полезен. Результат остатка от деления берет свой знак от делимого, а результат модуля — от делителя: –5 % 2 равно –1
**	Возведение в степень	JavaScript применил фортрановскую двойную звездочку, чтобы придать языку необычный ретровид

Поразрядные операторы

В JavaScript имеется набор поразрядных операторов, похожих на те, что есть в C и других языках. Все они работают с числами JavaScript, сначала преобразуя их в 32-разрядные целые числа со знаком с помощью поразрядных операций, а затем обратно в числа JavaScript. Было бы лучше, если бы они работали с 54-разрядными безопасными целыми числами, но они этого не делают, поэтому 22 старших двоичных разряда могут быть утрачены без предупреждения.

В некоторых языках сдвиг может быть использован вместо умножения или деления или же поразрядный оператор *and* (И) применен в качестве оператора модуля. Если что-либо подобное проделать в этом языке, могут быть отброшены 22 самых старших разряда. В одних случаях это ни на что не повлияет, а в других обойтись без отрицательных последствий не удастся.

Поэтому поразрядные операторы используются в JavaScript значительно реже, чем в других языках. Но даже если эти операторы не применяются, они несут потенциальную синтаксическую опасность. Знаки амперсанда (&) и вертикальной черты (|) можно спутать с двумя знаками амперсанда (&&) и двойной вертикальной чертой (||). Двойной знак «меньше» (<<) и двойной знак «больше» (>>) можно спутать со знаками «меньше» (<) и «больше» (>). Мне непонятно, почему знак >> представляет собой оператор сдвига вправо, выполняющий расширение знака, а знак >>> — нет. В языке C расширение знака было определено типом, а в языке Java — оператором. В JavaScript скопирован неудачный выбор, сделанный для Java. Поэтому здесь нужно проявлять осмотрительность.

Единственным поразрядным унарным оператором является знак тильды (~), обозначающий поразрядное НЕ.

К поразрядным бинарным операторам относятся:

- `&` (амперсанд) — поразрядное И;
- `|` (вертикальная черта) — поразрядное ИЛИ;
- `^` (циркумфлекс) — поразрядное исключающее ИЛИ;
- `<<` (меньше, меньше) — сдвиг влево;
- `>>>` (больше, больше, больше) — сдвиг вправо;
- `>>` (больше, больше) — сдвиг вправо с расширением знака.

Объект Math

В объекте `Math` содержится важный набор функций, который следовало бы встроить в `Number`. Это очередной пример плохого влияния языка Java. Помимо тригонометрических и логарифмических функций, в нем содержатся практические функции, которые следовало бы предоставить в качестве операторов.

Обе функции, и `Math.floor`, и `Math.trunc`, производят из числа целое число. `Math.floor` выдает наименьшее целое число, а `Math.trunc` — то целое число, которое ближе к нулю. Какую из них использовать, зависит от того, что вы хотите получить из отрицательных чисел:

```
Math.floor(-2.5) // -3
Math.trunc(-2.5) // -2
```

Функции `Math.min` и `Math.max` возвращают наименьший или наибольший из аргументов.

Функция `Math.random` возвращает число в диапазоне от 0 до 1. Она вполне подойдет для игр, но только не для криптографических приложений или игр в казино.

Монстр

В JavaScript нет инструментария для разбора числа на компоненты, но написать такое средство не составит труда. Тогда мы сможем познать истинную природу чисел.

Я собираюсь использовать вместо значимой части числа коэффициент, поскольку хочу работать полностью в целочисленном пространстве, где все может быть понятнее и точнее. Дробная значимая часть числа требует более пространных объяснений.

```
function deconstruct(number) {
```

Эта функция выполняет разбор числа, переводя его в компоненты — знак, целочисленный коэффициент и экспоненту:

```
    number = sign * coefficient * (2 ** exponent)

    let sign = 1;
    let coefficient = number;
    let exponent = 0;
```

Убираем знак из коэффициента:

```
    if (coefficient < 0) {
        coefficient = -coefficient;
        sign = -1;
    }

    if (Number.isFinite(number) && number !== 0) {
```

Выполняем перевод коэффициента: получить экспоненту можно делением числа на 2 до тех пор, пока не будет получен нуль. Количество делений добавляем к `-1128`, то есть к экспоненте `Number.MIN_VALUE` за вычетом количества разрядов в значимой части числа и бонусного разряда:

```
        exponent = -1128;
        let reduction = coefficient;
        while (reduction !== 0) {
```

Этот цикл гарантирует достижения нуля. Каждая операция деления будет уменьшать экспоненту перевода. Когда она станет настолько малой, что ее невозможно будет уменьшить на единицу, тогда вместо этого внутренняя субнормальная значимая часть будет сдвинута вправо. В конечном итоге будут удалены все разряды:

```
            exponent += 1;
            reduction /= 2;
        }
```

Переводим экспоненту: когда она равна нулю, число может рассматриваться как целое. Если экспонента отлична от нуля, требуется корректировка коэффициента:

```
        reduction = exponent;
        while (reduction > 0) {
```

```
            coefficient /= 2;
            reduction -= 1;
        }
        while (reduction < 0) {
            coefficient *= 2;
            reduction += 1;
        }
    }
```

Возвращаем объект, содержащий три компонента и исходное число:

```
    return {
        sign,
        coefficient,
        exponent,
        number
    };
}
```

Теперь, имея оснащение, рассмотрим само чудовище.

Когда будет выполнен разбор `Number.MAX_SAFE_INTEGER`, мы получим:

```
{
    "sign": 1,
    "coefficient": 9007199254740991,
    "exponent": 0,
    "number": 9007199254740991
}
```

`Number.MAX_SAFE_INTEGER` — наибольшее число, помещающееся в 54-разрядное целое число со знаком.

После разбора числа 1 мы получим:

```
{
    "sign": 1,
    "coefficient": 9007199254740992,
    "exponent": -53,
    "number": 1
}
```

Заметьте, что `1 * 9007199254740992 * (2 ** -53)` равняется `1`.

А теперь усложним задачу и разберем число `0.1`. Одну десятую. Цент:

```
{
    "sign": 1,
    "coefficient": 7205759403792794,
    "exponent": -56,
    "number": 0.1
}
```

Если вычислить `1 * 7205759403792794 * 2 ** -56`, то получится значение не `0.1`, а более точно: `0.1000000000000000055511151231257827021181583404541015625`.

Хорошо известно, что JavaScript плохо справляется с обработкой десятичных дробей, в частности денежных величин. Когда в программу вводится `0.1` или большинство других десятичных дробей, JavaScript не способен в точности представить это значение, поэтому *использует иные величины*, подставляя значение, которое он способен представить.

Когда в программе набирается десятичная точка или же она считывает значение данных с десятичной точкой, вполне вероятно, что в программу вносится небольшая погрешность. Порой погрешности настолько малы, что их невозможно заметить. Иногда они взаимопоглощаются, а в некоторых случаях накапливаются.

При разборе `0.3` мы получим результат, отличный от получаемого при разборе `0.1 + 0.2`:

```
{
    "sign": 1,
    "coefficient":5404319552844595,
    "exponent":-54,
    "number": 0.3
}

{
    "sign": 1,
    "coefficient":5404319552844596,
    "exponent":-54,
    "number": 0.30000000000000004
}
```

Заметьте, что ни `0.2999999999999999888977697537484345957636833190917968 75`, ни `0.3000000000000000444089209850062616169452667236328125` не равны `0.3`.

Рассмотрим еще один пример. При разборе 100 / 3 мы получим:

```
{
    "sign": 1,
    "coefficient": 9382499223688534,
    "exponent": -48,
    "number": 33.333333333333336
}
```

Заметьте, JavaScript сообщил, что число равно `33.333333333333336`. Конечная цифра `6` свидетельствует о том, что JavaScript не в состоянии дать правильный или хотя бы приемлемый ответ. На самом деле все еще печальнее. JavaScript фактически полагает, что ответ в точности равен `33.33333333333333570180911920033395290374755859375`.

Системы с плавающей точкой поставляются с функциями, выполняющими преобразования между внутренним двоичным представлением и внешним, более привычным для людей десятичным представлением. Эти функции разработаны таким образом, чтобы по возможности максимально скрыть истинное положение дел. Есть вполне обоснованная обеспокоенность тем, что постоянное столкновение с реалиями стандарта IEEE 754 вызвало бы настоящий бунт с выдвижением требований об использовании чего-то более подходящего. А на практике мы не желаем

видеть возникающие погрешности в получаемых результатах и не хотим показывать их своим клиентам. Это выставило бы напоказ всю нашу некомпетентность. Лучше сделать вид, что все хорошо.

Первыми использовать двоичное представление чисел с плавающей точкой начали математики и ученые. Математики хорошо понимали, что компьютеры, будучи конечными устройствами, не могут точно представлять действительные числа, поэтому они полагались на методы численного анализа, чтобы получать от конечных систем полезные результаты. Ученые работали над экспериментальными данными, не лишенными погрешностей, поэтому неточность представления двоичных чисел с плавающей точкой не создавала для них существенных проблем. Но первые бизнес-пользователи отказались от двоичного представления с плавающей точкой, потому что их клиенты и закон требуют точных десятичных значений. По закону при подсчете денег вы должны получить правильную сумму.

Это было более полувека назад. С тех пор мы, похоже, забыли о компромиссах, связанных с двоичным представлением чисел с плавающей точкой. К настоящему времени мы должны были перейти к чему-то более подходящему. Непростительно, что в XXI веке мы не можем с достаточной степенью надежности сложить `0.1` и `0.2`, чтобы получить `0.3`. Я надеюсь, что язык, который придет на смену JavaScript, будет иметь один числовой тип, способный точно выражать десятичные дроби. Такая система до сих пор не способна точно представлять вещественные числа. И никакая конечная система не сделает этого. Но она сможет точно представлять числа, наиболее важные для человечества, — числа, состоящие из десятичных цифр.

А пока нужно стремиться как можно больше работать в безопасном целочисленном диапазоне. Я рекомендую переводить все денежные значения в центы, чтобы они могли быть точно обработаны как целые числа. Опасность такого подхода заключается во взаимодействии с кодом или системами, которые не делают того же самого. Подобные ошибки в организации взаимодействия могут привести к результатам, стократно отличающимся от правильных в ту или иную сторону. Конечно, это никуда не годилось бы. Возможно, здесь нас спасет инфляция, обесценивающая центы и позволяющая их просто проигнорировать.

Когда работа протекает вне безопасного целочисленного диапазона, числа, содержащие десятичную точку (`.`) или десятичную экспоненту `e`, могут не получать истинного числового значения. Сложение одинаковых чисел дает меньшую ошибку, чем сложение разных чисел. Поэтому вычисление общего итога из предварительно вычисленных промежуточных итогов дает более точный результат, чем вычисление общего итога на основе отдельно взятых значений.

Глава 3

Как работают большие целые числа

○ ○ ○ ● ●

> Он сказал пять, я сказал шесть. Он сказал восемь,
> я сказал девять. Он сказал десять, я сказал
> одиннадцать. Я ни за что не останавливаюсь.
> Я повышаю ставку. Я иду все выше, выше, выше.
>
> *Чико Маркс (Chico Marx)*

Одна из популярных претензий к JavaScript заключается в том, что этот язык не имеет 64-разрядных целых чисел. Тип `int64` способен содержать точные целые числа вплоть до `9223372036854775807`, что на три разряда больше получаемого от чисел JavaScript с их жалким значением `Number.MAX_SAFE_INTEGER`, равным `9007199254740991`.

Возникают вопросы по поводу необходимости простого введения нового числового типа. Казалось бы, здесь не должно быть никакой проблемы. В других языках есть несколько числовых типов. Почему бы JavaScript не стать более похожим на другие языки?

Когда в языке имеется один числовой тип, добавление еще одного является актом насилия. Это повлечет за собой огромные потери в простоте и существенный рост нового потенциала для формирования ошибок. Потенциальная ошибка — каждое объявление типа и каждое преобразование типа.

Возникает также вопрос: а хватит ли 64 разрядов? Возможно, нам нужно заглядываться на 72, или 96, или 128, или 256 разрядов. Какое бы число вы ни выбрали, есть не менее веский аргумент в пользу еще более высокого числа.

Полагаю, что добавление в язык больших целых чисел было бы ошибкой. Вместо этого должна быть библиотека. Большинство пользователей языка в таких числах не нуждаются, и они не решают самую серьезную проблему, которую приходится решать при работе с нынешними числовыми данными. При небольшом объеме программирования нет необходимости калечить язык. Точную целочисленную арифметику в любом распределении разрядов можно получить с помощью текущей версии JavaScript. Существует множество способов решения этой задачи.

Реализация, представляемая в этой книге, не была оптимизирована в области скорости или лаконичности, вместо этого вся ее оптимизация была направлена на объяснимость. Я хочу представить полную библиотеку, не отведя на это слишком много страниц.

Я собираюсь хранить большие целые числа в массивах. Массивы — удачный выбор, поскольку они могут иметь любой размер. (Неплохим вариантом станут строки, в которых каждый символ может рассматриваться как 16-разрядное целое число без знака.) В каждом элементе массива хранится число, содержащее разряды большого целого числа. Важный конструктивный вопрос: *сколько разрядов будет приходиться на один элемент*? Ответ: максимум 53, размер положительного безопасного целого числа. Язык подсказывает, что разумнее выбрать 32 разряда или еще меньше, поскольку это позволит задействовать поразрядные операторы. Если размер нашего слова будет больше 32 разрядов, тогда поразрядные операторы не смогут использоваться и реализация усложнится.

Но, учитывая потребности в умножении и делении, и 32 разряда будут слишком большой величиной. При реализации умножения и деления больших целых чисел хотелось бы использовать оператор умножения JavaScript, сохраняющий точность только до 53 разрядов. Это означает, что размер нашего слова должен быть не более половины этой величины. Я выбрал 24 разряда. Мог бы остановиться и на 26, но мне нравится число 24, оно мне ближе. (Моя первая программа работала на 24-разрядной универсальной машине Control Data Corporation 3150.) Назовем эти 24-разрядные блоки *мегацифрами*, потому что они могут представлять более чем в миллион раз больше значений, чем обычные цифры.

Я выбрал представление числовой величины со знаком. Нулевой элемент массива содержит знак числа — либо «+», либо «–». В первом элементе содержатся наименее значимые мегацифры. В последнем элементе — наиболее значимые мегацифры. В результате `9000000000000000000` будет выглядеть так:

```
["+", 8650752, 7098594, 31974]
```

Особой красоты в этом представлении нет, но оно работает, поскольку:

```
9000000000000000000 = 8650752 + ((7098594 + (31974 * 16777216)) * 16777216)
```

Чтобы легче было добраться до внутреннего устройства большого целого числа, я создал константы и функции.

```
const radix = 16777216;
const radix_squared = radix * radix;
const log2_radix = 24;
const plus = "+";
const minus = "-";
const sign = 0;
const least = 1;

function last(array) {
```

```
    return array[array.length - 1];
}

function next_to_last(array) {
    return array[array.length - 2];
}
```

Создадим несколько констант. В них нет насущной потребности, но код с ними будет проще читаться:

```
const zero = Object.freeze([plus]);
const wun = Object.freeze([plus, 1]);
const two = Object.freeze([plus, 2]);
const ten = Object.freeze([plus, 10]);
const negative_wun = Object.freeze([minus, 1]);
```

Для обнаружения как положительных, так и отрицательных больших целых чисел нам нужны предикативные функции:

```
function is_big_integer(big) {
    return Array.isArray(big) && (big[sign] === plus || big[sign] === minus);
}

function is_negative(big) {
    return Array.isArray(big) && big[sign] === minus;
}

function is_positive(big) {
    return Array.isArray(big) && big[sign] === plus;
}

function is_zero(big) {
    return !Array.isArray(big) || big.length < 2;
}
```

Функция `mint` удаляет последние слова из массива, если они нулевые. При наличии соответствия она выполняет подстановку одной из констант. Если соответствия нет, массив замораживается. Если позволить изменять массивы, то в некоторых случаях реализация могла бы работать быстрее, но тогда она усложнится и станет способна допускать больше ошибок. Наши большие целые числа неизменяемые, как и числа JavaScript:

```
function mint(proto_big_integer) {
```

Создание большого целого числа из его прототипа. Удаление лидирующих нулевых мегацифр. Подстановка по возможности популярных констант:

```
    while (last(proto_big_integer) === 0) {
        proto_big_integer.length -= 1;
    }
    if (proto_big_integer.length <= 1) {
        return zero;
    }
    if (proto_big_integer[sign] === plus) {
        if (proto_big_integer.length === 2) {
```

```
            if (proto_big_integer[least] === 1) {
                return wun;
            }
            if (proto_big_integer[least] === 2) {
                return two;
            }
            if (proto_big_integer[least] === 10) {
                return ten;
            }
        }
        } else if (proto_big_integer.length === 2) {
        if (proto_big_integer[least] === 1) {
            return negative_wun;
        }
    }
    return Object.freeze(proto_big_integer);
}
```

Наши первые практические функции — отрицание, абсолютное значение и извлечение знака:

```
function neg(big) {
    if (is_zero(big)) {
        return zero;
    }
    let negation = big.slice();
    negation[sign] = (
        is_negative(big)
        ? plus
        : minus
    );
    return mint(negation);
}

function abs(big) {
    return (
        is_zero(big)
        ? zero
        : (
            is_negative(big)
            ? neg(big)
            : big
        )
    );
}

function signum(big) {
    return (
        is_zero(big)
        ? zero
        : (
            is_negative(big)
            ? negative_wun
            : wun
        )
    );
}
```

Функция `eq` определяет, содержат ли два больших целых числа одни и те же значения разрядов:

```
function eq(comparahend, comparator) {
    return comparahend === comparator || (
        comparahend.length === comparator.length
        && comparahend.every(function (element, element_nr) {
            return element === comparator[element_nr];
        })
    );
}
```

Функция `abs_lt` определяет, меньше ли абсолютное значение большого целого числа абсолютного значения другого большого целого числа. Функция `lt` определяет, меньше ли значение большого целого числа со знаком значения другого большого целого числа со знаком. Этим функциям приходится труднее, если два больших целых числа одинаковой длины. Их работа может стать легче при наличии версии функции `reduce`, способной работать в обратном порядке, а также завершать работу на ранней стадии.

```
function abs_lt(comparahend, comparator) {
    return (
```

Если проигнорировать знак, то число, у которого больше мегацифр, будет больше. Если два числа содержат одинаковое количество мегацифр, придется исследовать каждую пару.

```
        comparahend.length === comparator.length
        ? comparahend.reduce(
            function (reduction, element, element_nr) {
                if (element_nr !== sign) {
                    const other = comparator[element_nr];
                    if (element !== other) {
                        return element < other;
                    }
                }
                return reduction;
            },
            false
        )
        : comparahend.length < comparator.length
    );
}

function lt(comparahend, comparator) {
    return (
        comparahend[sign] !== comparator[sign]
        ? is_negative(comparahend)
        : (
            is_negative(comparahend)
            ? abs_lt(comparator, comparahend)
            : abs_lt(comparahend, comparator)
        )
    );
}
```

При наличии функции `lt` будет легче выполнить другие сравнения путем дополнений и перестановок:

```
function ge(a, b) {
    return !lt(a, b);
}

function gt(a, b) {
    return lt(b, a);
}

function le(a, b) {
    return !lt(b, a);
}
```

Теперь создадим функции поразрядной обработки. Каждое из больших целых чисел содержит разряды. Мы будем исходить из того, что знаки не имеют никакого отношения к поразрядным операциям, поэтому игнорируются во входных данных, а в выходных данных игнорируется знак «+».

Нашими первыми поразрядными функциями станут `and`, `or` и `xor`. Для функции `and` желательно предусмотреть обработку самого короткого массива. Лишние слова в более длинном массиве ей неинтересны. Лишние слова обнуляются и стираются. Функциям `or` и `xor` требуется работа с самым длинным массивом.

```
function and(a, b) {
```

Превращение `a` в самый короткий массив:

```
    if (a.length > b.length) {
        [a, b] = [b, a];
    }
    return mint(a.map(function (element, element_nr) {
        return (
            element_nr === sign
            ? plus
            : element & b[element_nr]
        );
    }));
}

function or(a, b) {
```

Превращение `a` в самый длинный массив:

```
    if (a.length < b.length) {
        [a, b] = [b, a];
    }
    return mint(a.map(function (element, element_nr) {
        return (
            element_nr === sign
            ? plus
            : element | (b[element_nr] || 0)
        );
    }));
}
```

```
function xor(a, b) {
```

Превращение a в самый длинный массив:

```
    if (a.length < b.length) {
        [a, b] = [b, a];
    }
    return mint(a.map(function (element, element_nr) {
        return (
            element_nr === sign
            ? plus
            : element ^ (b[element_nr] || 0)
        );
    }));
}
```

Некоторые функции получают в качестве аргумента малое целое число. Функция int упрощает работу как с числами, так и с большими целыми числами:

```
function int(big) {
    let result;
    if (typeof big === "number") {
        if (Number.isSafeInteger(big)) {
            return big;
        }
    } else if (is_big_integer(big)) {
        if (big.length < 2) {
            return 0;
        }
        if (big.length === 2) {
            return (
                is_negative(big)
                ? -big[least]
                : big[least]
            );
        }
        if (big.length === 3) {
            result = big[least + 1] * radix + big[least];
            return (
                is_negative(big)
                ? -result
                : result
            );
        }
        if (big.length === 4) {
            result = (
                big[least + 2] * radix_squared
                + big[least + 1] * radix
                + big[least]
            );
            if (Number.isSafeInteger(result)) {
                return (
                    is_negative(big)
                    ? -result
                    : result
                );
            }
        }
    }
}
```

Функция `shift_down` уменьшает числа, удаляя наименее значимые разряды. С ее помощью можно уменьшить большое целое число. Это похоже на деление на степени числа 2. Данная операция обычно называется *сдвигом вправо* (`>>>`) и выглядит обманчиво, поскольку для уменьшения чисел использует знак «больше». Нумерация разрядов — вещь совершенно произвольная, поэтому представление о том, что разряды увеличиваются справа налево, сбивает с толку. Мы создаем значения из массивов, которые в обычном представлении растут слева направо, следовательно, рост происходит одновременно как справа налево, так и слева направо. В некоторых системах обычного письма строки идут слева направо, а в некоторых — справа налево, поэтому естественность прироста слева направо нельзя считать универсальной. Проблема направления роста разрядов (Endian Problem) уходит своими корнями именно в эту путаницу. Сдвиги влево и вправо дают менее точные результаты, чем уменьшение и увеличение чисел.

Если количество сдвигов кратно числу 24, сдвиг дается легко. В противном случае приходится перераспределять все разряды:

```
function shift_down(big, places) {
    if (is_zero(big)) {
        return zero;
    }
    places = int(places);
    if (Number.isSafeInteger(places)) {
        if (places === 0) {
            return abs(big);
        }
        if (places < 0) {
            return shift_up(big, -places);
        }
        let skip = Math.floor(places / log2_radix);
        places -= skip * log2_radix;
        if (skip + 1 >= big.length) {
            return zero;
        }
        big = (
            skip > 0
            ? mint(zero.concat(big.slice(skip + 1)))
            : big
        );
        if (places === 0) {
            return big;
        }
        return mint(big.map(function (element, element_nr) {
            if (element_nr === sign) {
                return plus;
            }
            return ((radix - 1) & (
                (element >> places)
                | ((big[element_nr + 1] || 0) << (log2_radix - places))
            ));
        }));
    }
}
```

Функция `shift_up` увеличивает числа, вставляя нули в менее значимые концевые разряды. Это похоже на умножение на степени числа 2. Тем самым можно увеличить большое целое число. В большинстве систем при сдвиге за пределы числа разряды могут быть утрачены, но в этой системе пределы не лимитированы, поэтому разряды не теряются:

```
function shift_up(big, places) {
    if (is_zero(big)) {
        return zero;
    }
    places = int(places);
    if (Number.isSafeInteger(places)) {
        if (places === 0) {
            return abs(big);
        }
        if (places < 0) {
            return shift_down(big, -places);
        }
        let blanks = Math.floor(places / log2_radix);
        let result = new Array(blanks + 1).fill(0);
        result[sign] = plus;
        places -= blanks * log2_radix;
        if (places === 0) {
            return mint(result.concat(big.slice(least)));
        }
        let carry = big.reduce(function (accumulator, element, element_nr) {
            if (element_nr === sign) {
                return 0;
            }
            result.push(((element << places) | accumulator) & (radix - 1));
            return element >> (log2_radix - places);
        }, 0);
        if (carry > 0) {
            result.push(carry);
        }
        return mint(result);
    }
}
```

Нам не помешало бы иметь функцию `not`, выполняющую дополнение всех разрядов, но у нас отсутствуют ограничения на количество разрядов, поэтому непонятно, сколько разрядов нужно переворачивать (flipped). Следовательно, есть функция `mask`, создающая большое целое число из конкретного количества единичных разрядов. Затем можно будет использовать `mask` и `xor` для создания `not`, но функции `not` нужно сообщить размер поля разрядов.

```
function mask(nr_bits) {
```

Создание строки единичных разрядов:

```
    nr_bits = int(nr_bits);
    if (nr_bits !== undefined && nr_bits >= 0) {
        let mega = Math.floor(nr_bits / log2_radix);
```

```
        let result = new Array(mega + 1).fill(radix - 1);
        result[sign] = plus;
        let leftover = nr_bits - (mega * log2_radix);
        if (leftover > 0) {
            result.push((1 << leftover) - 1);
        }
        return mint(result);
    }
}

function not(a, nr_bits) {
    return xor(a, mask(nr_bits));
}
```

Функция `random` создает произвольное (случайное) большое целое число. Она получает количество генерируемых разрядов, а также необязательную функцию генератора случайных чисел, возвращающую числа между 0 и 1. Если функция генератора случайных чисел не передается, используется функция `Math.random`, подходящая для того, чтобы достичь большинства поставленных целей, но только не для криптографических приложений:

```
function random(nr_bits, random = Math.random) {
```

Создание строки случайных разрядов. Если решаются вопросы обеспечения безопасности, этой функции можно передать более серьезный генератор случайных чисел.

Сначала создается строка единичных разрядов:

```
    const wuns = mask(nr_bits);
    if (wuns !== undefined) {
```

Для каждой мегацифры берется случайное число между `0.0` и `1.0`. Затем берется часть верхних разрядов и часть нижних разрядов и над ними проводится операция `xor`. Далее с помощью операции `and` формируется мегацифра, которая помещается в новое число:

```
        return mint(wuns.map(function (element, element_nr) {
        if (element_nr === sign) {
                return plus;
            }
            const bits = random();
            return ((bits * radix_squared) ^ (bits * radix)) & element;
        }));
    }
}
```

Сложение выполняется так же, как это делалось в школе, за исключением того, что вместо основания 10 используется основание 16 777 216. Чтобы сделать перенос доступным сумматору, применяется замкнутое выражение:

```
function add(augend, addend) {
    if (is_zero(augend)) {
```

```
        return addend;
    }
    if (is_zero(addend)) {
        return augend;
    }
```

Если знаки разные, код превращается в вычитание:

```
    if (augend[sign] !== addend[sign]) {
        return sub(augend, neg(addend));
    }
```

Знаки одинаковые. При сложении разрядов результату присваивается тот же знак. Можно складывать числа разной длины. К более длинному применяется метод `.map`, и для подстановки нулей вместо несуществующих элементов используется оператор `||`:

```
    if (augend.length < addend.length) {
        [addend, augend] = [augend, addend];
    }
    let carry = 0;
    let result = augend.map(function (element, element_nr) {
        if (element_nr !== sign) {
            element += (addend[element_nr] || 0) + carry;
            if (element >= radix) {
                carry = 1;
                element -= radix;
            } else {
                carry = 0;
            }
        }
        return element;
    });
```

Если возникает переполнение, то для возможности переноса добавляется еще один элемент:

```
    if (carry > 0) {
        result.push(carry);
    }
    return mint(result);
}
```

Вычитание также не составляет труда:

```
function sub(minuend, subtrahend) {
    if (is_zero(subtrahend)) {
        return minuend;
    }
    if (is_zero(minuend)) {
        return neg(subtrahend);
    }
    let minuend_sign = minuend[sign];
```

Если знаки разные, код превращается в сложение:

```
    if (minuend_sign !== subtrahend[sign]) {
        return add(minuend, neg(subtrahend));
    }
```

Вычитание меньшего из большего:

```
    if (abs_lt(minuend, subtrahend)) {
        [subtrahend, minuend] = [minuend, subtrahend];
        minuend_sign = (
            minuend_sign === minus
            ? plus
            : minus
        );
    }
    let borrow = 0;
    return mint(minuend.map(function (element, element_nr) {
        if (element_nr === sign) {
            return minuend_sign;
        }
        let diff = element - ((subtrahend[element_nr] || 0) + borrow);
        if (diff < 0) {
            diff += 16777216;
            borrow = 1;
        } else {
            borrow = 0;
        }
        return diff;
    }));
}
```

Умножение несколько сложнее. Мы используем вложенные функции `forEach`, поскольку нужно перемножить каждый элемент `multiplicand` с каждым элементом `multiplier`. Каждое из этих произведений может быть 48-разрядным, но в элементе могут содержаться только 24 разряда, поэтому переполнение должно быть перенесено:

```
function mul(multiplicand, multiplier) {
    if (is_zero(multiplicand) || is_zero(multiplier)) {
        return zero;
    }
```

Если знаки совпадают, знак результата будет «+»:

```
    let result = [
        multiplicand[sign] === multiplier[sign]
        ? plus
        : minus
    ];
```

Перемножение каждого элемента `multiplicand` с каждым элементом `multiplier` с распространением переноса:

```
    multiplicand.forEach(function (
        multiplicand_element,
```

```
        multiplicand_element_nr
    ) {
        if (multiplicand_element_nr !== sign) {
            let carry = 0;
            multiplier.forEach(function (
                multiplier_element,
                multiplier_element_nr
            ) {
                if (multiplier_element_nr !== sign) {
                    let at = (
                        multiplicand_element_nr + multiplier_element_nr - 1
                    );
                    let product = (
                        (multiplicand_element * multiplier_element)
                        + (result[at] || 0)
                        + carry
                    );
                    result[at] = product & 16777215;
                    carry = Math.floor(product / radix);
                }
            });
            if (carry > 0) {
                result[multiplicand_element_nr + multiplier.length - 1] = carry;
            }
        }
    });
    return mint(result);
}
```

Функция `divrem` выполняет деление, возвращая как частное, так и остаток. Для удобства предоставим также функцию `div`, возвращающую только частное от деления:

```
function divrem(dividend, divisor) {
    if (is_zero(dividend) || abs_lt(dividend, divisor)) {
        return [zero, dividend];
    }
    if (is_zero(divisor)) {
        return undefined;
    }
```

Придадим операндам положительные значения:

```
    let quotient_is_negative = dividend[sign] !== divisor[sign];
    let remainder_is_negative = dividend[sign] === minus;
    let remainder = dividend;
    dividend = abs(dividend);
    divisor = abs(divisor);
```

Выполним привычное по школьным временам деление в столбик. Оценим следующую цифру частного. Вычтем столько значений делителя, сколько получится в результате оценки делимого, а затем повторим это действие. Вместо основания 10 применим основание 16 777 216 и воспользуемся более системным подходом в прогнозировании следующей цифры частного.

Чтобы улучшить прогнозы, сначала обработаем делитель функцией `mint`. Сдвиг влево выполняется до тех пор, пока его самый старший разряд не станет `1`. На ту же величину выполняется сдвиг влево делимого. Описание алгоритма Algorithm 4.3.1D можно найти в книге *The Art of Computer Programming*.

Для определения количества сдвигов находим количество лидирующих `0`. Функция `clz32` выполняет вычисление в поле из 32 разрядов. Нас также интересует поле из 24 разрядов, поэтому вычитаем 8:

```
let shift = Math.clz32(last(divisor)) - 8;

dividend = shift_up(dividend, shift);
divisor = shift_up(divisor, shift);
let place = dividend.length - divisor.length;
let dividend_prefix = last(dividend);
let divisor_prefix = last(divisor);
if (dividend_prefix < divisor_prefix) {
    dividend_prefix = (dividend_prefix * radix) + next_to_last(dividend);
} else {
    place += 1;
}
divisor = shift_up(divisor, (place - 1) * 24);
let quotient = new Array(place + 1).fill(0);
quotient[sign] = plus;
while (true) {
```

Оценка не будет слишком маленькой, но может оказаться слишком большой. Если она слишком высока, вычитание из делимого произведения оценки и делителя дает отрицательный результат. Когда это происходит, нужно уменьшить оценку и попробовать еще раз:

```
    let estimated = Math.floor(dividend_prefix / divisor_prefix);
    if (estimated > 0) {
        while (true) {
            let trial = sub(dividend, mul(divisor, [plus, estimated]));
            if (!is_negative(trial)) {
                dividend = trial;
                break;
            }
            estimated -= 1;
        }
    }
```

Правильная оценка сохраняется в `quotient`. Если это был последний разряд, идем дальше:

```
    quotient[place] = estimated;
    place -= 1;
    if (place === 0) {
        break;
    }
```

Выполняем подготовку для следующего разряда. Обновляем `dividend_prefix` с использованием первых двух слов оставшегося делимого `dividend` и уменьшаем делитель `divisor`:

```
            if (is_zero(dividend)) {
                break;
            }
            dividend_prefix = last(dividend) * radix + next_to_last(dividend);
            divisor = shift_down(divisor, 24);
        }
```

Исправляем остаток:

```
        quotient = mint(quotient);
        remainder = shift_down(dividend, shift);
        return [
            (
                quotient_is_negative
                ? neg(quotient)
                : quotient
            ),
            (
                remainder_is_negative
                ? neg(remainder)
                : remainder
            )
        ];
    }

    function div(dividend, divisor) {
        let temp = divrem(dividend, divisor);
        if (temp) {
            return temp[0];
        }
    }
```

Возведение целого числа в целочисленную степень сводится к простому использованию квадрата числа и метода умножения:

```
    function power(big, exponent) {
        let exp = int(exponent);
        if (exp === 0) {
            return wun;
        }
        if (is_zero(big)) {
            return zero;
        }
        if (exp === undefined || exp < 0) {
            return undefined;
        }
        let result = wun;
        while (true) {
            if ((exp & 1) !== 0) {
                result = mul(result, big);
            }
```

```
        exp = Math.floor(exp / 2);
        if (exp < 1) {
            break;
        }
        big = mul(big, big);
    }
    return mint(result);
}
```

Для сокращения дробей применим функцию `gcd`:

```
function gcd(a, b) {
    a = abs(a);
    b = abs(b);
    while (!is_zero(b)) {
        let [ignore, remainder] = divrem(a, b);
        a = b;
        b = remainder;
    }
    return a;
}
```

Нам нужны функции для преобразования чисел и строк в большие целые числа и обратно. При преобразовании в строки и из строк нужна явная поддержка десятичной системы записи, а также следующих сигнатур: двоичной, восьмеричной, шестнадцатеричной и Base32. Дополнительные сведения о Base32 можно получить по адресу crockford.com/wrmg/base32.html.

Строка `digitset` позволяет отображать числа на символы. Объект `charset` отображает символы не числа. Для шестнадцатеричных, десятичных, восьмеричных и двоичных чисел может использоваться поднабор с аналогичным символьным отображением:

```
const digitset = "0123456789ABCDEFGHJKMNPQRSTVWXYZ*~$=U";
const charset = (function (object) {
    digitset.split("").forEach(function (element, element_nr) {
        object[element] = element_nr;
    });
    return Object.freeze(object);
}(Object.create(null)));
```

Функция `make` получает число или строку и необязательное основание системы счисления, а возвращает большое целое число. Преобразование выполняется с абсолютной точностью для всех целочисленных значений:

```
function make(value, radix_2_37) {
```

Функция `make` возвращает большое целое число. Параметр `value` является строкой, а необязательный параметр `radix` — либо целочисленным значением, либо значением большого целого числа (`big_integer`):

```
    let result;
    if (typeof value === "string") {
```

```
        let radish;
        if (radix_2_37 === undefined) {
            radix_2_37 = 10;
            radish = ten;
        } else {
            if (
                !Number.isInteger(radix_2_37)
                || radix_2_37 < 2
                || radix_2_37 > 37
            ) {
                return undefined;
            }
            radish = make(radix_2_37);
        }
        result = zero;
        let good = false;
        let negative = false;
        if (value.toUpperCase().split("").every(
            function (element, element_nr) {
                let digit = charset[element];
                if (digit !== undefined && digit < radix_2_37) {
                    result = add(mul(result, radish), [plus, digit]);
                    good = true;
                    return true;
                }
                if (element_nr === sign) {
                    if (element === plus) {
                        return true;
                    }
                    if (element === minus) {
                        negative = true;
                        return true;
                    }
                }
                return digit === "_";
            }
        ) && good) {
            if (negative) {
                result = neg(result);
            }
            return mint(result);
        }
        return undefined;
    }
    if (Number.isInteger(value)) {
        let whole = Math.abs(value);
        result = [(
            value < 0
            ? minus
            : plus
        )];
        while (whole >= radix) {
            let quotient = Math.floor(whole / radix);
            result.push(whole - (quotient * radix));
            whole = quotient;
```

```
        }
        if (whole > 0) {
            result.push(whole);
        }
        return mint(result);
    }
    if (Array.isArray(value)) {
        return mint(value);
    }
}
```

Функция number преобразует большое целое число в число JavaScript. Это делается с абсолютной точностью, если значение находится в безопасном целочисленном диапазоне:

```
function number(big) {
    let value = 0;
    let the_sign = 1;
    let factor = 1;
    big.forEach(function (element, element_nr) {
        if (element_nr === 0) {
            if (element === minus) {
                the_sign = -1;
            }
        } else {
            value += element * factor;
            factor *= radix;
        }
    });
    return the_sign * value;
}
```

Функция string преобразует большое целое число в строку. Преобразование выполняется с абсолютной точностью:

```
function string(a, radix_2_thru_37 = 10) {
    if (is_zero(a)) {
        return "0";
    }
    radix_2_thru_37 = int(radix_2_thru_37);
    if (
        !Number.isSafeInteger(radix_2_thru_37)
        || radix_2_thru_37 < 2
        || radix_2_thru_37 > 37
    ) {
        return undefined;
    }
    const radish = make(radix_2_thru_37);
    const the_sign = (
        a[sign] === minus
        ? "-"
        : ""
    );
    a = abs(a);
    let digits = [];
```

```
    while (!is_zero(a)) {
        let [quotient, remainder] = divrem(a, radish);
        digits.push(digitset[number(remainder)]);
        a = quotient;
    }
    digits.push(the_sign);
    return digits.reverse().join("");
}
```

Функция подсчета заполнения подсчитывает в большом целом числе количество разрядов, содержащих 1. Это может быть использовано для вычисления расстояния Хэмминга.

Получение общего количества единичных разрядов в 32-разрядном целом числе:

```
function population_32(int32) {
```

Подсчет 16 пар разрядов с получением 16 двухразрядных счетчиков (0, 1 или 2): для каждой пары выполняется вычитание самого старшего разряда из пары, что превращает два разряда в значение счетчика:

```
//                              HL – H = count
//                              00 – 0 = 00
//                              01 – 0 = 01
//                              10 – 1 = 01
//                              11 – 1 = 10

    int32 -= (int32 >>> 1) & 0x55555555;
```

Объединяя восемь пар двухразрядных счетчиков, получаем восемь четырехразрядных счетчиков в диапазоне от 0 до 4:

```
    int32 = (int32 & 0x33333333) + ((int32 >>> 2) & 0x33333333);
```

Объединяя четыре пары четырехразрядных счетчиков, получаем четыре восьмиразрядных счетчика в диапазоне от 0 до 8. Переполнение с переходом в соседние счетчики больше невозможно, поэтому после сложения нам понадобится только одна операция наложения маски:

```
    int32 = (int32 + (int32 >>> 4)) & 0x0F0F0F0F;
```

Объединяя две пары восьмиразрядных счетчиков, получаем два шестнадцатиразрядных счетчика в диапазоне от 0 до 16:

```
    int32 = (int32 + (int32 >>> 8)) & 0x001F001F;
```

И наконец, объединяя два шестнадцатиразрядных счетчика, получаем число в диапазоне от 0 до 32:

```
    return (int32 + (int32 >>> 16)) & 0x0000003F;
}

function population(big) {
```

Подсчет общего количества разрядов, содержащих 1:

```
        return big.reduce(
            function (reduction, element, element_nr) {
                return reduction + (
                    element_nr === sign
                    ? 0
                    : population_32(element)
                );
            },
            0
        );
    }

    function significant_bits(big) {
```

Подсчет общего количества разрядов, исключая лидирующие нули:

```
        return (
            big.length > 1
            ? make((big.length - 2) * log2_radix + (32 - Math.clz32(last(big))))
            : zero
        );
    }
```

И в завершение все эти полезные объекты экспортируются в виде модуля:

```
    export default Object.freeze({
        abs,
        abs_lt,
        add,
        and,
        div,
        divrem,
        eq,
        gcd,
        is_big_integer,
        is_negative,
        is_positive,
        is_zero,
        lt,
        make,
        mask,
        mul,
        neg,
        not,
        number,
        or,
        population,
        power,
        random,
        shift_down,
        shift_up,
        significant_bits,
```

```
        signum,
        string,
        sub,
        ten,
        two,
        wun,
        xor,
        zero
    });
```

Доступ к объекту большого целого числа в вашем модуле можно получить после его импортирования:

```
import big_integer from "./big_integer.js";
```

Глава 4

Как работают большие числа с плавающей точкой

○ ○ ● ○ ○

Не геройствуй, юноша. В этом нет никакого смысла.

Харлан Поттер (Harlan Potter)

Система, работающая с большими числами, способна решать множество задач, но явно ограничена только целыми числами, а решение ряда задач неподвластно целым числам. Поэтому создадим систему с числами с плавающей точкой. Она имеет дело с тремя числами: *мантиссой*, *экспонентой*, или *порядком*, и *основанием*. Значение определяется этими тремя числами:

```
значение = мантисса * (основание ** порядок)
```

У формата IEEE 754, используемого для чисел в JavaScript, основанием является число 2. Это позволяло реализовать его на оборудовании 1950-х годов. Закон Мура убрал ограничение, делающее 2 единственно разумным основанием, поэтому рассмотрим другие возможности.

Пакет больших целых чисел (big integer package) привязан к `2 ** 24`. Если взять за основание число 16 777 216, то некоторые операции выравнивания будут просто вставлять элементы в массив или удалять их из него. Это позволит добиться весьма высокой производительности. И это согласуется с широко распространенной практикой обмена точности на производительность.

Я полагаю, что основанием должно послужить число 10. При таком основании все десятичные дроби могут получить точное представление. Важность такого выбора обусловлена тем, что большинство людей используют десятичные дроби, следовательно, система чисел с плавающей точкой, имеющая основание 10, станет наиболее подходящей для них.

Большие целые числа являются идеальным представлением для мантиссы. Большинство странностей систем с плавающей точкой связаны с ограничением размера.

Они возникают, если размер не ограничен. Поскольку это способно вызывать ошибки, следует по возможности избавиться от странностей.

Большие целые числа можно использовать также для экспоненты, но это будет излишним. Вполне хватит и чисел JavaScript. Гигабайты памяти были бы исчерпаны еще до того, как свойство `Number.MAX_SAFE_INTEGER` стало бы ограничением.

Большие числа с плавающей точкой будут представлены в виде объектов со свойствами `coefficient` и `exponent`.

После того как большие целые числа взяты на вооружение, разобраться с числами с плавающей точкой не составит особого труда:

```
import big_integer from "./big_integer.js";
```

Функция `is_big_float` используется для обнаружения объекта больших чисел с плавающей точкой:

```
function is_big_float(big) {
    return (
        typeof big === "object"
        && big_integer.is_big_integer(big.coefficient)
        && Number.isSafeInteger(big.exponent)
    );
}

function is_negative(big) {
    return big_integer.is_negative(big.coefficient);
}

function is_positive(big) {
    return big_integer.is_positive(big.coefficient);
}

function is_zero(big) {
    return big_integer.is_zero(big.coefficient);
}
```

Отдельное значение `zero` представляет все нули:

```
const zero = Object.create(null);
zero.coefficient = big_integer.zero;
zero.exponent = 0;
Object.freeze(zero);

function make_big_float(coefficient, exponent) {
    if (big_integer.is_zero(coefficient)) {
        return zero;
    }
    const new_big_float = Object.create(null);
    new_big_float.coefficient = coefficient;
    new_big_float.exponent = exponent;
    return Object.freeze(new_big_float);
}

const big_integer_ten_million = big_integer.make(10000000);
```

Функция number превращает большое число с плавающей точкой в число JavaScript. Если число выходит за пределы безопасной зоны целых чисел, точность преобразования не гарантируется. Попробуем также разобраться с другими типами.

```
function number(a) {
    return (
        is_big_float(a)
        ? (
            a.exponent === 0
            ? big_integer.number(a.coefficient)
            : big_integer.number(a.coefficient) * (10 ** a.exponent)
        )
        : (
            typeof a === "number"
            ? a
            : (
                big_integer.is_big_integer(a)
                ? big_integer.number(a)
                : Number(a)
            )
        )
    );
}
```

Нам нужны функция абсолютного значения и функция смены знака:

```
function neg(a) {
    return make_big_float(big_integer.neg(a.coefficient), a.exponent);
}

function abs(a) {
    return (
        is_negative(a)
        ? neg(a)
        : a
    );
}
```

Сложение и вычитание даются легко: мы просто складываем мантиссы, но только при одинаковых экспонентах. Если экспоненты неодинаковы, их нужно привести в соответствие. Поскольку сложение и вычитание очень похожи, я создал функцию, выполняющую как функцию сложения, так и функцию вычитания. Если функции conform_op передать аргумент bi.add, получится функция сложения чисел с плавающей точкой. А если ей передать аргумент bi.sub, получится функция вычитания чисел с плавающей точкой:

```
function conform_op(op) {
    return function (a, b) {
        const differential = a.exponent - b.exponent;
        return (
            differential === 0
            ? make_big_float(op(a.coefficient, b.coefficient), a.exponent)
            : (
                differential < 0
                ? make_big_float(
```

```
                    op(
                        big_integer.mul(
                            a.coefficient,
                            big_integer.power(big_integer.ten, -differential)
                        ),
                        b.coefficient
                    ),
                    b.exponent
                )
                : make_big_float(
                    op(
                        a.coefficient,
                        big_integer.mul(
                            b.coefficient,
                            big_integer.power(big_integer.ten, differential)
                        )
                    ),
                    a.exponent
                )
            )
        );
    };
}
const add = conform_op(big_integer.add);
const sub = conform_op(big_integer.sub);
```

Умножение реализуется еще проще. Мы просто перемножаем мантиссы и складываем экспоненты:

```
function mul(multiplicand, multiplier) {
    return make_big_float(
        big_integer.mul(multiplicand.coefficient, multiplier.coefficient),
        multiplicand.exponent + multiplier.exponent
    );
}
```

Сложность деления заключается в том, чтобы не ошибиться с моментом остановки. Проще всего остановиться в целочисленном делении. Остановка выполняется, когда заканчиваются цифры. Также просто определить момент остановки, работая с числами с плавающей точкой, имеющими фиксированный размер. Остановка выполняется, когда заканчиваются разряды, но при работе с большими целыми числами такие ограничения отсутствуют. Деление может продолжаться до нахождения точного результата, но нет никакой гарантии, что его получится достичь. Так что оставим все это на усмотрение программиста. Функция `div` получает необязательный третий аргумент, указывающий на точность результата. Указывается место десятичной точки. Младший разряд (разряд единиц) находится на нулевой позиции. Дробные позиции выражены отрицательными числами. Деление возвращает как минимум столько знаков после точки, сколько было указано. По умолчанию используется значение `-4`, то есть четыре цифры после точки:

```
function div(dividend, divisor, precision = -4) {
    if (is_zero(dividend)) {
        return zero;
    }
```

```
    if (is_zero(divisor)) {
        return undefined;
    }
    let {coefficient, exponent} = dividend;
    exponent -= divisor.exponent;
```

Масштабируем мантиссу до нужной точности:

```
    if (typeof precision !== "number") {
        precision = number(precision);
    }
    if (exponent > precision) {
        coefficient = big_integer.mul(
            coefficient,
            big_integer.power(big_integer.ten, exponent - precision)
        );
        exponent = precision;
    }
    let remainder;
    [coefficient, remainder] = big_integer.divrem(
        coefficient,
        divisor.coefficient
    );
```

Округляем результат, если это необходимо:

```
    if (!big_integer.abs_lt(
        big_integer.add(remainder, remainder),
        divisor.coefficient
    )) {
        coefficient = big_integer.add(
            coefficient,
            big_integer.signum(dividend.coefficient)
        );
    }
    return make_big_float(coefficient, exponent);
}
```

Большое число с плавающей точкой нормализовано, если экспонента максимально приблизилась к нулю без потери значимости:

```
function normalize(a) {
    let {coefficient, exponent} = a;
    if (coefficient.length < 2) {
        return zero;
    }
```

Если экспонента равна нулю, значит, число уже нормализовано:

```
    if (exponent !== 0) {
```

Если экспонента — положительное число, выполняется умножение мантиссы на `10` ** экспоненты:

```
        if (exponent > 0) {
            coefficient = big_integer.mul(
                coefficient,
                big_integer.power(big_integer.ten, exponent)
```

```
            );
            exponent = 0;
        } else {
            let quotient;
            let remainder;
```

Если же экспонента — отрицательное число, а мантисса делится на 10, выполняется деление и к экспоненте добавляется единица.

Чтобы ускорить эту работу, сначала попробуем взять из младших разрядов блоки по 10 000 000, убирая сразу семь нулей:

```
            while (exponent <= -7 && (coefficient[1] & 127) === 0) {
                [quotient, remainder] = big_integer.divrem(
                    coefficient,
                    big_integer_ten_million
                );
                if (remainder !== big_integer.zero) {
                    break;
                }
                coefficient = quotient;
                exponent += 7;
            }
            while (exponent < 0 && (coefficient[1] & 1) === 0) {
                [quotient, remainder] = big_integer.divrem(
                    coefficient,
                    big_integer.ten
                );
                if (remainder !== big_integer.zero) {
                    break;
                }
                coefficient = quotient;
                exponent += 1;
            }
        }
    }
    return make_big_float(coefficient, exponent);
}
```

Функция `make` получает большое целое число, или строку, или число JavaScript и преобразует его в большое число с плавающей точкой. Преобразование выполняется с абсолютной точностью:

```
const number_pattern = /
    ^
    ( -? \d+ )
    (?: \. ( \d* ) )?
    (?: e ( -? \d+ ) )?
    $
/;

// Capturing groups
// [1] int
```

```
// [2] frac
// [3] exp

function make(a, b) {

// (big_integer)
// (big_integer, exponent)
// (string)
// (string, radix)
// (number)

    if (big_integer.is_big_integer(a)) {
        return make_big_float(a, b || 0);
    }
    if (typeof a === "string") {
        if (Number.isSafeInteger(b)) {
            return make(big_integer.make(a, b), 0);
        }
        let parts = a.match(number_pattern);
        if (parts) {
            let frac = parts[2] || "";
            return make(
                big_integer.make(parts[1] + frac),
                (Number(parts[3]) || 0) - frac.length
            );
        }
    }
```

Если работа ведется с числом, оно принимает вид числа с экспонентой с основанием 2 и мантиссой, а затем реконструируется в виде точного числа с плавающей точкой:

```
    if (typeof a === "number" && Number.isFinite(a)) {
        if (a === 0) {
            return zero;
        }
        let {sign, coefficient, exponent} = deconstruct(a);
        if (sign < 0) {
            coefficient = -coefficient;
        }
        coefficient = big_integer.make(coefficient);
```

Если экспонента имеет отрицательное значение, можно выполнить деление на `2 ** abs(exponent)`:

```
        if (exponent < 0) {
            return normalize(div(
                make(coefficient, 0),
                make(big_integer.power(big_integer.two, -exponent), 0),
                b
            ));
        }
```

Если экспонента больше нуля, мантиссу можно умножить на 2 ** экспоненты:

```
            if (exponent > 0) {
                coefficient = big_integer.mul(
                    coefficient,
                    big_integer.power(big_integer.two, exponent)
                );
                exponent = 0;
            }
            return make(coefficient, exponent);
        }
        if (is_big_float(a)) {
            return a;
        }
    }
```

Функция `string` преобразует большое число с плавающей точкой в строку. Преобразование выполняется с абсолютной точностью. Основная часть работы заключается во вставке десятичного знака и заполнении нулевых позиций. Аналогичная функция для двоичной плавающей точки была бы значительно сложнее.

```
    function string(a, radix) {
        if (is_zero(a)) {
            return "0";
        }
        if (is_big_float(radix)) {
            radix = normalize(radix);
            return (
                (radix && radix.exponent === 0)
                ? big_integer.string(integer(a).coefficient, radix.coefficient)
                : undefined
            );
        }
        a = normalize(a);
        let s = big_integer.string(big_integer.abs(a.coefficient));
        if (a.exponent < 0) {
            let point = s.length + a.exponent;
            if (point <= 0) {
                s = "0".repeat(1 - point) + s;
                point = 1;
            }
            s = s.slice(0, point) + "." + s.slice(point);
        } else if (a.exponent > 0) {
            s += "0".repeat(a.exponent);
        }
        if (big_integer.is_negative(a.coefficient)) {
            s = "-" + s;
        }
        return s;
    }
```

Существует два соглашения по представлению десятичного знака: в виде точки (`.`) и в виде запятой (`,`). В большинстве стран используется один из этих знаков. В преде-

лах той или иной страны вид знака не имеет значения. Оба они работают должным образом. Но такое разночтение препятствует международным связям, поскольку у числа вида 1,024 может быть различное толкование. Могу предсказать, что в итоге все остановятся на использовании точки, приняв ее за международный стандарт, поскольку в языках программирования применяется точка, а основная часть информационных потоков, проходящих через наши программы, написана на этих языках.

Функция `scientific` преобразует большое число с плавающей точкой в строку с e-нотацией:

```
function scientific(a) {
    if (is_zero(a)) {
        return "0";
    }
    a = normalize(a);
    let s = big_integer.string(big_integer.abs(a.coefficient));
    let e = a.exponent + s.length - 1;
    if (s.length > 1) {
        s = s.slice(0, 1) + "." + s.slice(1);
    }
    if (e !== 0) {
        s += "e" + e;
    }
    if (big_integer.is_negative(a.coefficient)) {
        s = "-" + s;
    }
    return s;
}
```

И наконец, все эти полезные объекты экспортируются в виде модуля:

```
export default Object.freeze({
    abs,
    add,
    div,
    eq,
    fraction,
    integer,
    is_big_float,
    is_negative,
    is_positive,
    is_zero,
    lt,
    make,
    mul,
    neg,
    normalize,
    number,
    scientific,
    string,
    sub,
    zero
});
```

Получается библиотека, подходящая для вычислений, обработки финансовых данных или любой другой работы, требующей корректного обращения с десятичными дробями. На данной стадии в ней есть только самое необходимое, но ее можно дополнить всеми функциями и операторами, которые когда-либо могут понадобиться.

Я воспользовался этой библиотекой, чтобы установить истину в главе 2. Несмотря на ее высокую эффективность, я полагаю, что многим приложениям больше подошел бы стандартный тип десятичных чисел с плавающей точкой, имеющих фиксированный размер. Проблема с числовым типом, имеющимся в JavaScript, кроется не в ограниченных диапазоне или точности. Дело в том, что он не в состоянии точно представить числа, наиболее интересные для практического применения людьми, — числа, состоящие из десятичных цифр. Поэтому более удачным выбором для следующего языка может стать что-то вроде DEC64: www.DEC64.com.

Ни двоичные числа с плавающей точкой, ни десятичные числа с плавающей точкой не могут в точности дать представление, к примеру, о результате действия `100/3`. Эта проблема будет рассматриваться и далее.

Глава 5

Как работают большие рациональные числа

○ ○ ● ○ ●

> Я изучал только обязательные предметы. ...
> Сначала мы, как полагается, Чихали и Пищали[1].
> А потом принялись за четыре действия
> Арифметики: Скольжение, Причитание, Умиление
> и Изнеможение[2].
>
> *Черепаха Квази*
> *(«Алиса в Стране чудес»[3])*

Рациональным называется число, которое может быть выражено отношением двух целых чисел. Если два целых числа являются большими целыми числами, то это способно стать весьма привлекательным представлением чисел. Ими можно точно выразить то же самое, что и двоичными числами с плавающей точкой. И они позволяют точно выразить все рациональные числа, чего не могут сделать другие представления.

Система рациональных чисел работает с двумя числами — *числителем* (numerator) и *знаменателем* (denominator). *Значение* (value) определяется двумя числами:

```
значение = числитель / знаменатель
```

Наши рациональные числа — это объекты со свойствами `numerator` и `denominator`, каждое из которых является большим целым числом. Знак значения определяется знаком числителя. Знаменатель не должен быть отрицательным.

---

1 В английском созвучно словам «читали» и «писали». — *Примеч. пер.*

2 В английском созвучно словам «сложение», «вычитание», «умножение», «деление». — *Примеч. пер.*

3 В переводе Н. Демуровой.

Эти обстоятельства допускают применение весьма привлекательных способов выполнения арифметических действий. Можно выстраивать их реализацию на основе использования больших целых чисел:

```
import big_integer from "./big_integer.js";
```

Начнем с некоторых предикатных функций:

```
function is_big_rational(a) {
    return (
typeof a === "object"
&& big_integer.is_big_integer(a.numerator)
&& big_integer.is_big_integer(a.denominator)
    );
}

function is_integer(a) {
    return (
big_integer.eq(big_integer.wun, a.denominator)
|| big_integer.is_zero(
            big_integer.divrem(a.numerator, a.denominator)[1]
        )
    );
}

function is_negative(a) {
    return big_integer.is_negative(a.numerator);
}
```

А это константы, которые я счел полезными. Можно без особого труда определить дополнительные константы:

```
function make_big_rational(numerator, denominator) {
const new_big_rational = Object.create(null);
new_big_rational.numerator = numerator;
new_big_rational.denominator = denominator;
return Object.freeze(new_big_rational);
}
const zero = make_big_rational(big_integer.zero, big_integer.wun);
const wun = make_big_rational(big_integer.wun, big_integer.wun);
const two = make_big_rational(big_integer.two, big_integer.wun);
```

Понадобятся также функции абсолютного значения и отрицания. В соответствии с соглашением знак определяется числителем. Знаменатель всегда положительное число.

```
function neg(a) {
    return make(big_integer.neg(a.numerator), a.denominator);
}

function abs(a) {
    return (
        is_negative(a)
        ? neg(a)
        : a
    );
}
```

Сложение и вычитание реализуются весьма просто. Если знаменатели равны, можно складывать или вычитать числители. В противном случае дополнительно выполняются два умножения, сложение и еще одно умножение, поскольку:

```
(a / b) + (c / d) = ((a * d) + (b * c)) / (b * d)
```

Схожесть сложения и вычитания позволила мне создать функцию, выполняющую функции `add` и `sub`:

```
function conform_op(op) {
    return function (a, b) {
        try {
            if (big_integer.eq(a.denominator, b.denominator)) {
                return make(
                    op(a.numerator, b.numerator),
                    a.denominator
                );
            }
            return normalize(make(
                op(
                    big_integer.mul(a.numerator, b.denominator),
                    big_integer.mul(b.numerator, a.denominator)
                ),
                big_integer.mul(a.denominator, b.denominator)
            ));
            } catch (ignore) {
        }
    };
}
const add = conform_op(big_integer.add);
const sub = conform_op(big_integer.sub);
```

Рациональное число можно увеличить на единицу, добавив знаменатель к числителю.

```
function inc(a) {
    return make(
        big_integer.add(a.numerator, a.denominator),
        a.denominator
    );
}

function dec(a) {
    return make(
        big_integer.sub(a.numerator, a.denominator),
        a.denominator
    );
}
```

Умножение также не вызывает затруднений. Мы просто перемножаем числители и знаменатели. Деление представляется тем же умножением, но с инвертированным вторым аргументом. В системе рациональных чисел выполнять деление в столбик практически не приходится. Мы просто увеличиваем делитель:

```
function mul(multiplicand, multiplier) {
    return make(
```

```
            big_integer.mul(multiplicand.numerator, multiplier.numerator),
            big_integer.mul(multiplicand.denominator, multiplier.denominator)
        );
    }

    function div(a, b) {
        return make(
            big_integer.mul(a.numerator, b.denominator),
            big_integer.mul(a.denominator, b.numerator)
        );
    }

    function remainder(a, b) {
        const quotient = div(normalize(a), normalize(b));
        return make(
            big_integer.divrem(quotient.numerator, quotient.denominator)[1]
        );
    }

    function reciprocal(a) {
        return make(a.denominator, a.numerator);
    }

    function integer(a) {
        return (
            a.denominator === wun
            ? a
            : make(big_integer.div(a.numerator, a.denominator), big_integer.wun)
        );
    }

    function fraction(a) {
        return sub(a, integer(a));
    }
```

Функция нормализации `normalize` сокращает дробь, избавляя ее от общих множителей. Разложение на множители больших чисел — весьма непростая задача. К счастью, нам не нужно выполнять разложение на множители для операции сокращения. Следует просто найти наибольший общий делитель, на который затем и выполняется сокращение.

Фактически нормализация не нужна. Значение в результате нормализации не изменяется. Большие целые числа внутри рационального объекта могут быть уменьшены, и это способно сократить потребность в памяти (что редко играет важную роль) и ускорить выполнение последующих арифметических операций.

```
    function normalize(a) {
```

Нормализация большого рационального числа выполняется делением двух его компонентов на наибольший общий делитель (НОД). Если НОД равен единице, значит, число уже нормализовано:

```
let {numerator, denominator} = a;
if (big_integer.eq(big_integer.wun, denominator)) {
    return a;
}
    let g_c_d = big_integer.gcd(numerator, denominator);
    return (
        big_integer.eq(big_integer.wun, g_c_d)
? a
: make(
big_integer.div(numerator, g_c_d),
big_integer.div(denominator, g_c_d)
    )
);
}
```

Чтобы определить равенство двух значений, нормализация не понадобится. Если:

```
a / b = c / d
```

то:

```
a * d = b * c
```

даже если они не нормализованы.

```
function eq(comparahend, comparator) {
    return (
        comparahend === comparator
        ? true
        : (
big_integer.eq(comparahend.denominator, comparator.denominator)
? big_integer.eq(comparahend.numerator, comparator.numerator)
: big_integer.eq(
big_integer.mul(comparahend.numerator, comparator.denominator),
big_integer.mul(comparator.numerator, comparahend.denominator)
            )
        )
    );
}

function lt(comparahend, comparator) {
    return (
is_negative(comparahend) !== is_negative(comparator)
? is_negative(comparator)
: is_negative(sub(comparahend, comparator))
    );
}
```

Функция `make` принимает два аргумента и создает объект, содержащий числитель `numerator` и знаменатель `denominator`. Преобразование выполняется с абсолютной точностью.

Функция принимает одно или два больших целых числа, а также строки вида `"33 1/3"` и `"98.6"` и выполняет их правильное преобразование. Она также принимает

любое конечное число JavaScript и превращает его в рациональное без дополнительных потерь.

```
const number_pattern = /
    ^
    ( -? )
    (?:
        ( \d+ )
        (?:
            (?:
                \u0020 ( \d+ )
            )?
            \/
            ( \d+ )
        |
            (?:
                \. ( \d* )
            )?
            (?:
                e ( -? \d+ )
            )?
        )
    |
        \. (\d+)
    )
    $
/;

function make(numerator, denominator) {
```

Если получены два аргумента, то оба будут преобразованы в большие целые числа. Возвращаемым значением станет объект, содержащий числитель и знаменатель.

При вызове с одним аргументом будет сделана попытка его осмысления. Если аргумент имеет строковое значение, будет предпринята попытка его парсинга в качестве смешанной дроби или десятичного литерала. В противном случае предполагается, что недостающим аргументом была единица.

```
    if (denominator !== undefined) {
```

Создание рационального числа из числителя и знаменателя — можно передать функции большие целые числа, простые целые числа или строки.

```
        numerator = big_integer.make(numerator);
```

Если числитель равен нулю, то знаменатель не нужен.

```
        if (big_integer.zero === numerator) {
            return zero;
        }
        denominator = big_integer.make(denominator);
        if (
!big_integer.is_big_integer(numerator)
```

```
|| !big_integer.is_big_integer(denominator)
|| big_integer.zero === denominator
        ) {
            return undefined;
        }
```

Если знаменатель — отрицательное число, знак переносится в числитель.

```
        if (big_integer.is_negative(denominator)) {
            numerator = big_integer.neg(numerator);
            denominator = big_integer.abs(denominator);
        }
        return make_big_rational(numerator, denominator);
    }
```

А если аргумент является строковым значением? В таком случае выполняется его парсинг.

```
    if (typeof numerator === "string") {
        let parts = numerator.match(number_pattern);
        if (!parts) {
                return undefined;
        }

// Определение групп:
// [1] sign
// [2] integer
// [3] top
// [4] bottom
// [5] frac
// [6] exp
// [7] naked frac

    if (parts[7]) {
        return make(
            big_integer.make(parts[1] + parts[7]),
            big_integer.power(big_integer.ten, parts[7].length)
        );
    }
    if (parts[4]) {
        let bottom = big_integer.make(parts[4]);
            if (parts[3]) {
                return make(
                    big_integer.add(
                        big_integer.mul(
                            big_integer.make(parts[1] + parts[2]),
                            bottom
                        ),
                        big_integer.make(parts[3])
                    ),
                    bottom
                );
            }
            return make(parts[1] + parts[2], bottom);
        }
```

```
        let frac = parts[5] || "";
        let exp = (Number(parts[6]) || 0) - frac.length;
        if (exp < 0) {
                return make(
                    parts[1] + parts[2] + frac,
                    big_integer.power(big_integer.ten, -exp)
                );
            }
            return make(
                big_integer.mul(
                    big_integer.make(parts[1] + parts[2] + parts[5]),
                    big_integer.power(big_integer.ten, exp)
                ),
                big_integer.wun
            );
        }
```

Аргумент является числом? Тогда оно подвергается разбору и реконструкции.

```
        if (typeof numerator === "number" && !Number.isSafeInteger(numerator)) {
            let {sign, coefficient, exponent} = deconstruct(numerator);
            if (sign < 0) {
                coefficient = -coefficient;
            }
            coefficient = big_integer.make(coefficient);
            if (exponent >= 0) {
                return make(
                    big_integer.mul(
                        coefficient,
                        big_integer.power(big_integer.two, exponent)
                    ),
                    big_integer.wun
                );
            }
            return normalize(make(
                coefficient,
                big_integer.power(big_integer.two, -exponent)
            ));
        }
        return make(numerator, big_integer.wun);
    }
```

Функция `number` преобразует большое рациональное число в число JavaScript. Преобразование не гарантирует абсолютную точность, если число находится за пределами безопасной целочисленной зоны:

```
    function number(a) {
        return big_integer.number(a.numerator) / big_integer.number(a.demoninator);
    }
```

Функция `string` преобразует большое рациональное число в строку. Преобразование выполняется с абсолютной точностью:

```
    function string(a, nr_places) {
        if (a === zero) {
```

```
            return "0";
        }
        let {numerator, denominator} = normalize(a);
```

Числитель делится на знаменатель. Если действие выполнено без остатка, значит, получен нужный результат.

```
        let [quotient, remains] = big_integer.divrem(numerator, denominator);
        let result = big_integer.string(quotient);
        if (remains !== big_integer.zero) {
```

Если предоставлен аргумент `nr_places`, результат имеет десятичный формат. Остатки сводятся к масштабу степени 10, и выполняется целочисленное деление. Если остаток составляет не менее половины знаменателя, производится округление в бóльшую сторону.

```
            remains = big_integer.abs(remains);
            if (nr_places === undefined) {
                let [fractus, residue] = big_integer.divrem(
                    big_integer.mul(
                        remains,
                        big_integer.power(big_integer.ten, nr_places)
                    ),
                    denominator
                );
                if (!big_integer.abs_lt(
                    big_integer.mul(residue, big_integer.two),
                    denominator
                )) {
                    fractus = big_integer.add(fractus, big_integer.wun);
                }
                result += "." + big_integer.string(fractus).padStart(
                    big_integer.number(nr_places),
                    "0"
                );
            } else {
```

Результат будет в виде смешанной дроби.

```
                result = (
                    (
                        result === "0"
                        ? ""
                        : result + " "
                    )
                    + big_integer.string(remains)
                    + "/"
                    + big_integer.string(denominator)
                );
            }
        }
        return result;
    }
```

Все эти полезные объекты экспортируются в виде модуля:

```
export default Object.freeze({
    abs,
    add,
    dec,
    div,
    eq,
    fraction,
    inc,
    integer,
    is_big_rational,
    is_integer,
    is_negative,
    lt,
    make,
    mul,
    neg,
    normalize,
    number,
    wun,
    reciprocal,
    remainder,
    string,
    sub,
    two,
    zero
});
```

Получить в модуле доступ к большому рациональному числу можно путем его импортирования:

```
import big_rational from "./big_rational.js";
```

Эта библиотека рациональных чисел весьма компактна, проста, медлительна и на удивление эффективна. Она все еще не в состоянии в точности отобразить число ϖ или квадратный корень из 2, но способна приблизиться к их отображению в нужной вам степени. Например:

```
const pi = big_rational.make(
    "31415926535897932384626433832795028841971693993751058209749445923078164 06",
    "10000000000000000000000000000000000000000000000000000000000000000000000000"
);
const sqrt_two = big_rational.make(
    "14142135623730950488016887242096980785696718753769480731766797379907324 78",
    "10000000000000000000000000000000000000000000000000000000000000000000000000"
);
```

Такой формат подойдет для 72 знаков после десятичной точки. Если нужно больше, можно получить и такой результат. С увеличением точности падает производительность. Зачастую мы не задумываясь жертвуем точностью ради производительности.

Эта библиотека позволяет выполнять в JavaScript вычисления, которые, как уже говорилось, не могут быть произведены в чистом JavaScript. Поэтому фактически не возникает надобности в добавлении в JavaScript новых числовых типов. Все, что нужно, реально сделать средствами самого языка.

Да, при этом возникают некоторые неудобства. Вместо `a + b` приходится делать запись `big_rational.add(a, b)`. Не думаю, что это кого-то очень напрягает, особенно если основной интерес — получение качественных результатов, а не экономия времени при наборе текста. Роль синтаксиса сильно преувеличена. Но если вы чрезвычайно озабочены синтаксической поддержкой, транспилятор может с легкостью переписать радующие ваш взор языковые конструкции в вызовы `big_rational`.

Глава 6

Как работают булевы значения

○ ○ ● ● ○

> Истина есть истина. У вас не может быть собственных представлений об истине.
>
> *Питер Шикели (Peter Schickele)*

Булев тип был назван в честь Джорджа Буля (George Boole), английского математика, разработавшего систему алгебраической логики. Клод Шеннон (Claude Shannon) адаптировал эту систему для проектирования цифровых схем. Именно поэтому мы называем компьютерные схемы логикой.

Булев тип состоит всего из двух значений: истины (`true`) и лжи (`false`). Обычно булевы значения генерируются операторами сравнения, манипулируют ими с помощью логических операторов, а потребляют их тернарный оператор и условная часть инструкций `if`, `do`, `for` и `while`.

Оператор `typeof` возвращает `"boolean"`, когда его операнд имеет значение `true` или `false`.

Операторы отношений	
=== (знак равенства, знак равенства, знак равенства)	Равно
!== (восклицательный знак, знак равенства, знак равенства)	Не равно
< (знак меньше)	Меньше
<= (знак меньше, знак равенства)	Меньше или равно
> (знак больше)	Больше
>= (знак больше, знак равенства)	Больше или равно

К великому сожалению, оператор равенства имеет вид `===`, а не `=`. Еще печальнее то, что оператор неравенства имеет вид `!==`, а не `≠`. По сути, все эти операторы работают ожидаемо, за исключением того, что при их использовании возникает масса абсурдных моментов:

```
undefined < null            // false
undefined > null            // false
undefined === null          // false
```

```
NaN === NaN                 // false
NaN !== NaN                 // true

"11" < "2"                  // true
"2" < 5                     // true
5 < "11"                    // true
```

В целом операторы `===` и `!==` работают правильно, за исключением случая, когда оба операнда имеют значение `NaN`. Операторы `===` и `!==` могут применяться для определения нулевых (`null`) или неопределенных (`undefined`) значений или любого значения, отличного от `NaN`. Чтобы протестировать *x* на значение `NaN`, нужно всегда использовать выражение `Number.isNaN(`*x*`)`.

Оператор `===` не должен применяться для тестирования условия завершения цикла, если переменная инициирования очередной итерации не относится к безопасному целочисленному диапазону. И даже тогда безопаснее воспользоваться оператором `>=`.

Операторы `<`, `<=`, `>` и `<=` в целом работают правильно, когда оба операнда являются или строками, или числами. В большинстве других случаев результат абсурдный. При сравнениях следует избегать смешанных типов. JavaScript не запрещает смешивать типы подобным образом, поэтому стоит полагаться на собственную дисциплинированность.

В JavaScript есть и еще менее надежные операторы сравнения. Я рекомендую отказаться от использования оператора `==`, а также `!=`. Они перед сравнением выполняют приведение типов, поэтому могут давать ложные срабатывания (и ложные несрабатывания). Вместо них следует пользоваться операторами `===` и `!==`.

Выражения, не являющиеся булевыми, но выдающие булевы результаты

В JavaScript имеется превосходный булев тип, но им невозможно воспользоваться в полной мере. Перечислю места в программе, где булевы выражения приносят наибольшую пользу:

- в качестве условия положения инструкции `if`;
- в качестве условия положения инструкции `while`;
- в качестве условия положения инструкции `for`;
- в качестве условия положения инструкции `do`;
- в качестве операнда оператора `!` *(восклицательный знак)*;
- в качестве обоих операндов оператора `&&` *(амперсанд, амперсанд)*;
- в качестве обоих операндов оператора `||` *(вертикальная черта, вертикальная черта)*;
- в качестве первого операнда тернарного оператора `?` *(восклицательный знак)* `:` *(двоеточие)*;

- в качестве возвращаемого значения аргумента функции к `Array`-методам `filter`, `find`, `findIndex`, `indexOf`.

В качественно проработанном языке в этих местах было бы разрешено использовать только булевы значения. А в JavaScript допускается применение любого типа данных. Получается, что все значения в языке относятся к булевоподобному типу, в котором они вычисляются либо как правдивые (truthy), либо как лживые (falsy).

К лживым значениям относятся:

- `false`;
- `null`;
- `undefined`;
- `""` (пустая строка);
- `0`;
- `NaN`.

Правдивыми являются все остальные значения, включая пустые объекты, пустые массивы и строки, которые могут выглядеть лживыми, например `"false"` и `"0"`.

Лживые значения зачастую ведут себя как значения `false`, но большинство из них, строго говоря, не являются `false`. Правдивые значения зачастую ведут себя как значения `true`, но большинство из них не являются `true`. Такая булевоподобность была ошибкой, но допущена она не случайно. Это сделано намеренно, чтобы разрешить в JavaScript идиомы языка C.

Язык C является неадекватно типизированным. Одно и то же значение в нем применяется для представления `0`, `FALSE`, `NULL`, конца строки и др. Для языка C они все одинаковы. Поэтому на месте условия инструкции `if` в языке C определяется, является значение нулем (`0`) или нет. В языке C есть стиль программирования, использующий это обстоятельство, чтобы укоротить условие.

В JavaScript имеется объявленный булев тип, но булевоподобность сильно его обесценивает. Условие должно быть либо `true`, либо `false`. Любые другие значения должны быть ошибкой, которая в идеале проявляется на этапе компиляции. Но к JavaScript это не относится. Условные выражения могут быть такими же лаконичными и загадочными, как и в языке C. Значения, попадающие на место условий совершенно случайно, как ошибки не помечаются. Однако они способны придать выполнению программы совершенно неожиданное направление. Язык Java требует, чтобы условия были булевыми значениями, устраняя тем самым данный класс ошибок. Мне хочется, чтобы то же самое делалось и в JavaScript.

Рекомендую создавать программы в предположении, что JavaScript в данном плане разработан корректно. Во всех условиях используйте булевы выражения. Создавая программы в стиле удачно разработанного языка, вы станете выдавать более качественный результат.

Логические операторы

Нелепости не обошли стороной и логические операторы.

! (восклицательный знак)	Логическое НЕ	Если операнд правдивый (truthy), результат — false. Если операнд лживый (falsy), результат — true
&& (знак амперсанда, знак амперсанда)	Логическое И	Если первый операнд лживый (falsy), результатом станет значение первого операнда, а второй операнд вычисляться не будет. Если первый операнд правдивый (truthy), результат будет определен значением второго операнда
‖ (вертикальная черта, вертикальная черта)	Логическое ИЛИ	Если операнд правдивый (truthy), результатом будет значение первого операнда, а второй операнд вычисляться не будет. Если первый операнд лживый (falsy), результат станет определяться значением второго операнда

Not!

Логические выражения могут усложняться. Упростить их могут формальные преобразования. К сожалению, булевоподобность и `NaN` способны привести к тому, что формальные преобразования станут выдавать ошибки.

Обычно имеет смысл упрощать двойные отрицания. В логической системе:

```
!!p === p
```

что в JavaScript истинно, только когда `p` является булевым значением. Если `p` относится к какому-либо другому типу, то `!!p` эквивалентно выражению `Boolean(p)` и не обязательно эквивалентно `p`.

У некоторых операторов сравнения есть антиподы, например, `<` является антиподом `>=`, а `>` является антиподом `<=`. Поэтому есть возможность упростить `!(a < b)`, применив выражение `a >= b`, поскольку:

```
!(a === b) === (a !== b)
!(a <=  b) === (a >   b)
!(a >   b) === (a <=  b)
!(a >=  b) === (a <   b)
```

Если либо `a`, либо `b` имеет значение `NaN`, то преобразование кода оказывается неудачным. Дело в том, что сравнение любого числа с `NaN` выдает результат `false` независимо от оператора сравнения. Следовательно:

```
7 < NaN                    // false
NaN < 7                    // false
!(7 < NaN) === 7 >= NaN    // false
```

Было бы разумнее, чтобы значение `NaN` было меньше всех остальных чисел или сравнение всех прочих чисел с `NaN` выдавало исключение. Но вместо этого в JavaScript

получается бессмыслица. Единственной значимой операцией в отношении `NaN` является `Number.isNaN(NaN)`. Во всех других контекстах от использования `NaN` нужно отказываться.

В упрощении логических выражений могут помочь законы де Моргана. Я ими пользуюсь довольно часто:

```
!(p && q) === !p || !q
!(p || q) === !p && !q
```

В JavaScript эти законы имеют силу, только если `p` и `q` не пострадали от бессмыслицы. Выражений, не являющихся булевыми, но выдающих булевы результаты, следует избегать. Используйте вместо них булевы выражения.

Глава 7

Как работают массивы

○○●●●

> Здесь все пронумеровано. Монстр проходит под нулевым номером.
>
> *Контролер планеты X*

Массивы — это самая почтенная структура данных. Массив представляет собой непрерывную область памяти, разделенную на равные части, каждая из которых связана с целым числом, по которому к ней можно получить быстрый доступ. В первый выпуск JavaScript массивы включить не удалось. Но весьма высокая эффективность объектов JavaScript практически нивелировала это упущение. Если проигнорировать вопросы производительности, объектам подвластно все, на что способны массивы.

С тех пор ситуация не изменилась. Индексом для массива способна послужить любая строка. В действительности массивы JavaScript являются объектами. В современном JavaScript массивы слегка отличаются от объектов четырьмя особенностями.

- У массивов есть волшебное свойство — их длина (`length`). Длина массива не обязательно отражает количество его элементов. Вместо этого она определяется как самое большое порядковое число элементов плюс один. Тем самым подтверждается то, что массивы JavaScript являются настоящими массивами, что позволяет им подвергаться обработке с использованием той же самой архаичной инструкции `for`, которую можно отыскать в программе на языке C полувековой давности.
- Массивы наследуются из прототипа `Array.prototype`, который содержит намного более богатую коллекцию методов, чем прототип `Object.prototype`.
- Массивы создаются с применением литералов массивов, а не литералов объектов. Литералы массивов синтаксически намного проще: от нуля и более выражений, разделенных запятыми (`,`), помещаются между левой (`[`) и правой (`]`) квадратными скобками.
- Отношение к массивам в JSON совершенно иное, нежели к объектам, хотя в JavaScript к ним в основном одинаковое отношение.

В самом JavaScript массивы создают некоторую путаницу. Когда оператору определения типа `typeof` попадается массив, он возвращает строку `"object"`, что в корне

неверно. Чтобы определить принадлежность значения к массиву, вместо него следует использовать функцию `Array.isArray(`*значение*`)`:

```
const what_is_it = new Array(1000);
typeof what_is_it                       // "object"
Array.isArray(what_is_it)               // true
```

Начало отсчета

Как только был изобретен счет, люди стали нумеровать предметы, начиная с тех слов, которые использовали для обозначения единицы. В середине 1960-х годов небольшая, но весьма влиятельная группа программистов установила, что мы вместо этого должны начинать отсчет с нуля. Все остальное человечество, включая большинство математиков, по-прежнему начинало считать с единицы. В начале отсчета математики обычно ставили цифру 0, но первый элемент упорядоченного набора большинство из них обозначало цифрой 1. Как так у них получилось, до сих пор остается загадкой.

Аргумент в пользу начала отсчета с нуля имеется, но он настолько слаб, что им можно пренебречь. Есть еще аргумент, согласно которому начало отсчета с нуля более корректно, так как уменьшает количество ошибок, занижения или завышения на единицу. Однако он также весьма сомнителен. Похоже, для объяснения того, почему программисты считают иначе, нежели все остальные, должна быть веская причина. Вероятно, однажды мы ее узнаем.

В JavaScript это обстоятельство отражается на нумерации элементов массива и в меньшей степени — на нумерации символов в строке. Мысль о том, что массивы должны обрабатываться поэлементно, возникла не позднее, чем был разработан Фортран. Более современная идея заключается в обработке массивов функциональным образом. Это упрощает код за счет упразднения явного управления циклом и создает потенциал для распределения работы по нескольким процессорам.

Качественно написанная программа на качественно разработанном языке не должна отвлекаться на то, с чего начинается подсчет элементов в массиве, с нуля или с единицы. Не стану уверять вас в том, что JavaScript — качественно разработанный язык, но все же он существенно улучшен, по сравнению с Фортраном, и, отказавшись по факту от модели Фортрана, нам не следует беспокоиться о том, как именно нумеруются элементы.

Но иногда приходится обращать на это внимание. Слово «первый» становится проблемным, поскольку ассоциируется со словом «один», поэтому вместо него я использую слово «нулевой» (zeroth). Это делает мою систему начала отсчета понятнее:

```
[0]     нулевой     0th
[1]     первый      1th
[2]     второй      2th
[3]     третий      3th
```

Если продолжить, то кажется, что на четвертом, пятом и шестом смысл несколько теряется, но со временем, когда наименьшие целые числа останутся позади, станет яснее, каким было начало отсчета.

Инициализация

Новый массив можно создать двумя способами:

- указав литерал массива;
- применив выражение `new Array(`*`целочисленное_значение`*`)`.

```
let my_little_array = new Array(10).fill(0);
         // получился my_little_array вида [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
let same_thing = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

my_little_array === same_thing                          // false
```

Обратите внимание на то, что `my_little_array` и `same_thing` абсолютно одинаковы. Но они также являются двумя отдельными, уникальными значениями. В отличие от строк, но подобно объектам, идентичные массивы считаются равными только в том случае, если это фактически один и тот же массив.

Стеки и очереди

У массивов есть методы, которые ведут себя с массивом как со стеком. Метод `pop` удаляет последний элемент массива и возвращает его значение. Метод `push` добавляет к массиву новый элемент.

Стеки зачастую используются в интерпретаторах и калькуляторах.

```
function make_binary_op(func) {
    return function (my_little_array) {
        let wunth = my_little_array.pop();
        let zeroth = my_little_array.pop();
        my_little_array.push(func(zeroth, wunth));
        return my_little_array;
    };
}

let addop = make_binary_op(function (zeroth, wunth) {
    return zeroth + wunth;
});

let mulop = make_binary_op(function (zeroth, wunth) {
    return zeroth * wunth;
});

let my_little_stack = [];           // my_little_stack имеет значение []
my_little_stack.push(3);            // my_little_stack имеет значение [3]
my_little_stack.push(5);            // my_little_stack имеет значение [3, 5]
my_little_stack.push(7);            // my_little_stack имеет значение [3, 5, 7]
```

```
mulop(my_little_stack);              // my_little_stack имеет значение [3, 35]
addop(my_little_stack);              // my_little_stack имеет значение [38]
let answer = my_little_stack.pop();  // my_little_stack имеет значение [],
                                     // answer          имеет значение 38
```

Метод `shift` аналогичен методу `pop`, за исключением того, что он удаляет нулевой элемент и возвращает его значение. Метод со странным названием `unshift` подобен методу `push`, за исключением того, что он вставляет новый элемент не в конец, а в начало массива. Методы `shift` и `unshift` могут работать намного медленнее методов `pop` и `push`, особенно с большим массивом. Совместное использование `shift` и `push` создает очередь, где новые элементы добавляются в конец, а забираются в конечном счете с начала.

Выполнение поиска

В JavaScript имеются методы для сквозного поиска в массиве. Метод `indexOf` получает значение, которое сравнивает с каждым элементом массива с самого начала. Если значение совпадает с элементом, поиск останавливается и возвращается порядковый номер элемента.

Если совпадение не обнаружено, метод возвращает `-1`. Заметьте, что это предпосылка для ошибки типа, поскольку число `-1` весьма похоже на любое другое число. Если воспользоваться возвращаемым значением `indexOf` без предварительной проверки на `-1`, то вычисления, в которых это значение применяется, могут пойти не так, как задумано, без какого-либо предупреждения. В JavaScript есть множество ничтожно малых (bottom) значений. Одно из них и нужно взять.

Функция `lastIndexOf` похожа на `indexOf`, за исключением того, что начинает свою работу с конца массива и ведет поиск в обратном направлении. В ней в качестве кода неудачного завершения также используется `-1`.

Функция `includes` похожа на `indexOf`, но она возвращает `true`, если значение найдено, и `false` — в противном случае.

Свертка

Метод `reduce` свертывает массив в одно значение. Он получает функцию, которая получает два аргумента. Метод `reduce` раз за разом вызывает эту функцию с парами значений, пока не останется всего одно значение, которое и станет результатом.

Метод `reduce` можно создать двумя способами. Один из них заключается в том, чтобы заставить его вызывать функцию для каждого элемента массива. Для начала нужно предоставить исходное значение:

```
function add(reduction, element) {
    return reduction + element;
}

let my_little_array = [3, 5, 7, 11];

let total = my_little_array.reduce(add, 0);     // total имеет значение 26
```

Значение свертки передается функции add для каждого элемента наряду с текущим элементом массива. В качестве исходного значения свертки было явно передано число 0. Функция add видит аргументы в такой последовательности:

```
(0, 3)                                  // 3
(3, 5)                                  // 8
(8, 7)                                  // 15
(15, 11)                                // 26
```

Каждое значение, возвращаемое add, становится значением свертки для следующего вызова add.

Исходное значение свертки не всегда должно быть нулевым. Если reduce передается функция multiply, то исходным значением свертки должно быть число 1. Если reduce передается Math.max, исходным значением свертки должно быть -Infinity. Здесь есть шанс допустить ошибку, поэтому при выборе исходного значения свертки нужно быть внимательными.

Другой способ создания reduce не требует исходного значения свертки. Вместо этого функция вызывается на один раз меньше. При первом вызове она получает нулевой и первый элементы. Нулевой элемент становится ее исходным значением.

```
total = my_little_array.reduce(add);            // 26
```

Теперь функция add видит аргументы в такой последовательности:

```
(3, 5)                                  // 8
(8, 7)                                  // 15
(15, 11)                                // 26
```

Функция вызывается на один раз меньше, и возможность ошибки при выборе неверного исходного значения свертки исключается.

У JavaScript есть замечательная особенность, заключающаяся в том, что его метод reduce работает в любом случае. Если ему передается исходное значение свертки, функция вызывается для каждого элемента. Если исходное значение свертки не передается, то для первого элемента функция не вызывается. Вместо этого первый элемент становится исходным значением свертки. Таким образом, работают оба примера свертки путем сложения, показанные ранее.

Функция reduceRight действует практически аналогично, за исключением того, что начинает свою работу с конца массива. Мне хотелось бы, чтобы она называлась reduce_reverse.

Я использую метод reduce для вычисления контрольной цифры ISBN-номера данной книги:

```
function isbn_13_check_digit(isbn_12) {
    const string_of_digits = isbn_12.replace(/-/g, "");
    if (string_of_digits.length === 12) {
        const check = string_of_digits.split("").reduce(
            function (reduction, digit, digit_nr) {
                return reduction + (
                    digit_nr % 2 === 0
```

```
                    ? Number(digit)
                    : Number(digit) * 3
                );
            },
            0
        ) % 10;
        return (
            check > 0
            ? 10 - check
            : check
        );
    }
}
isbn_13_check_digit("978-1-94-981500") // 9
```

Итерация

Одной из наиболее распространенных операций над массивами является определенная работа с каждым из его элементов. Исторически это делалось с помощью инструкции `for`. В JavaScript предлагается более современный подход.

Метод `forEach` получает массив и функцию, которую он вызывает для каждого элемента массива. У функции может быть три параметра: `element`, `element_nr` и `array`. Параметр `element` — это элемент массива. Параметр `element_nr` — порядковый номер элемента, который может пригодиться, если вычисление проводится над другим массивом и нужна координация. Параметр `array` был придуман ошибочно, его здесь быть не должно. Это приглашение к модификации обрабатываемого массива, что зачастую является весьма неудачной затеей.

К сожалению, мы не располагаем методом, способным обрабатывать массив в обратном порядке, как это делает `reduceRight`. Но есть метод `reverse`, который можно вызвать предварительно, правда, он имеет разрушительный характер и не применяется к замороженным массивам.

Метод `forEach` игнорирует значение вызываемой им функции. Если уделить внимание возвращаемому значению, можно создать весьма интересные методы.

- Метод `every` проверяет возвращаемое значение. Если оно лживое, метод останавливает обработку и возвращает `false`, если правдивое — продолжает обработку. Если метод `every` дойдет до конца массива, он возвращает `true`.
- Метод `some` настолько похож на метод `every`, что непонятно, зачем он вообще нужен в языке. Если функция возвращает правдивое значение, метод останавливает обработку и возвращает `true`, если значение лживое — продолжает обработку. Если метод `some` доходит до конца массива, он возвращает `false`.
- Метод `find` похож на метод `some`, но вместо возвращения `true` или `false` он возвращает элемент, который был обработан, когда функция возвратила что-либо правдивое.

- Метод `findIndex` похож на метод `find`, но вместо возвращения обработанного элемента, когда функция возвращает что-либо правдивое, он возвращает его порядковый номер.
- Метод `filter` также похож на метод `find`, но он всегда ведет обработку до конца и возвращает массив, собирающий все элементы, для которых функция возвратила правдивое значение. Стало быть, метод `find` возвращает первый соответствующий условию элемент, а метод `filter` — все соответствующие ему элементы.
- Метод `map` похож на метод `forEach`, но он собирает все возвращаемые значения и возвращает их в новом массиве. Этот метод — идеальный способ выполнения преобразований путем создания нового массива, являющегося усовершенствованной или развитой версией оригинала.

Эти методы обеспечивают более совершенный способ обработки массивов без использования цикла `for`. Но этот набор методов еще не полон.

Методы `forEach` и `find` способны на выход на ранних стадиях. (Формами `forEach` с возможностью выхода являются `every` и `some`.) А методы `map`, `reduce` и `filter` не имеют вариантов с ранним выходом. У метода `reduce` есть вариант, работающий в обратном направлении, — `reduceRight`, а у методов `forEach`, `map`, `filter` и `find` таких вариантов нет.

Эти функциональные пробелы не позволяют отказаться от применения инструкции `for`.

Сортировка

В JavaScript имеется метод `sort`. К сожалению, с ним возникает ряд проблем.

Он выполняет сортировку на месте, изменяя сортируемый массив. Это означает, что замороженный массив отсортировать невозможно, также небезопасно сортировать совместно используемый массив.

```
let my_little_array = ["unicorns", "rainbows", "butterflies", "monsters"];
my_little_array.sort()
          // my_little_array имеет значение ["butterflies", "monsters",
          //                                 "rainbows", "unicorns"]
```

Используемая им по умолчанию функция сравнения выстраивает значения таким образом, будто это строки, даже если они являются числами, например:

```
let my_little_array = [11, 2, 23, 13, 3, 5, 17, 7, 29, 19];
my_little_array.sort();
     // my_little_array имеет значение [11, 13, 17, 19, 2, 23, 29, 3, 5, 7]
```

Это не только абсолютно неэффективно, но и неверно. К счастью, этот негатив можно сгладить, передав методу `sort` функцию сравнения. Этой функции передаются

два элемента. Ожидается, что она возвратит отрицательное число, если первый элемент должен быть первым, положительное число, если первым должен быть второй элемент, и нуль, если функции сравнения будет нечего сказать.

Эта функция способна правильно отсортировать числовой массив, если, конечно, все элементы будут конечными числами:

```
function compare(first, second) {
    return first - second;
}
```

Если нужно сравнивать неконечные числа вроде `NaN` и `Infinity`, функции сравнения придется поработать интенсивнее.

Следующим в списке проблем, связанных с функцией `sort`, станет дефицит стабильности. Функция `sort` стабильна, если порядок равных сравниваемых элементов (для которых функция сравнения возвратила нуль) остается неизменным. JavaScript не гарантирует стабильности. При сортировке массива строк или массива чисел это неактуально, а вот при сортировке массива объектов и массива массивов становится серьезной проблемой. Сложная сортировка может потребовать сортировки по фамилиям, а при их совпадении — по именам. Один из способов решения задачи — предварительная сортировка по именам, а затем повторная — по фамилиям. Но этот прием не работает, поскольку сортировка нестабильна и информация, добавленная в ходе сортировки по именам, может быть утрачена.

Уменьшить нестабильность реально за счет применения более сложной функции сравнения. Чтобы упростить задачу, создадим фабрику по производству функций сравнения:

```
function refine(collection, path) {
```

Она получает массив или объект и путь в форме массива строк и возвращает значение в конце пути. Если значения нет, возвращается неопределенность.

```
    return path.reduce(
        function (refinement, element) {
            try {
                return refinement[element];
            } catch (ignore) {}
        },
        collection
    );
}

function by(...keys) {
```

Эта фабрика создает функцию сравнения, чтобы помочь в сортировке массива объектов или массива массивов. Аргументами служат одна или несколько строк или целых чисел, определяющих сравниваемые свойства или элементы. Если первый аргумент видит связь, пробуется второй, затем третий...

Преобразование каждого ключа в массив строк:

```
    const paths = keys.map(function (element) {
        return element.toString().split(".");
    });
```

Сравнение каждой пары значений, пока не будет найдено несоответствие — если несоответствия не будет, значит, два значения равны:

```
    return function compare(first, second) {
        let first_value;
        let second_value;
        if (paths.every(function (path) {
            first_value = refine(first, path);
            second_value = refine(second, path);
            return first_value === second_value;
        })) {
            return 0;
        }
```

Если два значения относятся к одному и тому же типу, мы сможем сравнить их. Если типы разные, то нужна некая политика, позволяющая справиться со странностями. Нашей простой политикой будет сравнение имен типов по принципу `boolean < number < string < undefined`. (Возможно, было бы лучше отказаться от сортировки элементов, не совпадающих по типу.)

```
        return (
            (
               typeof first_value === typeof second_value
               ? first_value < second_value
               : typeof first_value < typeof second_value
               )
            ? -1
            : 1
        );
    };
}
```

Пример:

```
let people = [
    {first: "Frank", last: "Farkel"},
    {first: "Fanny", last: "Farkel"},
    {first: "Sparkle", last: "Farkel"},
    {first: "Charcoal", last: "Farkel"},
    {first: "Mark", last: "Farkel"},
    {first: "Simon", last: "Farkel"},
    {first: "Gar", last: "Farkel"},
    {first: "Ferd", last: "Berfel"}
];

people.sort(by("last", "first"));

// [
//         {"first": "Ferd", "last": "Berfel"},
```

```
//         {"first": "Charcoal", "last": "Farkel"},
//         {"first": "Fanny", "last": "Farkel"},
//         {"first": "Frank", "last": "Farkel"},
//         {"first": "Gar", "last": "Farkel"},
//         {"first": "Mark", "last": "Farkel"},
//         {"first": "Simon", "last": "Farkel"},
//         {"first": "Sparkle", "last": "Farkel"}
// ]
```

Попурри

Метод `concat` получает два и более массива и объединяет их для создания нового массива:

```
let part_zero = ["unicorns", "rainbows"];
let part_wun = ["butterflies", "monsters"];
let whole = part_zero.concat(part_wun);
    // whole имеет значение ["unicorns", "rainbows", "butterflies", "monsters"]
```

Метод `join` получает массив, содержащий строковые значения и строку-разделитель. Он создает длинную строку, объединяющую все элементы. Если разделение не требуется, следует в качестве разделителя воспользоваться пустой строкой. Этот метод — антипод метода `split`, работающего со строками.

```
let string = whole.join(" & ");
    // string имеет значение "unicorns & rainbows & butterflies & monsters"
```

Метод `reverse` разворачивает элементы в массиве, чтобы они оказались в нем стоящими в обратном порядке. Этот метод разрушающий, как и функция `sort`.

```
whole.reverse();
    // whole имеет значение ["monsters", "butterflies", "rainbows", "unicorns"]
```

Метод `slice` способен создать копию массива или его части. Нулевой параметр определяет, с какого порядкового номера начинать. Первый параметр — это нулевой параметр плюс количество копируемых элементов. Если первый параметр не указан, будут скопированы все оставшиеся элементы.

```
let element_nr = whole.indexOf("butterflies");
let good_parts;
if (element_nr !== -1) {
    good_parts = whole.slice(element_nr);
}
      // good_parts имеет значение ["butterflies", "rainbows", "unicorns"]
```

Чистый и нечистый

Некоторые из методов работы с массивами относятся к чистым, не изменяющим введенные в них данные. Другие — нет. Некоторые из них должны быть чистыми, но таковыми не являются. Методы, которые не могут быть чистыми по своей природе, все же представляют определенную ценность.

При работе с двойственностью «чистый — нечистый» важно знать, что является чистым, а что — нет.

Чистые методы:

```
concat
every
filter
find
findIndex
forEach
indexOf
join
lastIndexOf
map
reduce
reduceRight
slice
some
```

Нечистые методы:

```
fill
pop
push
shift
splice
unshift
```

Нечистые методы, которые должны быть чистыми:

```
reverse
sort
```

Глава 8

Как работают объекты

○ ● ○ ○ ○

> Избавляйтесь от вещей с чувством благодарности. Испытывая благодарность, вы завершаете отношения с этим объектом, и это чувство облегчает вам расставание с ним.
>
> *Мари Кондо (Marie Kondo)*

Слово «объект» в JavaScript сильно перегружено. В этом языке абсолютно все (за исключением двух ничтожно малых значений, `null` и `undefined`) является объектом. Но обычно, особенно в этой главе, слово «объект» означает нечто более конкретное.

Первичная структура данных в JavaScript называется *объектом*. Объект является контейнером для свойств или элементов. У каждого свойства есть имя и значение. Имя является строкой. Значение может быть любого типа. В других языках такой тип объекта называется хеш-таблицей, отображением, записью, структурой, ассоциативным массивом, словарем, а в некоторых слишком грубых языках — диктом (dict).

Новый объект создается с помощью литерала объекта. Этот литерал создает значение, которое может храниться в переменной, или в объекте, или в массиве, или же передаваться функции, или возвращаться из нее.

Литерал объекта заключается в фигурные скобки (`{ }`). Внутри них должно быть ноль или более свойств, разделенных запятыми (`,`). Свойствами могут быть:

- строковое значение, за которым следуют двоеточие (`:`) и выражение. Имя свойства определяется строковым значением. Значением свойства является значение выражения;
- имя, за которым следуют двоеточие и выражение. Именем свойства является имя, преобразованное в строковое значение, значением свойства — значение выражения;
- имя. Именем свойства является имя, преобразованное в строковое значение. Значением свойства является значение переменной или параметра с таким же именем;

- имя, за которым следуют список параметров, заключенный в круглые скобки (( )), и тело функции, заключенное в фигурные скобки ({ }). Это сокращение для имени, после которого стоит двоеточие, а затем выражение функции. Оно позволяет не указывать `: function`, но при этом разборчивость кода ухудшается, так что пользоваться им, пожалуй, не стоит.

Например:

```
let bar = "a long rod or rigid piece of wood or metal";
let my_little_object = {
    "0/0": 0,
    foo: bar,
    bar,
    my_little_method() {
        return "So small.";
    }
};
```

Обратиться к свойству объекта можно с помощью точечной записи с указанием имени:

```
my_little_object.foo === my_little_object.bar     // true
```

Можно также обратиться к свойству объекта, воспользовавшись скобочной записью. Ее применяют для доступа к свойству, чье имя не является допустимым идентификатором. Скобочная запись используется также для вычисляемых имен свойств. Если нужно, выражение в скобках вычисляется и преобразуется в строку.

```
my_little_object["0/0"] === 0                     // true
```

Если с помощью точечной или скобочной записи запрашивается имя, которое не может быть найдено в объекте, то вместо свойства предоставляется ничтожно малое значение `undefined`. Запрос несуществующего или утраченного свойства не считается ошибкой. Это обычная операция, выдающая `undefined`:

```
my_little_object.rainbow                          // undefined
my_little_object[0]                               // undefined
```

Новое свойство можно добавить к объекту с помощью операции присваивания. Таким же образом могут быть заменены значения существующих свойств.

```
my_little_object.age = 39;
my_little_object.foo = "slightly frilly or fancy";
```

Я рекомендую не хранить `undefined` в объектах. JavaScript это допускает и корректно возвращает значение `undefined`, но это `undefined` ничем не отличается от того, что означает отсутствие свойства. Стоит ли создавать путаницу, без которой легко обойтись? Желательно, чтобы сохранение `undefined` приводило к удалению свойства, но этого не происходит. Свойство можно удалить с помощью оператора `delete`:

```
delete my_little_object["0/0"];
```

Из объектов можно выстраивать сложные структуры данных, поскольку в объектах могут храниться ссылки на объекты. Можно построить разнообразные графы и циклические структуры. Глубина вложения не лимитирована, но благоразумие здесь не помешает.

Когда оператору `typeof` предоставляется объект, он выдает строку `"object"`:

```
typeof my_little_object === "object" // true
```

Регистр

Проверка на соответствие ключевому значению чувствительна к регистру символов. Поэтому `my_little_object.cat` — это не то же самое, что `my_little_object.Cat` или `my_little_object.CAT`. Соответствие строковых значений имен свойств определяется оператором `===`.

Копирование

Функция `Object.assign` способна скопировать свойства из одного объекта в другой. Копию объекта можно создать, присвоив его пустому объекту.

```
let my_copy = Object.assign({}, my_little_object);
my_copy.bar             // "a long rod or rigid piece of wood or metal"
my_copy.age                                         // 39
my_copy.age += 1;
my_copy.age                                         // 40
delete my_copy.age;
my_copy.age                                         // undefined
```

Объекту может присваиваться материал из множества объектов. Таким образом, сложный объект реально сконструировать путем сбора материала из более простых.

Наследование

В JavaScript объект может быть создан наследованием из другого объекта. Это совершенно иной тип наследования, чем тот, что практикуется в языках, имеющих такие тесно связанные программные структуры, как классы. В JavaScript это связанные данные, которые могут существенно уменьшить уязвимость, способную охватить всю структуру приложения.

Метод `Object.create(`*`прототип`*`)` получает существующий объект и возвращает в качестве его наследника новый объект. Существующий объект становится прототипом нового объекта. Прототипом может стать любой объект. Объект — наследник прототипа может также стать прототипом еще более новых объектов. Ограничений на длину цепочки прототипов не существует, но лучше делать ее покороче.

При попытке обращения к несуществующему свойству перед возвращением `undefined` система сначала обращается к прототипу, а затем к его прототипу и т. д.

Если свойство с таким же именем будет найдено в цепочке прототипов, оно будет выдано, как будто было найдено в интересовавшем нас объекте.

При обращении к объекту изменяется только самый близкий объект. Объектов по цепочке прототипов изменения не касаются.

```
let my_clone = Object.create(my_little_object);
my_clone.bar                 // "a long rod or rigid piece of wood or metal"
my_clone.age                                          // 39
my_clone.age += 1;
my_clone.age                                          // 40
delete my_clone.age;
my_clone.age                                          // 39
```

Чаще всего прототипы задействуются в качестве места для хранения функций. Этот прием используется и в самом JavaScript. Объект, созданный с помощью литерала объекта, является наследником `Object.prototype`. Аналогично этому массивы наследуют методы от `Array.prototype`. Числа наследуют методы от `Number.prototype`. Строки наследуют методы от `String.prototype`. Даже функции наследуют методы от `Function.prototype`. Методы массивов и строк весьма полезны, а вот методы в `Object.prototype` в большинстве своем бесполезны.

Поскольку речь идет о наследовании, теперь у нас есть два типа свойств: *собственные*, присущие самому верхнему объекту, и *унаследованные*, присущие цепочке прототипов. В большинстве случаев они работают одинаково. Иногда необходимо знать, действительно ли свойство принадлежит самому объекту. Большинство объектов наследуют функцию `hasOwnProperty(`*`строка`*`)`, но, к сожалению, ее надежность оставляет желать лучшего. Она получает строковое значение и возвращает `true`, если объект содержит свойство с указанным именем и это свойство неунаследованное. Если же у объекта есть свойство с именем `hasOwnProperty`, то вместо унаследованного метода `Object.prototype.hasOwnProperty` будет вызвано именно оно. Это может привести к сбоям или путанице. Было бы лучше, чтобы `hasOwnProperty` был не методом, а оператором, тогда состояние объекта не смогло бы привести к сбойному вызову `hasOwnProperty`. Еще лучше, если бы не было унаследованных свойств, что сделало бы этот проблемный метод ненужным.

```
my_little_object.hasOwnProperty("bar")                // true
my_copy.hasOwnProperty("bar")                         // false
my_clone.hasOwnProperty("bar")                        // false
my_clone.hasOwnProperty = 7;
my_clone.hasOwnProperty("bar")                        // ИСКЛЮЧЕНИЕ!
```

При наличии свойства `hasOwnProperty` воспользоваться унаследованным методом `hasOwnProperty` невозможно. Вместо него вызов будет обращен к собственному свойству объекта.

Тот же риск сбоев характерен и для метода `Object.prototype.toString`. Но, даже когда он работает, без разочарований не обходится.

```
my_clone.toString                                  // "[object Object]"
```

Не нужно говорить, что объект является объектом. Это не секрет. Хотелось бы посмотреть на его содержимое. Если нужно преобразовать объект в строку, `JSON.stringify` справится с этим намного лучше.

Преимущество `Object.create(`*`прототип`*`)` над `Object.assign(Object.create({}),` *`прототип`*`)` заключается в меньшей потребности в памяти. В большинстве случаев экономия незначительна. Прототипы могут привнести ряд странностей, не дав особых дополнительных преимуществ.

Существует также проблема непреднамеренного наследования. Возможно, вам захочется воспользоваться объектом наподобие хеш-таблицы, но объект наследует имена, например `"toString"`, `"constructor"` и т. д., которые могут зависеть от реализации. Потенциально их можно спутать с принадлежащими вам именами.

К счастью, `Object.create(null)` создает объект, не обремененный наследственностью. Не возникает никакой путаницы ни с унаследованными свойствами, ни с непреднамеренным наследованием. В объекте нет ничего, кроме того, что в него помещено явным образом. Я сейчас пользуюсь `Object.create(null)` весьма часто.

Ключи

Функция `Object.keys(`*`объект`*`)` может взять все собственные (неунаследованные) имена свойств, имеющихся у объекта, и возвратить их в виде строкового массива. Это позволяет воспользоваться методами работы с массивами для обработки свойств объекта.

Строковые значения в массиве будут располагаться в том порядке, в котором они были вставлены. Если нужен иной порядок, можно воспользоваться `Array`-методом `sort`.

Заморозка

Функция `Object.freeze(`*`объект`*`)` способна получить объект и заморозить его, сделав неизменяемым. Неизменяемость позволяет создавать более надежные системы. Если объект создается в соответствии с вашими предпочтениями, он может быть заморожен, что гарантирует невозможность его повреждения или вмешательства в его содержимое. Заморозка неглубока, она воздействует только на объекты верхнего уровня.

Неизменяемые объекты могут когда-нибудь обрести ценные характеристики производительности. Если известно, что объект нельзя изменить, в реализации языка появляется возможность применения весьма эффективного набора средств оптимизации.

У неизменяемых объектов имеются великолепные средства обеспечения безопасности. Их важность определяется общепринятой современной практикой,

позволяющей устанавливать в ваши системы сомнительный код. Когда-нибудь неизменяемость поможет превратить все это в безопасный метод работы. Неизменяемые объекты можно будет снабдить высококачественными интерфейсами, способными их защитить.

Функция `Object.freeze(`*`объект`*`)` и инструкция `const` действуют абсолютно поразному. `Object.freeze` оперирует значениями, а `const` — переменными. Если поместить изменяемый объект в `const`-переменную, вероятность изменения объекта сохранится, но вы не сможете заменить объект каким-либо другим значением. Если поместить неизменяемый объект в обычную переменную, вы не измените объект, но сможете присвоить переменной другое значение.

```
Object.freeze(my_copy);
const my_little_constant = my_little_object;

my_little_constant.foo = 7;                    // разрешено
my_little_constant = 7;                        // СИНТАКСИЧЕСКАЯ ОШИБКА!
my_copy.foo = 7;                               // ИСКЛЮЧЕНИЕ!
my_copy = 7;                                   // разрешено
```

Прототипы и заморозка не смешиваются

Одна из задач, решаемых с помощью прототипов, — это создание легких копий объектов. У нас есть объект, заполненный данными. Нам нужен другой точно такой же объект, но с одним измененным свойством. Как уже было показано, сделать это реально с помощью метода `Object.create`. Это позволит сэкономить немного времени на создании нового объекта, но извлечение свойств может обойтись дороже, поскольку придется просматривать цепочку прототипов.

К сожалению, это не сработает, если прототип заморожен. Если свойство в прототипе неизменяемое, экземпляр не способен иметь собственную версию этого свойства. В некоторых стилях функционального программирования для повышения надежности на основе неизменяемости требуется, чтобы все объекты были заморожены. Поэтому вместо создания копии для внесения изменения было бы неплохо получить возможность создать экземпляр, получающий наследство от замороженного прототипа, изменить его, а затем заморозить. Но работоспособных вариантов просто нет. При обновлении экземпляра выдается исключение. Кроме того, в целом замедляется вставка новых свойств. Когда вставляется новое свойство, то, чтобы определить, что у предка нет неизменяемого свойства, в поисках ключа нужно просмотреть всю цепочку прототипов. Избежать этого при создании объектов позволяет использование функции `Object.create(null)`.

Одна из конструктивных ошибок в JavaScript заключается в том, что именами свойств в объектах должны быть строки. Бывают ситуации, когда требуется в качестве ключа воспользоваться объектом или массивом. К сожалению, объекты JavaScript в таком случае ведут себя неправильно, преобразуя ключ-объект

в ключ-строку с помощью метода `toString`, что, как мы уже видели, приводит к разочарованиям.

Вместо предоставления нам объекта, правильно работающего для всех ключей, JavaScript дает другой тип объектов — `WeakMap`, который позволяет быть ключами объектам, а не строкам и у которого совершенно иной интерфейс.

Объект	**WeakMap**
object = Object.create(null);	weakmap = new WeakMap();
object[*ключ*]	weakmap.get(*ключ*)
object[*ключ*] = *значение*;	weakmap.set(*ключ*, *значение*);
delete object[*ключ*];	weakmap.delete(*ключ*);

В таком существенном синтаксическом различии двух типов объектов, работающих одинаково, нет никакого смысла. Также нет смысла в том, что их два, а не один. Вместо одного типа объекта, который позволяет задействовать в качестве ключей только строки, и другого, позволяющего использовать только объекты, должен быть один тип объекта, позволяющий брать в качестве ключей как строки, так и объекты.

При всем этом `WeakMap` — великолепный тип объекта. Рассмотрим два примера его использования.

Нам нужно поместить в объект секретное свойство. Для получения доступа к секретному свойству нужны доступ к объекту и секретный ключ. Получить доступ к свойству невозможно, пока не будет и того и другого. Это можно сделать с `WeakMap`, рассматривая его в качестве секретного ключа:

```
const secret_key = new WeakMap();
secret_key.set(object, secret);

secret = secret_key.get(object);
```

Чтобы раскрыть секрет, нужен доступ как к объекту, так и к секретному ключу. У этого приема есть весьма полезное качество: секретные свойства могут запросто добавляться к замороженным объектам.

Нам нужно получить возможность предоставления объекта некоторому коду, который способен сделать что-то полезное, например внести его в список или сохранить для дальнейшего извлечения, но при этом не хочется давать этому коду возможность изменить объект или вызвать какой-либо из его методов. В реальном мире нам хочется, чтобы работник запарковал машину, но при этом не рылся в бардачке и багажнике и не продал ее. В реальном мире доверять людям можно, но относиться с доверием к коду в компьютерной сети крайне неразумно.

Поэтому создадим устройство, называемое *установщиком пломбы* (sealer). Мы дадим ему объект для установки пломбы, а он вернет контейнер, который нельзя

будет открыть. Этот контейнер может быть отдан работнику. Чтобы получить обратно исходный объект, нужно отдать контейнер соответствующему *съемщику пломбы* (unsealer). Все эти функции довольно легко создаются с помощью `WeakMap`:

```
function sealer_factory() {
    const weakmap = new WeapMap();
    return {
        sealer(object) {
            const box = Object.freeze(Object.create(null));
            weakmap.set(box, object);
            return box;
        },
        unsealer(box) {
            return weakmap.get(box);
        }
    };
}
```

Объект `weakmap` не позволяет проверять свое содержимое. Из него невозможно извлечь значение, пока не будет ключа. `WeakMap` хорошо взаимодействует со сборщиком мусора JavaScript. Если копий ключа все еще не существует, свойство этого ключа в `weakmap` автоматически удаляется. Это помогает избежать утечек памяти.

В JavaScript имеется нечто похожее под названием `Map`, но там не обеспечены безопасность и защита от утечек памяти, имеющиеся у `WeakMap`. И хотя `WeakMap` (слабое отображение) далеко не самое подходящее название, `Map` (отображение) вносит еще бóльшую путаницу. Это название никак не связано с `Array`-методом `map`, и его нетрудно спутать с функцией картографии. Поэтому я не рекомендую использовать `Map`. Но мне нравятся `WeakMap` и `Array`-функция `map`.

В JavaScript имеется также функция под названием `Symbol`, способная выполнять одно из действий, присущих `WeakMap`. Пользоваться `Symbol` не стоит, поскольку в этом нет необходимости. Я стремлюсь к упрощению, избавляясь от ненужных мне избыточных функций.

Глава 9

Как работают строки

○ ● ○ ○ ●

> — Нечестно, нечестно! — зашипел он. — Ведь правда, моя прелесть, ведь нечестно спрашивать нас, что у него в мерзком кармашке?
>
> *Голлум*

Компьютеры хороши для манипуляции хитросплетениями битов. Людям же это несвойственно. Строки перебрасывают мост через пропасть между компьютерами и людьми. Отображение символов на целые числа было одним из важных достижений в разработке цифровых вычислительных устройств. Это был первый важный шаг к разработке пользовательских интерфейсов.

Мы не знаем, почему этот тип называется *строкой*. Почему он не назван *текстом*? Никто не знает. Строка JavaScript не похожа на фрагмент обычной строки. Можно говорить о строке символов и с такой же легкостью — о строке инструкций, или строке битов, или даже о строке сбоев. В реальном мире мы строки не объединяем. Мы их связываем.

Основы

Строка представляет собой неизменяемый массив из 16-разрядных целых чисел в диапазоне от 0 до 65 535. Строка может быть создана с помощью функции `String.fromCharCode`, получающей любое количество чисел. Обратиться к элементу строки можно с помощью метода `charCodeAt`. Элементы не могут быть изменены, поскольку строки всегда заморожены. У строк, как и у массивов, есть свойство длины — `length`.

```
const my_little_array = [99, 111, 114, 110];
const my_little_string = String.fromCharCode(...my_little_array);
my_little_string.charCodeAt(0) === 99                          // true
my_little_string.length                                        // 4
typeof my_little_string                                        // "string"
```

Для получения из строки отдельных значений можно применить скобочную запись, но она не предоставляет число, как в мире массивов. Вместо этого возвращается новая строка единичной длины, содержимым которой является число.

```
my_little_string[0] === my_little_array[0]                    // false
my_little_string[0] === String.fromCharCode(99)               // true
```

В прототипе `String.prototype` содержатся методы, работающие со строками. Методы `concat` и `slice` работают практически так же, как и аналогичные методы работы с массивами. А вот метод `indexOf` работает совершенно иначе. Аргументом для `indexOf` служит строка, а не число. Этот метод пытается сопоставить все элементы в аргументе последовательно со всеми элементами в первой строке:

```
my_little_string.indexOf(String.fromCharCode(111, 114))          // 1
my_little_string.indexOf(String.fromCharCode(111, 110))          // -1
```

Методы `startsWith`, `endsWith` и `contains` — всего лишь оболочки, заключающие в себе `indexOf` и `lastIndexOf`.

Оператор `===` считает строки с одинаковым содержимым равными. А подобные массивы равны только при условии, что это один и тот же массив.

```
my_little_array === my_little_array                           // true
my_little_array === [99, 111, 114, 110]                       // false
my_little_string === String.fromCharCode(99, 111, 114, 110)   // true
```

Равенство строк — весьма эффективный механизм. Он исключает необходимость в символьном типе, поскольку одинаковые строки считаются одним и тем же объектом, что совершенно не свойственно менее востребованным языкам типа Java.

Юникод

В качестве элементов строк JavaScript позволяет задействовать все 65 536 бит 16-разрядной раскладки. Но на практике каждый элемент считается символом, а стандарт Юникод определяет кодировку символов. В JavaScript имеется серьезная синтаксическая и методическая поддержка Юникода. Согласно стандарту Юникод некоторые коды не являются символами и использоваться не должны. Но для JavaScript это ничего не значит, и в нем допускается применение всех 16-битных кодов. Если предполагается взаимодействие ваших систем с системами, написанными на менее востребованных языках, то злоупотреблять Юникодом не стоит.

Строковые литералы записываются путем заключения нуля и более символов Юникода в двойные кавычки (`"`). (Можно использовать и одинарные кавычки (`'`), но лучше этого не делать, поскольку в этом нет необходимости.) Каждый символ представлен 16-разрядным элементом.

Для объединения применяется оператор в виде знака «плюс» (`+`). Нам уже встречался знак `+`, используемый и для сложения. Перед тем как выполнить объединение, следует убедиться, что хотя бы один операнд — это строка. Один из способов выполнения

этого условия предусматривает задействование в качестве одного из операндов строкового литерала. Другой способ предполагает передачу значения функции `String`:

```
my_little_string === "corn"                                    // true
"uni" + my_little_string                                       // "unicorn"
3 + 4                                                          // 7
String(3) + 4                                                  // 34
3 + String(4)                                                  // 34
```

Строковый литерал должен полностью помещаться в одной строчке кода. Обратный слеш (`\`) используется в качестве символа отмены действия специального символа, что позволяет включать в строковые литералы двойную кавычку (`"`), сам обратный слеш и символ перевода строки.

Для извлечения символов из строки могут применяться скобки. Представлением символа является строка единичной длины. Символьный тип отсутствует. Символ может быть представлен как число или строка.

Символьный тип есть во многих языках, и зачастую для его обозначения используется термин `char`, но, похоже, произношение этого термина в соответствии с его видом стандартным не стало. Мне приходилось слышать, что его произносят так же, как слова car, care, chair, char и share.

Все строки замораживаются при создании. Созданная строка не может быть изменена. Фрагменты строк можно скопировать. Каждый фрагмент является новой строкой. Новые строки могут создаваться путем объединения строк.

```
const mess = "monster";
const the_four = "uni" + my_little_string + " rainbow butterfly " + mess;
                // the_four имеет значение "unicorn rainbow butterfly monster"

const first = the_four.slice(0, 7);          // first имеет значение "unicorn"
const last = the_four.slice(-7);             // last имеет значение "monster"
const parts = the_four.split(" ");           // parts имеет значение [
                                             // "unicorn"
                                             // "rainbow"
                                             // "butterfly"
                                             // "monster"
                                             // ]
parts[2][0] === "b"                          // true
"*".repeat(5)                                // "*****"
parts[1].padStart(10, "/")                   // "///rainbow"
```

И снова Юникод

Исходной задачей Юникода было представление всех существующих на свете языков в 16 битах. Затем его характер изменился и он стал представлять все языки мира в 21 бите.

К сожалению, JavaScript был разработан в эпоху 16-битного Юникода. Символ JavaScript Юникодом разбивается на два новых способа выражения: *кодовую еди-*

ницу, или *кодовое значение* (code unit), и кодовую точку, или позицию (code point). Кодовая единица — это один из 16-битных символов. А кодовая точка — это число, соответствующее символу. Кодовая точка формируется из одной и более кодовых единиц.

В Юникоде определены 1 114 112 кодовых точек, разделенных на 17 плоскостей (planes) из 65 536 кодовых точек. Исходная плоскость называется основной многоязычной плоскостью — Basic Multilingual Plane (BMP). Остальные 16 плоскостей — дополнительные. JavaScript способен без каких-либо трудностей использовать символы в BMP, поскольку там кодовая точка может быть идентифицирована с одной кодовой единицей. Работать с дополнительными символами труднее.

JavaScript обращается к дополнительным символам, применяя суррогатные пары. Такие пары составляются из двух специализированных кодовых единиц. Существуют 1024 старшие суррогатные кодовые единицы и 1024 младшие суррогатные кодовые единицы. У старших суррогатных кодовых единиц коды ниже, чем у младших:

- от `0xD800` до `0xDBFF` — старшие суррогатные кодовые единицы;
- от `0xDC00` до `0xDFFF` — младшие суррогатные кодовые единицы.

Когда пары составлены правильно, каждая суррогатная кодовая единица вносит 10 бит для формирования 20-разрядного смещения, к которому добавляется 65 536 для формирования кодовой точки. (Дополнение было сделано, чтобы избавиться от путаницы, вызванной наличием двух разных последовательностей кодовых единиц, которые могли бы создавать одну и ту же кодовую точку. Полагаю, что это решение привнесло еще больше путаницы, чем та, которой удалось избежать. Более простым решением было бы запрещение использования суррогатных пар для представления кодовых точек в BMP. Это дало бы не 21-битный, а 20-битный набор символов.)

Рассмотрим кодовую точку `U+1F4A9` (или в десятичном представлении `128169`). Вычтем 65 536, что даст 62 633, или `0xF4A9`. Старшие 10 бит — `0x03D`. Младшие 10 бит — `0x0A9`. Прибавим `0xD800` к старшим битам и `0xDC00` к младшим битам и получим суррогатную пару. Таким образом, JavaScript хранит `U+1F4A9` как `U+D83D U+DCA9`. С точки зрения JavaScript `U+1F4A9` — это два 16-битных символа. Если операционная система в курсе, то они будут отображаться как один символ. Фактически JavaScript ничего не знает о дополнительных плоскостях, но не проявляет по отношению к ним открытой враждебности.

Существует два способа записи кодовой точки `U+1F4A9` в строковом литерале с применением эскейп-символа:

```
"\uD83D\uDCA9" === "\u{1F4A9}"                                    // true
```

Оба они приводят к созданию одной и той же строки длиной 2.

Есть функция `String.fromCodePoint`, способная создавать суррогатные пары:

```
String.fromCharCode(55357, 56489) === String.fromCodePoint(128169)  // true
```

Метод `codePointAt` аналогичен методу `charCodeAt`, за исключением того, что он может выполнять поиск также в следующем символе и, если символы формируют суррогатную пару, возвращает дополнительную кодовую точку:

```
"\uD83D\uDCA9".codePointAt(0)                                    // 128169
```

В BMP может быть найдено большинство тех символов, которые люди используют для глобального общения, поэтому суррогатные пары обычно не нужны. Но даже при этом дополнительные символы могут появиться в любой момент, и вашей программе следует ожидать их возникновения.

В Юникоде содержатся комбинированные и модифицирующие символы, добавляющие к буквам диакритические знаки и вносящие другие изменения. В нем также содержатся символы, управляющие направлением письма и другими сервисами. Иногда это способно усложнить ситуацию, и получится, что две строки, казалось бы содержащие одинаковые символы, могут не совпасть. В Юникоде существуют правила нормализации, которые определяют порядок появления символов и те случаи, когда базовые символы с модификаторами должны заменяться составными символами. Люди такими правилами не связаны, поэтому вам, возможно, придется работать с ненормативным материалом. В JavaScript правила нормализации заключены в методе `normalize`:

```
const combining_diaeresis = "\u0308";
const u_diaeresis = "u" + combining_diaeresis;
const umlaut_u = "\u00FC";

u_diaeresis === umlaut_u                                         // false
u_diaeresis.normalize() === umlaut_u.normalize()                 // true
```

В Юникоде содержится множество дубликатов и схожих символов, поэтому сохраняется вероятность наличия строк, одинаковых по виду, но неодинаковых по коду, даже после нормализации. Это вызывает путаницу и создает угрозу безопасности.

Шаблонные строковые литералы

Применение шаблонов — весьма популярная практика при разработке веб-приложений. Поскольку имеющийся в веб-браузере DOM API недоразвит и чреват появлением ошибок, обычной практикой стало создание вместо него HTML-представлений с использованием шаблонов. Это может существенно упростить работу по сравнению с борьбой с DOM-моделью, однако способно также стать причиной XSS-атак и других неприятностей в области веб-безопасности.

Имеющиеся в JavaScript шаблонные строковые литералы были предназначены для поддержки шаблонов и смягчения проблем безопасности, а иногда и частичного их решения.

Шаблонные строковые литералы — это строковые литералы, способные размещаться сразу на нескольких строках текста. В качестве разделителя используется символ ` (гравис).

```
const old_form = (
    "Can you"
    + "\nbelieve how"
    + "\nincredibly"
    + "\nlong this"
    + "\nstring literal"
    + "\nis?"
);

const new_form = `Can you
believe how
incredibly
long this
string literal
is?`;

old_form === new_form // true
```

Дает ли новая форма существенные преимущества по сравнению со старой? Есть небольшое преимущество в обозначениях, но оно обходится недешево: самые большие синтаксические структуры в языке представлены самыми незаметными символами клавиатуры. Из-за этого возрастает вероятность совершения ошибок. Добавляется также визуальная неразборчивость. Может ли случиться что-либо невероятное? При использовании старой формы ответ однозначно отрицательный.

В большинстве случаев текстовый материал крупных форм не должен храниться в программах. Он должен обрабатываться соответствующим инструментарием, например текстовым редактором, или редактором JSON, или инструментарием баз данных, а затем становиться доступным программе в виде привязки к ресурсу. Новый синтаксис поощряет выработку вредных привычек.

Зачастую строки применяются для записи слов естественного языка. Чтобы обслуживать всемирную аудиторию, такие тексты обычно должны быть переведены. Нам не хочется поддерживать отдельную версию каждого программного файла для каждого языка. Хочется иметь единую версию каждого программного файла, способную получать локализованный текст. Новая форма усложняет решение этой задачи, поскольку ограничивает способы представления программам шаблонных строк.

Преимущество нового синтаксиса в следующем: он позволяет использовать вставки. Допустимое выражение можно поместить внутрь конструкции в составе шаблонного строкового литерала, состоящей из знака доллара (`$`), открывающей фигурной скобки (`{`) и закрывающей фигурной скобки (`}`). Выражения вычисляются, преобразуются в строки и вставляются в шаблонный строковый литерал:

```
let fear = "monsters";

const old_way = "The only thing we have to fear is " + fear + ".";

const new_way = `The only thing we have to fear is ${fear}.`;

old_way === new_way // true
```

Опасность с обеих сторон в том, что контент может быть чем-то угрожающим, например:

```
fear = "<script src=themostevilserverintheworld.com/yourworstnightmare.js/>";
```

В веб-технологиях существует масса способов внедрения вредоносного материала. И шаблоны, похоже, усиливают уязвимость. Большинство инструментов работы с шаблонами предоставляют механизмы уменьшения опасности, но все же позволяют проявляться вредоносному коду. И что хуже всего, средства, подобные шаблонным строковым литералам с возможностью применения вставок, изначально небезопасны.

В этом случае возможность смягчения последствий выражается в особой функции, называемой функцией отличительного признака (тег-функцией). Функциональное выражение, предшествующее шаблонному строковому литералу, инициирует вызов функции с шаблонной строкой и значениями выражений в качестве ее аргументов. Смысл заключается в том, что тег-функции передаются все части, которые затем ею фильтруются, декодируются, собираются и возвращаются.

Функция `dump` — это тег-функция, которая просто возвращает все введенные в нее материалы в удобочитаемой форме:

```
function dump(strings, ...values) {
    return JSON.stringify({
        strings,
        values
    }, undefined, 4);
}

const what = "ram";
const where = "rama lama ding dong";

`Who put the ${what} in the ${where}?`
                    // "Who put the ram in the rama lama ding dong?"

const result = dump`Who put the ${what} in the ${where}?`;
                            // result имеет значение `{
                            //     "strings": [
                            //         "Who put the ",
                            //         " in the ",
                            //         "?"
                            //     ],
                            //     "values": [
                            //         "ram",
                            //         "rama lama ding dong"
                            //     ]
                            // }`
```

Странно, что строки передаются вместе в виде массива, а значения — в виде отдельных аргументов. При должной разработке тег-функции способны смягчить угрозы со стороны XSS и других уязвимостей безопасности, конечно, если эти функции действительно применяются. Но напомню еще раз: шаблонные строковые литералы с возможностью использования вставок изначально небезопасны.

Шаблонные строковые литералы добавили множество новых элементов синтаксиса и механизмов и усложнили язык. Они подталкивают к выработке вредных привычек, но дают некоторые преимущества в предоставлении больших строковых литералов.

Регулярные выражения

Строковые функции `match`, `replace`, `search`, `split` способны принимать в качестве аргументов объекты регулярных выражений. Эти объекты также могут содержать собственные методы `exec` и `test`.

Объекты регулярных выражений проверяют строки на соответствие шаблону. Объекты регулярных выражений лаконичны, что затрудняет их понимание, но очень выразительны. Они эффективны, однако их возможности далеко не безграничны. Например, эффективности объектов регулярных выражений не хватает для разбора текста в формате JSON. Дело в том, что для парсинга JSON требуется какое-то хранилище наподобие стека, позволяющее работать с вложенными структурами. У объектов регулярных выражений таких хранилищ нет. Объекты регулярных выражений могут разбивать текст в формате JSON на лексемы, что существенно упрощает создание JSON-парсера.

Разбор данных на лексемы

Компиляторы выполняют разбор исходных программ на лексемы, что является частью процесса компиляции. Есть и другие программы, также зависящие от разбора данных на лексемы: редакторы, оформители кода, анализаторы статистики, макропроцессоры и минификаторы. Программы с повышенной интерактивностью, такие как редакторы, нуждаются в быстром разборе на лексемы. К сожалению, язык JavaScript слишком сложен для разбора на лексемы.

Это связано со взаимными помехами, создаваемыми литералами регулярных выражений и автоматической вставкой точек с запятой. В результате возникают неоднозначности, затрудняющие интерпретацию программы, например:

```
return /a/i;                // возвращает объект регулярного выражения

return b.return /a/i;       // возвращает ((b.return) / a) / i
```

Из этого следует, что в целом корректный разбор программ JavaScript невозможен без одновременного проведения полного парсинга.

Ситуация еще больше усложняется при использовании шаблонных строковых литералов. В литерал может быть встроено выражение, а в него вложен еще один шаблонный строковый литерал. И такое построение способно уходить на любую глубину. В выражениях могут также содержаться литералы регулярных выражений и строки с грависами (backticks). Иногда очень сложно определить, закрывает гравис текущий литерал или же открывает вложенный литерал. Бывает также сложно определить, какова структура какой-либо строчки кода:

```
`${`${"\"}`"}`}`
```

Может быть и еще хуже. Эти выражения могут также содержать функции, которые, в свою очередь, могут содержать литералы регулярных выражений и вместо точек с запятыми еще какие-нибудь шаблонные строковые литералы.

Инструментальные средства, такие как редакторы, станут предполагать, что с программами все в порядке. Имеется поднабор JavaScript, который можно разобрать на лексемы без сопутствующего полного парсинга. Если от всего этого у вас наступает помрачение рассудка, то нужно ожидать, что ваш инструментарий не сработает.

Надеюсь, что следующий язык не создаст проблем с разбором на лексемы. А пока он не появился, я рекомендую не использовать вставки с шаблонными строковыми литералами.

Функция fulfill

Рассмотрим альтернативу шаблонным строковым литералам — функцию `fulfill`. Она получает строку, в которую могут включаться символические переменные, и объект или массив, содержащий значения, которые требуется заменить символическими переменными. Она также может принимать либо функцию, либо объект, состоящий из функций и способный кодировать замены.

Потенциальная проблема с шаблонными строками заключается в том, что все находящееся в области видимости доступно для включения кода. Это очень удобно, но одновременно угрожает безопасности. Функция `fulfill` имеет доступ только к контейнеру значений, переданных ей в явном виде, поэтому безопасна. Исходный кодировщик удаляет из определенного контекста символы, известные как источники угроз, поэтому контекст HTML изначально безопасен, в отличие от изначально небезопасных шаблонных строк. Но, как и включение шаблонных строк, такое включение способно стать опасным при неправильном использовании кодировщиков.

Строка может поступать из любого источника, такого как информационное наполнение JSON или строковый литерал. Ей не нужно находиться в том же самом файле JavaScript, что и программа. Это лучше вписывается в потребности локализованных приложений.

Символическая переменная заключается в фигурные скобки (`{ }`). Она не может содержать пробелов или скобок. В ней могут быть путь и дополнительно двоеточие, за которым следует название системы кодирования. Путь состоит из одного или нескольких имен или чисел, разделенных точками (`.`):

```
{a.very.long.path:hexify}
```

Контейнер является объектом или массивом, содержащим значения, заменяющие символические переменные. В нем могут содержаться вложенные объекты и массивы. Значения будут найдены с использованием пути. Если заменяемым значением является функция, она вызывается и ее возвращаемое значение становится

заменяющим значением. Сам контейнер может быть функцией. Она вызывается с передачей пути и системы кодирования, чтобы получить заменяющее значение.

Затем заменяющее значение может быть преобразовано кодировщиком. Это делается главным образом для смягчения проблем безопасности, исходящих от шаблонов, но способно служить и другим целям.

Необязательный аргумент кодировщика может быть объектом, содержащим функции кодирования. Та часть символической переменной, которая относится к кодировщику, выбирает один из кодировщиков. Если в объекте кодировщика нет подходящей функции, исходная символическая переменная не заменяется. Аргумент кодировщика может быть также функцией кодирования, вызываемой со всем содержимым символических переменных. Функции кодировщика передаются кандидат для замены, путь и система кодирования. Если она не возвращает строку или число, исходная символическая переменная не заменяется.

Если в символической переменной не указана система кодирования, предполагается, что она имеет вид "".

Путь содержит число, соответствующее элементу в массиве значений, или имя свойства в объекте значений. Путь может содержать точки, позволяющие находить свойства вложенных объектов и массивов.

Символические переменные заменяются, только если они правильно сформированы, имеется значение для замены и указана и правильно реализована система кодирования. Если что-то не так, символическая переменная остается на месте. Скобки, используемые в качестве литералов, можно помещать в строку, не устанавливая перед ними эскейп-символы. Если они не являются частью символической переменной, их оставляют в покое.

```
const example = fulfill(
    "{greeting}, {my.place:upper}! :{",
    {
        greeting: "Hello",
        my: {
            fabulous: "Unicorn",
            insect: "Butterfly",
            place: "World"
        },
        phenomenon : "Rainbow"
    },
    {
        upper: function upper(string) {
            return string.toUpperCase();
        },
        "": function identity(string) {
            return string;
        }
    }
); // example имеет значение "Hello, WORLD! :{"
```

Функция `entitiyify` делает текст безопасным для вставки его в HTML:

```
function entityify(text) {
    return text.replace(
        /&/g,
        "&amp;"
    ).replace(
        /</g,
        "&lt;"
    ).replace(
        />/g,
        "&gt;"
    ).replace(
        /\\/g,
        "&bsol;"
    ).replace(
        /"/g,
        "&quot;"
    );
}
```

Теперь заполним шаблон опасными данными:

```
const template = "<p>Lucky {name.first} {name.last} won ${amount}.</p>";

const person = {
    first: "Da5id",
    last: "<script src=enemy.evil/pwn.js/>"
};

// Теперь вызовем функцию fulfill.

fulfill(
    template,
    {
        name: person,
        amount: 10
    },
    entityify
)
// "<p>Lucky Da5id &lt;script src=enemy.evil/pwn.js/&gt; won $10.</p>"
```

Кодировщик `entityify` сделал потенциально вредоносный тег сценария безопасным для HTML.

А теперь посмотрим на код, который я использовал для подготовки оглавления книги, которую вы сейчас читаете!

Оглавление представлено в виде JSON-текста. В шаблонах имеются литералы скобок, но их наличие не вызывает никаких проблем. Здесь показаны вложенные вызовы функции `fulfill`, что позволяет избавиться от лексической сложности вложенных шаблонных строк:

```
const chapter_names = [
    "Сначала прочитайте меня!",
```

```
    "Как работают имена",
    "Как работают числа",
    "Как работают большие целые числа",
    "Как работают большие числа с плавающей точкой",
    "Как работают большие рациональные числа",
    "Как работают булевы значения",
    "Как работают массивы",
    "Как работают объекты",
    "Как работают строки",
    "Как работают ничтожно малые значения",
    "Как работают инструкции",
    "Как работают функции",
    "Как работают генераторы",
    "Как работают исключения",
    "Как работают программы",
    "Как работает this",
    "Как работает код без классов"
    "Как работают концевые вызовы",
    "Как работает чистота",
    "Как работает событийное программирование",
    "Как работает Date",
    "Как работает JSON",
    "Как работает тестирование",
    "Как работает оптимизация",
    "Как работает транспиляция",
    "Как работает разбиение на лексемы",
    "Как работает парсер",
    "Как работает генерация кода",
    "Как работает среда выполнения",
    "Как работают нелепости, или Что такое Wat!",
    "Как устроена эта книга"
];
const chapter_list = "<div>[</div>{chapters}<div>]</div>";
const chapter_list_item = `{comma}
<a href="#index">{"номер": {index}, "глава": "{chapter}"}</a>`;

fulfill(
    chapter_list,
    {
        chapters: chapter_names.map(function (chapter, chapter_nr) {
            return fulfill(
                chapter_list_item,
                {
                    chapter,
                    chapter_nr,
                    comma: (chapter_nr > 0)
                    ? ","
                    : ""
                }
            ).join("");
        })
    },
    entityify
)                       // См. оглавление в начале книги
```

Функция `fulfill` не очень большая:

```
const rx_delete_default = /[ < > & % " \\ ]/g;
const rx_syntactic_variable = /
    \{
    (
        [^ { } : \s ]+
    )
    (?:
        :
        (
            [^ { } : \s ]+
        )
    )?
    \}
/g;

// Определение групп:
//     [0] исходник (символическая переменная, заключенная в фигурные скобки)
//     [1] путь
//     [2] система кодирования

function default_encoder(replacement) {
    return String(replacement).replace(rx_delete_default, "");
}

export default Object.freeze(function fulfill(
    string,
    container,
    encoder = default_encoder
) {
```

Функция `fulfill` получает строку, содержащую символические переменные, функцию-генератор, или объект, или массив со значениями для замены символических переменных, и необязательную функцию кодировщика или объект, содержащий функции-кодировщики. Кодировщик, используемый по умолчанию, убирает все угловые скобки.

Основную часть работы выполняет принадлежащий `string` метод `replace`, который находит символические переменные, представляя их в виде исходной подстроки, строки пути и необязательной строки системы кодирования:

```
return string.replace(
    rx_syntactic_variable,
    function (original, path, encoding = "") {
        try {
```

Он использует путь для получения одиночной замены из контейнера значений. Путь содержит одно или несколько имен (или чисел), разделенных точками:

```
            let replacement = (
                typeof container === "function"
                ? container
                : path.split(".").reduce(
                    function (refinement, element) {
```

```
                    return refinement[element];
                },
                container
            )
        );
```

Если значением для замены является функция, ее вызывают, чтобы получить это значение.

```
        if (typeof replacement === "function") {
            replacement = replacement(path, encoding);
        }
```

Если предоставлен объект кодировщика, вызывается одна из его функций. Если кодировщик представлен функцией, то выполняется ее вызов.

```
        replacement = (
            typeof encoder === "object"
            ? encoder[encoding]
            : encoder
        )(replacement, path, encoding);
```

Если значения для замены относятся к булеву типу, они преобразуются в строку.

```
        if (
            typeof replacement === "number"
            || typeof replacement === "boolean"
        ) {
            replacement = String(replacement);
        }
```

Если значение для замены — строка, выполняется ее подстановка. В противном случае символическая переменная остается неизменной.

```
        return (
            typeof replacement === "string"
            ? replacement
            : original
        );
```

Если что-то пойдет не так, символическая переменная будет оставлена в исходном состоянии.

```
        } catch (ignore) {
            return original;
        }
    }
);
});
```

Глава 10

Как работают ничтожно малые значения

○ ● ○ ● ○

Ты — романс,
Ты — степи России,
Ты — штаны на билетере Рокси.
Я же — сломанная кукла, безделушка, пустячок.
И если я ничтожество,
То ты — само совершенство!

Коул Портер (Cole Porter)

Ничтожно малыми (bottom) называются специальные значения, показывающие окончание рекурсивной структуры данных или отсутствие значения. В языках программирования ничтожно малые значения могут носить имена `nil`, `none`, `nothing` и `null`.

В JavaScript два ничтожно малых значения — `null` и `undefined`. Значение `NaN` также может считаться ничтожно малым, так как оно показывает отсутствие числа. Обилие таких значений можно рассматривать как ошибку разработки.

Ничтожно малые значения `null` и `undefined` — это единственные значения в JavaScript, не являющиеся объектами в абстрактном смысле. Попытка дать свойству значение из этой пары приводит к выдаче исключения.

В чем-то `null` и `undefined` очень похожи друг на друга, а в чем-то ведут себя по-разному. Они частично, но не полностью взаимозаменяемы. Вызывает путаницу само наличие двух ничтожно малых значений, которые можно считать одним и тем же, но которые иногда действуют по-разному. Принятие решения, какое из них следует использовать, — пустая трата времени, а связанные с ними беспочвенные теории приводят к еще большей путанице, которая, в свою очередь, вызывает появление ошибок.

Удаление одного из них помогает создавать более качественные программы. Мы не в состоянии удалить одно из этих значений из языка, но можем получить выгоду, убрав его из своих программ. Следует удалить `null` и пользоваться исключительно `undefined`.

Обычно, когда нужно выбирать между двумя словами, я предпочитаю то, что короче. У `null` есть известное значение в контексте программирования и структуры данных, а у `undefined` его нет. Более того, `undefined` звучит непонятно. С математическим понятием *неопределенности* оно не имеет ничего общего. Оно даже не означает то, что программисты подразумевают под *неопределенностью*.

Получается, что `null` выглядит более подходящим именем, так почему же я предпочел `undefined`? Дело в том, что `undefined` — это значение, используемое самим JavaScript. Если определить переменную, применяя `let` или `var`, и не инициализировать ее явным образом, JavaScript инициализирует ее значением `undefined`, что вызывает путаницу, поскольку только что определенная переменная приобретает значение `undefined`, означающее, что она не определена. Если не передать функции достаточное количество аргументов, дополнительные параметры устанавливаются в `undefined`. При попытке получить у объекта отсутствующее свойство будет получено значение `undefined`. При попытке получить у массива отсутствующий элемент будет получено значение `undefined`.

Единственный случай, когда я применяю `null`, — использование метода `Object.create(null)` для создания нового пустого объекта. Ошибка спецификации не позволяет задействовать `Object.create()` и `Object.create(undefined)`.

Значения `null` и `undefined` могут быть протестированы с помощью оператора равенства:

```
function stringify_bottom(my_little_bottom) {
    if (my_little_bottom === undefined) {
        "undefined";
    }
    if (my_little_bottom === null) {
        return "null";
    }
    if (Number.isNaN(my_little_bottom)) {
        return "NaN";
    }
}
```

Иногда можно увидеть, что программисты со стажем пишут `(typeof my_little_bottom === "undefined")`, что также работает. Но `(typeof my_little_bottom === "null")` дает сбой, поскольку `typeof null` возвращает `"object"`, а не `"null"`. Это плохо, потому что `(typeof my_little_object === "object")` дает ложный положительный ответ, когда `my_little_object` имеет значение `null`, что приводит к ошибке, которую тест должен был предотвратить. Это еще одна причина избегать применения `null`.

Значение undefined, несомненно, лучше подходит для использования, чем null, но оно также страдает от проблемы пути. Вспомним, что значением отсутствующих свойств является undefined, что могло бы считаться положительным моментом, если бы undefined был замороженным пустым объектом, но он таковым не является. Значение undefined не объект, поэтому попытки получения из него свойства приводят к выдаче исключения. Это существенно усложняет написание путевых выражений. Например:

```
my_little_first_name = my_little_person.name.first;
```

выдает исключение, если в my_little_person нет свойства name или my_little_person имеет значение undefined. Поэтому нельзя рассматривать цепочку точек (.) и список индексов [ ] в качестве пути. Можно считать это последовательностью отдельно взятых операций, любая из которых способна дать сбой. Это приводит к созданию примерно следующего кода:

```
my_little_first_name = (
    my_little_person
    && my_little_person.name
    && my_little_person.name.first
);
```

Логический оператор И (&&) используют, чтобы избежать вычисления того, что находится справа от него, если то, что стоит слева, имеет лживое значение. Этот код весьма пространный, уродливый и медленный, но он избавляет от выдачи исключений и обычно делает то, что должен был бы делать код:

```
my_little_first_name = my_little_person.name.first;
```

если бы значение undefined работало как объект, а не как антиобъект.

Глава 11

Как работают инструкции

○●○●●

Когда я кивну головой, ты ударишь по ней молотком.

Буллвинкль Дж. Лось (Bullwinkle J. Moose)

Большинство языков программирования можно разбить на два класса: языки выражений и языки инструкций. Язык инструкций имеет инструкции (их еще называют операторами) и выражения. В языке выражений есть только выражения. У сторонников языков выражений в ходу совокупность теорий, где приводятся доводы о несомненном превосходстве этих языков. А вот обилия теорий с обоснованиями преимуществ языков инструкций что-то не наблюдается. Несмотря на это все популярные языки, включая JavaScript, — это языки инструкций.

Ранние программы были просто списками инструкций, иногда напоминающими предложения и даже заканчивающимися в некоторых языках точками. К сожалению, эти точки путали с десятичными, поэтому в более поздних языках символами завершения инструкций стали точки с запятой.

Структурированное программирование разрушило идею простого списка инструкций, позволив этим спискам быть вложенными в другие инструкции. Язык ALGOL 60 был одним из первых языков структурированного поколения, который в качестве разделителей блоков инструкций использовал `BEGIN` и `END`. В языке BCPL для замены `BEGIN` и `END` были введены фигурные скобки (`{` и `}`). Эта мода, пришедшая из 1960-х, сохраняла популярность многие десятилетия.

Объявления

В JavaScript имеется три инструкции, используемые для объявления переменных в функции или модуле: `let`, `function` и `const`. Есть также устаревшая инструкция `var`, применяемая с нелюбимым многими браузером Internet Explorer.

Поговорим об инструкции `let`. Она объявляет новую переменную в текущей области видимости. Каждый блок (строка инструкций, заключенная в фигурные скобки) создает область видимости. Переменные, объявленные в области видимости, невидимы за ее пределами. Инструкция `let` также допускает инициализацию, но не требует ее проведения. Если переменная не инициализирована, ей по умолчанию

дается значение undefined. Инструкции let позволено объявлять сразу несколько переменных, но я рекомендую объявлять каждую переменную ее собственной инструкцией let. Это упрощает чтение кода и поддержку программы.

Инструкция let допускает деструктурирование. Это довольно хитрый синтаксис, определяющий и инициализирующий сразу несколько переменных содержимым объекта или массива. Таким образом:

```
let {huey, dewey, louie} = my_little_object;
```

это сокращение для:

```
let huey = my_little_object.huey;
let dewey = my_little_object.dewey;
let louie = my_little_object.louie;
```

Аналогично:

```
let [zeroth, wunth, twoth] = my_little_array;
```

является сокращением для:

```
let zeroth = my_little_array[0];
let wunth = my_little_array[1];
let twoth = my_little_array[2];
```

Деструктурирование — не самое важное свойство, но оно способно улучшить некоторые шаблоны. Однако, применяя его, можно довольно легко ошибиться. Существуют синтаксические возможности для переименований и задания значений по умолчанию, но им сопутствует слишком много визуальных сложностей.

Объявление function создает объект функции и переменную, в которой она содержится. К сожалению, у этого объявления такой же синтаксис, как и у функций-выражений, что создает путаницу.

```
function my_little_function() {
    return "So small.";
}
```

является сокращением для:

```
let my_little_function = undefined;

my_little_function = function my_little_function() {
    return "So small.";
};
```

Следует заметить, что объявление function не заканчивается точкой с запятой, а let и инструкции присваивания заканчиваются этим символом.

Объявление function получает возможность подтянуться к началу. Части кода удаляются из того места, куда вы поместили объявление, и перемешаются к началу тела функции или модуля. Все инструкции let, созданные объявлением function, подтягиваются к началу, а за ними следуют все присваивания функциональных объектов для этих переменных. Поэтому объявление function не должно помещаться

в блок. Допустимо помещать объявление в тело функции или модуля, но не нужно — в инструкции `if`, `switch`, `while`, `do` или `for`. Функции рассматриваются в главе 12.

Инструкция `const` похожа на инструкцию `let`, но имеет два весьма важных отличия: для нее требуется обязательная инициализация, а откладывать присваивание значения переменной нельзя. Когда у меня есть выбор, я отдаю предпочтение использованию инструкции `const`, поскольку это позволяет программировать более чисто (см. главу 19).

Вполне очевидно, что `const` — сокращение от слова *constant*. Применять его было бы нежелательно, поскольку оно подразумевает постоянство или неподвластность времени. `Const` — это нечто эфемерное, способное исчезать, когда функция возвращает управление. При каждом запуске программы или вызове функции у нее могут быть разные значения. Следует также понимать, что если значение `const` изменяемое, подобное незамороженному объекту или массиву, то ей должно быть присвоено значение, а переменную ей присвоить невозможно. `Object.freeze` влияет на значения, а не на переменные, а `const` влияет на переменные, а не на значения. Важно понимать разницу между переменными и значениями: переменные содержат ссылки на значения. Значения никогда не содержат переменных.

```
let my_littlewhitespace_variable = {};
const my_little_constant = my_littlewhitespace_variable;
my_little_constant.butterfly = "free";              // {butterfly: "free"}
Object.freeze(my_littlewhitespace_variable);
my_little_constant.monster = "free";                // СБОЙ!
my_little_constant.monster                          // undefined
my_little_constant.butterfly                        // "free"
my_littlewhitespace_variable = Math.PI;   // my_littlewhitespace_variable имеет
                                          // значение, приближающееся к числу π
my_little_constant = Math.PI;                       // СБОЙ!
```

Выражения

В JavaScript допускается появление на месте инструкций любого выражения. Это весьма небрежный, но довольно популярный прием в конструкции языка. В JavaScript есть три типа выражений, имеющих смысл на месте инструкций: присваивания, вызовы и удаления. К сожалению, на месте инструкций допустимо применение и всех остальных типов выражений, что ослабляет возможности компилятора по выявлению ошибок.

Инструкция присваивания заменяет ссылку в переменной или вносит правку в изменяемый объект или массив. Инструкция присваивания состоит из четырех частей.

0. *Левая часть* — выражение, получающее значение. Это может быть переменная или выражение, производящее значение объекта или массива, и детализация — точка (`.`), за которой следует имя свойства, или открывающая квадратная скобка (`[`), за которой следует выражение, производящее имя свойства или номер элемента, и далее закрывающая квадратная скобка (`]`).

1. Оператор присваивания:
   - `=` — присваивание;
   - `+=` — присваивание со сложением;
   - `-=` — присваивание с вычитанием;
   - `*=` — присваивание с умножением;
   - `/=` — присваивание с делением;
   - `%=` — присваивание с остатком от деления;
   - `**=` — присваивание с возведением в степень;
   - `>>>=` — присваивание с расширением знака и со сдвигом вправо;
   - `>>=` — присваивание со сдвигом вправо;
   - `<<=` — присваивание со сдвигом влево;
   - `&=` — присваивание с побитовым И;
   - `|=` — присваивание с побитовым ИЛИ;
   - `^=` — присваивание с побитовым исключающим ИЛИ.
2. Выражение, чье значение будет сохранено.
3. Точка с запятой (`;`).

Я не рекомендую использовать операторы инкремента в виде сдвоенного знака «плюс» (`++`) или сдвоенного знака «минус» (`--`). Они были созданы в глубокой древности для работы с арифметикой указателей. С тех пор пришло понимание того, что арифметика указателей вредна, поэтому современные языки не допускают ее применения. Последним популярным языком, который может похвастаться наличием арифметики указателей, был C++ — язык настолько неудачный, что его назвали в честь оператора `++`.

Мы избавились от арифметики указателей, но все еще застреваем на применении оператора `++`. Теперь он добавляет к чему-нибудь единицу. Почему возникла мысль, что для добавления единицы нужна иная синтаксическая форма, чем для добавления любого другого значения? Разве в этом есть какой-либо смысл?

Ответ будет следующим: **это не имеет никакого смысла**.

Усугубляет ситуацию то, что у оператора `++` имеются форма предварительного увеличения на единицу и форма последующего увеличения на единицу. Порядок их следования легко перепутать и очень трудно обнаружить это при отладке. А оператор `++` был замешан в ошибках переполнения буфера и других сбоях системы безопасности. От ненужных и опасных функций следует отказываться.

Инструкции-выражения являются нечистыми. Инструкции присваивания и `delete` вызывают явные изменения. Вызов, игнорирующий возвращаемое значение, пол-

ностью зависит от побочных эффектов. Инструкция-выражение — единственная, которая не начинается с идентифицирующего ключевого слова. Данная синтаксическая оптимизация способствует нечистому программированию.

Ветвление

В JavaScript имеются две инструкции ветвления, `if` и `switch`, и здесь не поймешь, то ли одной слишком много, то ли двух.

Я не рекомендую использовать инструкцию `switch`, являющуюся жутким гибридом хоаровской инструкции `case` и фортрановской вычисляемой инструкции `goto`. Нет ничего такого, что может быть записано с инструкцией `switch`, чего нельзя было бы записать (зачастую компактнее) с помощью инструкции `if`. Инструкция `switch` предлагает воспользоваться явно заданной переменной-переключателем, но страдает от опасности сквозного прохода, способной привести к ошибкам и появлению дурных привычек. Существуют и стилистические проблемы: нужно ли `case` выравнивать под `switch` или же их следует набирать с отступом? Похоже, что правильного ответа на этот вопрос не существует.

В качестве альтернативы инструкции `switch` можно использовать объект. Заполните объект функциями, реализующими поведение для каждого случая. Ключом послужит значение, соответствующее переменной `case`:

```
const my_little_result = my_little_object[case_expression]();
```

Ваше выражение случая *`case_expression`* выбирает из объекта одну из функций, которая затем может быть вызвана. Если *`case_expression`* не соответствует одной из функций, выдается исключение.

К сожалению, эта форма не избавлена от потенциальных угроз безопасности, поскольку привязана к использованию ключевого слова `this`. Более подробно этот вопрос рассматривается в главе 16.

Инструкция `if` определенно лучше инструкции `switch`. Ее ключевое слово `else` — это не инструкция, а компонент, поэтому не должно иметь отступа, присущего инструкции. Компонент `else` нужно помещать на той строке, где стоит фигурная скобка (`}`), закрывающая предыдущий блок.

JavaScript предполагает, что часть, содержащая условие, будет булевоподобным значением. Я думаю, что это ошибка. В JavaScript нужно было настаивать на применении чисто булева значения. Я рекомендую предоставлять чисто булевы значения, несмотря на то что язык потворствует проявлению неряшливости.

Форма `else if` позволяет инструкции `if` заменять инструкцию `switch` без уродливых отступов по всему экрану. Применяя ее, легко ошибиться, что может вызвать путаницу в потоках управления. Форма `else if` должна использоваться только в `case`-подобных конструкциях. Если компонент `else` начинается с `if`, то, возможно,

было бы лучше не задействовать `else if`, а вместо этого поместить `if` в блок `else`. Форма `else if` не должна использоваться, если предыдущий блок закончился изменением потока выполнения инструкций.

Когда программа создается в чисто функциональном стиле, лучше воспользоваться тернарным оператором. Он часто применяется неправильно, что создало ему плохую репутацию. Заключите тернарный оператор в круглые скобки. Поставьте после открывающей скобки перевод строки и выровняйте условие и два последствия:

```
let my_little_value = (
    is_mythical
    ? (
        is_scary
        ? "monster"
        : "unicorn"
    )
    : (
        is_insect
        ? "butterfly"
        : "rainbow"
    )
);
```

Организация циклов

В JavaScript имеется три инструкции организации циклов, `for`, `while` и `do`, и здесь явный перебор то ли с двумя, то ли с тремя.

Инструкция `for` является потомком фортрановской инструкции `DO`. Обе они использовались для поэлементной обработки массивов, перекладывая основной объем работы по управлению индуктивной переменной на программиста. Вместо нее мы воспользуемся имеющимися в массивах методами, подобными `forEach`, которые управляют индуктивной переменной автоматически. Я ожидаю, что в будущих версиях языка методы массива, снабженные чистыми функциями для применения к элементам, смогут выполнять бóльшую часть работы в параллельном режиме. А бездарно написанные программы с инструкцией `for` по-прежнему будут вести поэлементную обработку.

В ходе обучения программированию начинающих я обнаружил, что управление, состоящее из трех частей — *инициализации*, *условия*, *приращения*, — усваивалось плохо. Было непонятно, что это за пункты и почему они располагаются в определенном порядке. К тому же нет никаких синтаксических интуитивно понятных свойств, облегчающих их обнаружение или запоминание.

Я не рекомендую использовать инструкцию `for`.

Обе инструкции, `while` и `do`, применяются для организации циклов общего назначения. Синтаксически они очень разные, но в смысле работы единственной разницей является то, что инструкция `while` проверяет свое условие на вершине цикла, а инструкция `do` — в самом низу. Странно, что при такой большой визуальной разнице получается столь незначительная разница в работе.

Я заметил, что многие из моих циклов не требуют отмены исключительно в начале или в конце. Обычно им нужна отмена где-нибудь в середине. Поэтому многие циклы имеют следующую форму:

```
while (true) {
    // какие-либо действия
    if (are_we_done) {
        break;
    }
    // какие-либо дополнительные действия
}
```

Лучше всего создавать циклы с использованием концевой рекурсии (см. главу 18).

Изменение потока выполнения инструкций

В JavaScript есть четыре инструкции, изменяющие поток выполнения кода, среди которых нет инструкции `goto`! Это `break`, `continue`, `throw` и `return`.

Инструкция `break` используется для выхода из цикла. Она может применяться также для выхода из инструкции `switch`, если та присутствует, но ситуация усложняется, если инструкция `switch` находится внутри цикла.

Инструкция `continue` подобна инструкции перехода `goto`, передающей управление в начало цикла. Мне еще не приходилось видеть программу, содержащую инструкцию `continue`, которая не была бы улучшена путем ее удаления. Инструкция `throw` выдает исключение (см. главу 14).

Моя любимая инструкция, изменяющая поток выполнения кода, — `return`. Она завершает выполнение функции и обозначает возвращаемое значение. Есть школа, в которой учат, что у функции должна быть только одна инструкция `return`. Я никогда не видел доказательств пользы такого учения. Полагаю, что было бы логичнее возвращаться из функции, когда ей известно, что это необходимо, а не направляться сначала к уникальной точке выхода из нее.

В JavaScript нет инструкции `goto`, но этот язык позволяет всем инструкциям иметь метки. Я думаю, это ошибка — инструкциям не нужны метки.

Попурри

Инструкция `throw`, инструкция `try` и компонент `catch` будут рассмотрены в главе 14.

Инструкции `import` и `export` рассмотрим в главе 15.

Инструкция `debugger` может вызвать приостановку выполнения наподобие выхода на контрольную точку. Инструкция `debugger` должна применяться только на стадии разработки. Перед передачей программы в эксплуатацию ее нужно удалить.

Пунктуация

В JavaScript `if` и `else`, а также инструкциям организации цикла разрешается получать либо одну инструкцию, либо блок. Всегда используйте блок, даже если он содержит всего одну инструкцию. Это повышает устойчивость кода. А также упрощает процесс улучшения кода, не внося в него ошибки. Не ставьте в коде никаких синтаксических ловушек, способных сбить с толку товарищей по команде. Программируйте с прицелом на то, чтобы код всегда работал хорошо.

Инструкции выражений, инструкция `do`, инструкции изменения потока выполнения кода и инструкция `debugger` должны завершаться точкой с запятой (`;`). В JavaScript имеется ущербная функция автоматической вставки точки с запятой (Automatic Semicolon Insertion), которая предназначалась для облегчения программирования начинающими, позволяя им не ставить точки с запятой. К сожалению, она может не справиться со своей задачей, поэтому набирайте код как профессионал. Было бы лучше, если бы грамматика JavaScript не использовала точку с запятой, но этого не случилось, поэтому не стоит притворяться, что так оно и было. Эта неразбериха способна привести к ошибкам.

Если инструкция слишком длинная, чтобы поместиться в одной строчке кода, ее следует разбить. Я делаю это после открывающей круглой скобки (`(`), открывающей квадратной скобки (`[`) или открывающей фигурной скобки (`{`). Соответствующие им закрывающая круглая скобка (`)`), закрывающая квадратная скобка (`]`) или закрывающая фигурная скобка (`}`) начинают новую строку с таким же отступом, что и в строке кода, с которой началось разбиение. Все, что между скобками, имеет отступ в четыре пробела.

Глава 12

Как работают функции

○ ● ● ○ ○

> Дела обстоят таким образом, что при доступности рациональных процедур все бы ими пользовались, на практике же преимущества их использования при создании программ трудно переоценить.
>
> *Ян Ф. Кюрри (Ian F. Currie)*

Первые программы назывались *списком команд*. Этот простой список команд (или инструкций) загружался в машину вместе с данными, и со временем, если, конечно, повезет, на выходе получался ответ.

Оказалось, что управлять отдельно взятым единым списком команд было трудно. Некоторые последовательности команд могли встречаться в нескольких списках или в одном и том же списке. В результате были изобретены подпрограммы. Набор полезных подпрограмм мог быть собран в библиотеку. Каждой подпрограмме присваивался номер вызова, подобный имеющемуся в библиотечных книгах. (Имена считались излишеством.)

Грейс Мюррей Хоппер (Grace Murray Hopper) разработала список команд под названием A-0. Это был первый компилятор. Ему нужны были список команд и номера вызовов, а также лента с библиотекой подпрограмм. Он находил подпрограммы, соответствующие номерам вызовов, и компилировал их в новую программу. Смысл *компиляции* был буквальным: выполнение работы за счет сбора информации из других источников. Отсюда произошло множество специальных терминов, используемых по сей день: ассемблер, компилятор, библиотека, исходник и, что важнее всего, вызов. Мы вызываем подпрограмму так же, как заказываем книгу в библиотеке. Мы не приступаем к выполнению подпрограммы и не активируем ее. Мы ее вызываем. Благодаря Грейс Мюррей Хоппер сложилась основа профессиональной лексики в нашей профессии. Это из-за нее мы называем вычислительные машины *компьютерами*, а не *электронными мозгами*. Именно ей я обязан тем, что зову себя *программистом*.

Было замечено, что подпрограммы были сродни математическим функциям. Для их ввода в действие в языке Фортран II имелись объявление `SUBROUTINE` и инструкция

CALL, а также объявление FUNCTION, способное возвращать результат, который мог предлагать значение выражению.

В языке C подпрограммы и функции были сведены к единому понятию, неформально называемому функцией. Описательное ключевое слово в C не используется. А в JavaScript такое слово есть — function.

Оператор function создает функциональные объекты. Он принимает список параметров и тело, являющееся блоком инструкций.

Каждое имя в списке параметров — это переменная, инициализируемая в выражении из списка аргументов. После каждого имени могут стоять необязательный знак равенства (=) и выражение. Если значением аргумента было undefined, тогда вместо него использовалось значение выражения.

```
function make_set(array, value = true) {

// Создание объекта с извлечением имен из массива строк.

    const object = Object.create(null);
    array.forEach(function (name) {
        object[name] = value;
    });
    return object;
}
```

Функциональный объект вызывается со списком аргументов, содержащим нуль и более выражений, разделенных запятыми. Каждое из этих выражений вычисляется и привязывается к параметру функции.

Список аргументов и список параметров не обязаны совпадать по длине. Лишние аргументы игнорируются. Отсутствующие аргументы дают значение undefined.

В списках аргументов и параметров разрешается применять оператор многоточия (...), называемый *spread* («развертывание»). Он развертывает массив, в результате чего каждый элемент массива рассматривается в качестве отдельно взятого аргумента. В списках параметров оператор многоточия называется *rest* («остальное»). Все остальные аргументы пакуются в массив, привязываемый к имени параметра. Оператор многоточия в списке параметров должен стоять последним. Это позволяет функции работать с переменным числом аргументов.

```
function curry(func, …zeroth) {
    return function (…wunth) {
        return func(…zeroth, …wunth);
    };
}
```

При вызове функции создается объект активации, остающийся для вас невидимым. Это скрытая структура данных, которая содержит информацию и связывает все то, что функция должна выполнить, а также содержит обратный адрес активации вызывающей функции.

В языках, подобных C, объекты активации размещаются в стеке. Они покидают стек (или выводятся из него), когда функция возвращает управление. В JavaScript происходит иначе. Объекты активации JavaScript размещает в куче, как обычные объекты. При возвращении функцией управления объекты активации не проходят автоматическую деактивацию. Вместо этого объект активации может выживать, пока на него есть ссылка. Объекты активации подпадают под сборку мусора, как и обычные объекты.

В объекте активации содержатся:

- ссылка на функциональный объект;
- ссылка на объект активации вызывающей функции. Она используется инструкцией `return` для возврата управления;
- информация о возвращении, которая применяется для продолжения выполнения кода после вызова. Обычно это адрес инструкции, выполняемой сразу же после вызова функции;
- параметры функции, инициализированные аргументами;
- переменные функции, инициализированные значением `undefined`;
- временные переменные, используемые функциями для вычисления сложных выражений;
- содержимое `this`, которое может быть ссылкой на интересующий объект, если функциональный объект был вызван как метод.

Функциональный объект похож на обычный изменяемый объект тем, что он может быть контейнером свойств. Но в этом нет ничего хорошего. В идеале функциональные объекты должны быть неизменяемыми. В некоторых сценариях обеспечения безопасности с помощью совместно используемых изменяемых функциональных объектов упрощают применение вредоносного кода.

У функционального объекта имеется свойство `prototype`. Его используют (что, в общем-то, не рекомендуется делать) в модели создания псевдоклассов. Свойство `prototype` хранит ссылку на объект, содержащий свойство `constructor`, включающий обратную ссылку на функциональный объект и ссылку делегирования на `Object.prototype`. Более подробно эти вопросы рассматриваются в главе 16.

Функциональный объект имеет ссылку делегирования на `Function.prototype`. По этой ссылке функциональный объект наследует ненужные методы `apply` и `call`.

В функциональном объекте содержатся также два скрытых свойства:

- ссылка на исполняемый код функции;
- ссылка на объект активации, который был активен в момент создания функционального объекта. Это делает возможным создание замыкания. Функция может использовать это скрытое свойство для доступа к переменным той функции, которая ее создала.

Иногда для описания используемых функцией переменных, которые были объявлены за ее пределами, применяется понятие *свободных переменных*. А с помощью понятия *связанных переменных* иногда описываются переменные, объявленные внутри самой функции, включая параметры.

Функции могут быть вложенными. Когда создается внутренний функциональный объект, он содержит ссылку на объект активации внешней функции, которая его создала.

Этот механизм — функциональный объект, сохраняющий ссылку на объект активации внешней функции, — называется *замыканием* (closure). Это самое важное открытие в истории языков программирования. Оно было сделано в языке Scheme. Особую популярность оно получило с появлением JavaScript и подогрело интерес к этому языку. Без него JavaScript был бы просто дымящейся кучей благих намерений, просчетов и классов.

Глава 13

Как работают генераторы

○ ● ● ○ ●

> Современные композиторы отказываются умирать.
>
> *Эдгар Варез (Edgard Varèse)*

В соответствии со стандартом ES6 в языке JavaScript появилась новая особенность — *генераторы* (generators). Это произошло на волне серьезной зависти составителей стандарта к возможностям языка Python. Я не могу рекомендовать использование ES6-генераторов по нескольким причинам.

То, что делает код, сильно отличается от того, как этот код выглядит. Генераторы похожи на обыкновенные функции, но работают совершенно иначе. Они создают функциональный объект, в свою очередь создающий объект, содержащий метод `next`, сделанный из тела `function*`. В этом теле имеется оператор `yield`, напоминающий оператор `return`, но он не дает ожидаемого значения. Вместо этого он создает объект, в котором имеется свойство `value`, содержащее ожидаемое значение. Крошечная звездочка (`*`) способна сильно изменить поведение.

У генераторов вероятен очень сложный поток управления. В Structured Revolution утверждалось, что потоки управления должны быть абсолютно понятными и предсказуемыми. Потоки управления генераторов могут быть сложными, поскольку они способны приостанавливаться и возобновляться. Также они могут иметь запутанные взаимодействия с `finally` и `return`.

Они вынуждают к использованию циклов. Поскольку мы отходим от применения объектов и стремимся работать с функциями, нужно избавляться от циклов. А генераторы вместо этого требуют *большего количества* циклов. Для создания значений основная часть генераторов содержит цикл с инструкцией `yield`. Большинство пользователей таких генераторов для получения значений берут цикл `for`. Получаются два цикла там, где они вообще не нужны.

В генераторах применяется весьма неуклюжий интерфейс ООП. Фабрика создает объект, и генератор создает объект. (В функциональной конструкции фабрика

создает функцию-генератор, а та, в свою очередь, создает значение. Функциональная конструкция проще, понятнее и легче в использовании.)

И наконец, что хуже всего, генераторы стандарта ES6 совершенно не нужны. Я не рекомендую использовать функциональные особенности — есть более удачные варианты.

Рассмотрим пример генераторов стандарта ES6:

```
function* counter() {
    let count = 0;
    while (true) {
        count += 1;
        yield count;
    }
}

const gen = counter();

gen.next().value                                    // 1
gen.next().value                                    // 2
gen.next().value                                    // 3
```

А вот что я рекомендую использовать взамен:

```
function counter() {
    let count = 0;
    return function counter_generator() {
        count += 1;
        return count;
    };
}

const gen = counter();

gen()                                               // 1
gen()                                               // 2
gen()                                               // 3
```

Вместо написания чего-то похожего на функцию, выдающего с помощью инструкции `yield` значение, фактически являющееся функцией, возвращающей объект, содержащий следующий метод, выдающий объект, содержащий свойство `value`, нужно просто написать функцию, возвращающую значение.

Более удачный способ применения

Генераторы нам не помешают. Давайте найдем им правильное применение. Начнем с функции, возвращающей функцию. Внешняя функция является *фабрикой*. Внутренняя функция представляет собой *генератор*. В общем виде получается следующий шаблон:

```
function фабрика (параметры фабрики) {

    Инициализация состояния генератора

    return function генератор (параметры генератора) {

        обновление состояния

        return значение;
    };
}
```

Разумеется, вариантов может быть множество. Состояние генератора надежно хранится в переменных фабрики.

Простейший полезный экземпляр такого шаблона — фабрика `constant`. Она получает значение, а возвращает генератор, который всегда возвращает это же значение:

```
function constant(value) {
    return function constant_generator() {
        return value;
    };
}
```

Более полезный экземпляр — фабрика `integer`. Она возвращает генератор, возвращающий при каждом вызове следующее целочисленное значение последовательности. В конце последовательности возвращается значение `undefined`. Оно служит сигналом о ее окончании.

```
function integer(from = 0, to = Number.MAX_SAFE_INTEGER, step = 1) {
    return function () {
        if (from < to) {
            const result = from;
            from += step;
            return result;
        }
    };
}
```

Фабрика `element` получает массив, а возвращает генератор, возвращающий при каждом вызове элемент этого массива. Когда элементы заканчиваются, возвращается значение `undefined`. Фабрика может получать второй необязательный аргумент — генератор, выдающий номера элементов, использующиеся для извлечения самих элементов. По умолчанию поочередно извлекаются все элементы.

```
function element(array, gen = integer(0, array.length)) {
    return function element_generator(…args) {
        const element_nr = gen(…args);
        if (element_nr !== undefined) {
            return array[element_nr];
        }
    };
}
```

Фабрика `property` делает то же самое в отношении объектов. Она возвращает каждое свойство объекта в форме массива, содержащего ключ и значение. Когда свойства заканчиваются, возвращается значение `undefined`. Фабрика может получать второй аргумент — генератор, выдающий ключи, используемые для извлечения свойств. По умолчанию извлекаются все принадлежащие объекту свойства в том порядке, в котором они в него вставлялись.

```
function property(object, gen = element(Object.keys(object))) {
    return function property_generator(…args) {
        const key = gen(…args);
        if (key !== undefined) {
            return [key, object[key]];
        }
    };
}
```

Фабрика `collect` получает генератор и массив. Она возвращает генератор, работающий точно так же, как и полученный ею генератор, за исключением того, что он также добавляет в массив интересные возвращаемые значения. Все аргументы, передаваемые новой функции, передаются и старой функции.

```
function collect(generator, array) {
    return function collect_generator(…args) {
        const value = generator(…args);
        if (value !== undefined) {
            array.push(value);
        }
        return value;
    };
}
```

Функция `repeat` используется в качестве драйвера. Она получает функцию и вызывает ее до тех пор, пока та не возвратит значение `undefined`. Это единственный нужный нам цикл. Его можно написать с применением инструкции `do`, но я предпочитаю концевую рекурсию (см. главу 18).

```
function repeat(generator) {
    if (generator() !== undefined) {
        return repeat(generator);
    }
}
```

Для сбора данных можно воспользоваться функцией `collect`.

```
const my_array = [];
repeat(collect(integer(0, 7), my_array));
                            // my_array имеет значение [0, 1, 2, 3, 4, 5, 6]
```

Для достижения нужного результата можно объединить функции `repeat` и `collect`. Функция `harvest` не является фабрикой или генератором, но получает генератор в качестве аргумента.

```
function harvest(generator) {
    const array = [];
    repeat(collect(generator, array));
    return array;
}

const result = harvest(integer(0, 7));
                              // result имеет значение [0, 1, 2, 3, 4, 5, 6]
```

Фабрика `limit` получает функцию и возвращает функцию, которая может использоваться ограниченное количество раз. Когда лимит будет исчерпан, функция не будет делать ничего, кроме возвращения `undefined`. Второй аргумент фабрики — допустимое количество вызовов новой функции.

```
function limit(generator, count = 1) {
    return function (…args) {
        if (count >= 1) {
            count -= 1;
            return generator(…args);
        }
    };
}
```

Фабрика `limit` может использоваться с любой функцией. Например, если ей передать функцию, исполняющую желания, и задать лимит 3, получится функция, исполняющая три желания.

Функция `filter` получает генератор и функцию предиката. Так называется функция, возвращающая `true` или `false`. Она возвращает новый генератор, работающий наподобие прежнего, за исключением того, что он доставляет только значения, для которых предикат возвращает `true`.

```
function filter(generator, predicate) {
    return function filter_generator(…args) {
        const value = generator(…args);
        if (value !== undefined && !predicate(value)) {
            return filter_generator(…args);
        }
        return value;
    };
}

const my_third_array = harvest(filter(
    integer(0, 42),
    function divisible_by_three(value) {
        return (value % 3) === 0;
    }
));
// my_third_array имеет вид [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39]
```

Фабрика `concat` может получить два и более генератора и объединить их для создания генератора, сочетающего последовательность их действий. Он получает

значения от первого генератора, пока тот не выдаст значение undefined, затем переключается на следующий генератор. Фабрика concat использует показанную ранее фабрику element, чтобы распорядиться генераторами должным образом.

```
function concat(…generators) {
    const next = element(generators);
    let generator = next();
    return function concat_generator(…args) {
        if (generator !== undefined) {
            const value = generator(…args);
            if (value === undefined) {
                generator = next();
                return concat_generator(…args);
            }
            return value;
        }
    };
}
```

Фабрика join получает функцию от одного или нескольких генераторов, возвращая новый генератор. При каждом вызове нового генератора вызываются все прежние генераторы и их результаты передаются функции. Фабрику join можно использовать с repeat для выполнения всех действий, которые вы привыкли делать с применением конструкции for of. Функциональный аргумент для join выполняет действия, запрограммированные в блоке. Фабрика join способна работать одновременно с потоками нескольких генераторов.

```
function join(func, …gens) {
    return function join_generator() {
        return func(…gens.map(function (gen) {
            return gen();
        }));
    };
}
```

Всем этим можно воспользоваться для создания функции map, работающей наподобие array-метода map. Она получает функцию и массив, а возвращает новый массив, в каждом элементе которого содержится результат вызова функции в отношении каждого элемента массива.

```
function map(array, func) {
    return harvest(join(func, element(array)));
}
```

Фабрика objectify предоставляет еще один способ конструирования объектов данных.

```
function objectify(…names) {
    return function objectify_constructor(…values) {
        const object = Object.create(null);
        names.forEach(function (name, name_nr) {
            object[name] = values[name_nr];
        });
```

```
        return object;
    };
}

let date_marry_kill = objectify("date", "marry", "kill");
let my_little_object = date_marry_kill("butterfly", "unicorn", "monster");
            // {date: "butterfly", marry: "unicorn", kill: "monster"}
```

Генераторы занимают пограничную позицию между чистыми и нечистыми функциями. Генераторы, созданные фабрикой `constant`, относятся к чистым, но большинство генераторов имеют безусловно нечистую природу. Генераторы могут быть с сохранением состояния, но они хранят состояние в скрытом виде в замыкании фабрики. Состояние обновляется только вызовом генератора. Они не создают побочных эффектов — досадной проблемы большинства нечистых функций. Это позволяет составлять из генераторов вполне успешно работающие комбинации.

Было бы неплохо придавать наибольшему количеству программ максимально возможную чистоту. Программы, взаимодействующие с внешним миром, не могут соблюдать абсолютную чистоту, поскольку такой чистотой не отличается и сам внешний мир. Как тогда узнать, что может быть чистым, а что должно быть нечистым? Правильный путь способны указать генераторы.

Глава 14

Как работают исключения

○ ● ● ● ○

> Нет! Пытаться нельзя. Делай или не делай.
> Попыток нет.
>
> *Йода*

Программисты всегда полны оптимизма. Они думают, что все в их программах всегда будет работать правильно, но даже оптимистам известно, что иногда что-то способно пойти не так.

Вызываемая функция может засбоить совершенно неожиданным образом. Особенно характерно это для использования кода сторонних разработчиков. Взаимосвязанность с ним выражается в появлении режимов сбоев. Итак, что же может произойти в результате сбоя такого кода? Приведет ли это к нарушению работы всей программы? Нужно ли предпринимать попытки повторных вызовов функции в надежде на лучший исход? И как следует сообщать о сбое?

Самый популярный подход к решению данной проблемы — обработка исключений, которая пытается вселить в нас прежний оптимизм при программировании. Отпадает необходимость в проверке наличия непонятных кодов ошибок в каждом возвращаемом значении. Не нужно опрашивать глобальный регистр *defcon*, чтобы выяснить, не стала ли программа нестабильной. Вместо этого мы предполагаем, что все работает правильно. Если случится что-то неожиданное, текущая деятельность остановится и обработчик исключений определит, что программе делать дальше.

Идея управления исключениями JavaScript была позаимствована у Java. А для Java эту идею взяли у C++. У языка C++ нет надлежащего управления памятью, поэтому, когда что-то идет не так, каждой функции в цепочке вызовов нужно явным образом высвободить любую распределенную ею память. В JavaScript очень качественное управление памятью, но этот язык страдает от последствий несовершенства модели памяти языка C++.

О сбое сигнализирует инструкция `throw`. В языке C++ применялась бы инструкция `raise`, как в языке Ada, но слово `raise` уже использовалось в библиотеках C, поэтому вместо него в C++ было зарезервировано слово, с которым никто никогда не хотел иметь дела.

JavaScript позволяет выдать с помощью инструкции `throw` любое значение. Обычно выдается что-нибудь сделанное конструктором `Error`, но это необязательное требование. Выдать можно буквально все. В C++ и Java объект исключения — это что-то предназначенное для дальнейшего развития действий. В качественно написанном коде JavaScript объект исключения не нужен.

```
throw "Это не вычисляется.";
```

Инструкция `try` прикрепляет обработчик исключений к блоку. Обработчик исключений оформлен как компонент `catch`. Допускается один параметр, определяющий порядок получения того, что было выдано в результате выполнения инструкции `throw`.

```
try {
    here_goes_nothing();
} catch {
    console.log("fail: here_goes_nothing");
}
```

Если что-либо привело к выдаче исключения в блоке `try`, запускается блок компонента `catch`.

Конструкция `try` позволяет добиться весьма скрупулезного управления исключениями. Каждая инструкция в функции может иметь собственные блоки `try` и `catch`. Допускается даже наличие в инструкциях `try` других инструкций `try`. И что еще хуже, допускается наличие `try` в компонентах `catch`, а инструкции `try` и компоненты `catch` могут содержать инструкции `throw`. В функции не должно быть более одной инструкции `try`.

Развитие событий

Важной целью, которой удалось добиться при управлении исключениями, явилось то, что при правильно запущенных программах исключение не приводит к снижению производительности. Она может снизиться при выдаче исключения, но это должно случаться нечасто, и даже при этом существенных потерь не происходит.

Компилятор JavaScript создает для каждой компилируемой функции отображение перехвата (catchmap). С его помощью место инструкции в теле функции отображается на тот компонент `catch`, с помощью которого это место обрабатывается. Отображение перехвата в ходе обычного выполнения программы не используется.

При выполнении инструкции `throw` выдается исключение и внимание сосредотачивается на отображении перехвата для текущей функции. При наличии предназначенного компонента `catch` ему передается управление и выполнение продолжается с кода этого компонента.

Если предназначенного компонента `catch` нет, внимание переключается на вызывающую функцию. Теперь текущей становится вызывающая функция, а инструкция, вызвавшая предыдущую функцию, — местом сбоя. Опять внимание переключается

на отображение перехвата. Если в нем есть предназначенный компонент `catch`, ему передается управление и выполнение продолжается с кода этого компонента.

Вот так все движется вниз по виртуальному стеку вызовов, пока не будет найден компонент `catch`. Когда стек вызовов опустеет, мы получим неперехваченное исключение.

Это простой и весьма удачный механизм. Он позволяет использовать стиль программирования, в котором основное внимание уделяется успешному выполнению кода, но без абсолютного исключения сбоев. Однако, применяя его, легко наделать ошибок.

Обычные исключения

Наиболее распространенная ошибка в работе с исключениями — их использование для передачи обычных результатов. Например, если взять функцию, считывающую содержимое файла, то ошибка *отсутствия файла* не должна быть исключением. Это вполне обычное дело. Исключения должны применяться только при неожиданных проблемах.

В языке Java неправильное применение исключений упрощает способ обхода проблем с его системой типов. Метод Java может возвратить результат только одного типа, поэтому исключения используются в качестве альтернативного канала для возвращения обычных результатов, чей тип не разрешен системой типов. Из-за этого несколько компонентов `catch` могут оказаться прикрепленными к одному и тому же блоку `try`. Ситуация становится запутанной, поскольку обычные результаты смешиваются с реальными исключениями.

Это напоминает присваивание значения инструкции `GOTO` в языке Фортран, при котором переменная содержит адрес назначения перехода. Чтобы гарантировать выбор правильного компонента, следует выстроить компоненты `catch` в правильном порядке. Выбор не основывается на любом типе равенства, как в инструкции `switch`, он основан на правилах приведения типов, что само по себе отвратительно. Любая система типов, требующая явного приведения типов, порочна.

Пути управления продиктованы методом, создавшим объект исключения. Таким образом, создается тесная связь между объектом, который выдал исключение, и тем, который его перехватывает, что идет вразрез с построением четкой модульной конструкции.

Через этот клубок может проходить очень много путей, что способно существенно усложнить задачу высвобождения распределенных ресурсов. Для смягчения данной проблемы был добавлен компонент `finally`. Он представляет собой функцию без параметров, подразумеваемо вызываемую из каждой точки выхода `try` и из каждого компонента `catch`.

В то же время в JavaScript имеется намного более понятный подход. Если блок `try` выполнен успешно, получается либо нужный результат, либо нужное объяснение.

Система типов в JavaScript обладает достаточной гибкостью для обработки всех исключительных случаев.

Если случится что-нибудь по-настоящему неожиданное, запустится блок кода компонента `catch` и начнется альтернативная история, либо все закрывающая, либо запускающая все сначала. Мы следуем плану А. Если он реализован удачно, то все хорошо. Если он дал сбой, переходим напрямую к плану Б:

```
try {
    plan_a();
} catch {
    plan_b();
}
```

Рассуждать о восстановлении после ошибок сложно, а о тестировании — еще сложнее. Поэтому мы будем придерживаться простой и надежной схемы. Сделаем все ожидаемые результаты возвращаемыми значениями, а исключения оставим для исключительных случаев.

Проблема в том, что существует множество программ, разработанных людьми, которых испортил опыт работы на других языках. Они используют `throw` там, где должна быть инструкция `return`. Они создают сложные компоненты `catch`, в которых пытаются решить безнадежные проблемы и выполнить неподъемную работу. И в попытке навести порядок они применяют компонент `finally`, в котором JavaScript не нуждается.

Непредвиденные обстоятельства

Исключения работают за счет возвращения значений из стека. Выданные значения сообщаются вызовам функций, находящимся ниже в стеке. При программировании с учетом непредвиденных обстоятельств стек опустошается после каждого хода. Путешествие во времени для передачи выданного значения на активацию, которой больше нет, невозможно. Польза от исключений носит ограниченный характер. Они могут сообщать только о локальных проблемах на текущем временном отрезке. Эта особенность рассматривается в главе 20.

Безопасность

Существует весьма важная модель безопасности, ограничивающая возможности функций за счет передачи им лишь тех ссылок, в которых они нуждаются для выполнения своей работы. Исключения, исходя из сложившейся практики, обеспечивают канал, через который могут вступить в сговор две не пользующиеся доверием функции.

Происходит следующее: объект 1 и объект 2 представляют собой два пакета, установленные по нашей беспечности на сервере. Мы сильно рискуем, но полагаем, что риск приемлем, поскольку объекты 1 и 2 порой могут пригодиться. Мы даем

объекту 1 доступ к сетевому сокету, а объекту 2 — доступ к нашему закрытому ключу. Если они не способны обмениваться данными друг с другом, то все в порядке.

Чтобы от объекта 1 была польза, ему требуется нечто дешифруемое, но мы не хотим, чтобы он напрямую обменивался данными с объектом 2. Поэтому создаем функцию-посредник, которой доверяем. Мы позволяем объекту 1 вызывать посредник. Посредник вызывает объект 2. Тот в свою очередь возвращает значение посреднику. Посредник проверяет возвращаемое значение, убеждается в том, что это ожидаемый простой текст, и возвращает его объекту 1.

Если объект 2 вместо этого выполняет следующее действие:

```
throw top_secret.private_key;
```

и если посредник его не перехватывает или перехватывает, но опять выдает (ужасная практика, популярной которую сделал язык C++), то объект 1 может перехватить закрытый ключ и передать его злоумышленникам в соответствии со сговором.

Надежность

Что может сделать программа, когда что-то идет не так? Одна функция вызывает другую в ожидании, что та сработает. Но предположим, что она не сработала. Объект исключения сообщает функции, что аргумент вышел за рамки допустимого, или что объект был неизменяемым, или что допущена какая-то другая ошибка. При современном уровне технологии не стоит ожидать, что функция сможет выправить ситуацию.

Подробности, содержащиеся в объекте исключения, способны сыграть важную роль и стать полезными для оповещения программиста. Вместо этого данная информация передается функции, которая не в состоянии толком ею воспользоваться. Мы испортили информационный поток. Эти сведения должны отправляться программисту, возможно, в виде журнальной записи. Вместо этого они поступают вниз, в стек вызова, где могут быть неправильно поняты и забыты. Исключения привели нас к неправильной разработке наших систем. Получается, что причина низкой надежности — сам механизм исключений.

Глава 15

Как работают программы

○ ● ● ● ●

Привет, программы!

Кевин Флинн (Kevin Flynn)

Программа на JavaScript поступает к месту исполнения в виде исходного кода. Самой первой задачей JavaScript было добавление веб-страницам интерактивности. Поскольку HTML имеет текстовый формат, было решено вставлять JavaScript в веб-страницы в виде исходного текста. Это весьма необычное решение. Программы, написанные на большинстве других языков, поступали к месту исполнения в виде инструкций в коде каждой отдельно взятой машины или же в лучше переносимом формате, например в потоке инструкций в байтовом представлении, который передавался интерпретатору или генератору кода.

JavaScript-обработчик на месте исполнения компилирует исходный код JavaScript, производя машинный код, или интерпретируемый код, или и то и другое. Благодаря этому JavaScript стал высокопереносимым языком. Программы не зависели от архитектуры используемой машины. Одна и та же версия программы на JavaScript может запускаться на любом обработчике JavaScript независимо от того, на чем он базируется.

Компилятор JavaScript способен быстро и легко проверять важные свойства безопасности программы: он способен гарантировать, что программа не в состоянии вычислить адреса памяти, или выполнить переход в места с ограниченным доступом, или нарушить ограничения типа. В Java предпринималась попытка сделать что-то подобное путем проверки его байт-кода, но это происходило не настолько быстро, или не настолько легко, или не настолько уверенно.

Некоторые синтаксические особенности JavaScript неоправданно усложнили парсинг. Но даже при этом обработчики JavaScript способны выполнять компиляцию, загрузку и запуск намного быстрее, чем Java может загружать и запускать свой код из файлов с расширением `.jar`.

JavaScript распространяется в блоках исходного кода. Обычно это файл с расширением `.js`, но может быть и в виде строки исходного кода из JSON-объекта или исходного текста из базы данных или системы управления исходным кодом.

В прежние времена блоки исходного кода в браузерах были частями страницы, как содержимое тега `<script>` или как встроенный обработчик событий. В следующем примере блоком кода является `alert("Hi");`:

```
<img src=hello.png onclick=alert("Hi");>
```

Сейчас принято считать, что встраивать блоки исходного кода JavaScript в страницу HTML — порочная практика. Это ухудшает конструкцию из-за того, что не разделены представление и поведение. Поэтому снижается производительность, так как код в странице не может сжиматься средством gzip или кэшироваться. Это уменьшает безопасность, поскольку позволяет проводить трудно распознаваемые атаки с применением межсайтовых сценариев (Cross Site Scripting, XSS). Принятая W3C политика защиты содержимого (Content Security Policy, CSP) должна стать постоянной практикой, чтобы можно было предотвратить любое использование встроенного исходного кода.

Блок исходного кода рассматривается в качестве тела функции, не имеющей параметров. Он компилируется как функция, а затем вызывается. При этом может быть вызвано создание функциональных объектов и, возможно, их выполнение.

Это может быть концом программы. Но в JavaScript блок исходного кода обычно регистрируется для получения событий или сообщений либо для экспорта функции, которой может воспользоваться другой блок исходного кода. В JavaScript не требуется создавать программы таким образом, но этот способ написания программ просто замечателен и JavaScript поддерживает данный стиль лучше многих других языков.

(Историческое замечание: встроенные обработчики событий работают несколько иначе. Блок исходного кода также рассматривался в качестве тела функции, но созданная функция обработчика событий не вызывалась в тот же момент. Вместо этого она привязывалась к DOM-узлу и вызывалась по наступлении события.)

Изначальные компоненты

Есть объекты и функции, которые становятся автоматически доступными каждому блоку исходного кода. Некоторые из них нам уже встречались — это `Number`, `Math`, `Array`, `Object` и `String`. Существует и множество других, и все они предоставляются в соответствии со стандартом ECMAScript. Кроме того, среда окружения хоста (например, браузер или Node.js) способна внести свои дополнения. И все функции в блоке исходного кода могут использовать весь этот арсенал.

Глобальные компоненты

В древней браузерной модели программирования все объявления переменных вне функции добавлялись к области видимости *страницы*, также известной как *глобальная* область видимости (как грандиозно это звучит!). Эта область видимости

включает в себя также переменные `window` и `self`, содержащие ссылки на область видимости страницы. Все переменные в области видимости страницы были видны всем блокам исходного кода, имеющимся на ней.

Ничего хорошего в этом не было. Это подталкивало к применению такого стиля программирования, при котором использовались общие глобальные переменные. Получались ненадежные, дефектные программы. Когда два независимых друг от друга блока исходного кода случайно брали одно и то же имя переменной, возникали непредвиденные ошибки. Кроме того, это серьезно снижало уровень безопасности и способствовало проведению вредоносных XSS-атак.

Глобальные переменные — несомненное зло.

К счастью, сейчас ситуация существенно выправилась.

Модуль

Лучше было бы все объявления переменных вне функции добавлять к области видимости *модуля*, и тогда их видели бы все функции, находящиеся в блоке исходного кода. Это правильный подход, поскольку конструкции в результате получаются более строгие, безопасные и более надежно защищенные.

Теперь взаимодействие с другими блоками исходного кода станет осуществляться явным образом с помощью инструкций `import` и `export`. Они могут применяться для создания высококачественных программ. Но используются и для написания плохих программ. Я сосредоточусь на создании качественных программ.

При экспортировании модуль открывает доступ к экспортируемому другим модулям. Модуль должен экспортировать что-либо одно — обычно функцию или объект, наполненный функциями. Экспортируемое может использоваться несколькими другими модулями, поэтому вполне логично заморозить экспорт, чтобы не допустить случайных неприятностей или взаимного перемешивания важных данных. Экспортирование, по сути, является интерфейсом. Интерфейсы должны быть простыми и понятными.

Тело блока исходного кода выполняется только один раз, а это значит, что все импортеры совместно используют одни и те же экспортированные данные. Если экспорт зависит от предыстории, а каждый импортер ожидает чего-то незатронутого и неразделяемого, может сложиться проблемная ситуация. В таком случае следует экспортировать фабричную функцию, способную создавать незатронутые и не прошедшие разделяемость экземпляры. Инструкция `export` не создает экземпляры.

Эта инструкция имеет следующий вид:

```
export default экспортируемое;
```

Жаль только, что присутствие слова `default` все здесь портит. *Экспортируемое* — это замороженный объект или функция, являющаяся предметом экспорта.

Импортирование позволяет блоку исходного кода получить функцию или объект из другого блока исходного кода. Для этого выбирается имя новой переменной, в которую желательно получить импорт. Также нужно указать поставщика воспользовавшись строковым литералом в двойных кавычках ("), содержащим некое имя или адрес. Ваша система может воспользоваться ими для определения местоположения блока исходного кода и, если нужно, считать его, откомпилировать, вызвать и доставить вам то, что он предлагает на экспорт.

Инструкцию `import` можно рассматривать как особую инструкцию `const`, получающую значение инициализации из какого-нибудь другого места. Она имеет следующий вид:

```
import имя from строковый_литерал;
```

Я рекомендую применять только экспорт, но вы можете использовать столько инструкций `import`, сколько вам нужно. Инструкции `imports` должны находиться в начале файла, а инструкция `export` — в конце.

Связывание и сцепление

Качественное программирование на микроуровне зависит от подходящих соглашений по написанию кода, способствующих увеличению визуальной дистанции между качественным и некачественным кодом, что упрощает обнаружение ошибок.

Качественное программирование на макроуровне зависит от качественной проработки модулей.

Качественные модули имеют сильное связывание. Это означает, что все элементы связаны и работают вместе для выполнения конкретных действий. Некачественные модули слабо связаны — зачастую из-за плохой организации и попытки выполнения слишком большого количества действий. Здесь может проявиться эффективность функций JavaScript, поскольку у нас есть возможность передать функцию, чтобы позаботиться об определенных деталях, которые не должны касаться модуля.

Качественные модули отличаются слабым сцеплением. Чтобы грамотно воспользоваться модулем, вам потребуются лишь ограниченные сведения о его интерфейсе. Подробности его реализации не нужны. Хороший модуль скрывает свою реализацию. А модуль, не скрывающий реализацию, предлагает применение тесного сцепления. Понятно, что взаимозависимые модули отличаются весьма тесным сцеплением. Это практически так же отвратительно, как и использование глобальных переменных.

Интерфейсы ваших модулей должны быть простыми и понятными. Сводите зависимости к минимуму. Чтобы придать программам достаточную структурированность, позволяющую им разрастаться без превращения в сплошную неразбериху, нужна хорошо проработанная архитектура.

Концепции сцепления и связывания были представлены в двух замечательных книгах:

- *Myers G.* Reliable Software Through Composite Design. John Wiley & Sons, 1975;
- *Yourdon E., Larry L.* Structured Design. Constantine. Yourdon Press, 1979.

Обе книги изданы раньше *Object Oriented Programming*, поэтому они вышли из моды и были по большому счету забыты. Но изложенные в них сведения о сильном связывании и слабом сцеплении по-прежнему верны и важны.

Глава 16

Как работает this

> Я изобрел термин «объектно-ориентированный»
> и могу сказать вам, что при этом не имел в виду C++.
>
> *Алан Кэй (Alan Kay)*

Язык Self — это диалект языка Smalltalk, в котором классы заменены прототипами. Объект мог наследоваться непосредственно из другого объекта. Классическая модель страдала от нестабильности и раздувания по причине сильного сцепления классов посредством расширений `extends`. Прототипы языка Self были весьма удачным упрощением. Модель прототипов легче и выразительнее.

В JavaScript реализована весьма странная модель прототипов.

Когда объект создан, может быть назначен прототип, состоящий из части или всего содержимого нового объекта:

```
const new_object = Object.create(old_object);
```

Объекты, по сути, всего лишь контейнеры для свойств, а прототипы — это просто объекты. Методы — это всего лишь функции, сохраненные в объектах.

При попытке извлечения значения свойства, которого нет у объекта, получается значение `undefined`. Но если у объекта имеется прототип (как у показанного ранее `new_object`), в результате получается значение свойства прототипа. Если и его извлечь не удается и если у прототипа есть свой прототип, получается значение свойства прототипа, принадлежащего первому прототипу. И все спускается ниже и ниже.

У многих объектов может быть общий прототип. Эти объекты могут рассматриваться как экземпляры класса, но, по сути, это всего лишь отдельные объекты с общим прототипом.

Чаще всего прототипы используются в качестве места хранения методов. У схожих объектов, скорее всего, имеются одинаковые методы, поэтому, если методы помещаются в один общий прототип, а не в каждый объект, можно сэкономить память.

А как функция в прототипе узнает, с каким объектом она работает? Здесь на сцену выходит привязка `this`.

Синтаксически вызов метода представляет собой тернарную операцию, использующую `.` и вызов `()` или ссылку `[]` и вызов `()`. Тремя подвыражениями в тернарном выражении являются:

- интересующий объект;
- имя метода;
- список аргументов.

Именованный метод ищут в объекте и в цепочке его прототипов. Если функция не найдена, выдается исключение. Это замечательно и подталкивает к использованию полиморфизма. Наследственность объекта вас волновать не должна. Нужно лишь побеспокоиться о его возможностях. Если возможностей у объекта нет, выдается исключение. Если они есть, мы ими пользуемся, не беспокоясь о том, каким образом они были приобретены.

Если функция найдена, она вызывается со списком аргументов. Функция также получает в качестве подразумеваемого параметр `this`, привязанный к интересующему объекту.

Если метод содержит внутреннюю функцию, последняя не получает доступа к `this`, поскольку внутренние функции вызываются в качестве функций, а привязку `this` получают только вызовы методов:

```
Old_object.bud = function bud() {
    const that = this;

// lou не может видеть принадлежащий bud параметр this, но lou может видеть
// принадлежащий bud параметр that.

    function lou() {
        do_it_to(that);
    }
    lou();
};
```

Привязка `this` работает только при вызове методов, поэтому вызов:

```
new_object.bud();
```

выполняется успешно, а:

```
const funky = new_object.bud;
funky();
```

нет. Здесь в `funky` содержится ссылка на ту же самую функцию, что и в `new_object.bud`, но `funky` вызывается как функция, поэтому не имеет привязки `this`.

Привязка `this` характерна тем, что имеет динамическую природу. Все остальные переменные имеют статическую привязку. И это создает путаницу.

```
function pubsub() {

// Фабрика pubsub создает объект публикации-подписки. Код, который
// получает доступ к объекту, может подписаться на функцию, получающую
// публикации, и опубликовать материал, который будет доставлен всем
// подписчикам.

// Для содержания функции subscriber используется массив подписчиков. Он
// скрыт от внешнего мира, поскольку находится в области видимости функции
// pubsub.

    const subscribers = [];
    return {
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            const length = subscribers.length;
            for (let i = 0; i < length; i += 1) {
                subscribers[i](publication);
            }
        }
    };
}
```

Функция `subscriber` получает привязку `this` к массиву `subscribers`, потому что:

```
subscribers[i](publication);
```

является вызовом метода. Он не обязательно похож на вызов метода, но суть от этого не меняется. Это дает каждой функции `subscriber` доступ к массиву `subscribers`, что может позволить `subscriber` нанести вред, например удалить всех остальных подписчиков или перехватить и изменить их сообщения.

```
my_pubsub.subscribe(function (publication) {
    this.length = 0;
});
```

Привязка `this` способна угрожать безопасности и надежности. Когда функция сохраняется в массиве, а позже вызывается, ей дается `this`, привязанная к массиву. Если наличие доступа к массиву для функций не предполагалось, как обычно и бывает, возникает вероятность неблагоприятного развития событий.

Функция `publish` может быть исправлена путем замены цикла `for` циклом `forEach`:

```
publish: function (publication) {
    subscribers.forEach(function (subscriber) {
        subscriber(publication);
    }
}
```

Все переменные статически привязаны, и это хорошо. Динамическая привязка есть только у `this`. Это означает, что ее привязка определяется вызывающей функцией, а не создателем функции. Данная аномалия становится источником путаницы.

У функционального объекта есть два прототипных свойства. В нем имеется делегационная ссылка на `Function.prototype`. Также у него есть свойство `prototype`, содержащее ссылку на объект, используемый в качестве прототипа объектов, созданных функцией, вызванной с префиксом `new`.

Вызов конструктора записывается путем помещения префикса `new` перед вызовом функции. Наличие префикса `new` приводит к следующим действиям:

- созданию значения `this` с `Object.create(`*`функция`*`.prototype)`;
- вызову *функции* с `this`, привязанной к новому объекту;
- принудительному возвращению `this`, если *функция* не возвращает объект.

Используя функцию `Object.assign` для копирования методов из одного прототипа в другой, можно организовать что-то вроде наследования. Но куда прочнее в обиход вошла замена свойства `prototype` функционального объекта объектом, созданным другим конструктором.

Поскольку каждая функция является потенциальным конструктором, узнать в нужный момент, следует ли в вызове применять префикс `new`, довольно трудно. Хуже того, когда он требуется, но по забывчивости не поставлен, никаких предупреждений не выдается.

Поэтому мы руководствуемся следующим соглашением: функции, предназначенные для вызова в качестве конструкторов с префиксом `new`, должны получать имена, начинающиеся с прописной буквы.

В JavaScript имеется также более привычный, похожий на классы синтаксис, созданный специально для тех разработчиков, которые не знают и никогда не узнают, как работает JavaScript. Он позволяет задействовать навыки работы с менее востребованными языками без обучения.

Синтаксис классов, несмотря на свой внешний вид, не реализует никакие классы. Это всего лишь «синтаксический сахар» в дополнение к странностям конструктора псевдоклассов. Он сохраняет самый худший аспект классической модели — расширения `extends`, создающие сильные сцепления между классами. Сильное сцепление становится причиной создания ненадежных и дефектных конструкций.

Избавление от this

В 2007 году было реализовано несколько исследовательских проектов, в рамках которых предпринимались попытки разработать безопасный поднабор JavaScript. Одной из важнейших проблем было управление привязкой `this`. В вызове метода `this` привязывается к интересующему объекту, что иногда полезно. Но, когда

этот же самый функциональный объект вызывается в качестве функции, `this` может быть привязана к глобальному объекту, что было крайне нежелательно.

Мое решение этой проблемы — полный запрет `this`. Аргументы: проблематичность и ненужность этой привязки. Если мы уберем `this` из языка, его полноценность по Тьюрингу не пострадает. Поэтому я начал программировать в диалекте, свободном от `this`, чтобы выяснить степень трудностей, связанных с отказом от нее.

Как же я удивился тому, что ситуация не усложнилась, а упростилась и мои программы стали меньше и качественнее.

Поэтому я рекомендую убрать `this`. Вы станете более квалифицированным и счастливым программистом, научившись работать без `this`. Я ничего у вас не отнимаю, а предлагаю путь к лучшей жизни через написание более качественных программ.

Программисты, использующие классы, так и покинут этот мир, не узнав, насколько они были несчастны. Привязка `this` не дает ничего хорошего.

Само слово *this* — это указательное местоимение. Наличие *this* в языке затрудняет разговор на нем. Он становится похож на программирование с участием пары знаменитых комиков — Эбботта и Костелло.

Глава 17

Как работает код без классов

●○○○●

> И думаешь, что ты умен вне всяких классов
> и свободен.
>
> *Джон Леннон (John Lennon)*

Одной из ключевых идей в разработке объектно-ориентированного программирования была модель обмена данными между частями программы. Имя метода и его аргументы нужно представлять в виде сообщений. Вызов метода посылает объекту сообщение. Каждый объект характеризуется собственным поведением, которое проявляется при получении конкретных сообщений. Отправитель полагает, что получатель знает, что ему делать с сообщением.

Одной из дополнительных выгод является полиморфизм. Каждый объект, распознавший конкретное сообщение, имеет право на его получение. Что происходит потом, зависит от специализации объекта. И это весьма продуктивная мысль.

К сожалению, мы стали отвлекаться на наследование — весьма эффективную схему повторного использования кода. Его важность связана с возможностью уменьшить трудозатраты при разработке программы. Наследование выстраивается на схожем замысле, за исключением *некоторых нюансов*. Можно сказать, что некоторый объект или класс объектов подобен какому-то другому объекту или классу объектов, но имеет некоторые важные отличия. В простой ситуации все работает замечательно. Следует напомнить, что современное ООП началось со Smalltalk — языка программирования для детей. По мере усложнения ситуации наследование становится проблематичным. Оно порождает сильное сцепление классов. Изменение одного класса может вызвать сбой в тех классах, которые от него зависят. Модули из классов получаются просто никудышными.

Кроме того, мы наблюдаем повышенное внимание к свойствам, а не к объектам. Особое внимание уделяется методам получения (get-методам) и присваивания (set-методам) значений каждому отдельно взятому свойству, а в еще менее удачных проектах свойства являются открытыми и могут быть изменены без ведома объекта. Вполне возможно ввести в обиход более удачный проект, где свойства скрыты, а методы обрабатывают транзакции, не занимаясь только лишь изменением свойств. Но такой подход применяется нечасто.

Кроме того, существует слишком сильная зависимость от типов. Типы стали особенностью Фортрана и более поздних языков, так как были удобны создателям компилятора. С тех времен мифология вокруг типов разрослась, обзаведясь экстравагантными заявлениями о том, что типы защищают программу от ошибок. Несмотря на преданность типам, ошибки не ушли из повседневной практики.

Типы вызывают уважение, их хвалят за раннее обнаружение просчетов на стадии компиляции. Чем раньше будет обнаружена оплошность, тем меньших затрат потребует ее ликвидация. Но при надлежащем тестировании программы все эти просчеты обнаруживаются очень быстро. Поэтому ошибки идентификации типов относятся к категории малозатратных.

Типы не виноваты в появлении труднообнаруживаемых и дорогостоящих ошибок. Их вины нет и в возникновении проблем, вызываемых такими ошибками и требующих каких-то уловок. Типы могут подтолкнуть нас к использованию малопонятных, запутанных и сомнительных методов программирования.

Типы похожи на диету для похудания. Диету не обвиняют в возвращении и увеличении веса. Ее также не считают причиной страданий или вызванных ею проблем со здоровьем. Диеты вселяют надежду, что вес придет в здоровую норму и мы продолжим есть нездоровую пищу.

Классическое наследование позволяет думать, что мы создаем качественные программы, в то время как мы допускаем все больше ошибок и применяем все больше неработоспособных наследований. Если игнорировать негативные проявления, типы представляются крупной победой. Преимущества налицо. Но если присмотреться к типам более пристально, можно заметить, что затраты превышают выгоду.

Конструктор

В главе 13 мы работали с фабриками — функциями, возвращающими функции. Что-то похожее теперь можем сделать с конструкторами — функциями, возвращающими объекты, которые содержат функции.

Начнем с создания `counter_constructor`, похожего на генератор `counter`. У него два метода, `up` и `down`:

```
function counter_constructor() {
    let counter = 0;

    function up() {
        counter += 1;
        return counter;
    }

    function down() {
        counter -= 1;
```

```
        return counter;
    }

    return Object.freeze({
        up,
        down
    });
}
```

Возвращаемый объект заморожен. Он не может быть испорчен или поврежден. У объекта есть состояние. Переменная `counter` — закрытое свойство объекта. Обратиться к ней можно только через методы. И нам не нужно использовать `this`.

Это весьма важное обстоятельство. Интерфейсом объекта являются исключительно методы. У него очень крепкая оболочка. Мы получаем наилучшую инкапсуляцию. Прямого доступа к данным нет. Это весьма качественная модульная конструкция.

Конструктор — это функция, возвращающая объект. Параметры и переменные конструктора становятся закрытыми свойствами объекта. В нем нет открытых свойств, состоящих из данных. Внутренние функции становятся методами объекта. Они превращают свойства в закрытые. Методы, попадающие в замороженный объект, являются открытыми.

В методах должны реализовываться транзакции. Предположим, к примеру, что у нас есть объект `person`. Может потребоваться изменить адрес того лица, чьи данные в нем хранятся. Для этого не нужен отдельный набор функций для изменения каждого отдельного элемента адреса. Нужен один метод, получающий объектный литерал, способный дать описание всех частей адреса, нуждающихся в изменении.

Одна из блестящих идей в JavaScript — объектный литерал. Это приятный и выразительный синтаксис для кластеризации информации. Создавая методы, потребляющие и создающие объекты данных, можно сократить количество методов, повышая тем самым целостность объекта.

Получается, у нас есть два типа объектов.

- Жесткие объекты содержат только методы. Эти объекты защищают целостность данных, содержащихся в замыкании. Они обеспечивают нас полиморфизмом и инкапсуляцией.
- Мягкие объекты данных содержат только данные. Поведение у них отсутствует. Это просто удобная коллекция, с которой могут работать функции.

Есть мнение, что ООП началось с добавления процедур к записям языка Кобол, чтобы обеспечить тем самым некое поведение. Полагаю, что сочетание методов и свойств данных было важным шагом вперед, но стать последним шагом не должно.

Если жесткий объект должен быть преобразован в строку, нужно включить метод `toJSON`. Иначе `JSON.stringify` увидит его как пустой объект, проигнорировав методы и скрытые данные (см. главу 22).

Параметры конструктора

Однажды я создал конструктор, получающий десять аргументов. Им было очень сложно пользоваться, поскольку никто не мог запомнить порядок следования аргументов. Позже было подмечено, что никто не использует второй аргумент, я мне захотелось убрать его из списка параметров, но это сломало бы весь уже разработанный код.

Будь я предусмотрительнее, у меня был бы конструктор, получающий в качестве параметра один объект. Обычно он берется из объектного литерала, но может поступать и из других источников, например из JSON-содержимого.

Это дало бы множество преимуществ.

- Ключевые строки придают коду задокументированный вид. Код легче читается, поскольку он сам сообщает вам, что представляет собой каждый аргумент вызывающей стороны.
- Аргументы могут располагаться в любом порядке.
- В будущем можно добавлять новые аргументы, не повреждая существующий код.
- Неактуальные параметры можно игнорировать.

Чаще всего параметр используют для инициализации закрытого свойства. Это делается следующим образом:

```
function my_little_constructor(spec) {
    let {
        name, mana_cost, colors, type, supertypes, types, subtypes, text,
        flavor, power, toughness, loyalty, timeshifted, hand, life
    } = spec;
```

Этот код создает и инициализирует 15 закрытых переменных, используя свойства с такими же именами из `spec`. Если в `spec` нет соответствующего свойства, происходит инициализация новой переменной, которой присваивается значение `undefined`. Это позволяет заполнять все пропущенное значениями по умолчанию.

Композиция

Яркая выразительность и эффективность JavaScript позволяют создавать программы в классической парадигме, хотя этот язык и не относится к классическим. JavaScript позволяет также вносить улучшения. Мы можем работать с функциональной композицией. Итак, вместо добавления *чего-то в качестве исключения* можно получить *понемногу того и этого*. Конструктор имеет следующий общий вид:

```
function my_little_constructor(spec) {
    let {компонент} = spec;
    const повторно_используемый = other_constructor(spec);
    const метод = function () {
        // могут применяться spec, компонент, повторно_используемый, метод
    };
```

```
    return Object.freeze({
        метод,
        повторно_используемый.полезный
    });
}
```

Ваш конструктор способен вызвать столько других конструкторов, сколько нужно для получения доступа к управлению состоянием и обеспечиваемому ими поведению. Ему даже можно передать точно такой же объект `spec`. Документируя `spec`-параметры, мы перечисляем свойства, нужные `my_little_constructor`, и свойства, необходимые другим конструкторам.

Иногда можно просто добавить полученные методы к замороженному объекту. В иных случаях у нас есть новые методы, вызывающие полученные методы. Тем самым достигается повторное использование кода, похожее на наследование, но без сильного сцепления. Вызов функции является исходной схемой повторного применения кода, и ничего лучше еще не придумано.

Размер

При таком подходе к конструированию объекта задействуется больше памяти, чем при использовании прототипов, поскольку каждый жесткий объект содержит все методы объекта, а прототипный объект содержит ссылку на прототип, содержащий методы. Существенна ли разница в потреблении памяти? Соизмеряя разницу с последними достижениями в повышении объема памяти, можно сказать: нет. Мы привыкли считать память в килобайтах. А теперь считаем ее в гигабайтах. На этом фоне разница совершенно не чувствуется.

Разницу можно сократить, улучшив модульность. Акцент на транзакциях, а не на свойствах позволяет уменьшить количество методов, а заодно улучшить связанность.

Классическая модель характеризуется однообразием. Каждый объект должен быть экземпляром класса. JavaScript снимает эти ограничения. Не все объекты нуждаются в соблюдении столь жестких правил.

Например, я полагаю, что нет никакого смысла в том, чтобы точки обязательно были жесткими объектами с методами. Точка может быть простым контейнером для двух или трех чисел. Точки передаются функциям, которые способны выполнять проекцию, или интерполяцию, или еще что-то, что можно делать с точками. Это может оказаться гораздо продуктивнее, чем создание подклассов точек, придающих им особое поведение. Пусть работают функции.

Глава 18

Как работают концевые вызовы

●○○●○

> Бывает момент, когда настоящий мужчина должен взять быка за рога и достойно справиться со сложившейся ситуацией.
>
> *У. К. Филдс (W. C. Fields)*

Ускорение наших программ зависит от оптимизации наших систем. Оптимизация — это ломка правил, но при этом отход от них должен быть таким, чтобы она не прослеживалась. В результате оптимизации хорошая программа не должна превращаться в плохую. Оптимизация не должна привносить ошибки.

Самое важное в оптимизации не только то, что она не добавляет ошибок, но и то, что она избавляет хорошие программы от целого класса ошибок, привнося новые парадигмы программирования. Я имею в виду *оптимизацию концевых вызовов* (Tail Call Optimization). Некоторые специалисты придают этой оптимизации настолько большое значение, что призывают не умалять его, называя данный прием оптимизацией. Они назвали его *правильным концевым вызовом* (Proper Tail Calls). Любой другой способ реализации концевых вызовов считается неправильным.

Я предпочитаю называть это оптимизацией, поскольку основная масса программистов не придает особого значения правильности, но любит оптимизацию, даже если она не дает ощутимого эффекта. Я же хочу, чтобы у них возникло желание пользоваться этим приемом.

Концевой вызов получается, когда своим последним действием функция возвращает результат вызова функции. В следующем примере `continuize` — это фабрика, получающая тип функции `any` и возвращающая функцию `hero`. Функция `hero` вызывает `any` и передает возвращаемое значение функции `continuation`:

```
function continuize(any) {
    return function hero(continuation, …args) {
        return continuation(any(…args));// <-- концевой вызов
    };
}
```

Когда функция возвращает результат вызова функции, мы называем этот вызов *концевым*. Плохо, что мы называем его *концевым*, а не *возвращающим*.

Оптимизация концевых вызовов относится к довольно простым операциям, но у нее масса вариантов применения. Если использовать обычный набор инструкций в качестве метафоры, то код, генерируемый `continuize`, может включать следующие машинные инструкции:

```
call     continuation         # вызов функции continuation
return                        # возвращение к функции, вызвавшей hero
```

Инструкция `call` помещает адрес следующей инструкции (в данном случае это `return`) в стек вызова. Затем она передает управление той функции, чей адрес числится в реестре с пометкой `continuation`. Когда функция `continuation` выполнит свою работу, адрес возврата будет извлечен из стека и на него будет выполнен переход. Инструкция `return` опять извлечет данные из стека и перейдет на инструкцию, вызвавшую `hero`.

Оптимизация заменит эти две инструкции одной:

```
jump     continuation         # переход к функции continuation
```

Теперь мы не помещаем в стек вызовов адрес инструкции `return`. Функция `continuation` возвращается к функции, вызвавшей `hero`, а не к самой `hero`. Концевой вызов похож на `goto` с аргументами, но без какой-либо опасности, присущей `goto` и подтолкнувшей к избавлению от нее.

Получается, что оптимизация сэкономила одну инструкцию, помещение данных в стек и извлечение их оттуда. Казалось бы, мелочь. Чтобы лучше разобраться, в чем тут преимущество, посмотрим, как на самом деле вызовы работают в JavaScript. Когда вызывается функция, происходит следующее.

- Вычисляются выражения аргументов.
- Создается объект активации, размер которого позволяет ему хранить все параметры и переменные функции.
- В новом объекте активации сохраняется ссылка на вызываемый функциональный объект.
- В новом объекте активации значения из аргументов сохраняются в параметрах. Пропущенные аргументы рассматриваются как содержащие значение `undefined`. Лишние аргументы игнорируются.
- Всем переменным в новом объекте активации присваивается значение `undefined`.
- Для поля активации `next instruction field` устанавливается значение, указывающее на инструкцию, следующую сразу же за вызовом.
- Для поля `caller` в новой активации устанавливается значение, указывающее на текущую активацию. Фактически это не стек вызовов, а связанный список активаций.
- Новая активация становится текущей.
- Начинается выполнение вызванной функции.

Оптимизация работает несколько иначе.

- Вычисляются выражения аргументов.
- Если текущая активация достаточно велика, в качестве нового объекта активации используется текущий.
- В противном случае:
  - создается объект активации, размер которого позволяет ему хранить все параметры и переменные функции;
  - из текущего объекта активации в новый объект копируется поле `caller`;
  - новый объект активации становится текущим.
- В новом объекте активации сохраняется ссылка на вызываемый функциональный объект.
- В новом объекте активации значения из аргументов сохраняются в параметрах. Пропущенные аргументы рассматриваются как содержащие значение `undefined`. Лишние аргументы игнорируются.
- Всем переменным в новом объекте активации присваивается значение `undefined`.
- Начинается выполнение вызванной функции.

Важность этого различия заключается в том, что если размер объекта активации достаточно велик, как чаще всего и бывает, то не нужно выделять еще один такой объект. Теперь можно повторно воспользоваться текущим объектом активации. Цепочки стека вызовов не очень длинные — обычно несколько сотен записей, поэтому, если ожидается применение концевых вызовов, есть смысл для реализации всегда выделять объекты активации одного максимального размера. Экономия времени на распределение памяти и сборку мусора может быть весьма существенной. Но это еще не все.

При оптимизации концевых вызовов рекурсивные вызовы способны стать такими же быстрыми, как и циклы. С функциональной точки зрения это важно, потому что циклы по своей природе не относятся к чистому коду. Чистота приходит с рекурсией. Благодаря данной оптимизации аргумент снижения производительности при рекурсии сведен на нет.

В общем виде структура цикла выглядит так:

```
while (true) {
    выполнение каких-либо действий
    if (done) {
        break;
    }
    выполнение дополнительных действий
}
```

А концевая рекурсивная функция может иметь следующий вид:

```
(function loop() {
    выполнение каких-либо действий
    if (done) {
        return;
    }
    выполнение дополнительных действий
    return loop();                          // <-- концевой вызов
}());
```

Здесь показано соотношение между циклами и рекурсивными функциями. В целом рекурсивные функции будут выглядеть более элегантно, передавая обновленное состояние в параметрах и возвращая значения вместо реализации присваивания.

Итак, у нас есть рекурсия, чтобы дойти до любой глубины, не опасаясь исчерпания памяти или искусственно созданных ошибок переполнения стека. Теперь качественные рекурсивные функции будут работать правильно. *Правильные концевые вызовы* (Proper Tail Calls) — это не какая-то особенность, это исправление дефекта. Стандарт требует реализации данной оптимизации, поскольку важно предотвращать сбои хороших программ.

Концевая позиция

Вызов концевой, если функцией возвращается не что иное, как ее возвращаемое значение. Следовательно:

```
return (
    typeof any === "function"
    ? any()                                 // <-- концевой вызов
    : undefined
);
```

является концевым вызовом. Концевыми вызовами могут также быть выражения, составленные из логического И (`&&`) и логического ИЛИ (`||`), но:

```
return 1 + any();                           // <-- не концевой вызов
```

не концевой вызов. Последним действием функции было возвращение результата сложения.

```
any();                                      // <-- не концевой вызов
return;
```

Это был не концевой вызов, поскольку им возвращается `undefined`, а не возвращаемое значение функции.

```
const value = any();                        // <-- не концевой вызов
return value;
```

Это также был не концевой вызов, поскольку им возвращается содержимое значения. Нам известно, что значение является результатом `any()`, но вызов был не в концевой позиции.

Рекурсия вообще не концевой вызов.

```
function factorial(n) {
    if (n < 2) {
        return 1;
    }
    return n * factorial(n - 1);             // <-- не концевой вызов
}
```

Рекурсивный вызов `factorial` не находится в концевой позиции, поэтому он создает запись активации при каждой итерации, но вызов можно переместить в концевую позицию:

```
function factorial(n, result = 1) {
    if (n < 2) {
        return result;
    }
    return factorial(n - 1, n * result);     // <-- концевой вызов
}
```

Эта версия будет оптимизирована. Рекурсивные вызовы не станут создавать объекты активации. Вместо этого произойдет переход в начало функции. Для освежения параметров здесь применяются аргументы, а не присваивание.

Возвращение нового функционального объекта не является концевым вызовом:

```
return function () {};                       // <-- не концевой вызов
```

если только новый функциональный объект тут же не будет вызван:

```
return (function () {}());                   // <-- концевой вызов
```

Исключительные случаи

Бывают ситуации, когда оптимизацию следует отменить, чтобы хорошие программы не превратились в плохие.

Концевой вызов внутри блока `try` оптимизировать невозможно. Оптимизация сэкономила бы время и память за счет отказа от привязки объекта активации для данного вызова к стеку вызовов. Но блок `try` может передать управление блоку `catch` этого вызова, и, чтобы это произошло, объект активации не следует оптимизировать. Кроме того, это станет еще одной веской причиной избавления от неправильного применения исключений.

Если рассматриваемая функция создает новый функциональный объект, имеющий какие-нибудь свободные переменные, то у нового функционального объекта должен быть доступ к объекту активации ее создателя, следовательно, объект активации не следует оптимизировать.

Стиль передачи продолжений

В стиле передачи продолжений (Continuation Passing Style) функции получают дополнительный параметр `continuation` — функцию, получающую результат. Фабрика `continuize`, показанная в начале этой главы, может превратить любую функцию в функцию, которая также получает продолжение. В данном стиле продолжением является функция, представляющая собой продолжение программы. Выполняемая программа всегда движется вперед, изредка нуждаясь в посещении предыдущих мест. Данный стиль — весьма эффективное средство управления случайностями. Он нашел свое применение, которое невозможно без оптимизации концевых вызовов, в транспилировании и др.

Отладка

Исключение активаций из стека вызова может усложнить отладку, поскольку больше не получится просмотреть все шаги, приводящие к текущему положению. Но хороший отладчик способен облегчить решение задачи, предложив режим, в котором он станет копировать состояние каждого объекта активации и всегда оставлять небольшой набор самых последних данных. Это не помещает проявлению преимуществ оптимизации, однако позволит проверить традиционную трассировку стека.

Глава 19

Как работает чистота

●○○●●

> Ценой стремления к чистоте становится перфекционизм.
>
> *Кельвин Триллин (Calvin Trillin)*

Функциональное программирование воспринимается как программирование с функциями. К сожалению, это неоднозначное определение. Оно может подразумевать программирование с математическими функциями, где интервалы значений отображаются на диапазоны значений. Или же это программирование с программными функциями, что в большинстве языков программирования означает параметризованные строки инструкций.

Математические функции считаются более чистыми по сравнению с программными. Чистота нарушается изменениями. Чистый результат функции определяется только ее входными данными. Кроме обеспечения выходных данных, функция не выполняет никаких других действий. На основе заданных входных данных всегда получаются одни и те же выходные данные.

А вот JavaScript, похоже, выглядит как какой-то праздник изменений. Основными механизмами изменений являются операторы присваивания, и в JavaScript их 15 (или 17, если вы считаете, что нужно опасаться еще и сдвоенных плюсов (++) и минусов (--)):

- `=` — присваивание;
- `+=` — присваивание со сложением;
- `-=` — присваивание с вычитанием;
- `*=` — присваивание с умножением;
- `/=` — присваивание с делением;
- `%=` — присваивание с остатком от деления;
- `**=` — присваивание с возведением в степень;
- `>>>=` — присваивание со сдвигом вправо;
- `>>=` — присваивание со сдвигом вправо и с расширением знака;

- `<<=` — присваивание со сдвигом влево;
- `&=` — присваивание с поразрядным И;
- `|=` — присваивание с поразрядным ИЛИ;
- `^=` — присваивание с поразрядным исключающим ИЛИ;
- `++` — преинкремент;
- `++` — постинкремент;
- `--` — предекремент;
- `--` — постдекремент.

Как и большинство широко востребованных языков, JavaScript любит всякий хлам, предпочитая изменения чистоте. В JavaScript символ равенства (`=`) выбран для изменений (нечистого кода), а не для обозначения эквивалентности (в чистом коде).

Благословение чистоте

Чистота дает несколько ценных преимуществ.

Чистота обычно предполагает возможность достижения превосходной модульности. В чистых функциях весьма сильное связывание. В функции есть все, что позволяет предоставить единый результат. Вне ее не происходит ничего, что могло бы ослабить ее связанность. Чистые функции характеризуются чрезвычайно слабым сцеплением. Чистая функция зависит только от своих входных данных. Создание качественных модулей может стать очень непростой задачей, но при соблюдении чистоты хорошая модульность приходит сама собой.

Чистые функции намного проще тестировать. Для этого не нужно использовать имитацию, фиктивный объект или заглушку. Поскольку известно, что чистая функция возвращает из входных данных правильные выходные данные, мы знаем, что никакие изменения среды окружения никогда не заставят чистую функцию возвратить какое-либо другое значение. Ошибки, исчезающие при выключении и включении питания, возникают из-за отсутствия чистоты.

Чистые функции обладают высокой сочетаемостью. Они легко собираются в более крупные и сложные функции, которые также могут быть чистыми и составными, поскольку у них нет побочных эффектов и внешних зависимостей или влияний.

Когда-нибудь чистота позволит добиться существенного повышения производительности. Чистота предоставляет отличные решения проблем надежности и производительности, мешающих потокам. Если в многопоточных системах два потока пытаются одновременно получить доступ к одной и той же области памяти, может возникнуть конфликт, переходящий в порчу данных или в системный сбой. Такие конфликты бывает очень трудно диагностировать. Смягчить конфликтную ситуацию способно взаимное исключение, но оно также может вызвать задержки, взаимные блокировки и сбои системы.

Чистые функции способны придать потокам безопасность и эффективность. Поскольку такие функции не вызывают изменений, совместное использование памяти не создает никакого риска. Метод массива `map` при условии применения чистой функции может распределять элементы массива по всем доступным ядрам. Вычисление способно выполняться очень быстро. Повышение производительности может иметь линейную зависимость. Чем больше ядер, тем быстрее идет вычисление. От этого выигрывает весьма обширный класс программ.

С целью включения в JavaScript рассматривается множество весьма нелепых компонентов. И многие из них входят в язык, отвлекая внимание на всякую чепуху. А нам, в частности, нужно распараллеливание чистых функций.

Как добиться чистоты

Понятно, что чистота дает огромные преимущества. Так можно ли добавить ее в язык? Нет. Чистота не является той самой характерной особенностью, которую можно было бы добавить. Она сродни надежности и безопасности. Добавить в систему надежность невозможно, реально лишь устранить ненадежность. Нельзя добавить в нее и безопасность. Нужно избавляться от опасности. Также нельзя добавить чистоту. Следует избавляться от нечистого кода. Он вносит искажения, допускающие отступление от математической модели функций.

ECMA не в состоянии удалить из JavaScript его неудачные составляющие, поэтому их количество продолжает расти, но ведь можно просто отказаться от них. Можно сделать язык чистым для себя, более дисциплинированно подходя к процессу программирования. Если отказаться от применения нечистой функции, она больше не сможет своей нечистотой искажать сущность используемых нами тел функций.

Итак, сделаем краткий обзор всех нечистых компонентов языка.

Разумеется, нужно избавиться от всех операторов присваивания, а также от инструкций `var` и `let`. Инструкцию `const` можно оставить. Она создает и инициализирует переменную, но не позволяет ей изменяться.

Следует избавиться от операторов и методов, изменяющих объекты, включая оператор `delete` и методы, подобные `Object.assign`. Нужно также избавиться от методов изменения массивов, таких как `splice` и `sort`. Метод работы с массивами `sort` мог бы быть чистой функцией, но его реализация изменяет исходный массив, поэтому решение отказаться от его использования остается в силе.

Нужно избавиться и от методов получения и установки значений (get- и set-методов). Set-методы, несомненно, являются инструментами внесения изменений и, как и get-методы, существуют для создания побочных эффектов. Все побочные эффекты разрушительны и должны быть устранены.

RegExp-метод `exec` изменяет свойство `lastIndex`, поэтому от него нужно отказаться. Его конструкция могла бы быть более удачной, но этого не произошло.

Задача инструкции `for` — изменение индуктивной переменной, поэтому ее тоже не стоит употреблять. Нужно отказаться и от `while` и `do`. Самая чистая форма итерации — это концевая рекурсия.

Следует отказаться от конструктора `Date`. При каждом его вызове мы получаем разные значения. Это не сообразуется с чистотой. Также нужно отказаться от `Math.random`. Неизвестно, какой результат всякий раз выдаст вам этот метод.

Нужно отказаться и от пользователей. Каждое взаимодействие с человеком может возвращать разные результаты. Такое общение невозможно признать чистым.

И наконец, нужно отказаться от сети. У лямбда-исчисления нет способа выражения информации, существующей на одной машине и отсутствующей на другой. По аналогии с этим у универсальной машины Тьюринга (Universal Turing Machine) нет связи по Wi-Fi.

Проблемы со Вселенной

Наша Вселенная не отличается чистотой. Она конечна, полностью распределена и в высокой степени параллельна. Наша модель программирования должна считаться с этой истиной. Если программы взаимодействуют со Вселенной, в частности с ее людскими агентами, то некоторые части программ могут быть нечистыми.

Нужно стремиться делать программы как можно более чистыми, поскольку это дает явные преимущества. Объекты по-прежнему ценны, даже если они обладают состоянием, а состояние представляет интерес, только если оно способно изменяться. Нужно конструировать объекты так, чтобы изменение их состояния было под строжайшим контролем.

Континуум

О чистоте было бы проще рассуждать, если бы она была бинарной: функция либо чистая, либо нечистая. К сожалению, все не так просто. Есть несколько способов представления чистоты.

Изменения не сообразуются с чистотой, но в конечном счете в глубинах даже самых чистых систем кроется что-то нечистое. Функциональная система всегда создает новые объекты активации, и процесс перемещения объектов из доступной памяти в активную и обратно требует изменений. В этом нет ничего плохого. В общем-то, хочется добиться как можно большего распространения чистоты, а то, что невозможно сделать чистым, следует спрятать.

Генераторы, рассмотренные в главе 13, и объекты из главы 17 хорошо скрывают свои состояния. Внешний код способен изменить состояние только через порядок, установленный функцией-генератором и методами. Генераторы и объекты нельзя изменить внешним присваиванием. Они не абсолютно чистые, но все же

существенно улучшены по сравнению с другими моделями ООП. Они представляют собой примеры возможности управления границей чистоты.

Одно из определений чистоты: это свобода от присваиваний и других средств внесения изменений. Тогда чист ли этот код?

```
function repeat(generator) {
    if (generator() !== undefined) {
        return repeat(generator);
    }
}
```

Функция `repeat` получает функцию-генератор и вызывает ее, пока не будет возвращено значение `undefined`. Она чиста телом, но не духом. Собственных нечистых действий она не совершает, но допускает нечистоту генератора. На этом среднем уровне находятся многие функции высшего порядка.

Функция может выполнять присваивание и при этом считаться чистой. Если она объявляет локальные переменные и использует циклы и присваивания для изменения этих переменных, это может выглядеть нечистым. Но мы могли бы присвоить такую функцию параллельной функции `map`, и она бы отлично работала. Это было бы чистотой духа, но не тела. Если смотреть на это как на черный ящик, то все будет чисто. Нечисто окажется, только если заглянуть внутрь.

Итак, существует некий континуум чистоты, где на самом верху находятся математические функции, ниже располагаются функции, чистоты которых достаточно для параллельных приложений, затем чистые функции более высокого порядка, после них функции более высокого порядка с состоянием, а затем мили разного хлама, загрязняющего сеть: применение классов, процедур и в самом низу все, в чем используются глобальные переменные.

Глава 20

Как работает событийное программирование

●○●○○

Мо! Ларри! Сыр!

Джером Лестер Хорвиц (Jerome Lester Horwitz)

В самом начале было *последовательное* программирование. Первые компьютеры были просто автоматическими калькуляторами. Компьютер подпитывали чередой данных и команд, и он выполнял по одной команде, пока не выдавал результат и не останавливался. Эта модель последовательного выполнения хорошо подходила для расчетов и массовой обработки данных.

Первые языки программирования были разработаны именно в эпоху последовательного программирования. Влияние этих языков оказалось столь сильным, что многие современные языки программирования прочно застряли в последовательной парадигме. Старая парадигма не предусматривала того, что сейчас делают наши программы, это неоправданно усложняет программирование современных систем, делая их ненадежными и небезопасными.

Характерная особенность последовательных языков — блокировка ввода и вывода. Если программа пытается считывать данные из файла или из сети, ее выполнение блокируется до завершения операции ввода. Для Фортрана в этом был смысл. Программа блокируется, когда ей нужно считать данные с перфокарт в считывателе. Программе нечего было делать, пока данные не были готовы, поэтому блокировка не вызывала проблем. В большинстве современных языков по-прежнему реализуется модель ввода-вывода языка Фортран. Но не в JavaScript. Главной задачей JavaScript было взаимодействие с людьми, поэтому в нем предпочтение отдано более удачной модели. JavaScript завяз в последовательной модели куда меньше большинства других языков.

Параллелизм

Последовательная модель сломалась, когда компьютеры начали взаимодействовать с людьми, а затем и с другими компьютерами. Это потребовало *параллельного* программирования — возможности одновременно выполнять несколько действий.

Однородный параллелизм обеспечивает поддержку многих похожих операций, происходящих в одно и то же время. Его пример уже был показан при использовании методов работы с массивами, получающими чистые функции и обрабатывающими сразу все элементы массива.

Неоднородный параллелизм поддерживает совместные действия специализированных процессов, каждый из которых имеет различные обязанности, но все они работают вместе как единая команда. Сложность состоит в том, чтобы убедиться, что компоненты команды не ведут себя как три балбеса. Это весьма серьезная проблема. Неразумное поведение — неизбежное следствие плохой архитектуры.

Потоки

Потоки — один из самых старых, но по-прежнему широко востребованных механизмов неоднородного параллелизма. Потоки (реальные или виртуальные) — это одновременно работающие центральные процессоры, совместно использующие одну и ту же память. Потоки хорошо справляются с работой, когда в них запускаются чистые функции. Если функции нечисты, поведение становится неразумным.

Здесь показаны два потока, Мо и Ларри, совместно использующие переменную. Мо прибавляет к переменной единицу, Ларри — двойку. Ожидается, что результатом станет тройка.

Мо	Ларри	variable
		0
variable += 1		1
	variable += 2	3

Здесь Мо внес изменение первым. А может случиться так, что первым это сделает Ларри. В данном конкретном случае не имеет значения, кто из них выиграет гонку, потому что операции коммуникативные. Если бы Ларри использовал вместо инструкции присваивания `+=` инструкцию `*=`, то результат зависел бы от того, кто из них вырвался вперед. Есть и другие, еще более проблематичные результаты.

Если посмотреть на этот код на более низком уровне, инструкция присваивания `+=` может быть откомпилирована в три простые машинные инструкции: `load` (загрузить), `add` (прибавить) и `store` (сохранить).

Мо	Ларри	variable
		0
load variable		0
add 1	load variable	0
store variable	add 2	1
	store variable	2

Здесь показано соревнование «чтение — изменение — запись». И Мо, и Ларри извлекают переменную, когда ее состояние равно нулю. Оба они прибавляют значение к извлеченной ими переменной, а затем каждый сохраняет получившуюся у него сумму. На этот раз изменение, внесенное Мо, переписывается. В другой раз переписывается изменение, внесенное Ларри. Возможно, бóльшую часть времени Мо и Ларри не извлекают одно и то же значение, поэтому код зачастую работает правильно.

У данного кода есть возможность работать правильно, но есть и вероятность делать это неправильно. Увидеть в двух простых строках программы потенциальную ошибку довольно трудно. В сложных программах такие ошибки способны прятаться очень хорошо.

Вычислительная нагрузка способна изменить чередование инструкций. В ходе разработки и тестирования код может работать вполне корректно, а в производственном режиме давать сбой. Или он хорошо работает целый год, а в декабре дает сбой. Периодически проявляющиеся ошибки в потоках считаются самыми худшими, самыми затратными из всех ошибок. Когда на поведение программы способна повлиять космическая случайность, воспроизвести условия возникновения ошибки может оказаться практически невозможно. Проверка типов такие ошибки не находит. Тесты их также не находят. Ошибки могут появляться крайне редко, поэтому чрезвычайно сложно их отладить или обрести уверенность в том, что исправление не усугубит ситуацию.

Опасность состязания потоков реально снизить посредством взаимного исключения, которое блокирует доступ к критическим участкам памяти, а при возникновении конфликта блокирует потоки, препятствуя их выполнению. Применение блокировок способно оказаться затратным для вычислений. Может быть заблокирован поток, который окажется не способен снять свои блокировки. Такое состояние называется *взаимной блокировкой* и является еще одним режимом сбоя, трудно поддающимся предотвращению, воспроизведению или исправлению.

Самой серьезной ошибкой проектировщиков в Java и во многих похожих языках стало то, что они не смогли решить, что именно требуется от самого языка — быть языком системы или языком приложений. Попытка сделать его и тем и другим потребовала применения потоков в приложениях, что непростительно.

Потоки в операционных системах могут быть неизбежным злом. А в приложениях это просто зло. К счастью, JavaScript не использует потоки таким вот образом. Для обеспечения параллелизма есть более удачный способ.

Событийное программирование

Событийной называется функция, немедленно возвращающая управление, возможно, еще до завершения затребованной от нее работы. Результат со временем будет сообщен через функцию обратного вызова или путем отправки сообщения, но не в виде немедленно возвращаемого значения.

Событийность обеспечивает способ управления широкой активностью без предоставления приложений потокам. Фактически имеются системы, в которых есть потоки приложений, зависящие от событийности управления своими пользовательскими интерфейсами только потому, что это более простой и надежный способ обработки событий с течением времени. Событийное программирование основано на двух принципах: на применении функций обратного вызова и использовании цикла обработки.

Функцией обратного вызова является функция, которая будет вызвана в будущем, когда случится что-нибудь интересное, например:

- поступит сообщение;
- завершится какая-то работа;
- человек станет общаться с программой;
- датчик на что-то отреагирует;
- истечет определенное время;
- произойдет некий сбой.

Функция обратного вызова передается функции, которая запускает или отслеживает активность. В более простых системах функция обратного вызова прикрепляется к объекту, представляющему активность. В браузерах функцию обратного вызова можно прикрепить к DOM-узлу, назначив обратный вызов определенному свойству этого узла:

```
my_little_dom_node.onclick = функция обратного вызова;
```

или вызвав в отношении объекта метод регистрации события:

```
my_little_dom_node.addEventListener("click", функция обратного вызова, false);
```

Работают обе представленные формы. Когда пользователь щелкает кнопкой мыши на конкретном DOM-узле, *функция обратного вызова* (она же *обработчик события*) вызывается, вероятно выполняя какое-то полезное действие в ответ на событие.

Еще одной идеей, заложенной в основу событийного программирования, является цикл обработки, известный также как цикл обработки событий или цикл обработки сообщений. Цикл обработки принимает из очереди событие или сообщение с наивысшим приоритетом и вызывает функцию обратного вызова, которая зарегистрирована для получения этого события или сообщения. Функция обратного вызова выполняется до завершения своей работы. Обратный вызов не нуждается в блокировке области памяти или применении взаимного исключения. Обратный вызов не прерывается, поэтому вероятности возникновения состязательных условий нет. Когда управление из функции обратного вызова возвращается, цикл обработки извлекает из очереди ее следующий элемент и работа продолжается уже с ним. Это очень надежная модель программирования.

Цикл обработки поддерживает очередь, известную как очередь событий или очередь сообщений, которая содержит входящие события или сообщения. События или сообщения могут быть инициированы функциями обратного вызова как часть их ответов. Скорее всего, события или сообщения поступают из вспомогательных потоков, управляющих пользовательским вводом данных, сетью, системой ввода-вывода и взаимодействием между процессами. Обмен данными с основным потоком, в котором выполняется ваша JavaScript-программа, осуществляется посредством очереди. Взаимное исключение ограничено этой единственной точкой контакта, поэтому событийные системы, как правило, очень эффективны и надежны.

Неожиданное преимущество этой модели заключается в том, что программы JavaScript в веб-браузере обладают удивительной устойчивостью. Если открыть отладчик, а затем выйти в Интернет, можно увидеть практически постоянный поток исключений и сбоев. Причиной может стать низкая квалификация веб-разработчиков. Однако практически все, похоже, работает.

В системах с потоками, если в одном потоке выдается исключение, его стек опустошается. И теперь этот поток может оказаться несогласованным с другими потоками, что способно привести к сбою каскадных потоков.

В JavaScript используется один поток. Основная часть его жизненного состояния находится в замыкании его функций, а не в стеке. Так что все, как правило, продолжает работать. Пока есть работоспособная кнопка, пользователи по-прежнему могут что-то делать, не подозревая о множестве ошибок, происходящих за кулисами.

Закон ходов

Каждая итерация в цикле обработки называется ходом. Это название пришло из таких игр, как шахматы и покер, где в каждый момент времени играть может только один игрок. Он делает ход, который после перемещения заканчивается, и, пока игра не завершена, начинается ход другого игрока.

У игр есть правила, и у событийной модели тоже есть правило — *закон ходов*.

Не ждать. Не блокировать. Быстро завершать

Закон ходов как прямо, так и косвенно применяется к функциям обратного вызова, вызываемым циклом обработки, и к каждой вызываемой ими функции. Функциям не разрешается зацикливаться в ожидании, что что-то произойдет. Функция не имеет права блокировать основной поток. В браузере такие функции, как `alert`, запрещены. В Node.js функции, имена которых содержат неприятный

суффикс `-Sync`, не допускаются. Функция, выполнение которой занимает много времени, запрещена.

Нарушения закона ходов превращают высокопроизводительную систему в систему с очень низкой производительностью. Нарушения вызывают задержки не только для текущего обратного вызова, но и для всего, что находится в очереди. Задержки могут накапливаться, что увеличивает очередь. Система перестает быть быстрой и отзывчивой.

Поэтому любая функция, нарушающая закон, должна быть исправлена или изолирована в отдельном процессе. Процесс похож на поток, который ни с кем не делит свою память. То есть для обратного вызова имеет смысл отправить часть работы процессу, а для этого процесса — отправить сообщение после его завершения. Сообщение попадает в очередь и в итоге доставляется.

JavaScript ничего не знает о процессах. Процессы — это сервисы, которые могут предоставляться системами, где выполняется JavaScript. Процессы — важная часть конечной модели, поэтому вполне вероятно, что они станут первейшей характерной чертой следующего языка.

Серверные проблемы

Язык JavaScript был создан для циклов событий и со своей ролью справляется весьма успешно. К сожалению, циклы сообщений потребовали от приложения гораздо больших усилий. Дело в том, что характер работы, выполняемой сервером, совершенно иной. В браузере основная часть работы программы — это реакция на события пользовательского интерфейса. Вызывается обработчик событий, он выполняет некоторую работу, обновляет отображение — и все.

У сервера типовой рабочий процесс гораздо сложнее. Вызывается приемник или обработчик сообщений, которому, прежде чем он сможет передать свой ответ, может потребоваться обмен данными с другими системами, возможно расположенными на других машинах. Иногда то, что он получит из одной системы, нужно передать другой системе. Эти операции способны выстраиваться в длинные цепочки. А результаты взаимодействия с другими системами будут передаваться с помощью обратных вызовов. Для этого есть весьма простое программное решение, но сначала давайте рассмотрим три самые популярные ошибки.

Первая ошибка называется *Callback Hell* (безрассудная череда обратных вызовов). Это шаблон, в котором каждая функция обратного вызова содержит код для запроса следующей элементарной операции. Этот запрос обеспечивает обратный вызов, который также запрашивает другую элементарную операцию и т. д. Программы, написанные таким образом, трудно поддаются чтению и обслуживанию и характеризуются невысокой надежностью.

Вторая ошибка называется *Promises* (промисы). В исходном виде промисы были отличной идеей. Они были созданы для поддержки разработки безопасных распреде-

ленных программ. К сожалению, когда промисы были перенесены в JavaScript, они потеряли все свои качества новой парадигмы. Остался лишь неуклюжий механизм потока управления. Промисы не были предназначены для управления локальным потоком управления, поэтому плохо справляются с этой задачей. Они, конечно же, лучше безрассудной череды обратных вызовов, но их работу все равно нельзя признать удовлетворительной.

Третья ошибка называется *Async Await* (асинхронные ожидания). Это пара ключевых слов, используемая для декорирования обычного последовательного кода и его волшебного превращения в событийный код. Прием похож на генераторы ES6 тем, что написанный вами код сильно отличается от получаемого кода. К его заслугам можно отнести сглаживание большей части разочарований, привнесенных промисами. Самое симпатичное в асинхронных ожиданиях — возможность продолжать создавать программы в стиле старой парадигмы. Но в этом кроется и самая серьезная проблема.

Новая парадигма играет важную роль. Понять ее может оказаться нелегко, потому что она новая, но именно так добиваются прогресса. Асинхронное ожидание дает нам возможность работать продуктивно, не стремясь двигаться вперед. Его пользователи пишут код, который сами не до конца понимают. А это плохо. Ширится проблема, связанная с программистами, которые повсюду ставят декораторы `async` и `await`. Поскольку они не осознают свои действия, то не понимают, как их правильно использовать. Не нужно отвергать или скрывать следующую парадигму. Ее нужно принять.

Общий признак всех трех ошибок — тесная связь логики и потока управления. Это снижает уровень связности, поскольку в одну структуру сводится слишком много разнородных действий. Их лучше разделить.

Запросчики

При конструировании нужно применять разбиение на модули. Каждая элементарная операция, которая может выполнять запрос к какому-то серверу или базе данных либо запускать процесс, должна стать отдельной функцией. Функция, которая просто выполняет элементарную операцию, обладает сильной связностью. Такие функции также принимают в качестве первого аргумента функцию обратного вызова. Когда элементарная операция завершена, результат передается обратному вызову. Это уменьшает зависимость от другого кода, давая нам слабое сцепление. Такова рекомендуемая норма модульного конструирования, помогающая нам в новой парадигме.

Для описания функции, принимающей обратный вызов и выполняющей элементарную операцию, которая может быть не завершена до будущего хода, используется слово *«запросчик»* (requestor):

```
function my_little_requestor(обратный_вызов, значение)
```

Функция обратного вызова принимает два аргумента — *значение* и *причину*. *Значение* — это результат работы запросчика или `undefined`, если запросчик дал сбой. Необязательный аргумент *причина* может использоваться для документирования сбоев:

```
function my_little_callback(значение, причина)
```

Функция запросчика способна дополнительно возвратить функцию отмены. Она может быть вызвана для отмены работы по любой причине. Это не отмена сделанного. Функция отмены предназначена только для прекращения ненужной работы. Таким образом, если запросчик запускает весьма затратную операцию на другом сервере и операция больше не требуется, то вызов принадлежащей запросчику функции отмены может отправить сообщение о прекращении на сервер.

```
function my_little_cancel(причина)
```

Фабрики запросчиков

Основная часть вашей работы заключается в создании фабрик, принимающих аргументы и возвращающих запросчики.

Эта фабрика служит простой оболочкой, которая может принять любую функцию с одним аргументом и возвратить запросчик.

```
function requestorize(unary) {
    return function requestor(callback, value) {
        try {
            return callback(unary(value));
        } catch (exception) {
            return callback(undefined, exception);
        }
    };
}
```

Следующая фабрика создает запросчик, способный читать файл на Node.js:

```
function read_file(directory, encoding = "utf-8") {
    return function read_file_requestor(callback, value) {
        return fs.readFile(
            directory + value,
            encoding,
            function (err, data) {
                return (
                    err
                    ? callback(undefined, err)
                    : callback(data)
                );
            }
        );
    };
}
```

Наиболее интересные фабрики создают запросчики, обменивающиеся данными с другими сервисами в диалогах, которые способны растягиваться на несколько ходов. Они могут принимать следующую форму:

```
function фабрика(адрес_сервиса, аргументы) {

// Функция фабрика возвращает функцию requestor, способную выполнять
// элементарную операцию.

    return function requestor(callback, value) {

// Работа начинается в блоке 'try' на тот случай, если функция,
// отправляющая сообщение рабочему сервису, даст сбой.

        try {

// Когда вызывается функция requestor, она отправляет сервису сообщение
// с предписанием о начале работы. Когда результат будет получен, его
// доставит функция callback. В данном примере предполагается, что система
// сообщений отправит свой результат, используя соглашение
// '(результат, исключение)'.

            return отправить_сообщение(
                callback,
                адрес_сервиса,
                старт,
                value,
                аргументы
            );
        } catch (exception) {

// Если произошел перехват исключения, мы сигнализируем о сбое.

            return callback(undefined, exception);
        }

// Нам разрешено возвращать функцию отмены cancel, которая будет пытаться
// отменить запрос, когда результат больше не нужен.

        return function cancel(reason) {
            return отправить_сообщение(
                undefined,
                адрес_сервиса,
                стоп,
                аргументы
            );
        };
    };
}
```

Parseq

Для управления потоком между функциями-запросчиками я разработал библиотеку Parseq. Фабрики Parseq пакуют массивы ваших функций-запросчиков с режимом потока управления: параллельным (parallel), последовательным (sequence),

альтернативным (fallback) или состязательным (race). Каждая фабрика возвращает новый запросчик. В результате запросчики легко поддаются компоновке.

Библиотека Parseq может решать проблемы истечения лимита времени. Допустим, некая работа должна быть завершена за 100 мс. Если срок истечет, ее нужно будет считать неудавшейся и перейти к альтернативному ответу. Это и есть то, что очень трудно сделать, используя любую из рассмотренных ранее трех ошибок. Применяя Parseq, вам останется всего лишь указать лимит времени в миллисекундах.

```
parseq.sequence(
    массив_запросчиков,
    миллисекунды
)
```

Фабрика `sequence` принимает массив функций-запросчиков и необязательный лимит времени, а возвращает запросчик, вызывающий каждый запросчик, имеющийся в массиве. Значение, переданное новому запросчику, передается нулевому запросчику в массиве. Получаемое значение передается первому запросчику. Таким образом, значения передаются от запросчика к запросчику. Результат последнего запросчика и есть результат последовательности.

```
parseq.parallel(
    обязательный_массив,
    необязательный_массив,
    миллисекунды,
    параметр_времени,
    регулятор
)
```

Фабрика `parallel` запускает все свои запросчики одновременно. Она не добавляет параллелизм в JavaScript, зато разрешает JavaScript использовать природный параллелизм Вселенной. Тем самым допускается распределение вычислительной нагрузки среди многих серверов. Это может дать огромное преимущество в производительности. Если мы обрабатываем все запросчики параллельно, то затрачиваемое время — это время выполнения работы самым медленным запросчиком, а не общее время работы всех запросчиков. Это весьма важная оптимизация.

Фабрику `parallel` следует задействовать только в том случае, если у запросчиков нет взаимозависимости. Всем запросчикам дается одно и то же *значение*. Результатом становится массив, содержащий результаты всех запросчиков. Этим созданный запросчик напоминает метод массива `map`.

Фабрика `parallel` способна принимать два массива запросчиков. В первом из них содержатся обязательные запросчики. Для успешного завершения параллельной операции все обязательные запросчики должны успешно завершить свою работу.

Во втором массиве содержатся необязательные запросчики. Они могут давать сбой, не вызывая сбоя параллельной операции.

Может предоставляться лимит времени в миллисекундах. Им определяется время, за которое запросчики должны завершить свою работу. Параметр *параметр_времени* определяет, как именно лимит времени влияет на необязательные запросчики.

параметр_времени	**Эффект**
Undefined	Необязательные запросчики должны завершить свою работу до того, как закончится работа обязательных запросчиков. Обязательные запросчики должны завершить свою работу до истечения лимита времени, если таковой имеется
True	Обязательные и необязательные запросчики должны завершить свою работу до истечения лимита времени
False	У обязательных запросчиков нет лимита времени. Необязательные запросчики должны завершить свою работу до окончания работы обязательных запросчиков или истечения лимита времени, в зависимости от того, что наступит позже

По умолчанию все запросчики стартуют одновременно. К сожалению, это может пагубно отразиться на некачественно спроектированных системах. Поэтому здесь присутствует необязательный *регулятор*, ограничивающий количество одновременно активируемых запросчиков.

```
parseq.fallback(
    массив_запросчиков,
    миллисекунды
)
```

Фабрика `fallback` похожа на фабрику `sequence`, но ее работа считается успешно завершенной, когда успешный результат получен любым запросчиком. Если один запросчик дает сбой, пробуют следующий. Сбой `fallback` происходит при сбое каждого запросчика.

```
parseq.race(
    массив_запросчиков,
    миллисекунды
    регулятор
)
```

Фабрика `race` похожа на фабрику `parallel`, но она объявляет об успешном завершении работы при первом же удачном результате работы запросчика. Если успешно завершилась работа одного запросчика, то работа `race` считается успешной.

Пример:

```
let getWeather = parseq.fallback([
    fetch("weather", localCache),
    fetch("weather", localDB),
    fetch("weather", remoteDB)
]);

let getAds = parseq.race([
    getAd(adnet.klikHaus),
    getAd(adnet.inUFace),
    getAd(adnet.trackPipe)
]);

let getNav = parseq.sequence([
    getUserRecord,
    getPreference,
    getCustomNav
]);

let getStuff = parseq.parallel(
    [getNav, getAds, getMessageOfTheDay],
    [getWeather, getHoroscope, getGossip],
    500,
    true
);
```

Исключения

Исключения слишком слабы, чтобы справляться со сбоями, происходящими на протяжении многих ходов. Исключение может опустошить стек, но от хода к ходу ничто в стеке не выживает. У исключений нет возможности сообщать будущему ходу о каком-то сбое в ранее состоявшемся ходе, и исключения не способны совершить путешествие во времени к источнику сбойного запроса.

Фабрике разрешено выдавать исключение, потому что ссылка на вызвавший ее код все еще находится в стеке. Запросчикам вообще не разрешается выдавать исключения, поскольку в стеке нет ничего, чтобы их перехватывать. Запросчики не должны допускать сброса случайных исключений. Все исключения должны быть перехвачены и переданы в функцию обратного вызова в качестве причины.

Реализация Parseq

Посмотрим на реализацию Parseq. Она не слишком большая. В ее состав входят четыре открытые и четыре закрытые функции.

Первая закрытая функция называется `make_reason`. Она создает объект ошибки:

```
function make_reason(factory_name, excuse, evidence) {
```

Создание объекта причины `reason` — эти объекты используются для исключений и отмен и создаются из объектов `Error`:

```
        const reason = new Error("parseq." + factory_name + (
            excuse === undefined
            ? ""
            : ": " + excuse
        ));
        reason.evidence = evidence;
        return reason;
    }
```

Функция обратного вызова с двумя параметрами:

```
    function check_callback(callback, factory_name) {
        if (typeof callback !== "function" || callback.length !== 2) {
            throw make_reason(factory_name, "Not a callback.", callback);
        }
    }
```

Нужно убедиться, что все элементы в массиве — это функции-запросчики:

```
    function check_requestor_array(requestor_array, factory_name) {
```

Массив `requestor` содержит только запросчики. Запросчик — это функция, принимающая один или два аргумента: функцию обратного вызова `callback` и необязательное исходное значение `initial_value`.

```
        if (
            !Array.isArray(requestor_array)
            || requestor_array.length < 1
            || requestor_array.some(function (requestor) {
                return (
                    typeof requestor !== "function"
                    || requestor.length < 1
                    || requestor.length > 2
                );
            })
        ) {
            throw make_reason(
                factory_name,
                "Bad requestors array.",
                requestor_array
            );
        }
    }
```

Функция `run` является сердцевиной Parseq. Она запускает запросчики и управляет соблюдением лимита времени, отменой и регулятором.

```
    function run(
        factory_name,
        requestor_array,
```

```
        initial_value,
        action,
        timeout,
        time_limit,
        throttle = 0
    ) {
```

Функция `run` выполняет работу, общую для всех фабрик Parseq. Она принимает имя фабрики, массив запросчиков, исходное значение, то или иное действие обратного вызова, функцию обратного вызова по истечении лимита времени, лимит времени в миллисекундах и регулятор.

Если все получается, вызываются все функции-запросчики, содержащиеся в массиве. Каждая из них может возвратить функцию отмены, хранящуюся в `cancel_array`:

```
    let cancel_array = new Array(requestor_array.length);
    let next_number = 0;
    let timer_id;
```

Нам нужны функции `cancel` и `start_requestor`:

```
    function cancel(reason = make_reason(factory_name, "Cancel.")) {
```

Остановка всех незавершенных дел — эта функция может быть вызвана, когда запросчик дает сбой. Ее также вызывают, когда запросчик завершает свою работу успешно, например, когда в состязательном режиме останавливается работа всех, кто проиграл, или в параллельном режиме останавливается работа тех необязательных запросчиков, которые еще не завершили работу.

Если таймер запущен, его следует остановить.

```
        if (timer_id !== undefined) {
            clearTimeout(timer_id);
            timer_id = undefined;
        }
```

Если происходит что-то еще, это действие нужно отменить.

```
        if (cancel_array !== undefined) {
            cancel_array.forEach(function (cancel) {
                try {
                    if (typeof cancel === "function") {
                        return cancel(reason);
                    }
                } catch (ignore) {}
            });
            cancel_array = undefined;
        }
    }

    function start_requestor(value) {
```

Теперь функция `start_requestor` совсем не рекурсивная. Она не вызывает сама себя напрямую, но возвращает функцию, которая может вызвать `start_requestor`.

Запуск выполнения запросчика, если таковой его еще ожидает:

```
if (
    cancel_array !== undefined
    && next_number < requestor_array.length
) {
```

У каждого запросчика есть номер:

```
    let number = next_number;
    next_number += 1;
```

Вызов следующего запросчика, передача функции обратного вызова, сохранение функции отмены, которую может возвратить запросчик:

```
    const requestor = requestor_array[number];
    try {
        cancel_array[number] = requestor(
            function start_requestor_callback(value, reason) {
```

Функция обратного вызова вызывается запросчиком, когда он завершает свою работу. Если работы больше нет, вызов игнорируется. Например, это может быть результат, отправленный назад по истечении лимита времени. Данная функция обратного вызова может быть вызвана только один раз:

```
                if (
                    cancel_array !== undefined
                    && number !== undefined
                ) {
```

Больше нам не нужна отмена, связанная с этим запросчиком:

```
                    cancel_array[number] = undefined;
```

Вызов функции `action` для информирования запросчика о случившемся:

```
                    action(value, reason, number);
```

Очистка номера, чтобы этот обратный вызов не мог быть использован еще раз:

```
                    number = undefined;
```

Если есть еще какие-нибудь запросчики, ожидающие запуска, то происходит запуск следующего запросчика. Если следующий запросчик встроен в последовательность, осуществляется передача ему самого последнего значения. Другие запросчики получают исходное значение `initial_value`:

```
                    return start_requestor(
                        factory_name === "sequence"
                        ? value
                        : initial_value
                    );
                }
            },
            value
        );
```

Запросчики должны сообщать о своем сбое через обратный вызов. Им не разрешено выдавать исключения. Если будет перехвачено исключение, это будет считаться сбоем:

```
            } catch (exception) {
                action(undefined, exception, number);
                number = undefined;
                start_requestor(value);
            }
        }
    }
```

Теперь, располагая функциями `cancel` и `start_requestor`, можно приступать к работе.

Если запрошено отслеживание превышения лимита времени, запускаем таймер:

```
    if (time_limit !== undefined) {
        if (typeof time_limit === "number" && time_limit >= 0) {
            if (time_limit > 0) {
                timer_id = setTimeout(timeout, time_limit);
            }
        } else {
            throw make_reason(factory_name, "Bad time limit.", time_limit);
        }
    }
```

Если есть намерение запустить параллельный или состязательный режим, нужно запустить все запросчики одновременно. Но если установлен регулятор, запускается разрешенное им количество запросчиков, а когда завершит работу каждый запросчик, запускается следующий.

Фабрики `sequence` и `fallback` устанавливают регулятор `throttle` равным 1, поскольку они обрабатывают запросчики поочередно и всегда запускают другой запросчик, когда завершает работу его предшественник:

```
    if (!Number.isSafeInteger(throttle) || throttle < 0) {
        throw make_reason(factory_name, "Bad throttle.", throttle);
    }
    let repeat = Math.min(throttle || Infinity, requestor_array.length);
    while (repeat > 0) {
        setTimeout(start_requestor, 0, initial_value);
        repeat -= 1;
    }
```

Возвращаем `cancel`, что позволяет запросчику завершить работу:

```
        return cancel;
    }
```

Теперь рассмотрим четыре открытые функции.

Функция `parallel` самая сложная из них из-за массива необязательных запросчиков:

```
    function parallel(
        required_array,
```

```
    optional_array,
    time_limit,
    time_option,
    throttle,
    factory_name = "parallel"
) {
```

Фабрика `parallel` самая сложная из этих фабрик. Она может принимать второй массив запросчиков, получающих более щадящую политику сбоев. И возвращает запросчик, создающий массив значений:

```
let number_of_required;
let requestor_array;
```

Возможны четыре варианта, поскольку как массив `required_array`, так и массив `optional_array` может быть пустым:

```
if (required_array === undefined || required_array.length === 0) {
    number_of_required = 0;
    if (optional_array === undefined || optional_array.length === 0) {
```

Если оба массива пустые, то это, вероятно, ошибка:

```
        throw make_reason(
            factory_name,
            "Missing requestor array.",
            required_array
        );
    }
```

Если присутствует только `optional_array`, то он становится `requestor_array`:

```
        requestor_array = optional_array;
        time_option = true;
    } else {
```

Если имеется только `required_array`, то этот массив становится `requestor_array`:

```
    number_of_required = required_array.length;
    if (optional_array === undefined || optional_array.length === 0) {
        requestor_array = required_array;
        time_option = undefined;
```

Если предоставлены оба массива, мы их объединяем:

```
        } else {
                requestor_array = required_array.concat(optional_array);
                if (time_option !== undefined && typeof time_option !== "boolean") {
                    throw make_reason(
                        factory_name,
                        "Bad time_option.",
                        time_option
                    );
                }
            }
        }
```

Проверяем массив и возвращаем запросчик:

```
check_requestor_array(requestor_array, factory_name);
return function parallel_requestor(callback, initial_value) {
    check_callback(callback, factory_name);
    let number_of_pending = requestor_array.length;
    let number_of_pending_required = number_of_required;
    let results = [];
```

`run` его запускает:

```
    let cancel = run(
        factory_name,
        requestor_array,
        initial_value,
    function parallel_action(value, reason, number) {
```

Функция действия получает результат работы каждого запросчика, имеющегося в массиве. Функции `parallel` требуется возвращение массива всех видимых ею значений:

```
        results[number] = value;
        number_of_pending -= 1;
```

Если запросчик был одним из обязательных, нужно убедиться, что он завершил свою работу успешно. Если же выдал сбой, параллельная операция считается сбойной. Если выдал сбой необязательный запросчик, можно продолжить работу:

```
        if (number < number_of_required) {
            number_of_pending_required -= 1;
            if (value === undefined) {
                cancel(reason);
                callback(undefined, reason);
                callback = undefined;
                return;
            }
        }
```

Если все было обработано или все обязательные запросчики завершили работу успешно и у нас не было `time_option`, значит, мы справились с задачей:

```
        if (
            number_of_pending < 1
            || (
                time_option === undefined
                && number_of_pending_required < 1
            )
        ) {
            cancel(make_reason(factory_name, "Optional."));
            callback(
                factory_name === "sequence"
                ? results.pop()
                : results
            );
```

```
                callback = undefined;
            }
        },
        function parallel_timeout() {
```

Когда срабатывает таймер, работа прекращается, если только она не велась под параметром времени со значением `false`. Это значение не накладывает лимит времени на работу обязательных запросчиков, позволяя необязательным работать до тех пор, пока не завершится работа обязательных запросчиков или не истечет лимит времени в зависимости от того, что наступит позже.

```
        const reason = make_reason(
            factory_name,
            "Timeout.",
            time_limit
        );
        if (time_option === false) {
            time_option = undefined;
            if (number_of_pending_required < 1) {
                cancel(reason);
                callback(results);
            }
        } else {
```

Время истекло. Если все запросчики завершили свою работу успешно, то параллельная операция будет считаться успешно завершенной.

```
                cancel(reason);
                if (number_of_pending_required < 1) {
                    callback(results);
                } else {
                    callback(undefined, reason);
                }
                callback = undefined;
            }
        },
        time_limit,
        throttle
    );
    return cancel;
};
}
```

Функция `race` намного проще, чем `parallel`, поскольку она не нуждается в аккумулировании всех результатов. Ей нужен всего лишь один результат.

```
function race(requestor_array, time_limit, throttle) {
```

Фабрика `race` возвращает запросчик, запускающий одновременно все запросчики, имеющиеся в массиве `requestor_array`. Первый же успешно завершивший работу запросчик считается победителем.

```
    const factory_name = (
        throttle === 1
```

```
            ? "fallback"
            : "race"
        );

        check_requestor_array(requestor_array, factory_name);
        return function race_requestor(callback, initial_value) {
            check_callback(callback, factory_name);
            let number_of_pending = requestor_array.length;
            let cancel = run(
                factory_name,
                requestor_array,
                initial_value,
                function race_action(value, reason, number) {
                    number_of_pending -= 1;
```

У нас есть победитель. Отменяем работу проигравших и передаем значение обратному вызову.

```
                    if (value !== undefined) {
                        cancel(make_reason(factory_name, "Loser.", number));
                        callback(value);
                        callback = undefined;
                    }
```

Победителя нет. Сигнализируем о сбое.

```
                    if (number_of_pending < 1) {
                        cancel(reason);
                        callback(undefined, reason);
                        callback = undefined;
                    }
                },
                function race_timeout() {
                    let reason = make_reason(
                        factory_name,
                        "Timeout.",
                        time_limit
                    );
                    cancel(reason);
                    callback(undefined, reason);
                    callback = undefined;
                },
                time_limit,
                throttle
            );
            return cancel;
        };
    }
```

Альтернативный режим (fallback) — это всего лишь регулируемый состязательный режим (race):

```
    function fallback(requestor_array, time_limit) {
```

Фабрика `fallback` возвращает запросчик, поочередно пробующий запускать каждый запросчик, имеющийся в массиве `requestor_array`, пока не найдет запросчик, успешно завершивший свою работу:

```
        return race(requestor_array, time_limit, 1);
    }
```

Последовательный режим (sequence) представляет собой все тот же регулируемый параллельный режим (parallel) с распространяемыми значениями:

```
    function sequence(requestor_array, time_limit) {
```

В последовательном режиме запускается по порядку каждый запросчик, который передает результаты следующему запросчику, если его работа завершается успешно. Последовательный режим — это регулируемый параллельный режим.

```
        return parallel(
            requestor_array,
            undefined,
            time_limit,
            undefined,
            1,
            "sequence"
        );
    }
```

Доступ к библиотеке Parseq в вашем модуле можно получить, импортировав ее.

```
    import parseq from "./parseq.js";
```

Тонкости английского языка

Эдсгер Дейкстра (Edsger Dijkstra) был одним из первых, кто еще в 1962 году распознал проблему с потоками. Он разработал первый механизм взаимного исключения — *семафоры*. Семафор был реализован в виде двух функций — `P` и `V`. Функция `P` будет пытаться заблокировать критическую секцию, блокируя затем код, пытающийся установить блокировку, если эта секция уже заблокирована. Функция `V` будет снимать блокировку, позволяя ожидающему потоку заблокировать критическую секцию и запуститься.

Пер Бринч Хансен (Per Brinch Hansen) и Чарльз Энтони Ричард Хоар (C. A. R. Hoare) интегрировали семафоры в классы, чтобы создать *мониторы* — синтаксическую форму, которая, как они надеялись, будет удобнее и устойчивее к ошибкам.

В языке JavaScript есть нечто похожее под названием *synchronized* («синхронизировано»). Ключевое слово `synchronized` применяется для декорирования кода, нуждающегося во вставке семафоров. К сожалению, выбранное слово не имеет смысла.

«Синхронный» (synchronous) означает «существующий в то же время или использующий те же часы». Музыканты в оркестре синхронизированы, потому что все

играют в ритме, задаваемом дирижером. Когда был создан язык Java, конструкторы искали слово, которое было бы связано с временем. К сожалению, похоже, при этом они были недостаточно усердны.

Когда компания Microsoft изыскивала возможность добавить совершенно неприемлемую событийную поддержку в C#, ее специалисты обратились к Java: взяли неправильное слово и превратили его в еще более неправильное, снабдив префиксом `a-` для присвоения противоположного значения. Получившееся слово `async` не имеет никакого смысла.

У большинства программистов недостаточно образования или опыта работы в сфере параллельного программирования. Одновременное управление несколькими потоками активности им незнакомо и непривычно. Внедрение в их сознание непонятного языка, который в корне неверен, не облегчает решения задачи. Неосведомленность приводит к еще большему невежеству.

Глава 21

Как работает Date

●○●○●

С чем вы столкнетесь, открывая дверь, будет ли это таинственное свидание мечтой или подделкой?

Милтон Брэдли (Milton Bradley)

Наш календарь отображает движение Солнца и Луны на постоянный ход времени, но он был разработан задолго до того, как мы поняли, как работает Солнечная система. Он претерпел множество коррекций, но никогда не подвергался полному обновлению. Наш уродливый календарь в ходе завоеваний и торговли навязывали все большему и большему количеству сообществ. В итоге его всучили всему мировому сообществу без предварительной корректировки или замены более подходящей конструкцией. Он работает, и весь мир использует его, но он может быть лучше.

Современный календарь основан на римском календаре, в котором сначала было десять месяцев и бонус зимнего сезона из оставшихся дней года. Первым месяцем был март (назван по имени Марса, бога войны). Десятым был декабрь, что и означает *десятый месяц.* Зимний сезон заменили двумя новыми месяцами — январем и февралем. Иногда в политических целях у февраля крали дни и использовали их для продления других месяцев. Каждые четыре года в календаре дрейфовал примерно один день, поскольку число дней в году не является целым числом. Римляне не виноваты в том, что число дней в тропическом году не целое число. Они попытались рационально перестроить свой календарь, объявив январь первым месяцем. При этом декабрь, который все еще означает *десятый месяц,* стал двенадцатым.

Юлий Цезарь ввел стандарт добавления високосного дня к февралю каждые четыре года. Это уменьшило сезонный дрейф, но полностью его не устранило. По указанию Папы Римского Григория XIII привели в соответствие со стандартом лучший алгоритм високосного года, но более простой и точный алгоритм так и не был принят. Григорианский алгоритм таков:

> добавлять високосный день в те годы, которые делятся на 4, за исключением того случая, когда они также делятся на 100, но все равно добавлять високосный день, если год делится на 400.

В результате средняя продолжительность года получилась 365,2425 дня, что довольно близко к тропическому году, продолжающемуся около 365,242 188 792 дня.

Есть и более удачный алгоритм:

> добавлять високосный день в те годы, которые делятся на 4, за исключением того случая, когда они также делятся на 128.

Он дает год, состоящий из 365,242 187 5 дня, что намного ближе к продолжительности тропического года. Мне, как программисту, представляется просто чудом включение в алгоритм двух степеней числа 2, хотя я был бы гораздо больше впечатлен точным соответствием продолжительности реального года. Если бы число дней в году было целым числом, то я мог бы подумать, что в теории разумного начала, возможно, кроется некая истина.

Следующим годом несоответствия алгоритмов 4|100|400 и 4|128 станет 2048-й. Нужно ввести алгоритм 4|128 в качестве стандарта до его наступления.

Високосный день добавлялся в конце года. К сожалению, когда январь стал первым месяцем, високосный день не был перенесен с февраля на декабрь (по-прежнему означающий *десятый месяц*).

У минут и секунд нулевые исходные значения, и это хорошо, но и те и другие подсчитываются по модулю 60, что уже не так хорошо. Часы имеют нулевое исходное значение, но в часах в 12-часовом формате 0 заменяется на 12, что абсолютно неестественно.

(Будь моя воля, вместо этого я использовал бы десятичасовой день, где час равен 100 минутам, а минута — 100 секундам. Моя секунда была бы немного короче, составив 86,4 % от нынешней секунды. Темпом секунд было бы приятное адажио, что стало бы легким ускорением темпа жизни по сравнению с нынешним ларго.)

Для месяцев и дней в качестве исходного значения используется единица, поскольку этот стандарт был введен до открытия нуля. Месяцы подсчитываются по модулю 12. Римляне пытались сделать месяцы по модулю 10, но не смогли ввести это в обиход. Дни в месяце подсчитываются по модулю 30 или 31 либо 28 или 29, в зависимости от месяца и года. Отсчет лет начинался с единицы, но ныне существующая нумерация была принята спустя множество веков после наступления нашей эры, поэтому данное неудобство можно спокойно проигнорировать.

Функция Date

Странности в стандартах учета времени становятся помехой для программ, в которых должно обрабатываться время. Класс `Date` в Java обеспечивает поддержку дат. Все должно было быть просто, но получилось очень сложно, продемонстрировав один из худших шаблонов проектирования классического программирования. В JavaScript могло бы быть сделано что-то более удачное, но вместо этого просто скопирован весьма неудачный пример из языка Java.

Нынешние объекты JavaScript `Date` содержат множество методов. Большинство из них — это просто получатели и установщики (get- и set-методы):

```
getDate
getDay
getFullYear
getHours
getMilliseconds
getMinutes
getMonth
getSeconds
getTime
getTimezoneOffset
getUTCDate
getUTCDay
getUTCFullYear
getUTCHours
getUTCMilliseconds
getUTCMinutes
getUTCMonth
getUTCSeconds
getYear
setDate
setFullYear
setHours
setMilliseconds
setMinutes
setMonth
setSeconds
setTime
setUTCDate
setUTCFullYear
setUTCHours
setUTCMilliseconds
setUTCMinutes
setUTCMonth
setUTCSeconds
setYear
toDateString
toISOString
toJSON
toLocaleDateString
toLocaleString
toLocaleTimeString
toString
toTimeString
toUTCString
```

Есть метод `getDate`, который возвращает день месяца из объекта `Date`. Сразу бросается в глаза то, что слово `Date` имеет в одном и том же методе два совершенно разных значения. Усугубляет путаницу метод `getDay`, который возвращает день недели.

Метод `getMonth` вносит правку в месяц, делая исходным нулевое значение, поскольку программистам нравится работать со всем, что начинается с нуля. Получается, что `getMonth` возвращает числа от 0 до 11. Метод `getDate` не вносит правку в день, поэтому он возвращает числа от 1 до 31. Это разночтение порождает массу ошибок.

Методы `getYear` и `setYear` после 1999 года работают неправильно и больше не должны использоваться. Язык Java был выпущен в 1995 году и содержал методы дат, которые должны были дать сбой в 2000 году. Неужели разработчики языка ничего не слышали о проблеме 2000 года? Или они сомневались, что Java переползет на рынке через этот рубеж? Возможно, это так и останется тайной. Но нам уже известно, что язык Java необъяснимым образом выжил и что в языке JavaScript была допущена точно такая же ошибка. Вместо этих методов всегда нужно использовать методы `getFullYear` и `setFullYear`.

`Date` демонстрирует весьма неудачные приемы классического программирования. В объекте что-то должно инкапсулироваться. Формами взаимодействия с объектами должны быть транзакции и другие высокоуровневые действия. `Date` дает весьма низкоуровневое представление с get- и set-методами для каждого отдельно взятого компонента времени. При таком подходе объекты используются крайне нерационально.

ISO 8601

Новый конструктор `Date` может принимать строку, представляющую дату, и создавать объект для этой даты. К сожалению, в стандарте ECMAScript правила синтаксического анализа и распознавания не определены, поэтому в соответствии со стандартом не гарантируется, что они будут работать.

За исключением дат ISO.

ISO 8601 — это международный стандарт представления даты и времени. Для правильного разбора дат ISO, например 2018-11-06, необходим JavaScript. Гораздо логичнее ставить наиболее значимые объекты на первое, а наименее значимые — на последнее место, в этом больше смысла, чем в стандарте США: 11/06/2018. Например, строки даты ISO хорошо поддаются сортировке.

Другой подход

Разумеется, уже слишком поздно что-либо исправлять. JavaScript не должен был копировать Java. Подобные действия всегда ошибочны. Многие из неправильностей JavaScript пришли из Java. Если создавать правильный вариант, то я предлагаю следующее.

Должно существовать три представления даты:

- количество миллисекунд, прошедших с начала эпохи;
- объект даты, в котором могут содержаться следующие свойства:
  - `year`;
  - `month`;
  - `day`;
  - `hour`;
  - `minute`;
  - `second`;
  - `zone`;
  - `week`;
  - `weekday`;
- строка в каком-либо стандартном формате.

Нам не нужен классический объект `Date` с методами. Нужны лишь простые функции для преобразований между тремя представлениями.

- `Date.now()` возвращает текущее время в виде числа. Эта функция уже существует. JavaScript открывает к ней доступ для всего кода, но я полагаю, что она должна быть привилегией, предоставляемой только доверенному

коду. Часть нечистого вредоносного кода может воспользоваться `Date.now()` или `Math.random()` для изменения ее поведения, позволяя избежать ее обнаружения.

- `Date.object()` принимает либо число, либо строку и возвращает объект, содержащий информацию, доступную для извлечения. Если аргумент является строкой, могут существовать дополнительные необязательные аргументы, определяющие часовой пояс и стандарт форматирования, управляющий синтаксическим анализом.
- `Date.string ()` будет принимать число или объект данных, а также необязательный стандарт форматирования и часовой пояс и возвращать удобное для прочтения строковое представление времени.
- `Date.number()` будет принимать объект данных или строку, а также необязательный стандарт форматирования и часовой пояс и возвращать числовое представление времени.

Этот подход был бы намного проще и легче в использовании, более устойчив к ошибкам и, что уже не имеет никакого значения, готов к 2000 году. Вместо массы нечистых методов низкого уровня предлагаются всего лишь одна нечистая и три чистые функции. Сделать такое в Java было невозможно, поскольку в этом языке отсутствовали замечательные объектные литералы JavaScript. Я не знаю, почему так не сделано в JavaScript. Надеюсь, все это появится в следующем языке.

Эпоха, которая применяется в JavaScript, — это эпоха Unix с точкой начала отсчета 1970-01-01. Есть 32-разрядные системы Unix, которые выйдут из строя в 2038 году, когда все биты будут использованы и часы переполнены. Оказывается, 32 разрядов недостаточно для запуска системных часов с секундным разрешением.

Я бы предпочел взять в качестве точки начала отсчета эпохи дату 0000-01-01. Числа JavaScript не дадут сбоя в суммировании миллисекунд вплоть до 285 426 года. К тому времени либо будет разработан более удачный календарь, либо все мы вымрем. Хорошего вам дня.

Глава 22

Как работает JSON

●○●●○

> XML должен стать основой для формата информационного наполнения не из-за того, что он замечателен с технической точки зрения, а из-за достигнутой им необычайной широты внедрения. Именно на этом обстоятельстве теряется суть аргументации приверженцев JSON (или YAML), которые гордо указывают на технические преимущества своего формата: создать более удачный формат данных, чем XML, может самый непроходимый тупица.
>
> *Джеймс Кларк (James Clark)*

Здесь впервые в печатной форме я расскажу правдивую историю о происхождении самого любимого в мире формата обмена данными.

Изобретение

Формат JSON был придуман в 2001 году в сарае за домом Чипа Морнингстара (Chip Morningstar). Мы с Чипом основали компанию по разработке платформы для одностраничных веб-приложений. Замысел заключался в том, что с хорошей библиотекой JavaScript (которую написал я) и эффективным и масштабируемым сервером сеансов (который создал Чип) можно было создавать веб-страницы, которые работали бы так же хорошо, как и устанавливаемые приложения. Или даже еще лучше, потому что наша платформа, кроме прочего, поддерживает многопользовательскую совместную работу, а это то, чего еще не достигали в сети.

Я все еще нахожусь под впечатлением от созданного Чипом сервера сеансов. За прошедшие годы он подвергался многочисленным переделкам. Самая свежая версия называется Elko (elkoserver.org).

Единственный недостаток Elko — то, что он написан на Java. Я мечтаю, чтобы вскоре кто-нибудь заплатил Чипу, чтобы он переделал его еще раз, теперь уже

на JavaScript. Это стало бы существенным продвижением вперед по сравнению с тем, где мы сейчас находимся с Node.js. Мы получили бы повышенный уровень безопасности, лучшую масштабируемость и поддержку более широкого спектра приложений.

Но вернемся в сарай. Нам нужен был способ передачи информации между браузером и сервером. В то время индустрия программного обеспечения испытывала полную приверженность формату XML. Такие гиганты, как Microsoft, IBM, Oracle, Sun, HP и другие, решили, что следующее поколение программного обеспечения будет построено на XML, и с ними согласились как их последователи, так и клиенты.

Нам же нужен был обмен данными между программами, написанными на двух разных языках. Мы посмотрели на XML и решили, что он плохо подходит для нашей задачи. Модель работы с XML заключалась в том, что сначала отправлялся запрос на сервер, который отвечал XML-документом. Затем, чтобы получить данные, делались дополнительные запросы к документу XML. Почему сервер не может просто отправить данные в форме, которую программы способны бы сразу использовать? Наши данные просто не были похожи на документы.

В то время доступно было множество улучшенных XML-вариантов и замен этого формата, но ни один из них не получил поддержки. Мы думали о том, чтобы пойти собственным путем. И тут на меня нашло прозрение. Мы могли бы воспользоваться объектными литералами JavaScript. Их встроенность в JavaScript создавала реальные удобства на стороне этого языка. А в отношении поднабора, в котором мы нуждались, нетрудно было выполнить синтаксический анализ на стороне Java. Для этого понадобилось бы куда меньше усилий, чем при работе с XML.

Нам хотелось, чтобы наша платформа работала как с Microsoft Internet Explorer, так и с Netscape Navigator. Это было совсем не просто, поскольку обе компании, производящие эти браузеры, изобретали свои собственные функции. Там было мало общего. Microsoft добавила функцию `XMLHttpRequest`, которую можно было использовать для связи с сервером. К сожалению, в компании Netscape не было ничего подобного, поэтому задействовать ее браузер было невозможно.

У обоих браузеров был JavaScript (ES3), и оба они имели наборы фреймов. И мы применяли их вместе, чтобы создать канал обмена данными. Первое сообщение JSON, отправленное в браузер, имело следующий вид:

```
<html><head><script>
document.domain = "fudco.com";
parent.session.receive({to: "session", do: "test", text: "Hello world"});
</script></head></html>
```

Веб-страница содержала скрытый фрейм, по которому можно было перемещаться. Выполнение POST-запроса на скрытом фрейме приводило к отправке сообщения на сервер. Сервер отвечал, передавая документ, содержащий сценарий, который

вызвал метод `receive` объекта `session` в главном фрейме. Чтобы включить межфреймовый обмен данными, нам пришлось изменить `document.domain`. Синтаксический анализ сообщения проводился компилятором JavaScript.

Как бы ни хотелось мне сказать, что первое сообщение было доставлено успешно, но ничего из этого не вышло. Сбой произошел из-за ужасной политики зарезервированных слов ES3. В то время были зарезервированы следующие слова:

```
abstract boolean break byte case catch char class const continue debugger
default delete do double else enum export extends false final finally float
for function goto if implements import in instanceof int interface long native
new null package private protected public return short static super switch
synchronized this throw throws transient true try typeof var void volatile
while with
```

В соответствии с политикой зарезервированных слов ES3 ни одно из этих слов нельзя использовать в качестве имен переменных, или имен параметров, или имен свойств в точечной позиции, или *имен свойств в литералах объекта*. В сообщении было свойство `do`, поэтому сценарий содержал синтаксическую ошибку и не запускался.

С радостью сообщаю, что в ES5 ситуация была исправлена. Список зарезервированных слов был сокращен, а ограничения на имена свойств были сняты, но в 2001 году нам пришлось поставить кавычки вокруг слова *do*, чтобы оно заработало. Чип поместил зарезервированный список слов в свой кодировщик, и больше проблем у нас с ним не было.

Мы выяснили, что строка, содержащая последовательность символов `</`, может вызвать сбой. Дело в том, что браузер предполагает, что это закрывающий тег `script`, поэтому JavaScript не получит все информационное наполнение и возникнет синтаксическая ошибка. Нам нужно было обойти сложившуюся ситуацию и избежать применения слеша. Браузер вполне удовлетворяла комбинация символов `<\ /` (меньше, обратный слеш, слеш).

Мы назвали этот формат JSML (что рифмуется со словом dismal, то есть «мрачный»). Позже мы узнали, что такая же аббревиатура использовалась для чего-то в Java, поэтому быстро подобрали замену: JSON.

JSON прекрасно справлялся с обменом данными между JavaScript и Java. Мы также воспользовались им для межсерверной связи. Мы создали первую базу данных JSON.

Стандартизация

Нам было трудно добиться того, чтобы концепция JSON стала популярной у наших клиентов. Они говорили, что не могут воспользоваться этим форматом, потому что являются приверженцами XML. Они сказали, что новым форматом нельзя пользоваться, поскольку он не стандарт. Я сказал им, что это стандарт — подмно-

жество ECMA-262. Они не соглашались с этим. И тогда я решил взять на себя роль комитета по стандартизации.

Я приобрел домен json.org и приступил к работе по формализации JSON. До этого времени JSON существовал как джентльменское соглашение между Чипом, мной и JavaScript. При разработке стандарта мне пришлось принять ряд решений. Я руководствовался принципами сохранения формата в текстовом виде, в минимальных объемах и в подмножестве JavaScript.

Принцип минимализма был критически важен. Стандарты должны быть простыми и завершенными. Чем меньше объем материала, с которым нужно будет согласиться, тем легче мы сможем наладить взаимодействие. Принцип принадлежности к подмножеству JavaScript весьма эффективно препятствовал тому, чтобы я добавлял множество привлекательных, но ненужных функций.

JavaScript позволяет заключать строки как в одинарные (') так и в двойные (") кавычки. Принцип минимализма гласит, что нам не нужны обе формы.

Я решил сохранить требование заключать в кавычки имена свойств. Мне не хотелось помещать в стандарт список зарезервированных слов ES3, потому что это выглядело бы действительно глупо. Неизбежен вопрос *«Почему?»*, а ответом будет: *«Потому что это JavaScript»*. Мы старались убедить людей разрабатывать свои приложения на JavaScript, поэтому я не хотел, чтобы стандарт JSON высветил неудачные компоненты JavaScript. Если заключать в кавычки все имена, то проблемы не возникнет. Тем самым упрощалась и спецификация, поскольку из-за интернационализации нелегко стало отвечать на вопрос *«Что такое буква?»*. А так удается избежать еще и всех этих проблем. Именем свойства запросто может стать любая строка. Благодаря этому JSON стал больше похож на Python, что, как я думал, поможет принять данный формат.

Я добавил комментарии, потому что они были разрешены принципом принадлежности к подмножеству JavaScript, и подумал, что они будут весьма кстати. Затем я увидел, что некоторые из тех, кто рано стал приверженцем формата, помещали в комментарии инструкции по синтаксическому разбору. Это нарушило бы совместимость, а в ней и был весь смысл.

Поскольку больше всего кодеков JSON было разработано для других языков, я заметил, что примерно половина работы по разработке парсера JSON представляла собой обработку комментариев. Это было вызвано не сложностью комментариев, а простотой остального JSON. Комментарии тормозили принятие формата.

Затем ко мне обратились представители сообщества YAML. JSON случайно вышел похожим на подмножество YAML. Если бы и я, и они внесли некоторые изменения, то JSON мог бы стать подходящим подмножеством. Одним из спорных моментов были комментарии.

Формат JSON предназначался исключительно для обмена данными по сети между программами, написанными на разных языках. Комментарии всегда игнорируются,

поэтому, когда они используются, снижается производительность сети. Комментарии были привлекательным излишеством, создающим помехи, поэтому я их удалил.

Если возникает реальная потребность в комментариях, есть несколько способов применить их. Просто пропустите закомментированный текст через минификатор, например, jsmin. К сожалению, это решение не подходит для тех программистов, которым не хватает навыков для написания простого конвейера.

Другим решением способна стать формализация комментариев и помещение их в структуру JSON. Если комментарии критически важны, следует дать им имена, чтобы их можно было соответствующим образом сохранить и обработать.

Мне не удалось сделать JSON истинным подмножеством JavaScript. В Юникоде для старых систем обработки текста есть пара невидимых управляющих символов — *разделитель абзацев (PS)* и *разделитель строк (LS)*. JavaScript обрабатывает их как символы окончания строки, так же *как возврат каретки (CR)* и *перевод строки (LF)*. Я упустил этот момент, и JSON допускает их присутствие в виде строк. К счастью, *PS* и *LS* используются крайне редко. Мне неизвестны какие-либо проблемы, возникающие из-за них. В ES5 был добавлен встроенный парсер JSON, который обрабатывает *PS* и *LS*, устраняя несовместимость.

В Юникоде есть ряд символов, которые в нем символами не считаются. Некоторые фанатики Юникода настаивают на том, что JSON не должен допускать применения magritte-символов. JSON этот вопрос совершенно не волнует. JSON — это носитель информации, а не проверяющая программа. Получатель должен сам решать, что делать с символами, которые не считаются символами. JSON не гарантирует значимости для каждого получателя всего, что было передано. Если отправитель и получатель одинаково понимают что-то, то это может быть выражено в JSON.

Мне хотелось сделать JSON независимым от IEEE 754. И в Java, и в JavaScript применяется двоичное число с плавающей точкой стандарта IEEE 754, но на JSON это никак не отражается. JSON способен обеспечить обмен данными между языками с различными представлениями чисел. То есть языки, использующие двоичные числа с плавающей точкой, могут обмениваться данными с языками, применяющими формат чисел BCD, а языки, использующие большие десятичные числа с плавающей точкой, могут взаимодействовать с языками, имеющими весьма странное внутреннее представление, в котором три цифры упакованы в десятиразрядные поля. JSON способен пережить моральное устаревание стандарта IEEE 754.

Я исключил `Infinity` и `NaN`, поскольку их присутствие в данных указывает на ошибку, а мы не должны помещать в сеть неверные данные. Такие данные не должны распространяться. Это весьма пагубная практика.

В нарушение принципа минимизации в качестве десятичных маркеров экспоненты были включены символы `e` и `E`. Мне следовало исключить `E`, а также знак `+` после `e`.

Я использовал null, потому что он нравится компьютерным специалистам. В JSON слову null не придают никакого значения, поэтому решение о нем отдано на откуп пользователям JSON. Формат JSON не определяет никакого поведения. Он определяет только формат, простую грамматику для данных.

Я создал одностраничный веб-сайт, в котором JSON описывается тремя способами: в виде грамматики в формате Маккимана (McKeeman Form grammar), синтаксических диаграмм и неформального английского языка. Надеюсь, читателю будет понятен хотя бы один из этих способов.

Я не создавал торговую марку или логотип для JSON. Не помещал на странице значок авторских прав. Я хотел сделать стандарт JSON как можно более свободным и ничем не обремененным. Мне не хотелось наживаться на JSON. Я просто хотел, чтобы мне разрешили его использовать.

Я начал получать запросы на добавление ссылок на кодеки на разных языках на сайт json.org, что и сделал. Практически все они открыты и бесплатны. Также я получал подарки — переводы страницы. Преимущество JSON еще и в том, что страницу о нем легко переводить.

В 2006 году я написал информационный запрос на комментарии, RFC 4627, для IETF (рабочей группы инженеров Интернета) с целью присвоения JSON MIME-типа. Я надеялся получить text/json, но лучшим, что они пожелали дать, был MIME-тип application/json. Это меня сильно разочаровало.

Учитывая успех JSON, я думаю, что IETF должен сделать json типом мультимедиа первого класса, чтобы люди могли регистрировать такие MIME-типы, как json/geo и json/money.

В 2013 году я написал стандарт ECMA-404, который был принят ISO как ISO/IEC 21778. JSON стал настолько важным, что на него должны были ссылаться другие стандарты, и они нуждались в более формальном цитировании по сравнению с цитированием моей веб-страницы.

Основы работоспособности JSON

Формат JSON предназначался для предоставления программам, написанным на разных языках, возможности эффективно обмениваться данными. Задача эта непростая, поскольку особенности представления значений и структур данных в языках могут быть довольно сложными. Поэтому при разработке JSON я сконцентрировался на унифицированности.

Представления чисел в языках сильно различаются, но все согласны с тем, что число может быть представлено в виде строки из десяти основных цифр, возможно, с десятичной точкой в нем и, возможно, десятичным показателем. В некоторых языках есть целочисленный тип, а в некоторых, например в JavaScript, его нет. В JSON считается, что все мы лучше всего можем понять строчное представление цифр.

У языков разное представление о строках и наборах символов. Язык способен иметь внутреннее представление UTF-16 (например, JavaScript), или UTF-32, или UTF-8, или ASCII, или Half ASCII, или EBCDIC. Для формата JSON это неважно. Максимально возможный смысл придается строке символов, получаемой по сети, и ее превращению в приемлемые локальные представления.

Во всех языках имеется структура данных, представляющая собой линейную последовательность значений. Детали могут быть совершенно разными, но все языки с помощью JSON-декодера способны извлекать смысл, заложенный в ряд значений, разделенных запятыми и заключенных в квадратные скобки. Языки с отсчетом элементов, начинающимся с нуля, могут обмениваться данными с языками, в которых отсчет начинается с единицы.

Большинство языков обладают некой структурой данных, связывающей имена со значениями. Детали могут сильно отличаться друг от друга, но все языки с помощью JSON-декодера способны извлечь смысл из серии именованных значений, разделенных запятыми и заключенных в фигурные скобки.

И этого вполне достаточно. Находя области пересечения всех языков программирования, программы, написанные на всех языках, могут обмениваться данными друг с другом. Критики сомневались в работоспособности такой схемы. И все же она работает.

Влияние сторонних факторов

Я смог предугадать переносимость объектных литералов благодаря своему опыту работы с двумя другими языками. Язык LISP имеет текстовые представления, называемые s-выражениями, которые используются как для программ, так и для данных. В языке Rebol применяется то же текстовое представление для программ и данных, но с гораздо более богатым синтаксисом.

Для Rebol вполне естественно использовать это текстовое представление для сериализации данных с целью их передачи. Та же идея была применена и к JavaScript.

Я не первый, кто задействовал JavaScript в качестве средства кодирования данных. До этого самостоятельно дошли многие еще в 1996 году. Я первым попытался продвинуть его в качестве стандарта.

Три языка — JavaScript, Python и NewtonScript — разработаны примерно в одно время, и у всех был одинаковый синтаксис для создания структур данных. С аналогичной системой записи немного раньше были созданы следующие списки свойств OpenStep Property Lists.

В комфорте нашего пост-XML-мира JSON выглядит как некая неизбежность. Но в ходе развития событий твердой уверенности в этом не было.

JSON-объект

В JavaScript поддержка JSON осуществляется с помощью двух функций, находящихся в JSON-объекте. Функции называются `parse` и `stringify`, и в этом всецело моя вина. Я выбрал название `parse`, взяв в качестве неудачного примера `Date`-функцию, которая, как уже было показано, имеет существенные недостатки. Название `stringify` появилось потому, что я полагал название `toString` неправильным. Следовало дать функциям имена `decode` и `encode`.

```
JSON.parse(текст, оживитель)
```

Функция `parse` принимает *`текст`* в формате JSON и декодирует его в данные JavaScript. Необязательная функция *`оживитель`* может выполнять преобразования. Ей даются ключ и значение, а она возвращает желаемое значение для этого ключа.

Например, можно создать *`оживитель`*, превращающий строки данных в объекты данных. Если ключ заканчивается на `_date` или если значение является строкой в формате данных ISO, то значение заменяется объектом `Date`, созданным из значения. Формально JSON не содержит дат, но даты могут быть удобно закодированы в виде строк, а затем на другой стороне превращены в объекты данных.

```
const rx_iso_date = /
    ^ \d{4} - \d{2} - \d{2}
    (?:
        T \d{2} : \d{2} : \d{2}
        (?:
            \. \d*
        )? Z
    )? $
/;

const text = JSON.parse(text, function (key, value) {
    return (
        typeof value === "string" && (
            key.endsWith("_date")
                || rx_iso_date.match(value)
            )
            ? new Date(value)
            : value
        );
    };
```

```
JSON.stringify(значение, заменитель, пробел)
```

Функция `stringify` принимает *`значение`* и кодирует его в текст JSON. Необязательная функция *`заменитель`* может выполнять преобразования. Она получает ключ и значение, а возвращает желаемое для этого ключа значение.

К примеру, если нужно автоматически превратить `Date`-объекты в ISO-строки, можно предоставить функцию *`заменитель`*. Результат работы функции *`заменитель`* передается функции `JSON.stringify` и включается в текст.

```
const json_text = JSON.stringify(my_little_object, function (key, value) {
    return (
        value instanceof Date
        ? value.toISOString()
        : value
    );
});
```

(Получается, что нам этого не нужно делать, поскольку все это выполняется автоматически методом `Date.prototype.toJSON`. См. далее описание метода `toJSON`.)

Аргумент *`заменитель`* может быть также массивом строк. В текст включаются только те свойства, имена которых есть в массиве. Это белый список, с помощью которого отфильтровываются неинтересные на данный момент свойства. Этот массив может быть и отдельным аргументом, не перезагружающим функцию *`заменитель`*.

Формат JSON допускает применение в тексте JSON пробельных символов, облегчающих его чтение человеком. По умолчанию `JSON.stringify` не добавляет пробельные символы, чтобы текст был компактнее для передачи. Если используется параметр *`пробел`*, применяется перевод строки и вставляются отступы. Корректным значением для параметра *`пробел`* будет число 4. И это вполне очевидно. Мне нужно было сделать это значение булевым.

```
toJSON()
```

Любой объект может иметь метод `toJSON`, вызываемый методом `JSON.stringify`, когда объект получает строковое представление. В результате JSON-представление могут иметь свободные от классов объекты. Подобный объект, содержащий только функции, получает строковое представление как пустой объект. Но если у него есть метод `toJSON`, то объект преобразуется в строку, как и возвращаемое значение метода `toJSON`.

Объекты `Date` наследуют метод `toJSON`, который кодирует объекты данных в виде строк ISO.

Последствия для безопасности

Первые пользователи JSON применяли для декодирования текста JSON JavaScript-функцию `eval` или ее эквивалент. Этой функции присущи определенные риски безопасности, но в данном случае потенциально опасный текст поступал с того самого сервера, который предоставлял HTML и все сценарии, поэтому его безопасность была не хуже, чем в Интернете в целом.

Ситуация изменилось, когда набрала популярность практика загрузки JSON с других серверов с тегами сценариев. Сценарии вычислялись с помощью функции `eval`, и не было никакой гарантии, что информационное наполнение состояло из чистого JSON, а не из кода XSS-атаки. Пользоваться этим было удобно, но легкомысленно.

Текст JSON ни в коем случае не должен создаваться путем объединения, поскольку фрагмент вредоносной строки может содержать символы кавычек и обратный слеш, из-за чего вероятна неправильная интерпретация текста. Текст JSON непременно должен создаваться с помощью таких кодировщиков, как `JSON.stringify`, или аналогичных инструментов, не допускающих подобного смешивания.

Именно поэтому результат работы метода `toJSON` и функции *`заменитель`* обрабатывается методом `JSON.stringify`. Если бы возвращаемые значения этих функций были вставлены в текст в чистом виде, то этими функциями можно было бы манипулировать, упрощая задачу подмешивания в строку вредоносного кода. Было бы лучше, если бы строки не начинались и не заканчивались одним и тем же символом, но такое случается. И это дает нам еще один повод для разумного подхода к программированию.

Форма Маккимана

Форма Маккимана представляет собой систему записи для выражения грамматики. Она была предложена Биллом Маккиманом (Bill McKeeman) из Дартмутского колледжа. Это упрощенная форма Бэкуса — Наура (Backus — Naur Form) с широким применением пробельных символов и минимальным использованием метасимволов. Мне она нравится, поскольку обладает полноценной минимальной адекватностью.

Грамматику в форме Маккимана можно выразить в самой форме Маккимана.

Элемент Юникода U+0020 используется в качестве пробела. А элемент Юникода U+000A применяется в качестве символа перевода строки:

```
пробел
    '0020'
перевод_строки
    '000A'
```

Имя — это последовательность букв:

```
имя
    последовательность_букв
    последовательность_букв имя
последовательность_букв
    'a' . 'z'
    'A' . 'Z'
```

Отступ состоит из четырех пробелов:

```
отступ
    пробел пробел пробел пробел
```

Грамматика представляет собой список из одного или нескольких правил:

```
грамматика
    правила
```

Каждое правило отделено переводом строки. У правила есть однострочное имя с альтернативами, указываемыми с отступом под ним.

```
правила
    правило
    правило перевод_строки правила
правило
    имя перевод_строки ничего альтернативы
```

Если первая строка после имени правила имеет вид пустых кавычек (""), то правило может не совпадать ни с чем. Это позволяет создавать необязательные правила.

```
ничего
    ""
    отступ '"' '"' перевод_строки
```

Каждая альтернатива указывается с отступом на отдельной строке. И она содержит один или несколько элементов, за которыми следует перевод строки:

```
отступы
    альтернатива
    альтернатива альтернативы
альтернатива
    отступ элемент перевод_строки
```

Элементы отделены друг от друга пробелами. Элемент является литералом или именем правила:

```
элементы
    элемент
    элемент пробел элементы
элемент
    литерал
    имя
```

Бывает две формы литералов. Одинарные кавычки могут указывать на кодовую точку или на кодовую точку из диапазона. Двойные кавычки могут указывать на несколько символов:

```
литерал
    ''' кодовая_точка ''' диапазон
    '"' символы '"'
```

В одинарные кавычки может быть помещена любая кодовая точка Юникода, за исключением 32 управляющих кодов. В одинарные кавычки может быть помещен также шестнадцатеричный код любой кодовой точки Юникода. Шестнадцатеричный код способен содержать четыре, пять или шесть шестнадцатеричных цифр (*hex*):

```
кодовая_точка
    '0020' . '10FFFF'
    шестнадцатеричный_код
```

```
шестнадцатеричный_код
    "10" hex hex hex hex
    hex hex hex hex hex
    hex hex hex hex
hex
    '0' . '9'
    'A' . 'F'
```

Диапазон указывается с помощью точки, за которой стоит еще одна кодовая точка. Далее дополнительно могут быть указаны знаки «минус» и исключаемые кодовые точки:

```
диапазон
    ""
    пробел '.' пробел ''' кодовая_точка ''' exclude
exclude
    ""
пробел '-' пробел ''' кодовая_точка ''' диапазон
```

Символ в двойных кавычках может быть любой кодовой точкой Юникода, за исключением 32 управляющих кодов и символа двойной кавычки. Определение символа показывает пример диапазона кодовых точек и исключения:

```
символы
    символ
    символ символы
символ
    '0020' . '10FFFF' - '"'
```

Грамматика JSON

Далее представлена грамматика JSON в форме Маккимана:

```
json
    элемент
значение
    объект
    массив
    строка
    число
    "true"
    "false"
    "null"
объект
    '{' ws '}'
    '{' компоненты '}'
компонент
    компонент
    компонент ',' компоненты
компонент
    ws строка ws ':' элемент
массив
    '[' ws ']'
    '[' элементы ']'
элементы
```

```
    элемент
    элемент ',' элементы
элемент
    ws значение ws
строка
    '"' символы '"'
символы
    ""
    символ символы
символ
    '0020' . '10FFFF' - '"' - '\'
    '\' переход
переход
    '"'
    '\'
    '/'
    'b'
    'n'
    'r'
    't'
    'u' hex hex hex hex
hex
    разряд
    'A' . 'F'
    'a' . 'f'
число
    int frac exp
int
    разряд
    разряды от 0 до 9
    '-' разряд
    '-' разряды от 0 до 9
разряды
    разряд
    разряд разряды
разряды
    '0'
    от 0 до 9
от 0 до 9
    '1' . '9'
frac
    ""
    '.' разряды
exp
    ""
    'E' знаковые разряды
    'e' знаковые разряды
знак
    ""
    '+'
    '-'
ws
    ""
    '0009' ws
    '000A' ws
    '000D' ws
    '0020' ws
```

Совет разработчикам стандартов обмена данными

Я не предполагаю, что JSON будет последним стандартом обмена данными. JSON разработан для конкретной цели, и он для нее действительно подходит. Сейчас он успешно используется и для многих других целей. Но для некоторых из них должны существовать более подходящие альтернативные варианты. Поэтому, если вы собираетесь разрабатывать следующий стандарт, я хочу дать несколько советов.

0. Пожалуйста, не ломайте JSON.

Лучшее, что я сделал с JSON, — не стал давать ему номер версии. Если чему-то дается версия 1.0, то подразумевается, что будет еще 1.1, 1.2 и т. д., пока дело не дойдет до версии 3.0.

У JSON только одна стандартная версия. Изменить JSON, не сломав в нем все, невозможно. Это позволяет избавиться от проблем управления версиями, характеризующихся болезненными и обременительными побочными эффектами постепенной подгонки и вечной бета-версии. В стеке программных средств может быть изменен каждый уровень, кроме уровня JSON. Формат JSON всегда будет таким, какой он сейчас. Он стабилен. Я не думаю, что есть какая-либо особенность ценнее этой.

1. Существенно улучшите свой стандарт.

Мне часто предлагали добавить в JSON еще одну особую функцию. Но тогда появятся два стандарта, в основном, но не полностью совместимые друг с другом. На пользователей обоих стандартов ляжет бремя обеспечения совместимости, выраженное в форме сбоев и проблем с настройками.

Чтобы компенсировать тяготы перехода на новый стандарт, снабдите его достаточным количеством реальных ценностей. Не вносите мелкие и косметические изменения. Сделайте новый стандарт содержательным и достойным усилий перехода к нему.

2. Придумайте для него более подходящее название.

Мне приходилось видеть множество предложений, где к имени JSON добавлялась буква или цифра. Не делайте этого. Дайте своему стандарту более подходящее название. В программировании немало усилий тратится на придумывание удачных имен. Покажите, на что вы способны.

Самое худшее в JSON — это имя. Оно означает JavaScript Object Notation — способ записи объектов JavaScript.

JS означает JavaScript. Проблема этой части имени заключается в том, что она вводит людей в заблуждение. Кто-то думает, что JSON имеет прямое отношение к JavaScript и работает только с этим языком. А кто-то полагает, что JSON определяется стандартом ECMAScript, что неверно, поскольку JSON определяется ECMA-404 — стандартом JSON. И есть те, кто до сих пор путает JavaScript с Java.

Мы упомянули JavaScript в имени, потому что хотели отдать должное его источнику. Мы не пытались спроецировать на название часть авторитета JavaScript. В 2001 году JavaScript был самым ненавистным языком программирования в мире. Его ненавидели все, кому нравились Java, C#, PHP, Python, Ruby, Perl, C++ и т. д. Мало того, JavaScript ненавидели почти все, кто использовал JavaScript. Времена благосклонности к этому языку тогда еще не наступили.

O означает Object. В JavaScript *объект* является неформальной коллекцией пар «имя:значение», а JSON взял это себе на вооружение. В других языках *объект* означает экземпляр хрупкой иерархии классов. Можно было бы ожидать, что JSON — это формат сериализации объекта, но это не так. JSON — формат сериализации данных. Соответственно, возникает путаница.

N означает Notation. Со способом *записи*, или *нотацией*, нет никаких проблем. Если захотите употребить в названии своего формата слово notation, я возражать не стану.

Глава 23

Как работает тестирование

> Тестирование программ может применяться
> для демонстрации наличия ошибок, но никогда
> не используется для демонстрации их отсутствия!
>
> *Эдсгер Дейкстра (Edsger Dijkstra)*

Компьютерные программы — самое сложное, что делают люди. Ничто иное не состоит из такого количества запутанных составляющих, от которых требуются идеальная сочетаемость и совместная работа. Готовая программа должна быть совершенной во всем для всех входных данных, состояний, условий и времен. Наш контракт с компьютером заключается в том, что, если мы дадим ему недоработанную программу, у него будет лицензия на худшие действия в самый неподходящий момент и не он в этом виноват. Вина лежит на вас.

Теста на совершенство не существует. Можно доказать, что в программе есть недоработки, но нет такого теста, с помощью которого реально было бы доказать, что программа не содержит дефектов. Отсутствие подобного тестирования послужило мотивом для разработки доказательств правильности (Proofs of Correctness). Идея заключалась в том, что программа будет разработана с математическими доказательствами ее совершенства. В компьютерной науке эта нерешенная задача считалась самой важной, и, к сожалению, она так и осталась нерешенной. Доказательства представлялись намного более сложными, чем программы, чье совершенство пытались доказать с их помощью. Сложность получалась просто запредельной.

Робин Мильнер (Robin Milner) и другие специалисты предположили, что более практичной альтернативой доказательствам способна стать корректность типов. Типы легче интегрируются в программы, чем доказательства. Мильнер уверенно заявил, что качественно набранные программы не могут «пойти не так». Но и в языках с развитыми системами типов, таких как Haskell, и в языках с отвратительными системами типов, таких как C++ и Java, программы все равно работают неправильно. Возможно, когда-нибудь типы и окупятся. Но такие времена еще не настали.

Итак, из-за несостоявшихся надежд на доказательства и применение типов мы запускаем в производство явно несовершенные программы, надеясь, что найдем

и исправим ошибки до того, как кто-нибудь о них узнает. Это крайне неразумно. Но пока все именно так.

Должен существовать более разумный выход. Но если он и есть, то до сих пор не найден.

Все это вынуждает нас вернуться к тестированию. Качественно выполненное тестирование может выявить дефекты, а затем вселить в нас уверенность в том, что дефекты были исправлены. Но даже самое качественное тестирование не в состоянии доказать отсутствие ошибок. А его высокое качество обеспечивается не всегда.

В конце 1970-х и начале 1980-х годов компания Atari опередила любые другие компании мира по объему продаж программных средств и количеству их покупателей. Основная часть программ была записана в ПЗУ (микросхеме памяти, предназначенной только для чтения). Исправить ошибки в ПЗУ невозможно. Их приходилось уничтожать и изготавливать новые микросхемы памяти. Atari создал домашний компьютер с операционной системой, записанной в ПЗУ. Я был этим очень впечатлен. До какой степени нужно быть уверенными в результатах труда своих программистов, чтобы рискнуть поместить их в ПЗУ! Как это у них получалось?

Я устроился исследователем в корпоративную лабораторию Atari и получил возможность постичь их секрет: они гоняли свое изделие во всех режимах и, если все работало, отправляли его на фабрику. Им просто везло. Везение закончилось, когда руководство решило, что игра под названием E. T. больше не нуждается в тестировании. Этого решения компания не пережила.

В те времена программы были намного меньше и проще. Если программа занимает всего несколько килобайтов, то ее грамотное составление и тестирование способно дать неплохие результаты. Сегодняшние программы разрослись до громадных размеров. Всецело осознать их суть не в состоянии никто. Как же можно быть уверенными в том, что такой сложный для понимания продукт работает правильно?

Ошибки

Впервые ошибки назвал багами (bugs — «жучки») Томас Альва Эдисон (Tomas Alva Edison). В то время он разрабатывал свой фонограф — устройство, записывающее и воспроизводящее звук с помощью иголки и листа фольги, обернутого вокруг вращающегося цилиндра. Его прототип издавал щелчок на каждом обороте, когда иголка пересекала край листа фольги. Эдисон сказал репортеру, что он не спал всю ночь, пытаясь достать *жучка* из фонографа.

Слово прижилось. За долгие годы, прошедшие с той поры, накопилось множество историй о сумасшедших изобретателях, которые разбогатели бы, если бы смогли извлечь последних «жучков» из своих изобретений.

На слуху также широко известная история о мотыльке, попавшем в реле компьютера Harvard Mark II. Грейс Мюррей Хоппер (Grace Murray Hopper) приклеила

насекомое в свой журнал и написала: «Первый фактический случай обнаружения ошибки». Это была не первая ошибка, но, возможно, первая, вызванная насекомым. Причинами большинства наших ошибок по-прежнему являются люди, хотя высказывают не лишенные основания тревоги по поводу вероятности появления ошибок в результате деятельности искусственного интеллекта.

Нужно стараться по максимуму избавиться от путаницы в программах. Если программа делает совсем не то, что от нее ожидается, значит, мы в ней запутались. Чтобы избежать путаницы, нужно стремиться к максимальной простоте и чистоте программ. *Ошибка* — синоним *беспорядка*. Его устранение гораздо продуктивнее тестирования.

Раздутость программ

Одна из самых серьезных проблем в разработке программных средств — их тучность, или раздутость. Программы просто становятся слишком большими. Это может быть связано с неразумным выбором функций, но чаще всего становится следствием плохой архитектуры. Популярное средство повторного использования кода — наследование, но его работа оставляет желать лучшего, поэтому вместо него зачастую применяются копирование и вставка кода. Не следует сбрасывать со счетов и чрезмерную зависимость от библиотек, платформ и пакетов, тесно связанных со многими другими библиотеками, платформами и пакетами. Раздутость может быть побочным эффектом приемов гибкой разработки. Чтобы справиться с ней, увеличивают численность команды разработчиков, но это порождает еще большую раздутость.

Раздутость создает проблемы безопасности, расширяя фронт атаки и давая ошибкам больше места, чтобы спрятаться. Раздутые системы гораздо труднее адекватно протестировать.

Кэширование способно сгладить ряд признаков раздутости в веб-браузере, но, к сожалению, веб-браузеры не очень хорошо справляются с кэшированием. Существуют такие инструментальные средства, как «ленивые» загрузчики (lazy loaders) и стряхиватели лишнего кода (tree shakers), которые пытаются задержать загрузку или удалить часть ненужного кода, но, устраняя симптомы, они могут стимулировать раздутость.

Лучший способ справиться с раздуванием программ — не допускать его. Приоритетом при разработке и реализации программы нужно сделать ее «худобу». Следует избегать внедрения в практику раздутых пакетов и инструментов, способствующих раздуванию. Обходитесь без классов. Нанимайте небольшие квалифицированные команды разработчиков. И активно практикуйте удаление кода. Создайте резерв из нескольких циклов разработки с целью удаления ненужного кода и избавления от проблемных пакетов. Радуйтесь, когда количество строк кода в проекте уменьшается. Придерживайтесь *принципа наименьшей раздутости*.

Широкие функциональные возможности дают определенные преимущества, но за все приходится платить. Если не осознавать, какова будет цена, придется платить раздуванием.

TDD

Как методология разработка на основе тестирования (Test Driven Development (TDD)) мне нравится. А вот как религию я ее просто ненавижу. Ярые приверженцы TDD сказали мне, что небрежно скроенный ненадежный код вполне допустим и даже приветствуется TDD. Предполагается, что тесты выявят все ошибки, поэтому нет необходимости в дисциплинированном стиле программирования.

Именно так рациональные приемы программирования превращаются в весьма пагубную практику. Истина заключается в том, что нельзя полагаться на тесты при поиске всех ошибок. Нужно вкладываться в предотвращение ошибок. Рекомендуемая практика программирования заключается в малозатратных вложениях в качество. С годами мой собственный стиль программирования изменился, поскольку я наблюдаю за тем, как формируются ошибки и каким образом можно снизить вероятность их появления.

Я получил сообщение от сторонника использования JSLint об ошибке этого анализатора кода. Он включил в свою программу функцию, отклоненную JSLint, и сказал, что, видимо, в JSLint есть что-то неправильное, поскольку эта функция прошла все свои блочные тесты. Внеплановая проверка показала, что JSLint был прав. Он обнаружил ошибку в регулярном выражении, которую не нашли тесты. Ошибки были в самих тестах. Ложные отрицательные результаты всегда исправляются быстро, а вот ложные положительные — неистребимы. Такие тесты дают нам ложную уверенность, но не добавляют качества. Так какими же должны быть настоящие тесты?

Для низкоуровневого кода весьма эффективны блочные тесты. Например, упомянутая в главе 3 библиотека Big Integer низкоуровневая и слабо зависит от чего-либо еще. Я написал для нее множество блочных тестов, и они очень помогли в разработке.

По мере того как мы взбираемся наверх, блочные тесты постепенно становятся все менее эффективными. С ростом зависимостей значимость тестов уменьшается. Необходимость разработки заглушек, имитаторов и фиктивных объектов приводит к росту затрат на создание тестов. (Мне приходилось наблюдать, как разработчики впустую тратят время на споры о том, является ли тот или иной элемент фиктивным объектом или имитатором.) И при переходе на более высокие уровни сложность смещается от компонентов к соединениям между компонентами.

Когда теряется чистота, количество ошибок возрастает, но блочные тесты не проверяют степень чистоты. В случае плохой модульности вероятность появления ошибок возрастает, но блочные тесты не проводят проверку на модульность. С те-

чением времени код становится раздутым, но блочные тесты не проверяют его на раздутость. На пределе возможностей такого рода тестирования мы тестируем фиктивные объекты и имитаторы, а не программу. Я не говорю, что блочные тесты никуда не годятся. Я говорю, что их недостаточно. Мы пишем все больше и больше тестов, которые обнаруживают все меньше и меньше ошибок. К этому привыкают, но это называется *попустительством*.

От этих фраз веет безнадежностью, но без тестирования не обойтись. Качественная, хорошо продуманная конструкция и ее программная реализация играют весьма важную роль, но этого недостаточно. Нам еще предстоит провести эффективное тестирование. Нужно все тщательно проверить.

Ты не должен пройти

Большинство библиотек тестирования поддерживают вызовы такого рода:

```
assertEquals("add 3 + 4", 7, add(3, 4));
```

Это позволяет получать удовольствие при написании теста, но допустимо ли полагать, что при одном только сложении можно обнаружить слабовыраженную ошибку в функции `add`? Требуется провести множество тестов в гораздо более широком диапазоне значений, но кому захочется создавать все эти тесты? Также эта форма не способна тестировать последующие программы. Она способна работать только со строго последовательными функциями.

Именно поэтому под впечатлением от инструментального средства QuickCheck из арсенала Haskell я написал библиотеку тестирования JSCheck. Она создает различные обстоятельства, автоматически выполняя множество случайных попыток. JSCheck поддерживает также событийное программирование для тестирования серверных и браузерных приложений.

```
jsc.claim(имя, предикат, сигнатура, классификатор)
```

Самая важная из предоставляемых JSCheck функций — `claim` (утверждение). Утверждение является суждением о вашей программе.

`Имя` — это описательная строка, используемая в отчетах.

`Предикат` — это функция, выдающая `true`, если программа работает правильно. Функция *`предикат`* принимает функцию обратного вызова *`вердикт`*, которая используется для выдачи результата каждой попытки. Остальные аргументы определяются аргументом *`сигнатура`*.

`Сигнатура` — это массив из функций-генераторов, производящих аргументы для функции *`предикат`*. Генераторы легко создаются со спецификаторами, предоставляемыми JSCheck.

Функция *классификатор* — необязательная. Она может отбрасывать недействительные попытки, а также классифицировать попытки, чтобы упростить оценку шаблонов.

```
jsc.check(конфигурация)
```

Можно создать сколько угодно утверждений, а затем вызвать функцию `jsc.check` для проверки того, что все утверждения оправдались.

Функция `jsc.check` принимает объект *конфигурация*, который может содержать любое из следующих свойств:

- `time_limit` в миллисекундах. Для каждой попытки возможны три исхода: `PASS`, `FAIL` и `LOST`. Любая попытка, не выдавшая вердикт до истечения времени, считается утраченной — `LOST`. В ряде ситуаций запоздавшее получение положительного ответа приравнивается к сбою;
- `on_pass` — обратный вызов для каждой удачной попытки;
- `on_fail` — обратный вызов для каждой неудачной попытки;
- `on_lost` — обратный вызов для каждой попытки, которой не удалось выдать вердикт;
- `on_report` — обратный вызов для отчета;
- `on_result` — обратный вызов для сводной информации;
- `nr_trials` — количество попыток, выполняемых для каждого утверждения;
- `detail` — уровень детализации отчета:
  - 0 — отчета не будет;
  - 1 — будет минимальный отчет, показывающий подсчет пройденных попыток каждого утверждения;
  - 2 — будет отчет об отдельных случаях сбоя;
  - 3 — дополнительно будет отчет по классификационным сводкам;
  - 4 — будет отчет по всем случаям.

Вместо тестирования отдельно каждой функции в библиотеке Big Integer я сконструировал тесты, включающие несколько совместно работающих функций. Например, в тесте `demorgan` применяются `random`, `mask`, `xor`, `or` и `eq`. JSCheck выдает случайные целые числа, используемые для создания случайных больших целых чисел, которые применяются с законом Де Моргана (DeMorgan Law).

```
jsc.claim(
    "demorgan",
    function (verdict, n) {

// !(a && b) === !a || !b
        let a = big_integer.random(n);
        let b = big_integer.random(n);
```

```
        let mask = big_integer.mask(n);
        let left = big_integer.xor(mask, big_integer.and(a, b));
        let right = big_integer.or(
            big_integer.xor(mask, a),
            big_integer.xor(mask, b)
        );
        return verdict(big_integer.eq(left, right));
    },
    [jsc.integer()]
);
```

Я создал генератор, выдающий большие целые числа для некоторых моих тестов. Спецификаторы, предоставляемые JSCheck, высокоэффективны, но им ничего не известно о больших числах, поэтому я написал собственный спецификатор.

```
function bigint(max_nr_bits) {
    return function () {
        let nr_bits = Math.floor(Math.random() * max_nr_bits);
        let result = big_integer.random(nr_bits);
        return (
            Math.random() < 0.5
            ? big_integer.neg(result)
            : result
        );
    }
}
```

Тестирование умножения и деления выполняется совместно. Я предоставляю функцию-классификатор, чтобы отбросить попытки, в которых будет деление на нуль.

```
jsc.claim(
    "mul & div",
    function (verdict, a, b) {
        let product = big_integer.mul(a, b);
        return verdict(big_integer.eq(a, big_integer.div(product, b)));
    },
    [bigint(99), bigint(99)],
    function classifier(a, b) {
        if (!big_integer.is_zero(b)) {
            return "";
        }
    }
);
```

Я опять тестирую умножение и деление вместе, но на этот раз с учетом остатков. Классификатор выделяет знаки двух значений, производя классификацию `"--"`, `"-+"`, `"+-"` и `"++"`. Это помогло изолировать ошибки, связанные с обработкой знаков.

```
jsc.claim("div & mul & remainder", function (verdict, a, b) {
    let [quotient, remainder] = big_integer.divrem(a, b);
    return verdict(big_integer.eq(
        a,
        big_integer.add(big_integer.mul(quotient, b), remainder)
    ));
}, [bigint(99), bigint(99)], function classifier(a, b) {
```

```
        if (!big_integer.is_zero(b)) {
            return a[0] + b[0];
        }
    });
```

Я выстроил тесты вокруг тождественностей. Например, прибавление `1` к строке из *n* единичных битов будет иметь такой же результат, что и `2 ** n`:

```
    jsc.claim("exp & mask", function (verdict, n) {
        return verdict(
            big_integer.eq(
                big_integer.add(big_integer.mask(n), big_integer.wun),
                big_integer.power(big_integer.two, n)
            )
        );
    }, [jsc.integer(100)]);
```

Здесь еще одно тождество, `(1 << n) - 1`, должно быть тождественно *n* единичным битам:

```
    jsc.claim("mask & shift_up", function (verdict, n) {
        return verdict(big_integer.eq(
            big_integer.sub(
                big_integer.shift_up(big_integer.wun, n),
                big_integer.wun
            ),
            big_integer.mask(n)
        ));
    }, [jsc.integer(0, 96)]);
```

Я сконструировал большой набор подобных тестов. Этот стиль тестирования дает мне гораздо больше уверенности, чем можно было бы получить от `3 + 4`.

JSCheck

Далее показана реализация JSCheck. Наиболее интересные его составляющие — это спецификаторы, которые могут создавать тестовые данные. Большинство из них пригодны для компоновки интересными способами, поскольку спецификаторы передают значения через функцию `resolve`. Функция `resolve` возвращает свой аргумент, если он не является функцией:

```
    function resolve(value, …rest) {
```

Функция `resolve` принимает значение. Если это значение — функция, она вызывается для создания возвращаемого значения. В противном случае функция возвращает значение:

```
        return (
            typeof value === "function"
            ? value(…rest)
            : value
        );
    }
```

Ранее уже упоминалось применение `literal` в качестве `constant`. Это позволяет отключать функции, и у вас может быть функция, прошедшая все попытки:

```
function literal(value) {
    return function () {
        return value;
    };
}
```

Спецификатор `boolean` производит генератор, который создает булевы значения:

```
function boolean(bias = 0.5) {
```

Сигнатура способна содержать булеву спецификацию. Может быть предоставлен необязательный параметр смещения (bias). Если смещение равно 0,25, то примерно четверть произведенных булевых значений будет `true`:

```
    bias = resolve(bias);
    return function () {
        return Math.random() < bias;
    };
}
```

Спецификатор `number`, что неудивительно, производит числа в диапазоне:

```
function number(from = 1, to = 0) {
    from = Number(resolve(from));
    to = Number(resolve(to));
    if (from > to) {
        [from, to] = [to, from];
    }
    const difference = to - from;
    return function () {
        return Math.random() * difference + from;
    };
}
```

Спецификатор `wun_of` принимает массив значений и генераторов и возвращает генератор, выдающий эти значения случайным образом. Спецификатор `wun_of` может дополнительно принимать массив показателей веса, способный смещать выбор:

```
function wun_of(array, weights) {
```

У спецификатора `wun_of` имеется две сигнатуры:

```
// wun_of(array)
//    Из массива берется и разрешается один элемент.
//    Элементы выбираются случайным образом с равными вероятностями.

// wun_of(array, weights)
//    Оба аргумента являются массивами одинаковой длины.
//    Чем больше показатель веса, тем выше вероятность, что элемент будет выбран.
```

```
    if (
        !Array.isArray(array)
        || array.length < 1
        || (
            weights !== undefined
            && (!Array.isArray(weights) || array.length !== weights.length)
        )
    ) {
        throw new Error("JSCheck wun_of");
    }
    if (weights === undefined) {
        return function () {
            return resolve(array[Math.floor(Math.random() * array.length)]);
        };
    }
    const total = weights.reduce(function (a, b) {
        return a + b;
    });
    let base = 0;
    const list = weights.map(function (value) {
        base += value;
        return base / total;
    });
    return function () {
        let x = Math.random();
        return resolve(array[list.findIndex(function (element) {
            return element >= x;
        })]);
    };
}
```

Спецификатор `sequence` принимает массив значений и генераторов и возвращает генератор, выдающий эти значения по порядку:

```
function sequence(seq) {
    seq = resolve(seq);
    if (!Array.isArray(seq)) {
        throw "JSCheck sequence";
    }
    let element_nr = -1;
    return function () {
        element_nr += 1;
        if (element_nr >= seq.length) {
            element_nr = 0;
        }
        return resolve(seq[element_nr]);
    };
}
```

Спецификатор `falsy` возвращает генератор, выдающий лживые значения:

```
const bottom = [false, null, undefined, "", 0, NaN];

function falsy() {
    return wun_of(bottom);
}
```

Спецификатор `integer` возвращает генератор, выдающий целые числа в выбранном диапазоне. Если диапазон не указан, он возвращает генератор, выдающий случайные простые числа меньше 1000:

```
const primes = [
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
    127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
    233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
    283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
    353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
    419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
    467, 479, 487, 491, 499, 503, 509, 521, 523, 541,
    547, 557, 563, 569, 571, 577, 587, 593, 599, 601,
    607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
    661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
    739, 743, 751, 757, 761, 769, 773, 787, 797, 809,
    811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
    877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
    947, 953, 967, 971, 977, 983, 991, 997
];

function integer_value(value, default_value) {
    value = resolve(value);
    return (
        typeof value === "number"
        ? Math.floor(value)
        : (
            typeof value === "string"
            ? value.charCodeAt(0)
            : default_value
        )
    );
}

function integer(i, j) {
    if (i === undefined) {
        return wun_of(primes);
    }
    i = integer_value(i, 1);
    if (j === undefined) {
        j = i;
        i = 1;
    } else {
        j = integer_value(j, 1);
    }
    if (i > j) {
        [i, j] = [j, i];
    }
    return function () {
        return Math.floor(Math.random() * (j + 1 - i) + i);
    };
}
```

Спецификатор `character` возвращает генератор, выдающий символы. Если ему передается целое число или два числа, генератор создает символы, чьи кодовые точки входят в указанный диапазон. Если ему передаются две строки, он возвращает символы в диапазоне, составленном из первой кодовой точки каждой строки. Если передается одна строка, он возвращает символы из нее. По умолчанию возвращаются ASCII-символы:

```
function character(i, j) {
    if (i === undefined) {
        return character(32, 126);
    }
    if (typeof i === "string") {
        return (
            j === undefined
            ? wun_of(i.split(""))
            : character(i.codePointAt(0), j.codePointAt(0))
        );
    }
    const ji = integer(i, j);
    return function () {
        return String.fromCodePoint(ji());
    };
}
```

Спецификатор `array` возвращает генератор, выдающий массивы:

```
function array(first, value) {
    if (Array.isArray(first)) {
        return function () {
            return first.map(resolve);
        };
    }
    if (first === undefined) {
        first = integer(4);
    }
    if (value === undefined) {
        value = integer();
    }
    return function () {
        const dimension = resolve(first);
        const result = new Array(dimension).fill(value);
        return (
            typeof value === "function"
            ? result.map(resolve)
            : result
        );
    };
}
```

Если аргументом является массив значений и генераторов, результатом становится массив, содержащий значения и результаты работы генераторов:

```
let my_little_array_specifier = jsc.array([
    jsc.integer(),
    jsc.number(100),
```

```
    jsc.string(8, jsc.character("A", "Z"))
])

my_little_array_specifier()     // [179, 21.228644298389554, "TJFJPLQA"]
my_little_array_specifier()     // [797, 57.05485427752137,  "CWQDVXWY"]
my_little_array_specifier()     // [941, 91.98980208020657,  "QVMGNVXK"]
my_little_array_specifier()     // [11,  87.07735128700733,  "GXBSVLKJ"]
```

В противном случае он создает массив значений. Можно предоставить размерность массива или генератор, создающий целочисленные значения. Или предоставить значение или генератор значений. По умолчанию используются случайные простые числа.

```
let my_other_little_array_specifier = jsc.array(4);

my_other_little_array_specifier()     // [659, 383, 991, 821]
my_other_little_array_specifier()     // [479, 701,  47, 601]
my_other_little_array_specifier()     // [389, 271, 113, 263]
my_other_little_array_specifier()     // [251, 557, 547, 197]
```

Спецификатор `string` возвращает генератор, выдающий строки. По умолчанию строки составляются из ASCII-символов.

```
function string(…parameters) {
    const length = parameters.length;

    if (length === 0) {
        return string(integer(10), character());
    }
    return function () {
        let pieces = [];
        let parameter_nr = 0;
        let value;
        while (true) {
            value = resolve(parameters[parameter_nr]);
            parameter_nr += 1;
            if (value === undefined) {
                break;
            }
            if (
                Number.isSafeInteger(value)
                && value >= 0
                && parameters[parameter_nr] !== undefined
            ) {
                pieces = pieces.concat(
                    new Array(value).fill(parameters[parameter_nr]).map(resolve)
                );
                parameter_nr += 1;
            } else {
                pieces.push(String(value));
            }
        }
        return pieces.join("");
    };
}
```

Он также принимает значения и генераторы, объединяя результаты.

```
let my_little_3_letter_word_specifier = jsc.string(
    jsc.sequence(["c", "d", "f"]),
    jsc.sequence(["a", "o", "i", "e"]),
    jsc.sequence(["t", "g", "n", "s", "l"])
)]);

my_little_3_letter_word_specifier()     // "cat"
my_little_3_letter_word_specifier()     // "dog"
my_little_3_letter_word_specifier()     // "fin"
my_little_3_letter_word_specifier()     // "ces"
```

Если параметр создает целое число, за которым следует строковое значение, он может использоваться в качестве длины.

```
let my_little_ssn_specifier = jsc.string(
    3, jsc.character("0", "9"),
    "-",
    2, jsc.character("0", "9"),
    "-",
    4, jsc.character("0", "9")
);

my_little_ssn_specifier()         // "231-89-2167"
my_little_ssn_specifier()         // "706-32-0392"
my_little_ssn_specifier()         // "931-89-4315"
my_little_ssn_specifier()         // "636-20-3790"
```

Спецификатор `any` возвращает генератор, выдающий случайные значения различных типов.

```
const misc = [
    true, Infinity, -Infinity, falsy(), Math.PI, Math.E, Number.EPSILON
];

function any() {
    return wun_of([integer(), number(), string(), wun_of(misc)]);
}
```

Спецификатор `object` возвращает генератор, выдающий объекты. По умолчанию создаются небольшие объекты со случайными ключами и случайными значениями.

```
function object(subject, value) {
    if (subject === undefined) {
        subject = integer(1, 4);
    }
    return function () {
        let result = {};
        const keys = resolve(subject);
        if (typeof keys === "number") {
            const text = string();
            const gen = any();
            let i = 0;
            while (i < keys) {
```

```
                result[text()] = gen();
                i += 1;
            }
            return result;
        }
        if (value === undefined) {
            if (keys && typeof keys === "object") {
                Object.keys(subject).forEach(function (key) {
                    result[key] = resolve(keys[key]);
                });
                return result;
            }
        } else {
            const values = resolve(value);
            if (Array.isArray(keys)) {
                keys.forEach(function (key, key_nr) {
                    result[key] = resolve((
                        Array.isArray(values)
                        ? values[key_nr % values.length]
                        : value
                    ), key_nr);
                });
                return result;
            }
        }
    };
}
```

Если ему передается массив имен и значение, он создает объект, используя эти имена в качестве имен свойств и задавая этим свойствам значения. Например, можно выдать массив, содержащий от трех до шести имен, где имена состоят из четырех букв в нижнем регистре, а значения являются булевыми:

```
let my_little_constructor = jsc.object(
    jsc.array(
        jsc.integer(3, 6),
        jsc.string(4, jsc.character("a", "z"))
    ),
    jsc.boolean()
);

my_little_constructor()
// {"hiyt": false, "rodf": true, "bfxf": false, "ygat": false, "hwqe": false}
my_little_constructor()
// {"hwbh": true, "ndjt": false, "chsn": true, "fdag": true, "hvme": true}
my_little_constructor()
// {"qedx": false, "uoyp": true, "ewes": true}
my_little_constructor()
// {"igko": true, "txem": true, "yadl": false, "avwz": true}
```

Если передан объект, создается объект, имеющий те же имена свойств:

```
let my_little_other_constructor = jsc.object({
    left: jsc.integer(640),
    top: jsc.integer(480),
```

```
    color: jsc.wun_of(["black", "white", "red", "blue", "green", "gray"])
});

my_little_other_constructor() // {"left": 305, "top": 360, "color": "gray"}
my_little_other_constructor() // {"left": 162, "top": 366, "color": "blue"}
my_little_other_constructor() // {"left": 110, "top": 5, "color": "blue"}
my_little_other_constructor() // {"left": 610, "top": 61, "color": "green"}
```

Можно составить множество тестовых данных. Но если есть некая форма данных, которую трудно создать путем составления этих спецификаторов, можно легко создать собственные спецификаторы. Спецификатор — просто функция, которая возвращает функцию.

А теперь приступим к работе с JSCheck.

Функция `crunch` перемалывает числа и подготавливает отчеты:

```
const ctp = "{name}: {class}{cases} cases tested, {pass} pass{fail}{lost}\n";

function crunch(detail, cases, serials) {
```

Прохождение через все примеры; сбор всех неудачных примеров; создание подробного отчета и сводных данных:

```
    let class_fail;
    let class_pass;
    let class_lost;
    let case_nr = 0;
    let lines = "";
    let losses = [];
    let next_case;
    let now_claim;
    let nr_class = 0;
    let nr_fail;
    let nr_lost;
    let nr_pass;
    let report = "";
    let the_case;
    let the_class;
    let total_fail = 0;
    let total_lost = 0;
    let total_pass = 0;

    function generate_line(type, level) {
        if (detail >= level) {
            lines += fulfill(
                " {type} [{serial}] {classification}{args}\n",
                {
                    type,
                    serial: the_case.serial,
                    classification: the_case.classification,
                    args: JSON.stringify(
                        the_case.args
                    ).replace(
                        /^\[/,
```

```
                    "("
                ).replace(
                    /\]$/,
                    ")"
                )
            }
        );
    }
}

function generate_class(key) {
    if (detail >= 3 || class_fail[key] || class_lost[key]) {
        report += fulfill(
            " {key} pass {pass}{fail}{lost}\n",
            {
                key,
                pass: class_pass[key],
                fail: (
                    class_fail[key]
                    ? " fail " + class_fail[key]
                    : ""
                ),
                lost: (
                    class_lost[key]
                    ? " lost " + class_lost[key]
                    : ""
                )
            }
        );
    }
}

if (cases) {
    while (true) {
        next_case = cases[serials[case_nr]];
        case_nr += 1;
        if (!next_case || (next_case.claim !== now_claim)) {
            if (now_claim) {
                if (detail >= 1) {
                    report += fulfill(
                        ctp,
                        {
                            name: the_case.name,
                            class: (
                                nr_class
                                ? nr_class + " classifications, "
                                : ""
                            ),
                            cases: nr_pass + nr_fail + nr_lost,
                            pass: nr_pass,
                            fail: (
                                nr_fail
                                ? ", " + nr_fail + " fail"
                                : ""
                            ),
```

```
                        lost: (
                            nr_lost
                            ? ", " + nr_lost + " lost"
                            : ""
                        )
                    }
                );
                if (detail >= 2) {
                    Object.keys(
                        class_pass
                    ).sort().forEach(
                        generate_class
                    );
                    report += lines;
                }
            }
            total_fail += nr_fail;
            total_lost += nr_lost;
            total_pass += nr_pass;
        }
        if (!next_case) {
            break;
        }
        nr_class = 0;
        nr_fail = 0;
        nr_lost = 0;
        nr_pass = 0;
        class_pass = {};
        class_fail = {};
        class_lost = {};
        lines = "";
    }
    the_case = next_case;
    now_claim = the_case.claim;
    the_class = the_case.classification;
    if (the_class && typeof class_pass[the_class] !== "number") {
        class_pass[the_class] = 0;
        class_fail[the_class] = 0;
        class_lost[the_class] = 0;
        nr_class += 1;
    }
    if (the_case.pass === true) {
        if (the_class) {
            class_pass[the_class] += 1;
        }
        if (detail >= 4) {
            generate_line("Pass", 4);
        }
        nr_pass += 1;
    } else if (the_case.pass === false) {
        if (the_class) {
            class_fail[the_class] += 1;
        }
        generate_line("FAIL", 2);
        nr_fail += 1;
    } else {
        if (the_class) {
```

```
                    class_lost[the_class] += 1;
                }
                generate_line("LOST", 2);
                losses[nr_lost] = the_case;
                nr_lost += 1;
            }
        }
        report += fulfill(
            "\nTotal pass {pass}{fail}{lost}\n",
            {
                pass: total_pass,
                fail: (
                    total_fail
                    ? ", fail " + total_fail
                    : ""
                ),
                lost: (
                    total_lost
                    ? ", lost " + total_lost
                    : ""
                )
            }
        );
    }
    return {losses, report, summary: {
        pass: total_pass,
        fail: total_fail,
        lost: total_lost,
        total: total_pass + total_fail + total_lost,
        ok: total_lost === 0 && total_fail === 0 && total_pass > 0
    }};
}
```

Модуль экспортирует конструктор, возвращающий `jsc`-объект. Этот объект обладает состоянием, так как содержит утверждения, которые нужно тестировать, поэтому каждый пользователь должен получать свежий экземпляр.

Значение `reject` используется для идентификации попыток, которые следует отбросить:

```
const reject = Object.freeze({});
```

Мы экспортируем функцию `jsc_constructor`. Функции `check` и `claim` обладают состоянием, поэтому они здесь и создаются. Я замораживаю конструктор, поскольку мне нравится все замораживать:

```
export default Object.freeze(function jsc_constructor() {
    let all_claims = [];
```

Работу выполняет функция `check`:

```
    function check(configuration) {
        let the_claims = all_claims;
        all_claims = [];
        let nr_trials = (
            configuration.nr_trials === undefined
            ? 100
```

```
        : configuration.nr_trials
    );

    function go(on, report) {
```

Вызов функции обратного вызова:

```
        try {
            return configuration[on](report);
        } catch (ignore) {}
    }
```

Функция `check` проверяет все попытки. Результаты предоставляются функциям обратного вызова:

```
    let cases = {};
    let all_started = false;
    let nr_pending = 0;
    let serials = [];
    let timeout_id;

    function finish() {
        if (timeout_id) {
            clearTimeout(timeout_id);
        }
        const {
            losses,
            summary,
            report
        } = crunch(
            (
                configuration.detail === undefined
                ? 3
                : configuration.detail
            ),
            cases,
            serials
        );
        losses.forEach(function (the_case) {
            go("on_lost", the_case);
        });
        go("on_result", summary);
        go("on_report", report);
        cases = undefined;
    }


    function register(serial, value) {
```

Эта функция используется функцией `claim` для регистрации нового примера, а также самим примером для вынесения вердикта. Эти два применения сопоставляются с помощью порядкового номера.

Если объект `cases` утрачен, все прибывшие позже потерянные результаты должны быть проигнорированы:

```
            if (cases) {
                let the_case = cases[serial];
```

Если порядковый номер не наблюдался, регистрируется новый пример и добавляется к коллекции примеров. Порядковый номер добавляется к коллекции порядковых номеров. Количество отложенных примеров увеличивается.

```
                if (the_case === undefined) {
                    value.serial = serial;
                    cases[serial] = value;
                    serials.push(serial);
                    nr_pending += 1;
                } else {
```

Теперь существующий пример получает свой вердикт. Если у него совершенно неожиданно уже имеется результат, выдается исключение. У каждого примера должен быть только один результат.

```
                if (
                    the_case.pass !== undefined
                    || typeof value !== "boolean"
                ) {
                    throw the_case;
                }
```

Если результат является булевым значением, пример обновляется и отправляется `on_pass` или `on_fail`:

```
                if (value === true) {
                    the_case.pass = true;
                    go("on_pass", the_case);
                } else {
                    the_case.pass = false;
                    go("on_fail", the_case);
                }
```

Этот пример больше не откладывается. Если все примеры были сгенерированы и дали результаты, нужно завершить работу.

```
                nr_pending -= 1;
                if (nr_pending <= 0 && all_started) {
                    finish();
                }
            }
        }
        return value;
    }
    let unique = 0;
```

Обработка каждого утверждения:

```
    the_claims.forEach(function (a_claim) {
        let at_most = nr_trials * 10;
        let case_nr = 0;
        let attempt_nr = 0;
```

Проход по генерации и тестированию примеров:

```
        while (case_nr < nr_trials && attempt_nr < at_most) {
            if (a_claim(register, unique) !== reject) {
                case_nr += 1;
                unique += 1;
            }
            attempt_nr += 1;
        }
    });
```

Были вызваны все предикаты примера:

```
    all_started = true;
```

Если все примеры возвратили вердикты, создание отчета:

```
    if (nr_pending <= 0) {
        finish();
```

В противном случае запуск таймера:

```
        } else if (configuration.time_limit !== undefined) {
            timeout_id = setTimeout(finish, configuration.time_limit);
        }
    }
```

Функция `claim` используется для подачи каждого утверждения. Все утверждения проверяются сразу при вызове функции `check`. Утверждение состоит:

- из описательного имени, отображаемого в отчете;
- функции-предиката, применяющей утверждение и возвращающей `true`, если оно соблюдается;
- функции с массивом сигнатур, определяющей типы и значения для функции-предиката;
- необязательной функции-классификатора, принимающей значения, создаваемые сигнатурой, и возвращающей строку для классификации попыток, или `undefined`, если предикату не нужно давать данный набор сгенерированных аргументов:

```
    function claim(name, predicate, signature, classifier) {
```

Функция размещается в наборе всех утверждений:

```
        if (!Array.isArray(signature)) {
            signature = [signature];
        }

        function the_claim(register, serial) {
            let args = signature.map(resolve);
            let classification = "";
```

Если была предоставлена функция-классификатор, она используется для получения классификации. Если классификация не является строкой, пример отбрасывается:

```
            if (classifier !== undefined) {
                classification = classifier(…args);
                if (typeof classification !== "string") {
                    return reject;
                }
            }
```

Создание функции вердикта, которая заключает в себя функцию регистрации:

```
            let verdict = function (result) {
                return register(serial, result);
            };
```

Регистрация объекта, представляющего данную попытку:

```
            register(serial, {
                args,
                claim: the_claim,
                classification,
                classifier,
                name,
                predicate,
                serial,
                signature,
                verdict
            });
```

Вызов предиката, передача ему функции вердикта и всех аргументов примера; чтобы сигнализировать о результате примера, предикат должен использовать обратный вызов вердикта:

```
                return predicate(verdict, …args);
            }
            all_claims.push(the_claim);
        }
```

И наконец, создается и возвращается экземпляр:

```
        return Object.freeze({
```

Спецификаторы:

```
            any,
            array,
            boolean,
            character,
            falsy,
            integer,
            literal,
            number,
            object,
            wun_of,
            sequence,
            string
```

Основные функции:

```
            check,
            claim
```

```
    });
});
```

Ecomcon

Если у файла имеются сильные зависимости от другого кода, я хочу поместить тесты этих отношений в тот же файл. Единственный способ определить, что модуль A хорошо работает с модулями B и C, — это запустить их вместе в едином контексте.

Я разработал простой инструмент включения условных комментариев под названием ecomcon (Enable Comments Conditionally). Я могу поместить тестирование и регистрацию в исходный код в виде снабженных тегами комментариев:

```
// Код тега
```

Тегом способно стать любое слово, например `test`. За ним может следовать код JavaScript. В обычной обстановке роль этих комментариев не меняется и они игнорируются. При минификации — удаляются. Функция `ecomcon` может их включить, удалив двойной слеш (`//`) и тег и оставив код JavaScript, который теперь может быть выполнен.

Можно создать особый вариант с массой проверок в ходе выполнения программы, а также с регистрационными записями и анализом. Будет разрешен доступ к переменным, которые обычно скрыты в областях видимости функций. Можно будет отслеживать значения по мере того, как они принимаются и сбрасываются. А еще тестировать целостность программы, убеждаясь, что особо важные ресурсы не повреждены.

Комментарии могут задействоваться также в качестве документации, поскольку в них показываются внутренние особенности файла и приводятся примеры его операций.

```
function ecomcon(исходная_строка, массив_тегов)
```

`Массив_тегов` содержит строки, разрешенные к использованию в качестве тегов.

В функции `ecomcon` нет ничего сложного:

```
const rx_crlf = /
    \n
|
    \r \n?
/;

const rx_ecomcon = /
    ^
    \/ \/
    ( [ a-z A-Z 0-9 _ ]+ )
    \u0020?
    ( .* )
```

```
    $
/;

// Определение групп:
// [1] Разрешенный тег комментария
// [2] Остальная часть строки

const rx_tag = /
    ^
    [ a-z A-Z 0-9 _ ]+
    $
/;

export default Object.freeze(function ecomcon(source_string, tag_array) {
    const tag = Object.create(null);
    tag_array.forEach(
        function (string) {
            if (!rx_tag.test(string)) {
                throw new Error("ecomcon: " + string);
            }
            tag[string] = true;
        }
    );
    return source_string.split(rx_crlf).map(
        function (line) {
            const array = line.match(rx_ecomcon);
            return (
                Array.isArray(array)
                ? (
                    tag[array[1]] === true
                    ? array[2] + "\n"
                    : ""
                )
                : line + "\n"
            );
        }
    ).join("");
});
```

Глава 24

Как работает оптимизация

●●○○○

> Большинство компьютерных прегрешений совершаются во имя эффективности (при этом не факт, что она будет достигнута), а не по любой другой причине, включая беспросветную тупость.
>
> *Уильям Вульф (William Wulf)*

Несколько первых поколений компьютеров были по современным стандартам весьма медленными. Поскольку запоздалое получение правильного ответа может быть приравнено к неспособности получить его, оптимизация производительности стала в программировании глубоко укоренившейся навязчивой идеей.

Современные устройства работают намного быстрее. Поэтому может показаться, что больше не стоит беспокоиться о производительности. Но к большинству приложений это не относится. Количество имеющихся у нас процессоров превышает потребности, и большинство их большую часть времени простаивает. У нас огромные избыточные мощности.

Некоторые вычисления все еще выполняются недостаточно быстро. Порой причина в том, что проблемы становятся масштабнее производительности. Большие данные становятся все больше. Медлительность может наблюдаться даже в области общения с человеком, где доминирует JavaScript.

Когда люди слишком долго не получают ответов, они раздражаются, потом разочаровываются и злятся. Если мы хотим добиться того, чтобы они были удовлетворены и лояльны, то наши системы должны быть отзывчивыми.

Поэтому вопросы повышения производительности с повестки дня не снимаются, но мы должны с ними справиться. Оптимизация может запросто усугубить ситуацию.

Существует весьма распространенное мнение, что полезна оптимизация любой мелочи, поскольку все сэкономленные наносекунды складываются. Это не так. Оптимизацию нужно применять только там, где можно получить существенное улучшение. Оптимизация в мелочах — пустая трата времени. Весь смысл оптимизации заключается в экономии времени, поэтому нам нужно оптимизировать саму оптимизацию.

Измерения

Компьютерное программирование известно также как *компьютерная наука* и *разработка программного обеспечения*. Эти названия выдают желаемое за действительное. Нашего понимания в этой области еще недостаточно, чтобы по-настоящему заниматься наукой и разработкой. У нас нет теории, отвечающей на самые важные вопросы в сфере управления программным проектом: сколько ошибок осталось и сколько времени понадобится, чтобы их устранить?

Существенная доля нашего творчества не поддается количественной оценке, но мы можем измерить производительность. Можно выполнить программу и проследить, сколько времени на это ушло. Эти цифры способны помочь нам лучше понять свои системы или же нас запутать.

Распространенная практика заключается в том, чтобы взять две функции языка, поместить их в циклы, а затем засечь время прохождения циклов. Самый быстрый подскажет, какую из функций использовать. Но при таком подходе могут возникать проблемы.

Результат способен оказаться бессмысленным. Что именно мы тестируем, производительность функции или способность среды выполнения оптимизировать эту функцию, когда она изолирована и выполняется миллион раз? Можно ли этот результат спроецировать на конкретную обстановку? Будут ли другие среды выполнения работать так же? А что еще более важно, будут ли работать так же будущие среды выполнения?

Результат способен оказаться несущественным. В определенном контексте в реальной программе может и не быть заметной разницы. Получается, что время потрачено на принятие решения на основе бессмысленных данных.

Вместо этого нужно выбирать функции, из которых получаются легко читаемые и обслуживаемые программы. Быстрая, но дефектная программа не станет нам подспорьем.

И опять измерения

У плотников есть поговорка: *семь раз отмерь, один раз отрежь*. Вроде бы здесь легко можно применить оптимизацию: один раз отмерь и отрежь. Повышенное внимание сокращает число ошибок, не позволяя тратить впустую ни время, ни материалы. А это дает немалый выигрыш.

Программисты должны руководствоваться следующим принципом: *измерить, затем вырезать, затем еще раз измерить*. Перед оптимизацией нужно измерить производительность оптимизируемого кода. Это делается, чтобы установить базовый уровень и показать, что код замедляет всю программу. Если код погоды не делает, нужно искать в другом месте. Затем проводится тщательная оптимизация кода. После чего снова выполняются измерения. Если изменения не позволяют

существенно превзойти базовый уровень, они отклоняются. Они не позволили добиться ожидаемого улучшения, а сбои мы не проверяем.

Большинство оптимизаций усложняют код, добавляя альтернативные пути и удаляя общий код. Это увеличивает объем кода, сложность его поддержки и затрудняет проведение адекватного тестирования. Если будет получено существенное ускорение, то можно считать, что овчинка стоит выделки. *Если же существенного ускорения нет, изменение можно считать ошибкой.* Оно приводит к снижению качества кода, либо обеспечивая минимальную компенсацию, либо вовсе ее не давая. Чистый код легче поддается осмыслению и поддержке. И мы не хотим от этого отказываться.

Основная часть кода незначительно влияет на производительность. Оптимизировать код, который не замедляет работу программы, — пустая трата времени.

Тщетность усилий

Излишняя возня с кодом редко ускоряет его, поскольку не устраняет основную причину медлительности. Приведу ряд наиболее ярких примеров тщетности усилий.

- **Неспособность к распараллеливанию.** Parseq для ускорения наших творений позволяет воспользоваться преимуществами параллелизма, присущего Вселенной. Если взамен заставить все выполняться последовательно, мы от него откажемся. В больших масштабах параллельное побеждает последовательное.
- **Нарушение закона о ходах (Law of Turns).** Когда блокируется цикл обработки, задержки добавляются ко всему, что происходит далее, пока очередь наконец не опустеет. Накопление задержек может стать помехой для опустошения очереди.
- **Слабая связность.** Если наши модули не обладают сильной связностью, то, скорее всего, они заняты тем, в чем нет необходимости. Ненужная работа замедляет выполнение программы.
- **Сильное сцепление.** Когда модули сильно сцеплены друг с другом, мы жертвуем локальностью. Это может привести к чрезмерной загруженности протоколов ненужным обменом данными, добавляющим каждому этапу обработки сетевые задержки.
- **Неверный алгоритм.** Небрежно составленная функция $O(n \log n)$ может при достаточно больших значениях n легко превзойти тонко проработанную, изощренно оптимизированную функцию $O(n^2)$. А когда значения n невелики, то, какая из них работает быстрее, не так уж и важно.
- **Переполнение памяти.** Раньше эта проблема касалась систем виртуальной памяти, и она по-прежнему наблюдается при веб-кэшировании. Через кэш

проходит столько мусора из Интернета, что многое полезное оттуда просто вымывается, прежде чем дело дойдет до его повторного использования.

- **Раздутость.** Вероятность полностью разобраться во всем, что делается в весьма объемном и излишне раздутом коде, довольно низка. В нем может выполняться гораздо больше работы, чем фактически необходимо. Не зацикливайтесь на скорости выполнения, лучше сосредоточьтесь на объеме кода.
- **Код, созданный другими людьми.** Возможно, ваш код зависит от применения других пакетов, библиотек, платформ, браузеров, серверов и операционных систем. Мне неизвестно, с чем вам приходится запускать свои программы, но я уверен, что возня с вашим кодом не заставит их код работать быстрее. Если ваш код выполняется влет, большинство веб-пользователей не заметят большой разницы.

Язык

Возможно, лучшим с точки зрения оптимизации станет вклад в сам механизм реализации языка. Проведенная в нем оптимизация станет выгодна всем его пользователям. Это позволит сосредоточиться на написании высококачественных программ. А механизм реализации придаст им высокую скорость выполнения.

Первые механизмы реализации JavaScript были оптимизированы для быстрого завоевания рынка. Неудивительно, что они не могли похвастаться скоростью выполнения программ, из-за чего сложилось мнение, что JavaScript — игрушечный язык.

Но, как бы то ни было, скорости JavaScript хватало для большинства веб-приложений. Основная масса проблем с производительностью в браузере была связана с неправильным использованием сети и последовательной загрузкой ресурсов вместо конвейерной передачи. И существовал крайне неэффективный DOM API. DOM-модель не является частью JavaScript, но в низкой производительности DOM обвинили JavaScript. Но, как бы ни была плоха DOM-модель, большинство веб-приложений работало вполне приемлемо. При этом скорость выполнения кода JavaScript весьма редко оказывалась решающим фактором.

С тех пор механизмы реализации JavaScript стали работать намного быстрее, но это далось нелегко. Все простые работы по оптимизации уже были выполнены. И добиться чего-то нового становится все сложнее. Есть оптимизации, способные масштабно ускорить выполнение программ, но они требуют увеличения времени запуска. Это тот самый случай, который может заставить людей огорчиться и рассердиться.

Поэтому ведется весьма сложная игра с быстрым начальным созданием медленного кода и дальнейшей оптимизацией в соответствии с поведением программы. До сих пор эта игра велась неплохо, хотя давалась нелегко. Сделать быстрее

абсолютно все невозможно, поэтому пошла игра на ставки. Оптимизируется то, что, как представляется, принесет наибольшую пользу самым привередливым разработчикам.

По мере усложнения языка игра становится все сложнее. Число монстров лишь растет и никогда не уменьшается. В итоге сложность станет запредельной и игра будет окончена. Более простой, чистый и обыкновенный язык было бы намного легче оптимизировать.

Прослеживается также весьма неприятная обратная связь. Механизмы реализации ускоряют выполнение некоторых функций. Программисты обнаруживают это и начинают активно работать с данными функциями. Возникает своеобразная модель использования, влияющая на наблюдающих за ней разработчиков механизмов реализации языка. Эта замкнутая модель стимулирует вклад в оптимизацию, но не факт, что при этом выдерживается направление на оптимизацию хороших программ.

Глава 25

Как работает транспиляция

●●○○●

Я не он[1].

Нео

Важность языка JavaScript в качестве цели компиляции неуклонно возрастает. Транспиляция — это специализированная форма компиляции, превращающая один язык программирования в другой, в роли которого зачастую выступает JavaScript. Тем самым JavaScript благодаря своим универсальности и стабильности рассматривается в качестве переносимого исполняемого формата. JavaScript стал виртуальной машиной Интернета. Мы всегда думали, что в этом качестве выступит виртуальная машина Java (JVM), но вышло так, что роль досталась JavaScript. Этот язык в качестве минифицированного исходного кода, несомненно, легче переносится и выполняется со скоростью, достаточной для разбивки на лексемы и синтаксического анализа.

Порой исходный язык представляет собой диалект JavaScript, иногда это другой существующий язык, а бывает, что это совершенно новый язык, разработанный специально для транспиляции.

Создание транспилятора можно обосновать множеством соображений.

- **Проведением экспериментов.** Использование транспиляторов — идеальный способ создания и тестирования экспериментальных языков и функций. Замыслы могут быть протестированы в JavaScript с гораздо меньшими затратами времени и усилий, чем в ином случае.
- **Специализацией.** Транспиляторы могут применяться для реализации небольших по объему языков, предназначенных для весьма узких целей. Такие языки способны уменьшить функциональную нагрузку на повторяющиеся задачи.

---

[1] I'm not the wun. Перефразирована цитата Нео из фильма «Матрица»: I'm not the One («Я не Избранный»). — *Примеч. ред.*

- **Использованием ранее разработанного.** Транспиляторы предоставляют способ защиты вложений в языки, не набравшие достаточной популярности, позволяющий воспользоваться всеми достижениями JavaScript. За счет транспиляции в JavaScript старые программы потенциально могут запускаться на чем угодно. Чтобы воспользоваться JavaScript, не нужно понимать, как этот язык работает. (Неспособность разобраться в том, что вы делаете, — плохой признак.)
- **Следованием определенной моде.** Программисты могут стать приверженцами какой-либо моды. Транспиляция позволяет создавать программы с соответствующим этой моде синтаксисом.
- **Возможностями раннего доступа.** Стандартизация, внедрение и повсеместное использование новых функций языка происходят порой весьма медленно и нерешительно. Иногда транспиляция делает такие функции доступными еще до их широкого внедрения, переводя код следующего выпуска JavaScript в код текущего выпуска. Обычно это не настоятельная необходимость, а просто дань новой моде. Но иногда мода не может ждать.
- **Соображениями безопасности.** В JavaScript немало слабых мест в области обеспечения безопасности. Транспилятор может улучшить ситуацию, удалив проблемные функции и добавив проверку в среде выполнения и косвенное обращение с целью поставить потенциально вредоносные программы под контроль.
- **Повышением производительности.** Разработки наподобие ASM.js и Web Assembly пытаются получить преимущество в производительности, убирая из JavaScript все подходящее для удаления.

Транспиляторы не должны использоваться в производственном режиме. Мне нравятся транспиляторы, поскольку они приносят пользу в образовательном процессе и исследованиях, но недальновидное следование определенной моде и образ мышления, характерный для прошлого, могут через несколько лет вылиться в весьма негативные последствия. Мой совет — писать превосходные программы на JavaScript.

Neo

Neo представляет собой транспилируемый язык. В следующих главах я покажу, как он реализован в таких стадиях: разбиения на лексемы, парсинга, генерации кода и времени выполнения.

Neo — образовательный язык. Он должен помочь перейти к следующему языку, исправляя ряд самых больших ошибок JavaScript и удаляя функции, которые сильнее всего укоренились в старой парадигме. В частности, удален синтаксис языка C. Трудно смотреть в будущее тем, кто застрял в 1970-х.

Neo во многом похож на JavaScript, но есть в нем и существенные отличия. Некоторые из них носят поверхностный характер, а некоторые уходят весьма глубоко.

- В нем отсутствуют зарезервированные слова.
- Имена могут содержать пробелы. Имена могут оканчиваться вопросительным знаком (`?`).
- Комментарии создаются с использованием знака решетки (`#`). Комментарии заканчиваются в конце строки.
- В Neo не требуется применять точку с запятой, в нем значимую роль играют пробельные символы. Длинные инструкции можно разбивать после левой круглой скобки (`(`), левой квадратной скобки (`[`), левой фигурной скобки (`{`) и знака `ƒ`.
- В Neo есть несколько уровней приоритета. Приоритетность операторов исключает необходимость использования круглых скобок, но если уровней приоритета слишком много, все их будет трудно запомнить и появятся ошибки. Посмотрите на эти уровни, перечисленные от самых слабых к самым сильным.

  0. `? !` *(тернарный оператор)* `| |` *(значение по умолчанию)*.

  1. `/\` *(логическое И)* `\/` *(логическое ИЛИ)*.

  2. `=` *(равно)* `≠` *(не равно)* `<` *(меньше)* `>` *(больше)* `≤` *(меньше или равно)* `≥` *(больше или равно)*.

  3. `~` *(объединение)* `≈` *(объединение с пробелом)*.

  4. `+` *(сложение)* `-` *(вычитание)* `<<` *(минимум)* `>>` *(максимум)*.

  5. `*` *(умножение)* `/` *(деление)*.

  6. `.` *(уточнение)* `[ ]` *(список индексов)* `( )` *(вызов)*.

- `null` объединяет `null`, `undefined` и `NaN` в один объект, исключая путаницу, связанную с тем, какое из этих значений и когда использовать. `null` — пустой неизменяемый объект. Получение свойства из `null` не приводит к сбою, просто выдается `null`, поэтому выражения пути работают. Попытки изменить или вызвать `null` окажутся неудачными.
- Префиксные операторы не применяются. Вместо них имеются унарные функции. Внешнее различие заключается в том, что для функций требуются прародители, а для операторов они необязательны. Функции более гибкие, и, используя исключительно их, мы устраняем источник путаницы. Другие операторы я оставляю. Если заменить функциями все операторы, то с большой долей вероятности в итоге я случайно смогу заново изобрести LISP. Что было бы неплохо.
- В Neo один числовой тип — большие десятичные числа (Big Decimal). Это означает, что десятичная арифметика может быть точной. Включено также каждое числовое значение, которое может быть точно представлено

языком JavaScript. Поскольку числа большие, нет необходимости использовать `MIN_VALUE`, `EPSILON`, `MAX_SAFE_INTEGER`, `MAX_VALUE` или `Infinity`.

```
0.1 + 0.2 = 0.3  # наконец-то истина
```

- Последовательность символов называется текстом.
- В Neo есть тип данных под названием *«запись»* (record), который объединяет объекты и коллекции пар «ключ — значение» (weakmaps) JavaScript. Теперь термин *«объект»* можно использовать для включения в язык всех типов. Запись не наследуется. Запись — это контейнер *полей*. У поля есть *ключ* — это текст, запись или массив. У поля есть *значение* — любое, кроме `null`. При изменении значения поля на `null` оно удаляется из записи.
- Массивы создаются с помощью литералов массива и функции `array(`*`размер`*`)`, которая создает новый массив, задает длину и инициализирует элементы значением `null`. Индексы являются неотрицательными целыми числами, не превышающими *`размер`*. Попытка хранения вне этого диапазона приводит к сбою. К массиву можно добавить специальную форму `let`, которая также увеличивает *`размер`* на единицу.

```
def my little array: array(10)
let my little array[10]: 666 # СБОЙ
let my little array[]: 555

my little array[0] # null
my little array[10] # 555
length(my little array) # 11
```

- Функция `stone` (превратить в камень) выполняет глубокую заморозку, заменяя функцию `Object.freeze`. Проблема с названием `freeze` (заморозить) заключается в том, что оно оставляет надежду на разморозку. А превращенное в камень возвратить невозможно.
- В тернарном операторе используются символы `?` и `!`. Условие должно вычисляться в булево значение.

```
call (
    my hero = "monster"
    ? blood curdling scream
    ! (
        my hero = "butterfly" \/ my hero = "unicorn"
        ? do not make a sound
        ! sing like a rainbow
    )
)()
```

- Логическими операторами, выполняющими короткозамкнутые вычисления, являются `/\` (косая черта и обратная косая черта) для логического И, а также `\/` (обратная косая черта и косая черта) для логического ИЛИ. Логически оператор `not` (НЕ) — унарная функция. Если операторам `/\`, `\/` и `not` передать не булевы значения, они дадут сбой.

- К числу унарных арифметических функций относятся `abs`, `fraction`, `integer` и `neg`. Это унарные функции, а не операторы. Следует заметить, что `-` (знак «минус»), превращающий значение в отрицательное, работает только с числовыми литералами.
- Оператор `+` (знак «плюс») выполняет сложение, а не объединение. Другими арифметическими операторами являются `-` (знак «минус») для вычитания, `*` (звездочка) для умножения, `/` (слеш) для деления, `>>` (два знака «больше») для максимума, `<<` (два знака «меньше») для минимума.
- Оператор `~` (тильда) выполняет объединение, оператор `≈` (двойная тильда) — объединение с разделительным пробелом:

```
"Hello" ~ "World"   # "HelloWorld"
"Hello" ≈ "World"   # "Hello World"
"Hello" ≈ ""        # "Hello"
"Hello" ≈ null      # null
```

  Они пытаются принудить свои операнды превратиться в текст.
- Поразрядные функции `bit mask`, `bit shift up`, `bit shift down`, `bit and`, `bit or` и `bit xor` работают с целыми числами любого размера.
- Оператор `typeof` заменен следующими функциями-предикатами: `array?`, `boolean?`, `function?`, `number?`, `record?`, `text?`.
- Функции `Number.isInteger` и `Number.isSafeInteger` заменены функцией `integer?`.
- Функция `Object.isFrozen` заменена функцией `stone?`.
- Функция `char` принимает кодовую точку, а возвращает текст.
- Функция `code` принимает текст, а возвращает первую кодовую точку.
- Функция `length` принимает массив или текст, а возвращает количество элементов или символов.
- Функция `array` создает массив. Способ создания зависит от ее аргумента.
  - Если аргумент — неотрицательное целое число, создается массив, размер которого равен этому числу. Если предоставлен еще один аргумент:
    - если это `null`, все элементы инициализируются значением `null`;
    - если это функция, то выполняется ее вызов для создания значений инициализации для элементов;
    - в противном случае этот аргумент предоставляет значение инициализации для элементов.
  - Если аргумент является массивом, то делается поверхностная копия этого массива. Дополнительные аргументы могут дать стартовую и конечную позиции для копирования части массива.
  - Если аргумент — это запись, то создается массив из текстовых ключей.

  - Если аргумент является текстом и имеется второй аргумент, то этот второй аргумент используется для разбиения текста на массив, составленный из фрагментов этого текста.
- Функция `number` принимает текст, а возвращает число. Она может принимать необязательное основание системы счисления.
- Функция `record` создает запись. Способ зависит от ее аргумента.
  - Если аргумент является массивом, то элементы массива используются в качестве ключей. Значения зависят еще от одного аргумента:
    - если он имеет значение `null`, то все значения превращаются в `true`. Иногда это называется *набором*;
    - если это массив, то значениями становятся его элементы. Массивы должны иметь такую же длину;
    - если это функция, то она применяется для генерирования значений;
    - в противном случае аргумент используется в качестве исходного значения для всех полей.
  - Если аргумент — это запись, то создается поверхностная копия записи, при этом копируются только текстовые ключи. Если есть еще один аргумент — массив ключей, то копируются только эти ключи.
  - Если аргумент имеет значение `null`, создается пустая запись.
- Функция `text` создает текст. Способ создания зависит от ее аргумента.
  - Если это число, то оно преобразуется в текст. Может приниматься необязательное основание системы счисления.
  - Если это массив, то все его элементы объединяются. Еще один аргумент способен предоставить текст разделителя.
  - Если это текст, то его часть может быть обозначена двумя дополнительными аргументами. К сожалению, эти аргументы определяют кодовые единицы. В идеале текст должен иметь внутреннее представление UTF-32. Для этого нам придется подождать появления следующего языка.
- Функциональные объекты создаются с помощью оператора *`ƒ`* (символ флорина). Используются две формы:

```
ƒ список параметров (выражение)

ƒ список параметров {
    тело функции
}
```

- *`Список параметров`* представляет собой список имен необязательных значений по умолчанию, разделенных запятыми (`,`), и многоточие (…). Функции безымянные. Чтобы дать функциям имена, используется инструкция `def`. Функциональные объекты изменениям не поддаются.

- Функциональное выражение возвращает значение выражения. Тело функции должно возвращать значение в явном виде.
- Оператором значения по умолчанию является `|`*`выражение`*`|`. Если слева от выражения или параметра стоит значение или `null`, то находящееся справа *`выражение`*, заключенное в символы вертикальной черты, вычисляется для получения значения по умолчанию. Это краткая форма записи.
- Многоточие (…) аналогично применяемому в JavaScript, за исключением того, что оно всегда ставится после обозначения массива, а не перед ним:

```
def continuize: ƒ any (
    ƒ callback, arguments… (callback(any(arguments…)))
)
```

- Использование *`ƒ`* в качестве префикса перед оператором создает *указатель на функцию (functino)*, позволяя задействовать оператор как обычную функцию. То есть *`ƒ+`* создает функцию сложения с двумя операндами, которая может быть передана функции свертки для получения сумм. *`ƒ+`*`(3, 4)` возвращает `7`.

```
ƒ/\ и ƒ\/ или ƒ= равно ƒ≠ не равно ƒ< меньше ƒ≥ больше или равно ƒ>
больше ƒ≤ меньше или равно ƒ~ объединение ƒ≈ объединение с пробелом
ƒ+ сложение ƒ- вычитание ƒ>> максимум ƒ<< минимум ƒ* умножение ƒ/ деление
ƒ[] получение ƒ() решение ƒ?! тернарная функция ƒ|| получение значения
по умолчанию
```

- Функция может быть вызвана как метод. Например:

```
my little function.method(x, y)
```

  делает то же самое, что и:

```
my little function("method", [x, y])
```

  Этот простой механизм позволяет использовать функцию в качестве посредника для записи.
- Инструкция `def` заменяет инструкцию `const`.
- Инструкция `var` объявляет переменные.
- Инструкция `let` может изменить значение переменной, или поля записи, или элемента массива. Инструкция `let` — единственное место, где разрешено изменение. Операторов присваивания нет. Я не могу избавиться от присваивания, но способен его ограничить.

```
def pi: 3.14159265358979323846264338327950288419716939937510582097494459
var counter: 0
let counter: counter + 1
```

- Область видимости блоков отсутствует, поскольку нет самих блоков. Есть область видимости функций.

- Инструкция `call` позволяет вызывать функцию и игнорировать ее возвращаемое значение:

```
call my little impure side effect causer()
```

  В Neo есть `if` и `else`. Есть также `else if` в качестве замены инструкции `switch`. Если условное выражение не выдает булево значение, происходит сбой. Лживые значения отсутствуют.

```
if my hero = "monster"
    call blood curdling scream()
else if my hero = "butterfly" \/ my hero = "unicorn"
    call do not make a sound()
else
    call sing like a rainbow()
```

- Инструкция `loop` заменяет инструкции `do`, `for` и `while`. Я хочу, чтобы вы перестали применять циклы. Инструкция `loop` — это простой бесконечный цикл. Для выхода нужно использовать `break` или `return`. У циклов не бывает меток. Циклы могут быть вложенными. В них не могут создаваться новые функции.
- Исключения заменяются сбоями. У функции может быть обработчик сбоя. Инструкция `try` отсутствует.

```
def my little function: ƒ x the unknown {
    return risky operation(x the unknown)
failure
    call launch all missiles()
    return null
}
```

- Инструкция `fail` сообщает о сбое. Она не принимает объект исключения или какое-то другое сообщение. Если нужно сообщить причину сбоя, ее следует зарегистрировать до инструкции `fail` каким-либо иным способом.
- У модуля может быть несколько инструкций `import`:

```
import имя: текстовый литерал
```

- У модуля может быть одна инструкция `export`:

```
export выражение
```

Пример

В Neo используется реверсная функция свертки `reduce reverse`. Она допускает ранний выход и работает в обратном направлении. Принимает три аргумента: массив `array`, функцию обратного вызова `callback function` и исходное значение `initial value`. Функция обратного вызова получает четыре значения: текущее значение свертки, текущий элемент массива, число индекса текущего элемента и функцию выхода. Если функции обратного вызова `callback function` потребуется

ранняя остановка операции, она возвращает результаты вызова функции выхода с итоговым значением.

```
def reduce reverse: f array, callback function, initial value {

# Свертка массива до одного значения.

# Если исходное значение не предоставлено, используется нулевой элемент,
# а первая итерация пропускается.

    var element nr: length(array)
    var reduction: initial value
    if reduction = null
        let element nr: element nr - 1
        let reduction: array[element nr]

# Функция обратного вызова получает функцию выхода,
# которую она может вызвать для остановки операции.

    def exit: f final value {
        let element nr: 0
        return final value
    }

# Выполнение цикла, пока массив не исчерпается или не будет запрошен ранний выход.
# Вызов в каждой итерации функции обратного вызова со следующим инкрементом.

    loop
        let element nr: element nr - 1
        if element nr < 0
            break
        let reduction: callback function(
            reduction
            array[element nr]
            element nr
            exit
        )
    return reduction
}
```

Следующий язык

Neo — это не следующий язык. Это даже не полноценный язык — не хватает многих важных компонентов. Сама по себе эта проблема несерьезна: мы знаем, что добавить материал в язык нетрудно.

В Neo нет поддержки JSON. Ни один серьезный язык XXI века не может существовать без встроенного механизма кодирования и декодирования JSON.

В Neo отсутствует форма сопоставления с текстовым шаблоном. В JavaScript для этого используются регулярные выражения. Следующий язык должен поддерживать контекстно-свободные языки с менее таинственной системой записи.

Следующему языку нужно лучше поддерживать Юникод. Например, должна существовать какая-то форма разбиения текста на глифы, учитывающая наличие комбинированных символов.

В качестве внутреннего представления символов в следующем языке должен использоваться UTF-32, как бы экстравагантно это ни выглядело. Я помню времена, когда расточительством считались 8 бит на символ. Тогда память измерялась в килобайтах. Сейчас она измеряется в гигабайтах. Объем памяти вырос настолько, что размером символа можно пренебречь. С UTF-32 разница между единицами и точками кода исчезает, что упрощает написание программ, правильно работающих в международном формате.

Следующий язык должен непосредственно поддерживать BLOB-объекты. Требуется, чтобы некоторые данные были большими, запутанными, красиво упакованными и таинственными.

В следующем языке должна существовать более широкая поддержка событийного программирования, включая цикл обработки и механизм для отправки сообщений и упорядоченной доставки.

В следующем языке должна быть более качественная поддержка безопасной сетевой работы.

Следующий язык должен поддерживать управление процессами: запуском, обменом данными и уничтожением. Должна иметься возможность связывать процессы с целью их группового самоуничтожения. Если один из них дает сбой, он считается общим для всех и возникает возможность повторно запускать их с заново сформированным состоянием.

В следующем языке должна существовать поддержка параллельной обработки чистых функций. Процессоры не становятся быстрее, но их становится больше. В конечном счете наибольший прирост производительности достигается за счет параллелизма.

Neo — это не следующий язык, но он поможет нам не бояться следующей парадигмы. Далее мы будем создавать Neo.

Глава 26

Как работает разбиение на лексемы

●●○●○

> Никогда не посылайте человека делать
> работу машины.
>
> *Агент Смит*

Первый шаг в обработке программы — ее разбиение на лексемы. Лексема состоит из последовательности символов, формирующей значащую характерную особенность исходного кода, такую как имя, знак пунктуации, числовой или текстовый литерал. Объект лексемы создается для каждой имеющейся в программе лексемы.

Для разбиения текста исходного кода на строки, а строк — на лексемы используются регулярные выражения.

```
const rx_unicode_escapement = /
    \\ u \{ ( [ 0-9 A-F ]{4,6} ) \}
/g;
```

`rx_crfl` соответствует переводу строки, возврату каретки, а также возврату каретки с переводом строки. Мы до сих пор работаем с кодами устройств для электромеханических телетайпов середины XX века.

```
const rx_crlf = /
    \n
|
    \r \n?
/;
```

`rx_token` соответствует лексеме Neo — комментарию, имени, числу, строке, знаку пунктуации.

```
const rx_token = /
    ( \u0020+ )
|
    ( # .* )
|
    (
        [ a-z A-Z ]
```

```
        (?:
            \u0020 [ a-z A-Z ]
        |
            [ 0-9 a-z A-Z ]
        )*
        \??
    )
|
    (
        -? \d+
        (?: \. \d+ )?
        (?: e \-? \d+ )?
    )
|
    (
        "
        (?:
            [^ " \\ ]
        |
            \\
            (?:
                [ n r " \\ ]
            |
                u \{ [ 0-9 A-F ]{4,6} \}
            )
        )*
        "
    )
|
    (
        \. (?: \. \.)?
    |
        \/ \\?
    |
        \\ \/?
    |
        > >?
    |
        < <?
    |
        \[ \]?
    |
        \{ \}?
    |
        [ ( ) } \] . , : ? ! ; ~ ≈ = ≠ ≤ ≥ & | + \- * % ƒ $ @ \^ _ ' ` ]
    )
/y;

// Определение групп
//     [1] Пробел
//     [2] Комментарий
//     [3] Символ
//     [4] Число
//     [5] Строка
//     [6] Знак препинания
```

Фабрика разбиения на лексемы выставляется на экспорт.

```
export default Object.freeze(function tokenize(source, comment = false) {
```

`tokenize` принимает исходный код и создает из него массив объектов-лексем. Если исходный код `source` не является массивом, он разбивается на строки по признаку присутствия символов возврата каретки и перевода строки. Если `comment` имеет значение `true`, то комментарий включается в качестве объекта-лексемы. Парсеру комментарии не нужны, а вот инструментам для работы с программными средствами они могут понадобиться.

```
    const lines = (
        Array.isArray(source)
        ? source
        : source.split(rx_crlf)
    );
    let line_nr = 0;
    let line = lines[0];
    rx_token.lastIndex = 0;
```

Фабрика возвращает генератор, который разбивает строки на объекты-лексемы. Эти объекты состоят из идентификатора, координат и другой информации. Пробельные символы на лексемы не разбиваются.

При каждом вызове генератор лексем производит очередную лексему.

```
    return function token_generator() {
        if (line === undefined) {
            return;
        }
        let column_nr = rx_token.lastIndex;
        if (column_nr >= line.length) {
            rx_token.lastIndex = 0;
            line_nr += 1;
            line = lines[line_nr];
            return (
                line === undefined
                ? undefined
                : token_generator()
            );
        }
        let captives = rx_token.exec(line);
```

Соответствие не найдено.

```
        if (!captives) {
            return {
                id: "(error)",
                line_nr,
                column_nr,
                string: line.slice(column_nr)
        };
    }
```

Соответствует пробельный символ:

```
if (captives[1]) {
    return token_generator();
}
```

Соответствует комментарий:

```
if (captives[2]) {
    return (
        comment
        ? {
            id: "(comment)",
            comment: captives[2],
            line_nr,
            column_nr,
            column_to: rx_token.lastIndex
        }
        : token_generator()
    );
```

Соответствует имя:

```
if (captives[3]) {
    return {
        id: captives[3],
        alphameric: true,
        line_nr,
        column_nr,
        column_to: rx_token.lastIndex
    };
}
```

Соответствует числовой литерал:

```
if (captives[4]) {
    return {
        id: "(number)",
        readonly: true,
        number: big_float.normalize(big_float.make(captives[4])),
        text: captives[4],
        line_nr,
        column_nr,
        column_to: rx_token.lastIndex
    };
}
```

Соответствует текстовый литерал:

```
if (captives[5]) {
```

Метод `.replace` используется для преобразования `\u{xxxxxx}` в кодовую точку, а `JSON.parse` — для обработки оставшихся эскейп-символов и кавычек:

```
        return {
            id: "(text)",
            readonly: true,
```

```
            text: JSON.parse(captives[5].replace(
                rx_unicode_escapement,
                function (ignore, code) {
                    return String.fromCodePoint(parseInt(code, 16));
                }
            )),
            line_nr,
            column_nr,
            column_to: rx_token.lastIndex
        };
    }
```

Соответствует знаку пунктуации:

```
        if (captives[6]) {
            return {
                id: captives[6],
                line_nr,
                column_nr,
                column_to: rx_token.lastIndex
            };
        }
    };
});
```

Глава 27

Как работает парсер

> Некоторые полагали, что нам не хватает языка программирования для описания вашего идеального мира. Но я верю, что как вид люди определяют свою реальность через муки и страдания.
>
> *Агент Смит*

При парсинге поток объектов-лексем сплетается в дерево. В ходе парсинга выполняется также поиск ошибок в источнике. Объекты-лексемы дополняются новыми свойствами. Наиболее важные свойства — `zeroth`, `wunth` и `twoth`. Они задают древовидную структуру. Например, два операнда сложения хранятся в свойствах `zeroth` и `wunth` лексемы `+`. Лексема `if` хранит выражение условия в `zeroth`, элемент `then` хранится в `wunth`, а элемент `else` — в `twoth`. Будут встречаться и другие свойства. Рассмотрим их в ходе изучения главы.

Отчет об ошибках создается с помощью функции `error`. Можно было бы сохранить список ошибок и продолжить, но я хочу остановиться после первой ошибки. Находясь в режиме разработки, я хочу, чтобы ошибка переместила курсор редактора к следующему проблемному месту. А в режиме сборки я не ищу список, а смотрю, прошла сборка или нет.

```
let the_error;

function error(zeroth, wunth) {
    the_error = {
        id: "(error)",
        zeroth,
        wunth
    };
    throw "fail";
}
```

В объекте `primordial` (фундаментальный объект) содержатся объекты, встроенные в язык. В их числе такие константы, как `true`, и такие функции, как `neg`. Объект `primordial` создан с помощью `Object.create(null)`, потому что я не хочу засорять объект цепочкой прототипов. К примеру, `Object.prototype` содержит метод

valueOf, но я не хочу его наследовать, поскольку может создаться видимость добавления к языку исконного метода valueOf.

```
const primordial = (function (ids) {
    const result = Object.create(null);
    ids.forEach(function (id) {
        result[id] = Object.freeze({
            id,
            alphameric: true,
            readonly: true
        });
    });
    return Object.freeze(result);
}([
    "abs", "array", "array?", "bit and", "bit mask", "bit or", "bit shift own",
    "bit shift up", "bit xor", "boolean?", "char", "code", "false", "fraction",
    "function?", "integer", "integer?", "length", "neg", "not", "number",
    "number?", "null", "record", "record?", "stone", "stone?", "text", "text?",
    "true"
]));
```

Свойство readonly блокирует инструкцию let.

По мере продвижения по потоку всегда будут видны три лексемы.

Поток объектов-лексем предоставляется функцией-генератором. Три лексемы видны как prev_token, token и next_token. Функция advance использует генератор для циклического прохождения через все объекты-лексемы, пропуская комментарии.

```
let the_token_generator;
let prev_token;
let token;
let next_token;

let now_function;     // Текущая обрабатываемая функция.
let loop;             // Массив состояния выхода из цикла.

const the_end = Object.freeze({
    id: "(end)",
    precedence: 0,
    column_nr: 0,
    column_to: 0,
    line_nr: 0
});
```

Функция advance выполняет переход к следующей лексеме. Ее партнерская функция prelude пытается разделить текущую лексему на две лексемы:

```
function advance(id) {
```

Переход к следующей лексеме с помощью генератора лексем. Если предоставлен идентификатор id, нужно убедиться, что текущая лексема соответствует ему:

```
    if (id !== undefined && id !== token.id) {
        return error(token, "expected '" + id + "'");
```

```
        }
        prev_token = token;
        token = next_token;
        next_token = the_token_generator() || the_end;
    }

    function prelude() {
```

Если лексема `token` содержит пробел, она разбивается и ее правая часть помещается в `prev_token`. В противном случае выполняется переход к следующей лексеме.

```
        if (token.alphameric) {
            let space_at = token.id.indexOf(" ");
            if (space_at > 0) {
                prev_token = {
                    id: token.id.slice(0, space_at),
                    alphameric: true,
                    line_nr: token.line_nr,
                    column_nr: token.column_nr,
                    column_to: token.column_nr + space_at
                };
                token.id = token.id.slice(space_at + 1);
                token.column_nr = token.column_nr + space_at + 1;
                return;
            }
        }
        return advance();
    }
```

Пробельные символы играют в этом языке весьма важную роль. Перевод строки может сигнализировать о конце инструкции или элемента. Отступ может означать конец условия. Справиться с этим помогают следующие функции:

```
    let indentation;

    function indent() {
        indentation += 4;
    }

    function outdent() {
        indentation -= 4;
    }

    function at_indentation() {
        if (token.column_nr !== indentation) {
            return error(token, "expected at " + indentation);
        }
    }

    function is_line_break() {
        return token.line_nr !== prev_token.line_nr;
    }

    function same_line() {
        if (is_line_break()) {
```

```
            return error(token, "unexpected linebreak");
        }
    }

    function line_check(open) {
        return (
            open
            ? at_indentation()
            : same_line()
        );
    }
```

Функция `register` объявляет в области видимости функции новую переменную. Функция `lookup` находит переменную в наиболее подходящей области видимости:

```
    function register(the_token, readonly = false) {
```

Добавление переменной к текущей области видимости:

```
        if (now_function.scope[the_token.id] !== undefined) {
            error(the_token, "already defined");
        }
        the_token.readonly = readonly;
        the_token.origin = now_function;
        now_function.scope[the_token.id] = the_token;
    }

    function lookup(id) {
```

Поиск определения в текущей области видимости:

```
        let definition = now_function.scope[id];
```

Если происходит сбой, поиск в области видимости прародителей:

```
        if (definition === undefined) {
            let parent = now_function.parent;
            while (parent !== undefined) {
                definition = parent.scope[id];
                if (definition !== undefined) {
                    break;
                }
                parent = parent.parent;
            }
```

Если и здесь происходит сбой, поиск в фундаментальном объекте:

```
            if (definition === undefined) {
                definition = primordial[id];
            }
```

Следует помнить, что это определение используется текущей функцией:

```
            if (definition !== undefined) {
                now_function.scope[id] = definition;
            }
        }
        return definition;
    }
```

В свойстве `origin` фиксируется функция, создавшая переменную. В свойстве `scope` хранятся все переменные, созданные или используемые в функции. Свойство `parent` указывает на функцию, создавшую данную функцию.

Функции, применяющиеся для парсинга особенностей языка, содержатся в трех объектах: `statements`, `prefixes` и `suffixes`.

В объектах `parse_statement`, `parse_prefix` и `parse_suffix` содержатся функции, которые выполняют специализированный парсинг. Для их создания используется метод `Object.create(null)`, поскольку нам не хочется, чтобы здесь вычищался какой-либо мусор, полученный из `Object.prototype`:

```
const parse_statement = Object.create(null);
const parse_prefix = Object.create(null);
const parse_suffix = Object.create(null);
```

Ядро этого парсера — функция `expression` (и ее помощник `argument_expression`). Выражение может рассматриваться как имеющее две части: левую и необязательную правую. Левая часть — это литерал, переменная или префикс, правая — оператор-суффикс, за которым могут следовать другие выражения. Если в правой части есть суффикс и у него более высокий приоритет, тогда левая часть передается парсеру правой части, в результате чего появляется новая левая часть. Вероятно, парсер правой части сам снова вызовет выражение, возможно, с другим приоритетом.

Выражения могут быть открытыми или закрытыми. Закрытое выражение должно полностью помещаться на одной строке. Открытые выражения должны начинаться с требуемого отступа и могут перед суффиксом иметь перевод строки:

```
function argument_expression(precedence = 0, open = false) {
```

Основу этого парсера составляет функция `expression`. В ней используется прием, называемый нисходящим приоритетом операторов (Top Down Operator Precedence).

Функция принимает необязательный параметр `open`, допускающий лояльное отношение к конкретным переводам строки. Если `open` имеет значение `true`, ожидается, что лексема будет находиться в точке отступа.

```
    let definition;
    let left;
    let the_token = token;
```

Чем является лексема, числовым или текстовым литералом?

```
    if (the_token.id === "(number)" || the_token.id === "(text)") {
        advance();
        left = the_token;
```

Является ли лексема буквенно-цифровой?

```
    } else if (the_token.alphameric === true) {
        definition = lookup(the_token.id);
```

```
            if (definition === undefined) {
                return error(the_token, "expected a variable");
            }
            left = definition;
            advance();
        } else {
```

Лексема может быть префиксом: (, [, {, *f*.

```
            definition = parse_prefix[the_token.id];
            if (definition === undefined) {
                return error(the_token, "expected a variable");
            }
            advance();
            left = definition.parser(the_token);
        }
```

У нас есть левая часть. Есть ли в правой части суффиксный оператор? Позволяет ли уровень приоритета использовать его? Если да, объединяем левую и правую части, чтобы сформировать новую левую часть.

```
        while (true) {
            the_token = token;
            definition = parse_suffix[the_token.id];
            if (
                token.column_nr < indentation
                || (!open && is_line_break())
                || definition === undefined
                || definition.precedence <= precedence
            ) {
                break;
            }
            line_check(open && is_line_break());
            advance();
            the_token.class = "suffix";
            left = definition.parser(left, the_token);
        }
```

После прохождения цикла ноль и более раз появляется возможность возвратить дерево парсинга выражения.

```
        return left;
    }

    function expression(precedence, open = false) {
```

Выражения проверяются на наличие пробельного символа, а для выражений аргументов такая проверка не нужна.

```
        line_check(open);
            return argument_expression(precedence, open);
    }
```

Свойство `precedence` определяет порядок парсинга суффиксного оператора. Свойство `parser` является функцией для парсинга префикса или суффикса. Свойство `class` может иметь значения `"suffix"`, `"statement"` или `undefined`.

Рассмотрим простой суффиксный оператор. Парсеру точки (`.`) передаются выражение слева и точка (`.`). Он проверяет допустимость выражения, расположенного слева. Также проверяет, является ли именем текущая лексема. Затем он собирает все в лексему `.` (точка) и возвращает ее.

```
function parse_dot(left, the_dot) {
```

Выражение слева должно быть переменной или выражением, способным возвратить объект (исключая литералы объектов).

```
    if (
        !left.alphameric
        && left.id !== "."
        && (left.id !== "[" || left.wunth === undefined)
        && left.id !== "("
    ) {
        return error(token, "expected a variable");
    }
    let the_name = token;
    if (the_name.alphameric !== true) {
        return error(the_name, "expected a field name");
    }
    the_dot.zeroth = left;
    the_dot.wunth = the_name;
    same_line();
    advance();
    return the_dot;
}
```

Парсер индекса (`[]`) немного интереснее. Ему передаются выражение слева и лексема левой квадратной скобки (`[`). Он проверяет допустимость выражения, расположенного слева. Затем вызывается выражение для получения содержимого в квадратных скобках. Если после левой квадратной скобки (`[`) был перевод строки, значит, это открытое выражение. Парсер проходит мимо закрывающей квадратной скобки (`]`).

```
function parse_subscript(left, the_bracket) {
    if (
        !left.alphameric
        && left.id !== "."
        && (left.id !== "[" || left.wunth === undefined)
        && left.id !== "("
    ) {
        return error(token, "expected a variable");
    }
    the_bracket.zeroth = left;
    if (is_line_break()) {
        indent();
        the_bracket.wunth = expression(0, true);
        outdent();
        at_indentation();
    } else {
        the_bracket.wunth = expression();
```

```
        same_line();
    }
    advance("]");
    return the_bracket;
}
```

Парсер многоточия `ellipsis` укомплектован не так, как другие парсеры суффиксных операторов, поскольку присутствие данного суффикса разрешено только в трех местах: в списках параметров, списках аргументов и литералах массивов. В других местах он запрещен, поэтому рассматривается как особый случай.

```
function ellipsis(left) {
    if (token.id === "…") {
        const the_ellipsis = token;
        same_line();
        advance("…");
        the_ellipsis.zeroth = left;
        return the_ellipsis;
    }
    return left;
}
```

Парсер вызова `()` проводит синтаксический разбор вызовов функций. Он вызывает для каждого аргумента `argument_expression`. В вызове в открытой форме аргументы перечисляются вертикально, без применения запятых.

```
function parse_invocation(left, the_paren) {

// вызов функции:
//    выражение
//    выражение…

    const args = [];
    if (token.id === ")") {
        same_line();
    } else {
        const open = is_line_break();
        if (open) {
            indent();
        }
        while (true) {
            line_check(open);
            args.push(ellipsis(argument_expression()));
            if (token.id === ")" || token === the_end) {
                break;
            }
            if (!open) {
                same_line();
                advance(",");
            }
        }
        if (open) {
            outdent();
            at_indentation();
```

```
        } else {
            same_line();
        }
    }
    advance(")");
    the_paren.zeroth = left;
    the_paren.wunth = args;
    return the_paren;
}
```

Функция `suffix` создает массив `parse_suffix`. Она принимает оператор и уровень приоритета, а также необязательный парсер. Он может предоставить функцию парсера, используемую по умолчанию и работающую для большинства операторов.

```
function suffix(
    id,
    precedence,
    optional_parser = function infix(left, the_token) {
        the_token.zeroth = left;
        the_token.wunth = expression(precedence);
        return the_token;
    }
) {
```

Создание инфиксного или суффиксного оператора:

```
    const the_symbol = Object.create(null);
    the_symbol.id = id;
    the_symbol.precedence = precedence;
    the_symbol.parser = optional_parser;
    parse_suffix[id] = Object.freeze(the_symbol);
}

suffix("|", 111, function parse_default(left, the_bar) {
    the_bar.zeroth = left;
    the_bar.wunth = expression(112);
    advance("|");
    return the_bar;
});

suffix("?", 111, function then_else(left, the_then) {
    the_then.zeroth = left;
    the_then.wunth = expression();
    advance("!");
    the_then.twoth = expression();
    return the_then;
});
suffix("/\\", 222);
suffix("\\/", 222);
suffix("~", 444);
suffix("≈", 444);
suffix("+", 555);
suffix("-", 555);
suffix("<<", 555);
suffix(">>", 555);
suffix("*", 666);
```

```
suffix("/", 666);
suffix(".", 777, parse_dot);
suffix("[", 777, parse_subscript);
suffix("(", 777, parse_invocation);
```

Чтобы защититься от ошибок $a < b \le c$, отношение к реляционным операторам выстраивается немного иначе.

```
const rel_op = Object.create(null);

function relational(operator) {
    rel_op[operator] = true;
    return suffix(operator, 333, function (left, the_token) {
        the_token.zeroth = left;
        the_token.wunth = expression(333);
        if (rel_op[token.id] === true) {
            return error(token, "unexpected relational operator");
        }
        return the_token;
    });
}

relational("=");
relational("≠");
relational("<");
relational(">");
relational("≤");
relational("≥");
```

Функция `prefix` создает массив `parse_prefix`. Следует обратить внимание на то, что левая круглая скобка (`(`) и левая квадратная скобка (`[`) также находятся в массиве `sparse_suffix`. В этом нет никакой проблемы. Какая-либо двусмысленность напрочь отсутствует. Префиксные операторы не нуждаются в приоритете.

```
function prefix(id, parser) {
    const the_symbol = Object.create(null);
    the_symbol.id = id;
    the_symbol.parser = parser;
    parse_prefix[id] = Object.freeze(the_symbol);
}

prefix("(", function (ignore) {
    let result;
    if (is_line_break()) {
        indent();
        result = expression(0, true);
        outdent();
        at_indentation();
    } else {
        result = expression(0);
        same_line();
    }
    advance(")");
    return result;
});
```

Парсер литерала массива вызывает для каждого элемента функцию `expression`. Элемент — это любое выражение, за которым может стоять многоточие (...). Существует три способа записи литерала массива:

- пустой — `[]`, массив нулевой длины;
- закрытый — весь литерал в одну строку, в качестве разделителей — запятые;
- открытый — после левой квадратной скобки (`[`) перевод строки, углубление отступа, правая квадратная скобка (`]`), которая восстанавливает прежний отступ, выражение с разделителями в виде запятой (`,`) или точки с запятой (`;`) и/или перевод строки.

С помощью точки с запятой (`;`) создаются двумерные массивы.

```
[[2, 7, 6], [9, 5, 1], [4, 3, 8]]
```

может быть записан в виде:

```
[2, 7, 6; 9, 5, 1; 4, 3, 8]
```

```
prefix("[", function arrayliteral(the_bracket) {
    let matrix = [];
    let array = [];
    if (!is_line_break()) {
        while (true) {
            array.push(ellipsis(expression()));
            if (token.id === ",") {
                same_line();
                advance(",");
            } else if (
                token.id === ";"
                && array.length > 0
                && next_token !== "]"
            ) {
                same_line();
                advance(";");
                matrix.push(array);
                array = [];
            } else {
                break;
            }
        }
        same_line();
    } else {
        indent();
        while (true) {
            array.push(ellipsis(expression(0, is_line_break())));
            if (token.id === "]" || token === the_end) {
                break;
            }
            if (token.id === ";") {
                if (array.length === 0 || next_token.id === "]") {
                    break;
                }
```

```
                same_line();
                advance(";");
                matrix.push(array);
                array = [];
            } else if (token.id === "," || !is_line_break()) {
                same_line();
                advance(",");
            }
        }
        outdent();
        if (token.column_nr !== indentation) {
            return error(token, "expected at " + indentation);
        }
    }
    advance("]");
    if (matrix.length > 0) {
        matrix.push(array);
        the_bracket.zeroth = matrix;
    } else {
        the_bracket.zeroth = array;
    }
    return the_bracket;
});

prefix("[]", function emptyarrayliteral(the_brackets) {
    return the_brackets;
});
```

Парсер литерала записи распознает четыре формы полей:

- *переменная*;
- *имя*: *выражение*;
- "*строка*": *выражение*;
- [*выражение*]: *выражение*.

```
prefix("{", function recordliteral(the_brace) {
    const properties = [];
    let key;
    let value;
    const open = the_brace.line_nr !== token.line_nr;
    if (open) {
        indent();
    }
    while (true) {
        line_check(open);
        if (token.id === "[") {
            advance("[");
            key = expression();
            advance("]");
            same_line();
            advance(":");
            value = expression();
        } else {
            key = token;
```

```
            advance();
            if (key.alphameric === true) {
                if (token.id === ":") {
                    same_line();
                    advance(":");
                    value = expression();
                } else {
                    value = lookup(key.id);
                    if (value === undefined) {
                        return error(key, "expected a variable");
                    }
                }
                key = key.id;
            } else if (key.id === "(text)") {
                key = key.text;
                same_line();
                advance(":");
                value = expression();
            } else {
                return error(key, "expected a key");
            }
        }
        properties.push({
            zeroth: key,
            wunth: value
        });
        if (token.column_nr < indentation || token.id === "}") {
            break;
        }
        if (!open) {
            same_line();
            advance(",");
        }
    }
    if (open) {
        outdent();
        at_indentation();
    } else {
        same_line();
    }
    advance("}");
    the_brace.zeroth = properties;
    return the_brace;
});

prefix("{}", function emptyrecordliteral(the_braces) {
    return the_braces;
});
```

Парсер функционального литерала создает новые функции. Он также предоставляет доступ к *указателям на функции* (functinos). Это название появилось у меня в результате опечатки.

```
const functino = (function make_set(array, value = true) {
    const object = Object.create(null);
```

```
    array.forEach(function (element) {
        object[element] = value;
    });
    return Object.freeze(object);
}([
    "?", "|", "/\\", "\\/", "=", "≠", "<", "≥", ">", "≤",
    "~", "≈", "+", "-", ">>", "<<", "*", "/", "[", "("
]));

prefix("ƒ", function function_literal(the_function) {
```

Если за символом `ƒ` следует суффиксный оператор, создается соответствующий указатель на функцию.

```
    const the_operator = token;
    if (
        functino[token.id] === true
        && (the_operator.id !== "(" || next_token.id === ")")
    ) {
        advance();
        if (the_operator.id === "(") {
            same_line();
            advance(")");
        } else if (the_operator.id === "[") {
            same_line();
            advance("]");
        } else if (the_operator.id === "?") {
            same_line();
            advance("!");
        } else if (the_operator.id === "|") {
            same_line();
            advance("|");
        }
        the_function.zeroth = the_operator.id;
        return the_function;
    }
```

Настройка новой функции:

```
    if (loop.length > 0) {
        return error(the_function, "Do not make functions in loops.");
    }
    the_function.scope = Object.create(null);
    the_function.parent = now_function;
    now_function = the_function;

// Параметры функции поступают в трех формах.
//        имя
//        имя | значение_по_умолчанию |
//        имя…
```

Список параметров может быть открытым или закрытым:

```
    const parameters = [];
    if (token.alphameric === true) {
        let open = is_line_break();
```

```
        if (open) {
            indent();
        }
        while (true) {
            line_check(open);
            let the_parameter = token;
            register(the_parameter);
            advance();
            if (token.id === "…") {
                parameters.push(ellipsis(the_parameter));
                break;
            }
            if (token.id === "|") {
                advance("|");
                parameters.push(parse_suffix["|"](the_parameter, prev_token));
            } else {
                parameters.push(the_parameter);
            }
            if (open) {
                if (token.id === ",") {
                    return error(token, "unexpected ','");
                }
                if (token.alphameric !== true) {
                    break;
                }
            } else {
                if (token.id !== ",") {
                    break;
                }
                same_line();
                advance(",");
                if (token.alphameric !== true) {
                    return error(token, "expected another parameter");
                }
            }
        }
        if (open) {
            outdent();
            at_indentation();
        } else {
            same_line();
        }
    }
    the_function.zeroth = parameters;
```

У функции может быть (*возвращаемое_выражение*) или {*тело_функции*}.

Парсинг возвращаемого выражения:

```
    if (token.id === "(") {
        advance("(");
        if (is_line_break()) {
            indent();
            the_function.wunth = expression(0, true);
            outdent();
```

```
            at_indentation();
        } else {
            the_function.wunth = expression();
            same_line();
        }
        advance(")");
    } else {
```

Парсинг тела функции — тело должно содержать явно указанную инструкцию возврата `return`. Подразумеваемый возврат по достижении последней инструкции в теле не применяется.

```
        advance("{");
        indent();
        the_function.wunth = statements();
        if (the_function.wunth.return !== true) {
            return error(prev_token, "missing explicit 'return'");
        }
```

Парсинг обработчика сбоев:

```
        if (token.id === "failure") {
            outdent();
            at_indentation();
            advance("failure");
            indent();
            the_function.twoth = statements();
            if (the_function.twoth.return !== true) {
                return error(prev_token, "missing explicit 'return'");
            }
        }
        outdent();
        at_indentation();
        advance("}");
    }
    now_function = the_function.parent;
    return the_function;
});
```

Функция `statements` выполняет парсинг инструкций, возвращая массив лексем этих инструкций. По мере надобности, чтобы отделить команду от лексемы, в ней используется функция `prelude`:

```
function statements() {
    const statement_list = [];
    let the_statement;
    while (true) {
        if (
            token === the_end
            || token.column_nr < indentation
            || token.alphameric !== true
            || token.id.startsWith("export")
        ) {
            break;
        }
```

```
            at_indentation();
            prelude();
            let parser = parse_statement[prev_token.id];
            if (parser === undefined) {
                return error(prev_token, "expected a statement");
            }
            prev_token.class = "statement";
            the_statement = parser(prev_token);
            statement_list.push(the_statement);
            if (the_statement.disrupt === true) {
                if (token.column_nr === indentation) {
                    return error(token, "unreachable");
                }
                break;
            }
        }
        if (statement_list.length === 0) {
            if (!token.id.startsWith("export")) {
                return error(token, "expected a statement");
            }
        } else {
            statement_list.disrupt = the_statement.disrupt;
            statement_list.return = the_statement.return;
        }
        return statement_list;
    }
```

Свойство `disrupt` помечает инструкции или списки инструкций, вызывающие отмену или возвращение. Свойство `return` помечает инструкции или списки инструкций, вызывающие возвращение.

Инструкция `break` вызывает отмену цикла. Парсеру передается лексема `break`. Она устанавливает условие выхода из текущего цикла:

```
    parse_statement.break = function (the_break) {
        if (loop.length === 0) {
            return error(the_break, "'break' wants to be in a loop.");
        }
        loop[loop.length - 1] = "break";
        the_break.disrupt = true;
        return the_break;
    };
```

Инструкция `call` вызывает функцию и игнорирует возвращаемое значение. Она используется для идентификации вызовов, происходящих исключительно из-за их побочных эффектов:

```
    parse_statement.call = function (the_call) {
        the_call.zeroth = expression();
        if (the_call.zeroth.id !== "(") {
            return error(the_call, "expected a function invocation");
        }
        return the_call;
    };
```

Инструкция `def` регистрирует переменные, предназначенные только для чтения:

```
parse_statement.def = function (the_def) {
    if (!token.alphameric) {
        return error(token, "expected a name.");
    }
    same_line();
    the_def.zeroth = token;
    register(token, true);
    advance();
    same_line();
    advance(":");
    the_def.wunth = expression();
    return the_def;
};
```

Инструкция `fail` — самый рискованный эксперимент в Neo. Механизмы исключений в большинстве языков были сведены к простым каналам связи. Инструкция `fail` пытается исправить положение:

```
parse_statement.fail = function (the_fail) {
    the_fail.disrupt = true;
    return the_fail;
};
```

У инструкции `if` есть необязательное продолжение в виде инструкций `else` или `else if`. Если прерывание или возвращение есть в обеих ветвях, то прерывание или возвращение происходят и в самой инструкции `if`:

```
parse_statement.if = function if_statement(the_if) {
    the_if.zeroth = expression();
    indent();
    the_if.wunth = statements();
    outdent();
    if (token.column_nr === indentation) {
        if (token.id === "else") {
            advance("else");
            indent();
            the_if.twoth = statements();
            outdent();
            the_if.disrupt = the_if.wunth.disrupt && the_if.twoth.disrupt;
            the_if.return = the_if.wunth.return && the_if.twoth.return;
        } else if (token.id.startsWith("else if ")) {
            prelude();
            prelude();
            the_if.twoth = if_statement(prev_token);
            the_if.disrupt = the_if.wunth.disrupt && the_if.twoth.disrupt;
            the_if.return = the_if.wunth.return && the_if.twoth.return;
        }
    }
    return the_if;
};
```

Инструкция `let` является в Neo инструкцией присваивания значений. Левая сторона — это не обычное выражение. На нее накладываются более серьезные

ограничения. Она называется `lvalue` и может быть переменной (`var`, а не `def`) или выражением, которое находит поле или элемент:

```
parse_statement.let = function (the_let) {
```

Инструкция `let` — единственное место, где разрешено вносить изменения.

Следующей лексемой должно быть имя:

```
same_line();
const name = token;
advance();
const id = name.id;
let left = lookup(id);
if (left === undefined) {
    return error(name, "expected a variable");
}
let readonly = left.readonly;
```

Теперь рассмотрим суффиксные инструкции `[]`, `.`, `[` и `{`.

```
while (true) {
    if (token === the_end) {
        break;
    }
    same_line();
```

В данной позиции символы `[]` указывают на операцию добавления элемента в массив:

```
        if (token.id === "[]") {
            readonly = false;
            token.zeroth = left;
            left = token;
            same_line();
            advance("[]");
            break;
        }
        if (token.id === ".") {
            readonly = false;
            advance(".");
            left = parse_dot(left, prev_token);
        } else if (token.id === "[") {
            readonly = false;
            advance("[");
            left = parse_subscript(left, prev_token);
        } else if (token.id === "(") {
            readonly = false;
            advance("(");
            left = parse_invocation(left, prev_token);
            if (token.id === ":") {
                return error(left, "assignment to the result of a function");
            }
        } else {
            break;
        }
    }
```

```
        advance(":");
        if (readonly) {
            return error(left, "assignment to a constant");
        }
        the_let.zeroth = left;
        the_let.wunth = expression();
```

В данной позиции символы `[]` указывают на операцию извлечения элемента из массива:

```
        if (token.id === "[]" && left.id !== "[]" && (
            the_let.wunth.alphameric === true
            || the_let.wunth.id === "."
            || the_let.wunth.id === "["
            || the_let.wunth.id === "("
        )) {
            token.zeroth = the_let.wunth;
            the_let.wunth = token;
            same_line();
            advance("[]");
        }
        return the_let;
    };
```

Инструкция `loop` сохраняет стек для работы с вложенными циклами. Записи в стеке являются условиями выхода из циклов. Если явного выхода нет, статус стека имеет значение `"infinite"`. Если единственный выход — это инструкция `return`, его статус — `"return"`. Если выход из цикла выполняется с помощью инструкции `break`, его статусом становится `"break"`. В данном случае `fail` не является явным условием выхода.

```
    parse_statement.loop = function (the_loop) {
        indent();
        loop.push("infinite");
        the_loop.zeroth = statements();
        const exit = loop.pop();
        if (exit === "infinite") {
            return error(the_loop, "A loop wants a 'break'.");
        }
        if (exit === "return") {
            the_loop.disrupt = true;
            the_loop.return = true;
        }
        outdent();
        return the_loop;
    };
```

Инструкция `return` изменяет статус циклов с `"infinite"` на `"return"`:

```
    parse_statement.return = function (the_return) {
        try {
            if (now_function.parent === undefined) {
                return error(the_return, "'return' wants to be in a function.");
            }
            loop.forEach(function (element, element_nr) {
                if (element === "infinite") {
```

```
                loop[element_nr] = "return";
            }
        });
        if (is_line_break()) {
            return error(the_return, "'return' wants a return value.");
        }
        the_return.zeroth = expression();
        if (token === "}") {
            return error(the_return, "Misplaced 'return'.");
        }
        the_return.disrupt = true;
        the_return.return = true;
        return the_return;
    } catch (ignore) {
        return the_error;
    }
};
```

Инструкция `var` объявляет переменную, значение которой может быть присвоено с помощью инструкции `let`. Если переменная не имеет явной инициализации, ее исходным значением становится `null`:

```
parse_statement.var = function (the_var) {
    if (!token.alphameric) {
        return error(token, "expected a name.");
    }
    same_line();
    the_var.zeroth = token;
    register(token);
    advance();
    if (token.id === ":") {
        same_line();
        advance(":");
    the_var.wunth = expression();
    }
    return the_var;
};
Object.freeze(parse_prefix);
Object.freeze(parse_suffix);
Object.freeze(parse_statement);
```

Инструкции `import` и `export` в `parse_statement` не включены, поскольку их местонахождение в исходном коде строго ограничено. Все инструкции `import` должны быть помещены до любых других инструкций. Разрешена только одна инструкция `export`, которая должна стоять самой последней.

```
function parse_import(the_import) {
    same_line();
    register(token, true);
    the_import.zeroth = token;
    advance();
    same_line();
    advance(":");
    same_line();
    the_import.wunth = token;
    advance("(text)");
```

```
    the_import.class = "statement";
    return the_import;
}

function parse_export(the_export) {
    the_export.zeroth = expression();
    the_export.class = "statement";
    return the_export;
}
```

Экспортируется одна функция `parse`. Она принимает генератор лексем и возвращает дерево. Нам не нужно создавать конструктор, поскольку `parse` никакого состояния между вызовами не сохраняет.

```
export default function parse(token_generator) {
    try {
        indentation = 0;
        loop = [];
        the_token_generator = token_generator;
        next_token = the_end;
        const program = {
            id: "",
            scope: Object.create(null)
        };
        now_function = program;
        advance();
        advance();
        let the_statements = [];
        while (token.id.startsWith("import ")) {
            at_indentation();
            prelude();
            the_statements.push(parse_import(prev_token));
        }
        the_statements = the_statements.concat(statements());
        if (token.id.startsWith("export")) {
            at_indentation();
            prelude();
            the_statements.push(parse_export(prev_token));
        }
        if (token !== the_end) {
            return error(token, "unexpected");
        }
        program.zeroth = the_statements;
        return program;
    } catch (ignore) {
        return the_error;
    }
};
```

Глава 28

Как работает генерация кода

●●●○○

> Отвлекись от всего, Нео. Страх, сомнения и неверие отбрось. Освободи свой разум.
>
> *Морфеус*

Следующий шаг заключается в том, чтобы взять дерево, сплетенное парсером, и превратить его в исполняемую форму. Это называется генерацией кода, но на самом деле это преобразование.

У нас богатый выбор целевых языков. Можно создать машинный код или код для виртуальной машины либо другого языка программирования. Имея подходящую среду выполнения, мы могли бы нацелиться на язык C, но собираемся ориентироваться на JavaScript.

JavaScript — на удивление удачный целевой язык. Он обеспечивает замечательный (и невидимый) механизм управления памятью, который зачастую является наиболее сложным аспектом доводки нового языка. Генерация кода для реальной машины может потребовать управления ограниченным набором регистров. JavaScript позволяет иметь какие угодно переменные. Объекты — это универсальная структура данных. За объекты JavaScript достоин гораздо более существенных похвал.

Начнем с создания некоторых наборов. Нам нужны наборы, производящие булевы значения и зарезервированные слова JavaScript.

```
function make_set(array, value = true) {
    const object = Object.create(null);
    array.forEach(function (element) {
        object[element] = value;
    });
    return $NEO.stone(object);
}

const boolean_operator = make_set([
    "array?", "boolean?", "function?", "integer?", "not", "number?", "record?",
    "stone?", "text?", "true", "=", "≠", "<", ">", "≤", "≥", "/\\", "\\/"
]);

const reserved = make_set([
```

```
    "arguments", "await", "break", "case", "catch", "class", "const",
    "continue", "debugger", "default", "delete", "do", "else", "enum", "eval",
    "export", "extends", "false", "finally", "for", "function", "if",
    "implements", "import", "in", "Infinity", "instanceof", "interface", "let",
    "NaN", "new", "null", "package", "private", "protected", "public", "return",
    "static", "super", "switch", "this", "throw", "true", "try", "typeof",
    "undefined", "var", "void", "while", "with", "yield"
]);
```

Объект `primordial` содержит всевозможные отображения из фундаментального пространства Neo в пространство JavaScript. Некоторые отображения направляются к среде выполнения Neo, а некоторые — к JavaScript.

```
const primordial = $NEO.stone({
    "abs": "$NEO.abs",
    "array": "$NEO.array",
    "array?": "Array.isArray",
    "bit and": "$NEO.bitand",
    "bit mask": "$NEO.bitmask",
    "bit or": "$NEO.bitor",
    "bit shift down": "$NEO.bitdown",
    "bit shift up": "$NEO.bitup",
    "bit xor": "$NEO.bitxor",
    "boolean?": "$NEO.boolean_",
    "char": "$NEO.char",
    "code": "$NEO.code",
    "false": "false",
    "fraction": "$NEO.fraction",
    "function?": "$NEO.function_",
    "integer": "$NEO.integer",
    "integer?": "$NEO.integer_",
    "length": "$NEO.length",
    "neg": "$NEO.neg",
    "not": "$NEO.not",
    "null": "undefined",
    "number": "$NEO.make",
    "number?": "$NEO.is_big_float",
    "record": "$NEO.record",
    "record?": "$NEO.record_",
    "stone": "$NEO.stone",
    "stone?": "Object.isFrozen",
    "text": "$NEO.text",
    "text?": "$NEO.text_",
    "true": "true"
});
```

В JavaScript пробельные символы играют куда более скромную роль, чем в Neo, поэтому можно создавать действительно уродливый код и JavaScript возражать не станет. Я думаю, что в мире слишком много безобразного. Когда есть возможность сделать что-то лучше, нужно так и делать, даже если никто этого не заметит.

```
let indentation;

function indent() {
```

```
        indentation += 4;
    }

    function outdent() {
        indentation -= 4;
    }

    function begin() {
```

В начале каждой строки вставляются перевод строки и пробельное заполнение.

```
        return "\n" + " ".repeat(indentation);
    }

    let front_matter;
    let operator_transform;
    let statement_transform;
    let unique;
```

Соглашения относительно использования имен в Neo и JavaScript не полностью совместимы. В Neo, в отличие от JavaScript, разрешены пробелы и вопросительные знаки. Поэтому при необходимости имена Neo коверкают, превращая в допустимые имена JavaScript. Neo допускает любые слова, но JavaScript некоторые из них резервирует. Когда в Neo применяется слово, зарезервированное в JavaScript, к этому слову при перемещении его в пространство JavaScript добавляется знак доллара. Чтобы программы легче читались, большие десятичные числа с плавающей запятой преобразуют в форму, напоминающую числовые литералы.

```
    const rx_space_question = / [ \u0020 ? ]/g;

    function mangle(name) {
```

В идентификаторах JavaScript не разрешены пробел или знак вопроса (`?`), поэтому их заменяют символом подчеркивания (`_`). Зарезервированным словам дается префикс в виде знака доллара (`$`).

Таким образом, `what me worry?` превращается в `what_me_worry_`, а `class` — в `$class`.

```
        return (
            reserved[name] === true
            ? "$" + name
            : name.replace(rx_space_question, "_")
        );
    }

    const rx_minus_point = / [ \- . ] /g;

    function numgle(number) {
```

Литералам больших десятичных чисел придают как можно более естественный вид путем превращения их в константы. Имя константы начинается с символа `$`. Символы `-` или `.` заменяют символом подчеркивания (`_`).

В результате `1` становится `$1`, `98.6` преобразуется в `$98_6`, а `-1.011e-5` — в `$_1_011e_5`.

```
    const text = big_float.string(number.number);
    const name = "$" + text.replace(rx_minus_point, "_");
    if (unique[name] !== true) {
        unique[name] = true;
        front_matter.push(
            "const " + name + " = $NEO.number(\"" + text + "\");\n"
        );
    }
    return name;
}
```

Большинство генераторов кода — это простые функции, преобразующие лексемы обратно в текст. Для превращения всех частей в текст они вызывают друг друга. Начнем с функции `op`, принимающей лексему оператора. Она выдает преобразованную форму оператора. Многие операторы придерживаются весьма простой схемы, поэтому их преобразованными формами являются строки. Если у лексемы есть прикрепленные операнды, то составляется вызов функции. Для операторов, не соответствующих данной схеме, преобразованием является функция, принимающая лексему и возвращающая строку.

```
function op(thing) {
    const transform = operator_transform[thing.id];
    return (
        typeof transform === "string"
        ? (
            thing.zeroth === undefined
            ? transform
            : transform + "(" + expression(thing.zeroth) + (
                thing.wunth === undefined
                ? ""
                : ", " + expression(thing.wunth)
            ) + ")"
        )
        : transform(thing)
    );
}
```

Функция `expression` обрабатывает лексемы выражений общего вида.

```
function expression(thing) {
    if (thing.id === "(number)") {
        return numgle(thing);
    }
    if (thing.id === "(text)") {
        return JSON.stringify(thing.text);
    }
    if (thing.alphameric) {
        return (
            thing.origin === undefined
            ? primordial[thing.id]
```

```
            : mangle(thing.id)
        );
    }
    return op(thing);
}
```

Эта функция создает массив литералов.

```
function array_literal(array) {
    return "[" + array.map(function (element) {
        return (
            Array.isArray(element)
            ? array_literal(element)
            : expression(element)
        );
    }).join(", ") + "]";
}
```

Из литералов записей Neo получаются объекты без прототипов. Пустые записи создаются с помощью `Object.create(null)`. Поля создаются путем присваивания. Присваивания заключаются в тут же вызываемое функциональное выражение. Из `{[foo bear]: 12.3, two part}` получается:

```
(function (o) {
    $NEO.set(o, foo_bear, $12_3);
    o["two part"] = two_part;
}(Object.create(null)))
```

Имена переменных в литералах записей коверкаются, а имена полей — нет.

```
function record_literal(array) {
    indent();
    const padding = begin();
    const string = "(function (o) {" + array.map(function (element) {
        return padding + (
            typeof element.zeroth === "string"
            ? (
                "o["
                + JSON.stringify(element.zeroth)
                + "] = "
                + expression(element.wunth)
                + ";"
            )
            : (
                "$NEO.set(o, "
                + expression(element.zeroth)
                + ", "
                + expression(element.wunth)
                + ");"
            )
        );
    }).join("") + padding + "return o;";
    outdent();
    return string + begin() + "}(Object.create(null)))";
}
```

В языке Neo нет ничего булевоподобного. Есть места, где язык ожидает предоставления булевых значений, например условная часть инструкции `if`. Если Neo передается значение, не являющееся булевым, он должен дать сбой. Чтобы получить такое же поведение в JavaScript, может потребоваться заключить такие значения в функцию `assert_boolean`.

```
function assert_boolean(thing) {
    const string = expression(thing);
    return (
        (
            boolean_operator[thing.id] === true
            || (
                thing.zeroth !== undefined
                && thing.zeroth.origin === undefined
                && boolean_operator[thing.zeroth.id]
            )
        )
        ? string
        : "$NEO.assert_boolean(" + string + ")"
    );
}
```

Массивы лексем-инструкций можно преобразовать в строку и заключить в блоки.

```
function statements(array) {
    const padding = begin();
    return array.map(function (statement) {
        return padding + statement_transform[statement.id](statement);
    }).join("");
}

function block(array) {
    indent();
    const string = statements(array);
    outdent();
    return "{" + string + begin() + "}";
}
```

Объект `statement_transform` содержит функции преобразований для всех инструкций. В большинстве инструкций нет ничего сложного. Сложность инструкции `if` заключается в существовании трех форм: без `else`, с `else` и с `else if`. Инструкция `let` — единственное место, где происходят изменения. Она должна работать с оператором `[]`, который в левой части выполняет добавление, а в правой — извлечение и работает с левосторонними значениями (lvalue).

```
statement_transform = $NEO.stone({
    break: function (ignore) {
        return "break;";
    },
    call: function (thing) {
        return expression(thing.zeroth) + ";";
    },
    def: function (thing) {
```

```
    return (
        "var " + expression(thing.zeroth)
        + " = " + expression(thing.wunth) + ";"
    );
},
export: function (thing) {
    const exportation = expression(thing.zeroth);
    return "export default " + (
        exportation.startsWith("$NEO.stone(")
        ? exportation
        : "$NEO.stone(" + exportation + ")"
    ) + ";";
},
fail: function () {
    return "throw $NEO.fail(\"fail\");";
},
if: function if_statement(thing) {
    return (
        "if ("
        + assert_boolean(thing.zeroth)
        + ") "
        + block(thing.wunth)
        + (
            thing.twoth === undefined
            ? ""
            : " else " + (
                thing.twoth.id === "if"
                ? if_statement(thing.twoth)
                : block(thing.twoth)
            )
        )
    );
},
import: function (thing) {
    return (
        "import " + expression(thing.zeroth)
        + " from " + expression(thing.wunth) + ";"
    );
},
let: function (thing) {
    const right = (
        thing.wunth.id === "[]"
        ? expression(thing.wunth.zeroth) + ".pop();"
        : expression(thing.wunth)
    );
    if (thing.zeroth.id === "[]") {
        return expression(thing.zeroth.zeroth) + ".push(" + right + ");";
    }
    if (thing.zeroth.id === ".") {
        return (
            "$NEO.set(" + expression(thing.zeroth.zeroth)
            + ", " + JSON.stringify(thing.zeroth.wunth.id)
            + ", " + right + ");"
        );
    }
```

```
        if (thing.zeroth.id === "[") {
            return (
                "$NEO.set(" + expression(thing.zeroth.zeroth)
                + ", " + expression(thing.zeroth.wunth)
                + ", " + right + ");"
            );
        }
        return expression(thing.zeroth) + " = " + right + ";";
    },
    loop: function (thing) {
        return "while (true) " + block(thing.zeroth);
    },
    return: function (thing) {
        return "return " + expression(thing.zeroth) + ";";
    },
    var: function (thing) {
        return "var " + expression(thing.zeroth) + (
            thing.wunth === undefined
            ? ";"
            : " = " + expression(thing.wunth) + ";"
        );
    }
});
```

Указатель на функцию — это встроенная функция, доступ к которой можно получить, поместив перед оператором префикс `ƒ`.

```
const functino = $NEO.stone({
    "?": "$NEO.ternary",
    "|": "$NEO.default",
    "/\\": "$NEO.and",
    "\\/": "$NEO.or",
    "=": "$NEO.eq",
    "≠": "$NEO.ne",
    "<": "$NEO.lt",
    "≥": "$NEO.ge",
    ">": "$NEO.gt",
    "≤": "$NEO.le",
    "~": "$NEO.cat",
    "≈": "$NEO.cats",
    "+": "$NEO.add",
    "-": "$NEO.sub",
    ">>": "$NEO.max",
    "<<": "$NEO.min",
    "*": "$NEO.mul",
    "/": "$NEO.div",
    "[": "$NEO.get",
    "(": "$NEO.resolve"
   });
```

Объект `operator_transform` содержит все преобразования операторов.

```
operator_transform = $NEO.stone({
    "?": function (thing) {
        indent();
        let padding = begin();
```

```
        let string = (
            "(" + padding + assert_boolean(thing.zeroth)
            + padding + "? " + expression(thing.wunth)
            + padding + ": " + expression(thing.twoth)
        );
        outdent();
        return string + begin() + ")";
    },
    "/\\": function (thing) {
        return (
            "(" + assert_boolean(thing.zeroth)
            + " && " + assert_boolean(thing.wunth)
            + ")"
        );
    },
    "\\/": function (thing) {
        return (
            "(" + assert_boolean(thing.zeroth)
            + " || " + assert_boolean(thing.wunth)
            + ")"
        );
    },
    "=": "$NEO.eq",
    "≠": "$NEO.ne",
    "<": "$NEO.lt",
    "≥": "$NEO.ge",
    ">": "$NEO.gt",
    "≤": "$NEO.le",
    "~": "$NEO.cat",
    "≈": "$NEO.cats",
    "+": "$NEO.add",
    "-": "$NEO.sub",
    ">>": "$NEO.max",
    "<<": "$NEO.min",
    "*": "$NEO.mul",
    "/": "$NEO.div",
    "|": function (thing) {
        return (
            "(function (_0) {"
            + "return (_0 === undefined) ? "
            + expression(thing.wunth) + " : _0);}("
            + expression(thing.zeroth) + "))"
        );
    },
    "…": function (thing) {
        return "…" + expression(thing.zeroth);
    },
    ".": function (thing) {
        return (
            "$NEO.get(" + expression(thing.zeroth)
            + ", \"" + thing.wunth.id + "\")"
        );
    },
```

```
    "[": function (thing) {
        if (thing.wunth === undefined) {
            return array_literal(thing.zeroth);
        }
        return (
            "$NEO.get(" + expression(thing.zeroth)
            + ", " + expression(thing.wunth) + ")"
        );
    },
    "{": function (thing) {
        return record_literal(thing.zeroth);
    },
    "(": function (thing) {
        return (
            expression(thing.zeroth) + "("
            + thing.wunth.map(expression).join(", ") + ")"
        );
    },
    "[]": "[]",
    "{}": "Object.create(null)",
    "ƒ": function (thing) {
        if (typeof thing.zeroth === "string") {
            return functino[thing.zeroth];
        }
        return "$NEO.stone(function (" + thing.zeroth.map(function (param) {
            if (param.id === "…") {
                return "…" + mangle(param.zeroth.id);
            }
            if (param.id === "|") {
                return (
                    mangle(param.zeroth.id) + " = " + expression(param.wunth)
                );
            }
            return mangle(param.id);
        }).join(", ") + ") " + (
            Array.isArray(thing.wunth)
            ? block(thing.wunth)
            : "{return " + expression(thing.wunth) + ";}"
        ) + ")";
    }
});
```

Здесь экспортируется функция генератора кода, которая принимает дерево и возвращает программу в исходном коде JavaScript.

```
export default $NEO.stone(function codegen(tree) {
    front_matter = [
        "import $NEO from \"./neo.runtime.js\"\n"
    ];
    indentation = 0;
    unique = Object.create(null);
    const bulk = statements(tree.zeroth);
    return front_matter.join("") + bulk;
});
```

Пример

Эта функция аналогична методу `map`, но работает и с несколькими массивами, скалярами и генераторами.

```
export ƒ function, arguments… {
    if length(arguments) = 0
        return null
    var index: 0
    def result: []
    var stop: false

    def prepare arguments: ƒ argument {
        def candidate: (
            array?(argument)
            ? argument[index]
            ! (
                function?(argument)
                ? argument(index)
                ! argument
            )
        )
        if candidate = null
            let stop: true
        return candidate
    }
    loop
        var processed: array(arguments, prepare arguments)
        if stop
            break
        let result[]: function(processed…)
        let index: index + 1
    return result
}
```

А это содержимое файла с расширением `.js`, произведенного с использованием `codegen(parse(tokenize(neo_source)))`:

```
import $NEO from "./neo.runtime.js";
const $0 = $NEO.number("0");
const $1 = $NEO.number("1");

export default $NEO.stone(function ($function, …$arguments) {
    if ($NEO.eq($NEO.length($arguments), $0)) {
        return undefined;
    }
    var index = $0;
    var result = [];
    var stop = false;
    var prepare_arguments = $NEO.stone(function (argument) {
        var candidate = (
            Array.isArray(argument)
            ? $NEO.get(argument, index)
            : (
                $NEO.function_(argument)
```

```
                ? argument(index)
                : argument
            )
        );
        if ($NEO.eq(candidate, undefined)) {
            stop = true;
        }
        return candidate;
    });
    while (true) {
        var processed = $NEO.array($arguments, prepare_arguments);
        if ($NEO.assert_boolean(stop)) {
            break;
        }
        result.push($function(…processed));
        index = $NEO.add(index, $1);
    }
    return result;
});
```

А затем:

```
import do: "example/do.neo"

var result: do(ƒ+, [1, 2, 3], [5, 4, 3])
# result is [6, 6, 6]

let result: do(ƒ/, 60, [1, 2, 3, 4, 5, 6])
# result is [60, 30, 20, 15, 12, 10]
```

что транспилируется в:

```
import $NEO from "./neo.runtime.js"
const $1 = $NEO.number("1");
const $2 = $NEO.number("2");
const $3 = $NEO.number("3");
const $5 = $NEO.number("5");
const $4 = $NEO.number("4");
const $60 = $NEO.number("60");
const $6 = $NEO.number("6");

import $do from "example/do.neo";
var result = $do($NEO.add, $60, [$1, $2, $3], [$5, $4, $3]);
result = $do($NEO.div, $60, [$1, $2, $3, $4, $5, $6]);
```

Глава 29

Как работает среда выполнения

●●●○●

> Я знаю, почему ты здесь, Нео. Знаю, что тебя гнетет...
> Почему ты плохо спишь, почему живешь один
> и почему ночи напролет сидишь за компьютером.
>
> *Тринити*

Среда выполнения — это программное средство, поддерживающее выполнение программ. Популярность JavaScript в качестве цели для транспиляции отчасти обусловлена качеством поддержки, предоставляемой этим языком во время их выполнения. Если семантика исходного языка отличается от целевой, должна быть добавлена специализированная поддержка среды выполнения. Форма среды выполнения Neo в JavaScript — это объект, содержащий функции, помогающие выполнению программ.

Два наиболее существенных семантических улучшения Neo — это усовершенствованные тип чисел и объект. Разработка числового типа была показана в главе 4. Записи языка Neo объединяют объекты и weakmap-коллекции. Подведем итоги.

- Ключом может быть текст, запись или массив. Перечислить с помощью функции `array` реально только текстовые ключи.
- Значение `null` приводит к удалению поля.
- Выражения пути с отсутствующими объектами не вызывают сбой — они выдают `null`.
- Массивы в качестве ключей принимают только целые числа и обеспечивают соблюдение их границ.

Начнем с централизованной функции `fail`.

```
function fail(what = "fail") {
    throw new Error(what);
}
```

Единственная переменная, сохраняющая состояние в этом файле, — это `weakmap_of_weakmaps`. Она связывает weakmap-коллекции с записями. Большинству записей

weakmap-коллекции не понадобятся, они нужны тем записям, которые действительно получают свои отображения из `weakmap_of_weakmaps`.

Функция `get` извлекает значение поля из записи или элемент из массива. В ней также путем возвращения функции реализован вызов функции в форме вызова метода. Если по какой-либо причине что-то пойдет не так, возвращается объект `null`, известный JavaScript как `undefined`.

```
let weakmap_of_weakmaps = new WeakMap();

function get(container, key) {
    try {
        if (Array.isArray(container) || typeof container === "string") {
            const element_nr = big_float.number(key);
            return (
                Number.isSafeInteger(element_nr)
                ? container[(
                    element_nr >= 0
                    ? element_nr
                    : container.length + element_nr
                )]
                : undefined
            );
        }
        if (typeof container === "object") {
            if (big_float.is_big_float(key)) {
                key = big_float.string(key);
            }
            return (
                typeof key === "string"
                ? container[key]
                : weakmap_of_weakmaps.get(container).get(key)
            );
        }
        if (typeof container === "function") {
            return function (…rest) {
                return container(key, rest);
            };
        }
    } catch (ignore) {
    }
}
```

Доступ к функции `get` можно получить через указатель на функцию, имеющий вид `ƒ[]`.

Функция `set` — это способ добавления, обновления или удаления поля записи или обновления элемента массива. Если что-то пойдет не так, происходит сбой. Функция `get` в этом смысле не слишком взыскательна, чего нельзя сказать о функции `set`.

```
function set(container, key, value) {
    if (Object.isFrozen(container)) {
        return fail("set");
    }
    if (Array.isArray(container)) {
```

Массивы в качестве ключей используют только большие числа с плавающей запятой.

```
        let element_nr = big_float.number(key);
        if (!Number.isSafeInteger(element_nr)) {
            return fail("set");
        }
```

Отрицательные индексы — это альтернативный способ доступа, поэтому применение `[-1]` приводит к установке на последний элемент.

```
        if (element_nr < 0) {
            element_nr = container.length + element_nr;
        }
```

Ключ должен находиться в выделенном для массива диапазоне.

```
        if (element_nr < 0 || element_nr >= container.length) {
            return fail("set");
        }
        container[element_nr] = value;
    } else {
        if (big_float.is_big_float(key)) {
            key = big_float.string(key);
        }
```

Если ключ представляет собой строковое значение, значит, это обновление объекта.

```
        if (typeof key === "string") {
            if (value === undefined) {
                delete container[key];
            } else {
                container[key] = value;
            }
        } else {
```

В противном случае это обновление weakmap-коллекции. Это будет weakmap-коллекция, связанная с каждой записью с помощью ключей в виде объектов. Следует заметить, что, когда ключом является массив, выражение `typeof key !== "object"` вычисляется в `false`.

```
            if (typeof key !== "object") {
                return fail("set");
            }
            let weakmap = weakmap_of_weakmaps.get(container);
```

Если же weakmap-колекции, связанной с этим контейнером, еще нет, то она создается.

```
            if (weakmap === undefined) {
                if (value === undefined) {
                    return;
                }
                weakmap = new WeakMap();
                weakmap_of_weakmaps.set(container, weakmap);
            }
```

Обновление weakmap-коллекции:

```
            if (value === undefined) {
                weakmap.delete(key);
            } else {
                weakmap.set(key, value);
            }
        }
    }
}
```

Далее следует группа функций, создающих массивы, числовые значения, записи и тексты:

```
function array(zeroth, wunth, …rest) {
```

Функция `array` выполняет работу `new Array`, `array.fill`, `array.slice`, `Object.keys`, `string.split` и многих других:

```
    if (big_float.is_big_float(zeroth)) {
        const dimension = big_float.number(zeroth);
        if (!Number.isSafeInteger(dimension) || dimension < 0) {
            return fail("array");
        }
        let newness = new Array(dimension);
        return (
            (wunth === undefined || dimension === 0)
            ? newness
            : (
                typeof wunth === "function"
                ? newness.map(wunth)
                : newness.fill(wunth)
            )
        );
    }
    if (Array.isArray(zeroth)) {
        return zeroth.slice(big_float.number(wunth), big_float.number(rest[0]));
    }
    if (typeof zeroth === "object") {
        return Object.keys(zeroth);
    }
    if (typeof zeroth === "string") {
        return zeroth.split(wunth || "");
    }
    return fail("array");
}

function number(a, b) {
    return (
        typeof a === "string"
        ? big_float.make(a, b)
        : (
            typeof a === "boolean"
            ? big_float.make(Number(a))
            : (
                big_float.is_big_float(a)
```

```
                ? a
                : undefined
            )
        )
    );
}

function record(zeroth, wunth) {
    const newness = Object.create(null);
    if (zeroth === undefined) {
        return newness;
    }
    if (Array.isArray(zeroth)) {
        if (wunth === undefined) {
            wunth = true;
        }
        zeroth.forEach(function (element, element_nr) {
            set(
                newness,
                element,
                (
                    Array.isArray(wunth)
                    ? wunth[element_nr]
                    : (
                        typeof wunth === "function"
                        ? wunth(element)
                        : wunth
                    )
                )
            );
        });
        return newness;
    }
    if (typeof zeroth === "object") {
        if (wunth === undefined) {
            return Object.assign(newness, zeroth);
        }
        if (typeof wunth === "object") {
            return Object.assign(newness, zeroth, wunth);
        }
        if (Array.isArray(wunth)) {
            wunth.forEach(function (key) {
                let value = zeroth[key];
                if (value !== undefined) {
                    newness[key] = value;
                }
            });
            return newness;
        }
    }
    return fail("record");
}

function text(zeroth, wunth, twoth) {
    if (typeof zeroth === "string") {
```

```
        return (zeroth.slice(big_float.number(wunth), big_float.number(twoth)));
    }
    if (big_float.is_big_float(zeroth)) {
        return big_float.string(zeroth, wunth);
    }
    if (Array.isArray(zeroth)) {
        let separator = wunth;
        if (typeof wunth !== "string") {
            if (wunth !== undefined) {
                return fail("string");
            }
            separator = "";
        }
        return zeroth.join(separator);
    }
    if (typeof zeroth === "boolean") {
        return String(zeroth);
    }
}
```

Функция `stone` выполняет глубокую заморозку:

```
function stone(object) {
    if (!Object.isFrozen(object)) {
        object = Object.freeze(object);
        if (typeof object === "object") {
            if (Array.isArray(object)) {
                object.forEach(stone);
            } else {
                Object.keys(object).forEach(function (key) {
                    stone(object[key]);
                });
            }
        }
    }
    return object;
}
```

Теперь группа функций-предикатов, используемых для идентификации типов:

```
function boolean_(any) {
    return typeof any === "boolean";
}

function function_(any) {
    return typeof any === "function";
}

function integer_(any) {
    return (
        big_float.is_big_float(any)
        && big_float.normalize(any).exponent === 0
    );
}

function number_(any) {
```

```
    return big_float.is_big_float(any);
}

function record_(any) {
    return (
        any !== null
        && typeof any === "object"
        && !big_float.is_big_float(any)
    );
}

function text_(any) {
    return typeof any === "string";
}
```

Далее следует группа логических функций. Есть и functino-версии. Они не выполняют короткозамкнутые вычисления. На «ленивое» вычисление своих операндов способны только версии в форме операторов.

```
function assert_boolean(boolean) {
    return (
        typeof boolean === "boolean"
        ? boolean
        : fail("boolean")
    );
}

function and(zeroth, wunth) {
    return assert_boolean(zeroth) && assert_boolean(wunth);
}

function or(zeroth, wunth) {
    return assert_boolean(zeroth) || assert_boolean(wunth);
}

function not(boolean) {
    return !assert_boolean(boolean);
}

function ternary(zeroth, wunth, twoth) {
    return (
        assert_boolean(zeroth)
        ? wunth
        : twoth
    );
}

function default_function(zeroth, wunth) {
    return (
        zeroth === undefined
        ? wunth
        : zeroth
    );
}
```

А это группа операторов отношений:

```
function eq(zeroth, wunth) {
    return zeroth === wunth || (
        big_float.is_big_float(zeroth)
        && big_float.is_big_float(wunth)
        && big_float.eq(zeroth, wunth)
    );
}

function lt(zeroth, wunth) {
    return (
        zeroth === undefined
        ? false
        : (
            wunth === undefined
            ? true
            : (
                (
                    big_float.is_big_float(zeroth)
                    && big_float.is_big_float(wunth)
                )
                ? big_float.lt(zeroth, wunth)
                : (
                    (typeof zeroth === typeof wunth && (
                        typeof zeroth === "string"
                        || typeof zeroth === "number"
                    ))
                    ? zeroth < wunth
                    : fail("lt")
                )
            )
        )
    );
}

function ge(zeroth, wunth) {
    return !lt(zeroth, wunth);
}

function gt(zeroth, wunth) {
    return lt(wunth, zeroth);
}

function le(zeroth, wunth) {
    return !lt(wunth, zeroth);
}

function ne(zeroth, wunth) {
    return !eq(wunth, zeroth);
}
```

Теперь группа арифметических операторов:

```
function add(a, b) {
    return (
```

```
        (big_float.is_big_float(a) && big_float.is_big_float(b))
        ? big_float.add(a, b)
        : undefined
    );
}

function sub(a, b) {
    return (
        (big_float.is_big_float(a) && big_float.is_big_float(b))
        ? big_float.sub(a, b)
        : undefined
    );
}

function mul(a, b) {
    return (
        (big_float.is_big_float(a) && big_float.is_big_float(b))
        ? big_float.mul(a, b)
        : undefined
    );
}

function div(a, b) {
    return (
        (big_float.is_big_float(a) && big_float.is_big_float(b))
        ? big_float.div(a, b)
        : undefined
    );
}

function max(a, b) {
    return (
        lt(b, a)
        ? a
        : b
    );
}

function min(a, b) {
    return (
        lt(a, b)
        ? a
        : b
    );
}

function abs(a) {
    return (
        big_float.is_big_float(a)
        ? big_float.abs(a)
        : undefined
    );
}

function fraction(a) {
```

```
    return (
        big_float.is_big_float(a)
        ? big_float.fraction(a)
        : undefined
    );
}

function integer(a) {
    return (
        big_float.is_big_float(a)
        ? big_float.integer(a)
        : undefined
    );
}

function neg(a) {
    return (
        big_float.is_big_float(a)
        ? big_float.neg(a)
        : undefined
    );
}
```

Далее идет группа поразрядных функций. Их работа выполняется с использованием `big_integer`:

```
function bitand(a, b) {
    return big_float.make(
        big_integer.and(
            big_float.integer(a).coefficient,
            big_float.integer(b).coefficient
        ),
        big_integer.wun
    );
}

function bitdown(a, nr_bits) {
    return big_float.make(
        big_integer.shift_down(
            big_float.integer(a).coefficient,
            big_float.number(nr_bits)
        ),
        big_integer.wun
    );
}

function bitmask(nr_bits) {
    return big_float.make(big_integer.mask(big_float.number(nr_bits)));
}

function bitor(a, b) {
    return big_float.make(
        big_integer.or(
            big_float.integer(a).coefficient,
            big_float.integer(b).coefficient
```

```
        ),
        big_integer.wun
    );
}

function bitup(a, nr_bits) {
    return big_float.make(
        big_integer.shift_up(
            big_float.integer(a).coefficient,
            big_float.number(nr_bits)
        ),
        big_integer.wun
    );
}

function bitxor(a, b) {
    return big_float.make(
        big_integer.xor(
            big_float.integer(a).coefficient,
            big_float.integer(b).coefficient
        ),
        big_integer.wun
    );
}
```

Возможно, вы помните из JSCheck указатель на функцию (functino) `ƒ()`:

```
function resolve(value, …rest) {
    return (
        typeof value === "function"
        ? value(…rest)
        : value
    );
}
```

Имеются также два оператора конкатенации. Оба они возвращают `null`, если значение `null` имеет один из аргументов. В противном случае они предпринимают попытку приведения своих аргументов к текстам. Если оба операнда не являются пустым текстом, указатель на функцию `ƒ≈` включает разделитель в виде пробела.

```
function cat(zeroth, wunth) {
    zeroth = text(zeroth);
    wunth = text(wunth);
    if (typeof zeroth === "string" && typeof wunth === "string") {
        return zeroth + wunth;
    }
}

function cats(zeroth, wunth) {
    zeroth = text(zeroth);
    wunth = text(wunth);
    if (typeof zeroth === "string" && typeof wunth === "string") {
        return (
```

```
            zeroth === ""
            ? wunth
            : (
                wunth === ""
                ? zeroth
                : zeroth + " " + wunth
            )
        );
    }
}
```

Далее следуют разнородные функции:

```
function char(any) {
    return String.fromCodePoint(big_float.number(any));
}

function code(any) {
    return big_float.make(any.codePointAt(0));
}

function length(linear) {
    return (
        (Array.isArray(linear) || typeof linear === "string")
        ? big_float.make(linear.length)
        : undefined
    );
}
```

Все эти ценности упаковываются в объект среды выполнения:

```
export default stone({
    abs,
    add,
    and,
    array,
    assert_boolean,
    bitand,
    bitdown,
    bitmask,
    bitor,
    bitup,
    bitxor,
    boolean_,
    cat,
    cats,
    char,
    code,
    default: default_function,
    div,
    fail,
    fraction,
    function_,
    ge,
    get,
```

```
    gt,
    integer,
    integer_,
    le,
    length,
    max,
    min,
    mul,
    ne,
        neg,
    not,
    number,
    number_,
    or,
    record,
    record_,
    resolve,
    set,
    stone,
    sub,
    ternary,
    text,
    text_
});
```

Глава 30

Как работают нелепости, или Что такое Wat!

●●●●○

> Хватит высмеивать отстойные языки. Давайте поговорим о JavaScript.
>
> *Гэри Бернхардт (Gary Bernhardt)*

Двенадцатого января 2012 года на конференции CodeMash, проходившей в аквапарке города Сандаски (штат Огайо), Гэри Бернхардт сделал блиц-доклад под названием Wat.

Бернхардт продемонстрировал ряд абсурдных аномалий в Ruby и JavaScript. После каждого примера он показывал забавную картинку с надписью WAT («Нелепость»). Присутствующим это понравилось. Это был классический ход. Слово Wat просто приклеилось к JavaScript.

Краткое выступление Бернхардта до сих пор можно найти на просторах Интернета. Подобные шоу стали проводить многие подражатели. Некоторые из них просто повторяли материал Бернхардта, а некоторые расширили представление, добавив новые нелепости. Кому-то, как Бернхардту, удалось вызвать смех аудитории. Кто-то просто напустил едкого туману.

В этой книге я постарался обойти стороной большинство неудачных мест JavaScript, но в данной главе собираюсь открыть неприглядную истину. Хочу показать вам некоторые нелепости из разряда Wat и подобных и объяснить, как они работают. Смешного или приятного здесь будет мало.

Многие насмешки связаны с алгоритмами приведения типов в JavaScript-операторах `==` и `+`. Используемые правила приведения типов сложны, трудны для запоминания, а в некоторых случаях, как будет показано далее, и работают неверно. Именно поэтому я не рекомендовал задействовать оператор `==`. В нем применяется регламентированный ECMAScript-стандартом алгоритм под названием Abstract Equality Comparison Algorithm (алгоритм абстрактного сравнительного метода). Это банка червей, которую не стоит открывать. Вместо этого оператора

нужно всегда использовать оператор `===`, состоящий из трех знаков равенства. Всегда.

Кому-то оператор `===` не нравится, поскольку он выглядит на 50 % бестолковее оператора `==`, который вдвое бестолковее оператора `=`. И все же правильным оператором равенства в JavaScript является `===`. А применения оператора `==` нужно избегать, поскольку он некорректен.

Не могу посоветовать отказаться от использования оператора `+`, поскольку он обеспечивает единственный практический способ сложения чисел. Поэтому смиритесь с червями из банки.

```
"" == false                    // true
[] == false                    // true
null == false                  // false
undefined == false             // false
```

Пустая строка — это логически лживое значение, поэтому дефектный оператор равенства может захотеть связать его с `false`. Пустой массив не является логически лживым, и все же он также сравнивается с `false`. А `null` и `undefined` *являются* логически лживыми значениями, но ни одно из них не сравнивается с `false`.

WAT!

```
[] == []                       // false
[] == ![]                      // true
```

Два пустых массива не являются одним и тем же объектом, поэтому они не равны. Но вторая строка вызывает недоумение. Получается, что JavaScript — это язык, в котором *x* и *не x* могут быть равны друг другу, что было бы серьезной, смеху подобной некомпетентностью. Происходит следующее: пустой массив считается логически правдивым, следовательно, `![]` — ложь (`false`). Дефектный оператор равенства хочет сравнить `[]` и `false` в качестве чисел, хотя ни один из операндов числом не является. Пустой массив приводится к пустой строке, которая приводится к нулю. `false` также приводится к нулю. Нуль равен нулю, следовательно, ответом будет `true`.

WAT!

```
[] + []                        // ""
[] + {}                        // "[object Object]"
{} + {}                        // "[object Object][object Object]"
```

Во всех эти случаях должно получаться значение `NaN`. Именно для этого оно и предназначено. А вместо этого, поскольку значения не являются числами, оператор `+` хочет их объединить. Сначала должно состояться их приведение к строкам. Метод `Array.prototype.toString()` преобразует пустой массив в пустую строку. Было бы лучше, если бы он работал как метод `JSON.stringify()` и возвращал `"[]"`. Беспо-

лезный метод `Object.prototype.toString()` визуализирует объекты как `"[object Object]"`. Затем эти строки объединяются.

WAT!

```
9999999999999999                    // 10000000000000000
1e23 + 2e23 === 3e23                // false
```

Многие целые числа, превышающие значение константы `Number.MAX_SAFE_INTEGER`, не могут быть представлены точно.

WAT!

```
"2" < 5                             // true
5 < "11"                            // true
"11" < "2"                          // true
```

Сравнение значений разных типов должно приводить к выдаче исключения. Вместо этого JavaScript пытается выполнить приведение типов значений, чтобы сравнение могло состояться. Правила приведения типов способны нарушать транзитивность.

WAT!

```
1 < 2 < 3                          // true
3 > 2 > 1                          // false
```

Эти случаи должны быть синтаксическими ошибками, потому что язык не хочет их правильно обрабатывать. В первом случае сравниваются 1 и 2 и получается `true`. Затем сравниваются `true` и `3`. В результате приведения типа `true` превращается в `1`. А `1` меньше `3`, в результате получается `true`. Таким образом, случайно дается правильный ответ. Ошибка, временами выдающая правильный ответ, может легко избежать обнаружения при тестировании.

Во втором случае сравниваются `3` и `2`, в результате получается `true`. Затем сравниваются `true` и `1`. В результате приведения типа `true` превращается в `1`. А `1` не меньше `1`, в результате получается `false`.

WAT!

```
"2" + 1                             // "21"
"2" - 1                             // 1
```

Другие арифметические операторы также выполняют приведение типов, но применяют совершенно другие правила, чем те, что использует оператор `+`. Во избежание этой небрежности нужно правильно управлять своими типами. Не выполняйте арифметические действия со строками. Для них предназначены числа.

WAT!

```
Math.min() > Math.max()             // true
```

Эти функции содержат дефекты. Когда им ничего не передается, они должны возвращать `undefined` или `NaN` или, возможно, выдавать исключение. Вместо этого `Math.min()` возвращает `Infinity`, а `Math.max()` возвращает `-Infinity`.

WAT!

```
Math instanceof Math                // выдача исключения
NaN instanceof NaN                  // выдача исключения
"wat" instanceof String             // false
```

Применение оператора `instanceof` в Java обычно свидетельствует о неспособности понять порядок использования полиморфизма. Оператор `instanceof`, принадлежащий JavaScript, отличается от оператора `instanceof`, принадлежащего Java. Я советую воздержаться от `instanceof`, работая с любым из этих языков.

WAT!

```
isNaN("this string is not NaN")     // true
```

Глобальные функции `isNaN` и `isFinite` имеют дефекты. Вместо них следует использовать `Number.isNaN` и `Number.isFinite`.

WAT!

```
((name) => [name])("wat")           // ["wat"]
((name) => {name})("wat")           // undefined
```

В первом случае показан пример функции жирной стрелки (или *fart*-функции). Fart-функции являются сокращенным способом записи функций. Все, что слева от fart-функции, — это список параметров, а все, что справа, — выражение. Возвращаемым значением функции является значение выражения. Не нужно набирать слово `function` или `return`. В данном случае возвращаемым значением будет массив, состоящий из аргумента функции. И это хорошо.

По идее, во втором случае должно быть возвращено значение `{name: "wat"}`, а не `undefined`. К сожалению, JavaScript полагает, что левая фигурная скобка после fart-функции является блоком, а не литералом объекта. В этом блоке содержится имя переменной. Автоматически точка с запятой помещается после имени, превращая его в бесполезное выражение, которое ничего не делает. В отсутствие оператора функция возвращает значение по умолчанию, то есть `undefined`.

Именно поэтому я рекомендую избегать применения fart-функций. К тому же их легко перепутать с операторами сравнения `<=` и `>=`. Несущественная экономия при наборе текста этого не стоит.

WAT!

```
function first(w, a, t) {
    return {
        w,
        a,
        t
    };
```

```
}
first("wat", "wat", "wat");    // {w: "wat", a: "wat", t: "wat"}

function second(w, a, t) {
    return
        {w, a, t};
}
second("wat", "wat", "wat");   // undefined
```

Первая инструкция `return` работает, как и ожидалось, возвращая новый объект.

Вторая инструкция `return` отличается всего лишь пробельным символом, и все же она возвращает `undefined`. Срабатывает механизм автоматической вставки после `return` точки с запятой. Теперь в позиции инструкции — левая фигурная скобка литерала объекта, где она рассматривается как блок. Этот блок содержит бесполезную инструкцию выражения. В позиции выражения запятая считается оператором, а не разделителем, поэтому данный код не вызывает синтаксической ошибки.

Автоматическая вставка точки с запятой не дает особого преимущества. Она создает опасные ситуации. Эта функция была добавлена в язык специально для начинающих, которые могут не знать, где поставить точку с запятой. Полагаться на автоматическую вставку точек с запятой можно только в том случае, если хочется создать впечатление, что код написан новичком.

WAT!

Глава 31

Как устроена эта книга

•••••

Я прошу одного — свободы. Свободны же бабочки.

Гарольд Скимпол (Harold Skimpole)

Функция include

Чтобы было проще собрать книгу, я создал событийную функцию `include`. И воспользовался этой функцией, чтобы собрать отдельные файлы глав в один большой файл книги, а также для вставки исполняемого исходного кода JavaScript в главы. Кроме того, она применялась для сбора фрагментов кода JavaScript в файлы с расширением `.js`, которые можно найти по адресу github.com/douglascrockford.

```
include(обратный_вызов, строка, получение_включения, максимальная_глубина)
```

Функция `include` заменяет выражения `@include` в строке другими строками. Если выражений `@include` нет, возвращается исходная строка.

Ваша функция *`обратный_вызов`*`(`*`результат`*`)` выдает в итоге получившуюся в результате обработки строку *`результат`*.

Ваша строка может содержать ноль и более выражений `@include`.

```
@include "ключ"
```

Между `@include` и открывающей двойной кавычкой (`"`) ставится пробел. Каждое выражение `@include` заменяется, если это возможно, строкой включения, связанной с ключом. Ключ, которым может быть имя файла, заключается в круглые скобки.

Функция *`получение_включения`*`(`*`обратный_вызов`*`, `*`ключ`*`)` принимает строковое значение *`ключ`* и в итоге передает получившуюся строку включения функции *`обратный_вызов`*`(`*`включение`*`)`. Функция *`получение_включения`* способна обращаться к файловой системе, базе данных, системе контроля версий, контент-менеджеру или объекту JSON. Если включения поступают из файлов и средой является Node.js, то функция *`получение_включения`* может выглядеть следующим образом:

```
function my_little_get_inclusion(callback, key) {
    return (
        (key[0] >= "a" && key[0] <= "z")
        ? fs.readFile(key, "utf8", function (ignore, data) {
            return callback(data);
        })
        : callback()
    );
}
```

Если каким-то образом пакет включения станет неконтролируемым (этим грешат управляемые пакеты), он не сможет причинить с помощью `my_little_get_inclusion` значительный ущерб, но смог бы нанести огромный вред, если бы имел прямой доступ к файловой системе.

Строка включения может содержать дополнительные выражения `@include`. Аргумент *`максимальная_глубина`* ограничивает глубину рекурсии, чтобы предотвратить бесконечные циклы включения.

Рассмотрим реализацию:

```
const rx_include = /
    @include \u0020 " ( [^ " @ ]+ ) "
/;

// Перехватываемые группы:
// [0] все выражение '@include'
// [1] Ключ

export default Object.freeze(function include(
    callback,
    string,
    get_inclusion,
    max_depth = 4
) {
```

Функция `include` не нуждается в непосредственном допуске к файловой системе или сведениях о ней, то же самое касается базы данных или чего-либо еще, поскольку эта возможность передана ей в качестве функции *`получение_включения`*. Благодаря этому функция `include` становится универсальной и заслуживающей доверия.

Ничто не возвращается. Результат в итоге сообщается через *`обратный_вызов`*.

```
    let object_of_matching;
    let result = "";
```

Всю работу делают функция `minion` (ближайшее окружение) и ее помощники. Основная функция `minion` ищет выражения `@include` и вызывает функцию *`получение_включения`* в том, что было найдено. Вспомогательная функция `assistant_minion` выполняет рекурсивный вызов `include` для обработки включения. Функция `junior_assistant_minion` добавляет обработанные включения в результат.

```
    function minion() {
```

Если строк для сканирования больше нет, выполняется доставка результата:

```
if (string === "") {
    return callback(result);
}
```

Попытка сопоставления регулярного выражения с оставшейся строкой:

```
object_of_matching = rx_include.exec(string);
```

Если соответствий не было, значит, наша работа выполнена:

```
if (!object_of_matching) {
    return callback(result + string);
}
```

Символы слева от выражения являются частью результата. Удаление отсканированного результата из строки:

```
result += string.slice(0, object_of_matching.index);
string = string.slice(
    object_of_matching.index + object_of_matching[0].length
);
```

Вызов функции `get_inclusion` для получения строки замены с передачей `assistant_minion` и ключа:

```
    return get_inclusion(
        assistant_minion,
        object_of_matching[1]
    );
}

function junior_assistant_minion(processed_inclusion) {
```

Прием включения, обработанного функцией `include`, и добавление его к результату — затем вызов функции `minion` для начала поиска следующего выражения `@include`:

```
    result += processed_inclusion;
    return minion();
}

function assistant_minion(inclusion) {
```

Если функция `get_inclusion` не доставила строку, к результату добавляется выражение `@include`, экономно оставляя эту часть строки без изменений:

```
    if (typeof inclusion !== "string") {
        result += object_of_matching[0];
        return minion();
    }
```

Включение может содержать собственные выражения `@include`, поэтому из обработки вызывается `include` с передачей `junior_assistant_minion`, добавляющей

к результату обработанное включение. Для защиты от бесконечных рекурсий значение `max_depth` уменьшается:

```
            return include(
                junior_assistant_minion,
                inclusion,
                get_inclusion,
                max_depth - 1
            );
        }
```

Таково ближайшее окружение. Теперь вернемся к `include`. Если происходит выход за пределы заданной нами глубины, вызывается `callback`:

```
        if (max_depth <= 0) {
            callback(string);
        } else {
```

Функция `include` создает три функции ближайшего окружения и вызывает основную функцию `minion`:

```
        minion();
    }
});
```

Благодарности

Я хочу поблагодарить Эдвина Аоки (Edwin Aoki), Владимира Баквански (Vladimir Bacvanski), Леонардо Боначчи (Leonardo Bonacci), Джорджа Буля (George Boole), Денниса Клайна (Dennis Cline), Роландо Димаандала (Rolando Dimaandal), Билла Франца (Bill Franz), Луи Готтлиба (Louis Gottlieb), Боба Хаблутцеля (Bob Hablutzel), Грейс Мюррей Хоппер (Grace Murray Hopper), Мэтью Джонсона (Matthew Johnson), Алана Карпа (Alan Karp), Готтрида Лейбница (Gottried Leibniz), Хокона Виума Ли (Håkon Wium Lie), Линду Мерри (Linda Merry), Джеффа Мейера (Jeff Meyer), Чипа Морнингстара (Chip Morningstar), Евгения Орехова, Бена Пардо (Ben Pardo), Клода Шеннона (Claude Shannon), Стива Соудерса (Steve Souders), Техути (Tehuti) и, самое главное, профессора Лизу фон Дрейк (Lisa von Drake).

Послесловие

В своей карьере программиста я прошел следующие смены парадигм.

- Языки высокого уровня.
- Структурное программирование.
- Объектно-ориентированное программирование.
- Функциональное программирование.

Крупнейшими бенефициарами каждого из этих прогрессивных изменений были программисты, получавшие возможность успешнее справляться с повышенным объемом работы, прикладывая при этом меньше усилий. Крупнейшими противниками этих прогрессивных изменений были те же программисты. Они встречали новые парадигмы с подозрением и недоверием. Весь свой интеллект и опыт они вкладывали в выработку контраргументов, казавшихся весьма убедительными, но на поверку ставших ошибочными. Они успешно применяли старую парадигму и не хотели принимать новую. Возможно, увидеть разницу между новой парадигмой и крайне неудачной идеей и в самом деле очень трудно.

Принятие каждой из смен парадигм заняло более 20 лет. Функциональное программирование прошло через это дважды. Причина, по которой этот процесс занимает так много времени, заключается в косности мышления. Приходится ждать, пока уйдет на пенсию или из жизни целое поколение программистов, прежде чем наберется критическая масса последователей следующей парадигмы.

Макс Планк (Max Planck) наблюдал подобное явление в физике, теперь оно известно как принцип Планка. Он писал: «Новая научная истина торжествует не потому, что ее противники признают свою неправоту, просто ее оппоненты со временем вымирают, а подрастающее поколение знакомо с нею с самого начала».

Он выразился и еще более емко: «Научная истина торжествует по мере того, как вымирают ее противники».

Полагаю, уже должно стать очевидным, что следующая парадигма — это распределенное событийное программирование (Distributed Eventual Programming). Эта идея не нова, она восходит к представлению модели акторов (1973 год). С тех пор было сделано несколько шагов вперед, но в то же время допущены ошибки в таких функциях, как удаленные вызовы процедур, которые пытаются выполнять распределенные действия, придерживаясь модели последовательного программирования. Я думаю, причина интереса к JavaScript заключается в том, что он был специально создан для выполнения распределенного событийного программирования. Когда язык разрабатывается всего за десять дней, невозможно все сделать правильно. К тому же это язык с несколькими парадигмами, поощряющий сохранение старой парадигмы. И с момента его создания поборники старой парадигмы пытаются вернуть JavaScript к его фортрановским корням.

И еще одно послесловие

Вы, вероятно, заметили, что в представленном материале довольно странно написано английское слово «один»: wun вместо привычного one[1]. Это могло отвлечь от чтения, сбить с толку и даже вызвать раздражение. Контраргументы носят традиционный характер. Мол, это не то, к чему мы привыкли. Это пишется не так, как

[1] В переводе книги такое написание сохраняется в примерах программного кода. — *Примеч. пер.*

нас учили. Это не то, что мы знаем. Все это правильно, но, учитывая условности английского произношения, wun должно быть верным написанием. Это просто исправление допущенной ошибки.

Наша эмоциональная реакция на wun сродни эмоциям при встрече с новой парадигмой.

Секрет принятия новой парадигмы до того, как к ней пропадет всяческий интерес, заключается в испытании ее положений на деле. В программировании это сделать проще, чем в физике, потому что все в нем сводится к написанию хороших программ. Я могу говорить о создании функции, которая возвращает функцию, а вы можете воспринимать сказанное и думать, что понимаете, о чем идет речь. Но фактически вы не способны осмыслить все это, пока не испытаете на практике. Нужно написать множество функций, возвращающих функции, а также множество функций, совершающих концевые вызовы. В какой-то момент все это перестанет восприниматься как неестественное. Но добраться до этого момента, просто читая теорию или обсуждая ее, вы не сможете. Если вы хорошо освоили написание программ на JavaScript, то JavaScript вас научит. Если же вы плохо пишете на JavaScript, то JavaScript вас накажет. Но для вас это уже, наверное, не секрет.

Возможно, в моей следующей книге мы что-нибудь сделаем с написанием слова two.

Заметки

Музыка — это пространство между нотами.

Клод Дебюсси (Claude Debussy)