

В. А. АНТОНЮК

Язык Julia
как инструмент
исследователя

Москва
Физический факультет МГУ им. М. В. Ломоносова
2019

Антонюк Валерий Алексеевич

Язык Julia как инструмент исследователя. –

М. : Физический факультет МГУ им. М. В. Ломоносова, 2019. – 48 с.

Пособие знакомит читателей со сравнительно новым языком программирования Julia (позиционируемым как язык для научного программирования), с его особенностями и возможностями. Спроектированный так, чтобы совместить удобство и простоту создания прототипа программы с последующим эффективным исполнением получаемого кода в рамках одного языка, он должен помочь всем, кто вынужден при написании программ сначала экспериментировать в рамках MATLAB или Python, а затем для реальной работы переписывать всю программу (или её наиболее критичные части) для компилируемого языка (C, C++ и т. п.), — тем более, что синтаксис его будет близок всем, кто имеет опыт работы с MATLAB или Python.

На простых примерах иллюстрируются основные идеи в реализации языка Julia (система типов, функциональная диспетчеризация, параллелизм исполнения и др.), так, чтобы каждый читатель мог сам для себя решить, оправданы ли усилия по освоению нового языка и насколько весомы те преимущества, которые можно будет получить в своей практической деятельности.

Рассчитано на студентов старших курсов физического факультета, но может быть полезным аспирантам и сотрудникам, желающим познакомиться с современными тенденциями развития языков программирования.

Автор — сотрудник кафедры математического моделирования и информатики физического факультета МГУ.

Рецензент: профессор кафедры математики физфака МГУ Боголюбов А.Н.

Подписано в печать 04.12.2019. Объем 3,0 п.л. Тираж 30 экз. Заказ № 190. Физический факультет им. М. В. Ломоносова, 119991 Москва, ГСП-1, Ленинские горы, д. 1, стр. 2.

Отпечатано в отделе оперативной печати физического факультета МГУ.

© Физический факультет МГУ
им. М. В. Ломоносова, 2019
© В. А. Антонюк, 2019

Оглавление

1.	Язык <i>Julia</i> — общие сведения	5
1.1.	Дистрибутивные файлы <i>Julia</i>	6
1.2.	Установка дистрибутива <i>Julia</i>	6
1.3.	Первые эксперименты	7
2.	<i>REPL</i> и работа в «Блокноте» (<i>Jupyter notebook</i>)	8
3.	Величины в языке <i>Julia</i>	9
3.1.	Типы числовых величин	9
3.2.	Целочисленные величины	10
3.3.	Вещественные числа	11
3.4.	Комплексные числа	11
3.5.	Рациональные числа	11
3.6.	<code>BigInt</code> , <code>BigFloat</code>	11
3.7.	Иррациональные числа	11
4.	Функции в языке <i>Julia</i>	12
4.1.	«Традиционное» определение функции	12
4.2.	Специальное (однотрочное) присваивание	13
4.3.	Безымянные (анонимные) функции	13
4.4.	Передача параметров функциям	13
4.5.	Операции, применимые к функциям	14
4.6.	Макроопределения (макросы)	14
4.7.	Некоторые часто используемые макросы	15
4.8.	Несколько практических примеров функций	15
4.9.	Пример: свойства последовательностей Коллатца	16
4.10.	Пример: реализация классического метода Рунге–Кутты	18
4.11.	Пример: оценка значения числа π методом Монте-Карло	21
4.12.	Пример: умножение матриц в идемпотентных алгебрах	22
4.13.	Пример: вычисление меры близости между ранжированиями	25
5.	Пакеты в языке <i>Julia</i>	26
5.1.	Модули и пакеты	26
5.2.	«Экосистема» <i>Julia</i> : <code>pkg.julialang.org</code> и <code>juliaobserver.com</code>	26
5.3.	Куда устанавливаются пакеты	27
5.4.	Пакеты визуализации данных	28
5.5.	Прочие пакеты	28
5.6.	Пример: <code>APL.jl</code> — интерпретатор строк APL-кода	29
6.	О плохом и хорошем (вместо заключения)	31
	ПРИЛОЖЕНИЕ. Поиск вариантов формул Штрассена	32

Предисловие

С точки зрения пользователя любой новый язык программирования — это в первую очередь необходимость по-новому формулировать свои мысли и пожелания. Поэтому переход будет оправдан, если уровень «входа» будет невысоким, а преимущества освоения — весомыми.

Созданный не так давно язык *Julia* — это открытый динамический компилируемый язык, ориентированный в первую очередь на производительные вычисления в научно-технических областях. До его появления казалось естественным, что интерактивность свойственна лишь языкам интерпретируемым (например, *Python*, *R*, *MATLAB*). Однако сочетание в нём JIT-компиляции, производимой лишь тогда, когда код необходимо исполнить, с множественной диспетчеризацией (означающей, что функции вызываются динамически во время исполнения — в зависимости от типа переданных им параметров), позволило достичь практически того же эффекта: язык получился интерактивным, при этом скорость исполнения его программ сравнима со скоростью программ, написанных на языке *C*.

В нём есть практически всё, что имеется в популярных языках настоящего времени (распространённые числовые типы вместе с величинами произвольной точности и богатый набор математических функций; современные коллекции данных: кортежи, словари и множества; интроспекция кода; встроенный менеджер пакетов; возможность взаимодействия с другими языками и библиотеками), но немало и новинок: высокоуровневые средства для параллельных и распределённых вычислений, поддержка метапрограммирования, а также возможность использования в тексте программ широкого диапазона допустимых символов, что приводит к некоторым трудностям в изложении, включая правильную синтаксическую раскраску *Julia*-кода и адекватное его воспроизведение в различных публикациях: статьях и книгах (это будет заметно по некоторым включённым в текст листингам).

Представленная брошюра даже не претендует на то, чтобы служить вводным пособием по языку *Julia*: в ней охвачены далеко не все аспекты языка, а те, что затронуты, не изложены в каком-либо стройном порядке. Скорее, можно считать брошюру неким эссе, призванным немного ознакомить читателя с предметом — для того, чтобы можно было решить, необходимо ли тратить время на более внимательное его изучение.

По этой причине оказались не затронутыми различные (но важные для полноценной работы с языком) темы: кортежи, словари, множества, параметрические типы и описание сложных типов, операции ввода-вывода, работа со строками, обработка исключений, создание сопрограмм, параллельная и распределённая обработка данных, взаимодействие с оболочкой операционной системы, интеграция с другими языками программирования и многое другое. Для их освоения правильнее будет обратиться к [руководству пользователя](#).

Тем не менее, отдельным базовым понятиям языка уделено особое внимание. В качестве таких понятий выбраны *величины*, *функции* и *пакеты*, поэтому вся структура изложения определяется ими, их свойствами и возможностями, а всё остальное, если и упоминается, то лишь для пояснения выбранных в качестве иллюстрации примеров.

Автор попытался применить разные конструкции языка к тем задачам, с которыми ему доводилось сталкиваться; что при этом получилось, изложено в приводимых далее простых практических примерах. Какие-то возможности языка остались при этом не охваченными, но исключительно потому, что соответствующие задачи пока ещё не встретились.

Главное же, в чём хотелось бы убедить читателя: язык *Julia* — это универсальный и мощный инструмент, который может помочь каждому исследователю практически в любой области научной или технической деятельности.

1. Язык *Julia* — общие сведения

Если рассматривать *Julia* просто как способ записи алгоритмов, то в первом приближении кажется, что всё в нём практически совпадает с тем, что можно написать в *MATLAB*...

(a) <i>Julia</i>	(b) <i>MATLAB</i>	(c) <i>Python</i>
<pre># Это комментарий в Julia</pre>	<pre>% Это комментарий в MATLAB</pre>	<pre># Это комментарий в Python</pre>
<pre>#= Многострочный комментарий =#</pre>	<pre>%{ Многострочный комментарий %}</pre>	<pre># # Многострочный комментарий #</pre>
<pre>if i <= N # Условие # Действия else # Альтернативные действия end</pre>	<pre>if i <= N % Условие % Действия else % Альтернативные действия end</pre>	<pre>if i <= N: # Условие # Действия else: # Альтернативные действия</pre>
<pre>while i <= N # Условие # Действия end</pre>	<pre>while i <= N # Условие % Действия end</pre>	<pre>while i <= N: # Условие # Действия</pre>
<pre>for i = 1:N # Действия end</pre>	<pre>for i = 1:N % Действия end</pre>	<pre>for i in range(N): # Действия</pre>

Рис. 1: Примеры базового синтаксиса в языках *Julia*, *MATLAB*, *Python*.

От *MATLAB* также «унаследованы»: индексация массивов с единицы и расположение их в памяти по столбцам, а также подавление вывода завершающей точкой с запятой.

Конечно, наиболее спорные «особенности» *MATLAB* в языке *Julia* отсутствуют: символ комментария не является знаком распространённой (в том числе — и в математике) операции, круглые скобки в доступе к элементам массивов заменены на более традиционные квадратные, даже функция `whos()` уже «забыта» в пользу более новой `varinfo()`...

Но подобное сходство не является случайным: оно свидетельствует о том, что создатели языка сознательно снизили и без того не очень высокий пороговый уровень первоначально «входа» в язык именно для пользователей *MATLAB*, видимо, рассчитывая на их интерес к бесплатной и более высокопроизводительной альтернативе, не обладающей пока столь же внушительным набором дополнительных пакетов.

В остальном же *Julia* — это новый, весьма интересный и нетривиальный язык общего назначения с многообещающими свойствами и характеристиками. Как уже говорилось выше, это открытый динамический компилируемый язык. Открытость его означает, что всем доступен исходный код языка и стандартных библиотек (*Julia* была провозглашена проектом с открытым кодом в 2012 году, через 3 года после появления языка). Каждый пользователь имеет право модифицировать код, адаптируя его для решения собственных задач. Кроме того, язык является бесплатным, т. е., не требуется делать никаких выплат за его использование (а в случае, например, таких популярных — но закрытых — языков как *MATLAB* и *Mathematica*, это не так).

Выражение «динамический язык» обозначает язык с динамической типизацией, где пользователь может писать программу, вообще не упоминая типы каких-либо величин, как это должно быть в языке со статической типизацией (например, *C*).

Объявление типов параметров, передаваемых функции, не является обязательным, поскольку *Julia* автоматически специализирует функцию по типам её параметров во время компиляции, т. е., типы параметров у функций являются своеобразным фильтром, позволяющим указать, какая конкретно функция и когда используется.

Можно сказать, что традиционное объектно-ориентированное программирование (в принятой в *C++* форме) использует диспетчеризацию только по одному (первому) параметру: выбор подходящего метода для вызова производится лишь на основании типа объекта, к которому он будет применён. А в *Julia* реализована т. н. **множественная диспетчеризация**: при исполнении выбирается метод или функция на основании типов **всех** аргументов.

Кроме того, *Julia* — компилируемый¹ язык (*Python*, *R*, *MATLAB* — интерпретируемые²). Это значит, что код *Julia* будет исполняться непосредственно на используемом процессоре. Всего процессорных платформ (архитектур), на которых работает *Julia*, пять: **i686** (32-bit), **x86-64** (64-bit), **ARMv7** (32-bit), **ARMv8** (64-bit), **PowerPC** (64-bit); поддерживаются операционные системы *Windows*, *macOS*, *Linux*, *FreeBSD*. В случае процессоров многоядерных, вполне обычных в настоящее время, можно использовать для работы все их ядра, запуская *Julia* специальным образом.

В языке имеется встроенная поддержка комплексных чисел, дробей и величин произвольной точности; реализованы популярные современные типы данных: векторы, матрицы, многомерные массивы, кортежи, словари, множества, битовые массивы, битовые строки.

Интересной особенностью является возможность использования многих символов *Unicode* в тексте программ: как в именах переменных и функций, так и в знаках операций; большинство этих символов можно ввести с помощью привычных L^AT_EX-последовательностей, завершаемых нажатием клавиши Tab.

1.1. Дистрибутивные файлы *Julia*

Дистрибутивные файлы для системы *Windows* — это исполняемые файлы (инсталляторы), поименованные по схеме `julia-<Версия>-<Система>.exe`, где *<Версия>* означает версию языка *Julia* (скажем, *1.3.0*), а *<Система>* — `win32` или `win64`. Для систем *Linux* используются обычные архивы с именами вида `julia-<Версия>-<Система>-<Архитектура>.tar.gz`, где *<Система>* — это `linux`, а *<Архитектура>* — `aarch64`, `armv7l` или `x86_64`. Все эти файлы находятся по адресу: <https://julialang.org/downloads/>.

На верхнем уровне любой файл архива содержит каталог с именем `julia-<Версия>`; там расположены: один файл с лицензией (`LICENSE.md`) и шесть каталогов (`bin`, `etc`, `include`, `lib`, `libexec`, `share`). В первом каталоге находится главный исполняемый файл языка (`julia`), во втором (в подкаталоге `julia`) — файл `startup.jl`, который может содержать команды, исполняемые при запуске *Julia* на компьютере³, в третьем (в его подкаталоге `julia`) находятся заголовочные файлы для *C*-компилятора, в четвёртом каталоге — файл динамической библиотеки `libjulia.so` со своими ярлыками и дополнительные библиотеки в подкаталоге `julia`, в пятом — исполняемый файл «распаковщика» `7z`, в шестом (в различных подкаталогах) собраны: документация, иконки (реально — одна в виде `.svg`-файла), ярлык для рабочего стола, описание приложения в виде файла `julia.appdata.xml` и тексты базовых библиотек (`base+stdlib`) с тестами.

1.2. Установка дистрибутива *Julia*

Для системы *Windows* пользователю следует использовать нужный файл инсталлятора, в различных вариантах *Linux* достаточно будет распаковать архив в нужное место.

Поскольку дистрибутив *Julia* для *Linux* — это просто архив каталогов, то имеется всего два варианта установки *Julia* в *Linux* и оба они должны будут осуществляться пользователем вручную. Первый — это распаковать архив в «пространстве» конкретного пользователя, второй — сделать это в каком-то месте, доступном всем пользователям⁴.

«Общедоступное» место тоже можно выбрать двумя способами: либо никак не связывать его с текущими путями поиска программ (и тогда понадобится ярлык исполняемого файла в пути их поиска), либо «вклеить» структуру каталогов дистрибутива *Julia* (каталоги `bin`,

¹Компиляция производится во время исполнения (т.н. JIT-компиляция), при этом для оптимизации создаваемого кода используется информация о типах величин, которые к этому моменту уже известны.

²Более точно следовало бы сказать так: язык *Python* интерпретирует откомпилированный байт-код.

³Для команд, исполняемых на уровне пользователя, предусмотрен файл `~/.julia/config/startup.jl`.

⁴Помимо каталога размещения дистрибутивного архива *Julia* для работы также создаётся (скрытый) каталог с именем `.julia` в «домашнем» каталоге пользователя; там будут располагаться: окружения, установленные пакеты (исходный код и откомпилированный вариант), сведения обо всех зарегистрированных пакетах (с разбивкой по первым буквам и названиям), журнальные файлы (чуть подробнее — см. стр. 27).

include, share и др.) в каталог `/usr/local`⁵ (при этом ярлык для запуска программы не понадобится, поскольку `/usr/local/bin` просматривается автоматически при поиске команд пользователя). Во втором способе надо лишь «избавиться» от «лишнего» каталога (вроде `julia-1.3.0` в архиве дистрибутива `julia-1.3.0-linux-armv7l.tar.gz`); для этого имеется специальный ключ `--strip-components` в программе распаковки:

```
sudo tar -xzf <Дистрибутив>.tar.gz -C /usr/local --strip-components 1
```

указывающий, с какого уровня вложенности нужно распаковывать архив (в данном случае — пропустить первый вложенный каталог `julia-1.3.0`).

Единственный недостаток такого способа — сравнительно сложная деинсталляция программы при необходимости (а это тоже придётся делать вручную), поскольку файлы дистрибутива будут перемешаны с другими файлами.

Если же распаковывать дистрибутивный архив, скажем, в каталог `/opt`, то пропускать первый вложенный каталог архива при этом не надо (это даст возможность установить несколько различных версий *Julia* параллельно), но для простоты обращения к версиям понадобится создать ярлыки (здесь иллюстрируется установка конкретной версии 1.3.0 в *Linux* для *ARM*):

```
sudo tar -xzf julia-1.3.0-linux-armv7l.tar.gz -C /opt
sudo ln -s /opt/julia-1.3.0/bin/julia /usr/local/bin/julia
```

При этом деинсталлировать какую-либо версию можно будет простым удалением каталога с соответствующим дистрибутивом (а также какой-то части каталога `.julia` у пользователя).

1.3. Первые эксперименты

Исполняемый файл языка *Julia* (`julia.exe` в *Windows* или `julia` — в других системах) может использоваться либо для запуска какой-либо программы, хранимой в файле с расширением `.jl` (тогда надо ввести в системной командной строке `julia <Программа>.jl`), либо для работы в интерактивном режиме (для чего исполняемый файл запускается без параметров — с командной строки или с помощью иконки). Интерактивная сессия начинается с воспроизведения в консольном окне сведений о версии языка и после вывода подсказки вида `julia>` (зелёного цвета) ожидается ответ пользователя. Данный режим называется *REPL* (сокращение от **read-eval-print loop**): ввод пользователя считывается, выполняется⁶, а результат исполнения выводится, после чего снова появляется *REPL*-подсказка и опять ожидается ответ пользователя. Выйти из этого интерактивного режима можно при помощи комбинации клавиш `Ctrl + D` или после ввода вызова функции `exit()`.

Режим *REPL* имеет четыре разных варианта, каждый со своей собственной подсказкой, причём другого цвета (чтобы их было легко различать): исполнение выражений *Julia* (упомянут выше и реализуется по умолчанию), справка (подсказка `help?>` будет жёлтого цвета и появится после ввода вопросительного знака `?` в начале строки), режим системной оболочки (подсказка `shell>` красного цвета; надо ввести точку с запятой `;`), поиск в истории введённых команд (подсказка белого цвета после `Ctrl + R`). Возврат к исполнению — нажатие клавиши `BackSpace` в самом начале строки (она же стирает всё введённое ранее).

В режиме исполнения выражений можно что-либо вычислить, используя *REPL* как калькулятор, или вызвать какие-то функции, например, информационные: `versioninfo()` сообщает сведения о языке *Julia* (версия, дата выпуска) и платформе исполнения (система, процессор, библиотеки), `varinfo()` выводит информацию о загруженных модулях, введённых в рамках сессии величинах и занимаемой ими памяти и т. д.

В справочном режиме будет выведена краткая цитата из документации в ответ на ввод символа, типа, переменной, функции или макроса — если то, что введено, известно *Julia*.

Режим системной оболочки используется для ввода из среды *Julia* системных команд.

Несмотря на то, что в практической деятельности предпочтительнее будет использовать «Блокнот», а не режим *REPL*, далее в брошюре везде приводятся примеры ввода-вывода в *REPL* — хотя бы потому, что в таком случае не требуется ничего, кроме установленного дистрибутива *Julia*.

⁵Установить таким образом несколько разных версий уже не получится.

⁶В случае *Julia* исполнение предваряется компиляцией введённого фрагмента (т. н. *JIT*-компиляция).

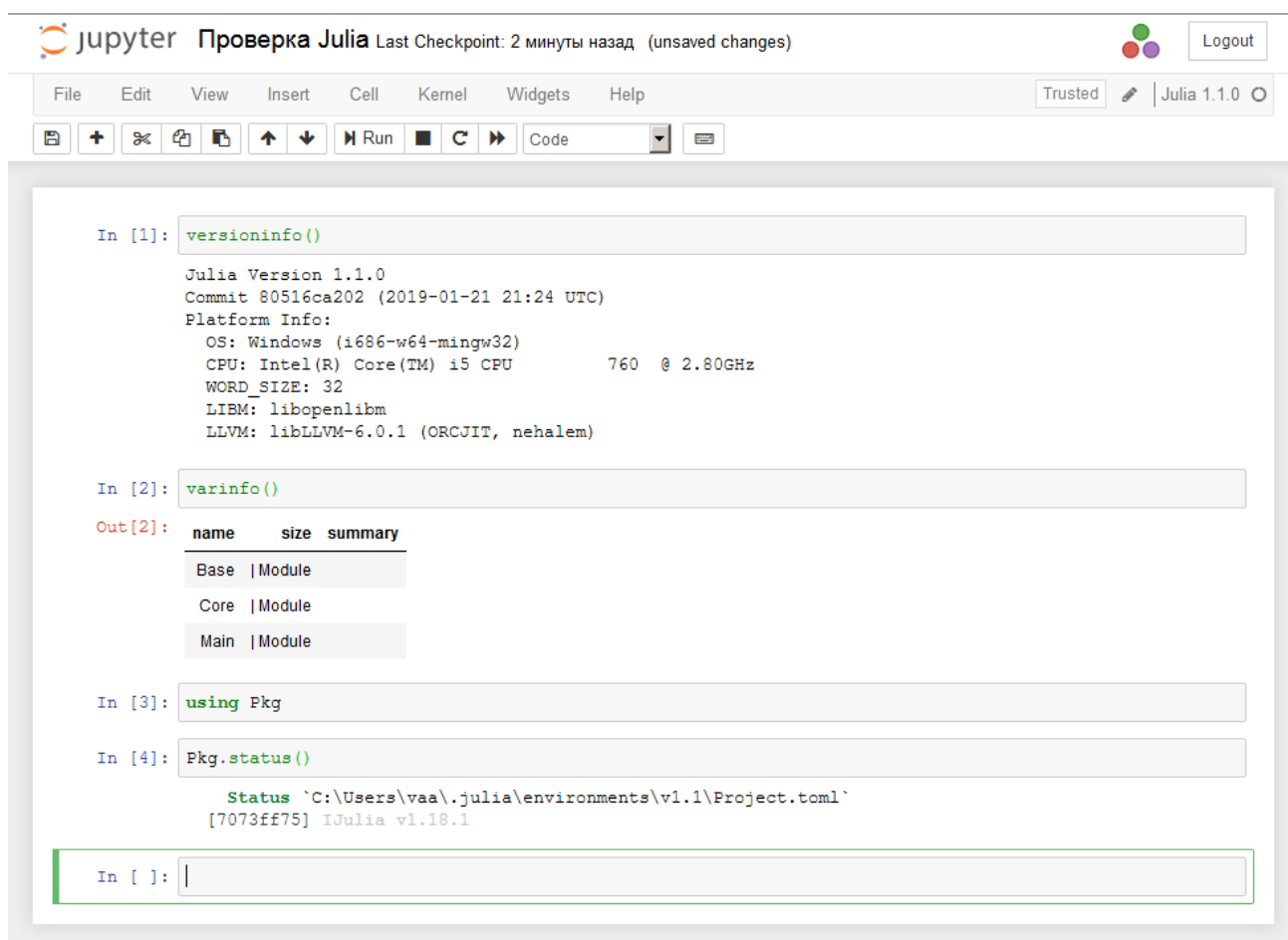
2. *REPL* и работа в «Блокноте» (*Jupyter notebook*)

В сессии *REPL* можно легко обращаться к ранее введённым командам, модифицировать их и запускать снова, однако это не идёт ни в какое сравнение с тем, что возможно в т. н. «Блокноте» (оригинальное название — *Jupyter*⁷ *notebook*).

Jupyter notebook — это интерактивное окружение в рамках обычного браузера. Оно сочетает в себе отображение кода, форматированного текста, изображений, видео и анимации, математических формул, графиков и иллюстраций в пределах одного документа, что позволяет эффективно сохранять результаты работы и/или распространять их.

Использование *Julia* в рамках «Блокнота» предполагает установку пакета `IJulia.jl` (для этого надо выполнить команду `add IJulia` из интерфейса менеджера пакетов, куда можно попасть, вводя символ `]` в *REPL*). Если пакет `IJulia.jl` установлен, то «Блокнот» запускается одним из двух способов: с командной строки системы — при помощи команды `jupyter-notebook` (иногда она может называться иначе), а непосредственно из сессии *Julia REPL* — импортированием модуля (`using IJulia`) с последующим вызовом функции `notebook()` (без параметра или с поименованным параметром: `notebook(dir=<Каталог>`), где `<Каталог>` — желательный текущий каталог).

В результате в браузере — после обращения к запущенному локальному веб-серверу — отобразится страница с адресом `localhost:8888` и дальнейшая работа будет происходить там (т. н. *dashboard*). Для создания нового документа следует выбрать «кнопку» **New** на этой странице справа и указать его тип (в нашем случае — *Julia* какой-либо версии); дождавшись появления новой закладки браузера с «пустым» документом, переименовать его нажатием надписи `Untitled1` в «шапке» страницы, после чего можно вводить команды (или текст) в документ (в текущий элемент ввода в зелёной рамке). Нажатие комбинации клавиш `Shift` + `Enter` «исполняет» введённый код.



The screenshot shows the Jupyter Notebook interface. At the top, there is a header with the Jupyter logo, the text "jupyter Проверка Julia Last Checkpoint: 2 минуты назад (unsaved changes)", and a "Logout" button. Below the header is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". To the right of the menu bar is a "Trusted" indicator and "Julia 1.1.0". Below the menu bar is a toolbar with icons for file operations, navigation, and execution. The main area contains a code cell with the following content:

```
In [1]: versioninfo()

Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (i686-w64-mingw32)
  CPU: Intel(R) Core(TM) i5 CPU          760 @ 2.80GHz
  WORD_SIZE: 32
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, nehalem)

In [2]: varinfo()

Out[2]: name      size  summary
        -----
        Base |Module
        Core |Module
        Main |Module

In [3]: using Pkg

In [4]: Pkg.status()

Status `C:\Users\vaa\.julia\environments\v1.1\Project.toml`
[7073ff75] IJulia v1.18.1

In [ ]: |
```

⁷Название программы *Jupyter* образовано из наименований трёх языков: *Julia*, *Python* и *R*.

3. Величины в языке *Julia*

Хотя весьма часто в простых программах можно обойтись без упоминания каких-либо типов вообще, типы величин в языке существуют, более того, образуют весьма стройную древовидную иерархию. Каждая величина обязательно имеет какой-то тип, но связан он не с именами переменных (как это было характерно для предварительно компилируемых языков вроде *C/C++*), а с самими величинами, поэтому обязательного объявления типа переменная не требует: он станет известен после присваивания ей величины⁸.

3.1. Типы числовых величин

Ниже приведены три фрагмента из деревьев типов *Julia* различных версий, начиная с (абстрактного) типа `Number` (*Число*); далее перечисляются возможные в указанной версии подтипы *Чисел*, а для них, соответственно, их подтипы и т. д.

(a) <i>Julia 0.3.11</i>	(b) <i>Julia 0.6.2</i>	(c) <i>Julia 1.0.0</i>
<pre>julia> ShowTypeTree(Number) Number Complex{T<:Real} Real FloatingPoint BigFloat Float16 Float32 Float64 Integer BigInt Bool Char Signed Int128 Int16 Int32 Int64 Int8 Unsigned UInt128 UInt16 UInt32 UInt64 UInt8 MathConst{sym} Rational{T<:Integer}</pre>	<pre>julia> ShowTypeTree(Number) Number Complex Real AbstractFloat BigFloat Float16 Float32 Float64 Integer BigInt Bool Signed Int128 Int16 Int32 Int64 Int8 Unsigned UInt128 UInt16 UInt32 UInt64 UInt8 Irrational Rational</pre>	<pre>julia> ShowTypeTree(Number) Number Complex Real AbstractFloat BigFloat Float16 Float32 Float64 AbstractIrrational Irrational Integer Bool Signed BigInt Int128 Int16 Int32 Int64 Int8 Unsigned UInt128 UInt16 UInt32 UInt64 UInt8 Rational</pre>

Рис. 2: Фрагменты иерархии типов в различных версиях *Julia*.

Эти фрагменты получены вызовом простой рекурсивной функции, названной здесь `ShowTypeTree()`; она выводит очередной тип в иерархии — с отступом⁹, соответствующим текущему уровню вложенности этого типа, переходя далее к перебору также всех его подтипов на следующем уровне — снова при помощи вызова `ShowTypeTree()`.

Определить тип какой-либо величины можно, вызывая для неё функцию `typeof()`, а для любого существующего типа всегда можно узнать, какие подтипы у него имеются

```
function ShowTypeTree(T, level=0)
    println(" " ^ level, T)
    for t in subtypes(T)
        ShowTypeTree(t, level+1)
    end
end
```

Рис. 3: Функция вывода иерархии типов.

⁸Важно также отметить, что в *Julia* — в отличие, скажем, от языка *Python*, — тип величины не может измениться в результате каких-то её преобразований (т. е., например, при последовательном возведении целого в квадрат оно останется целой величиной *того же типа*, что и ранее; попробуйте поэкспериментировать в *REPL* или «*Блокноте*» с чередованием команд `a = a*a` и `typeof(a)`, начиная с `a = 2`, и сравните, если есть такая возможность, с результатами аналогичных действий в *Python*).

⁹Операция `^`, применяемая здесь к строке и числу, просто повторно воспроизводит строку (пробелов) заданное число раз, формируя отступ, пропорциональный количеству вложенных вызовов.

(с помощью функции `subtypes()`), и у какого типа он сам является подтипом (используя `supertype()`).

Какие же выводы можно сделать, рассматривая приведённые варианты иерархий?

Видно, что по сравнению с ранними версиями названия типов при выводе стали проще (без фигурных скобок с пояснениями в них) и логичнее (без неоправданного поименования целых типов — вроде `UInt...` вместо `UInt...`). Исчезает как вариант целочисленного тип `Char`, а `BigInt` становится подтипом типа `Signed` — видимо, потому, что в представлении `BigInt` знак числа предусмотрен всегда. С определённого момента появляются величины *иррациональные* (тип `Irrational`) со своими интересными свойствами (см. стр. 11), но место их в иерархии «устоялось» не сразу; теперь они образуют отдельный подтип, получив свой родительский тип `AbstractIrrational`.

Все типы данных в *Julia* разделены на две большие группы: конкретные типы и абстрактные типы. Первые в дереве типов являются «листьями» (так как у них нет «потомков») и объекты только этих (конкретных) типов могут быть созданы, а вторые (т. е., «ветви», обладающие «потомками») необходимы для группового обозначения набора конкретных типов.

Резюмируя, можно сказать, что «встроенный» набор числовых типов — ничуть не хуже, чем у популярных в настоящее время языков для научных расчётов (*Python*, *R* и др.).

Остановимся кратко на некоторых особенностях числовых типов в языке *Julia*.

3.2. Целочисленные величины

Предельные параметры целочисленных типов *Julia* хорошо демонстрирует приводимый в [руководстве по языку](#) крошечный пример¹⁰:

```
for T in [Int8, Int16, Int32, Int64, Int128, UInt8, UInt16, UInt32, UInt64, UInt128]
    println("$ (lpad(T, 7)) : [$(typemin(T)), $(typemax(T))] ")
end
```

В нём последовательно перебираются все существующие в языке «встроенные» целочисленные типы (кроме `BigInt`) и выводятся их минимальные и максимальные значения:

```
Int8: [-128, 127]
Int16: [-32768, 32767]
Int32: [-2147483648, 2147483647]
Int64: [-9223372036854775808, 9223372036854775807]
Int128: [-170141183460469231731687303715884105728, 170141183460469231731687303715884105727]
UInt8: [0, 255]
UInt16: [0, 65535]
UInt32: [0, 4294967295]
UInt64: [0, 18446744073709551615]
UInt128: [0, 340282366920938463463374607431768211455]
```

Знаки доллара в текстовой строке вместе с круглыми скобками — признак т. н. *интерполяции*: выражения в круглых скобках вычисляются и полученные значения (имена типов с нужным числом дополняющих пробелов, а также минимальные и максимальные для типов величины) подставляются в каждую выводимую строку.

Как ни странно, у логической величины в *Julia* (являющейся, как нетрудно заметить, в иерархии типов подтипом целой и имеющей по определению логическое значение `true` или `false`) есть и числовое значение; оно может быть при необходимости получено, что позволяет применять приёмы, характерные для языка *C* (можно вычислять, например, выражения вроде `0+(2>1)` или `0>true`¹¹).

¹⁰<https://docs.julialang.org/en/latest/manual/integers-and-floating-point-numbers/>

¹¹Какие-то варианты могут оказаться невозможными, но исключительно из-за неоднозначности: скажем, не удастся вычислить значение выражения `0.+(2>1)`, поскольку здесь неясно, является ли точка частью числа `0` или же она — часть обозначения покомпонентной операции сложения; добавление пробела (вот так: `0.+(2>1)` или так: `0. +(2>1)`) легко разрешает ситуацию.

3.3. Вещественные числа

Результатами таких «неудобных» выражений как $1.0/0.0$, $1.0/(-0.0)$, $0.0/0.0$, вызывающих, например, в языке *Python* (единообразную) исключительную ситуацию деления на ноль, в *Julia* будут специальные значения `Inf`, `-Inf`, `NaN`, соответственно. В том, что это — обычные вещественные величины, легко убедиться с помощью функции `typeof()`; они также могут быть допустимой частью выражений и операции с ними будут давать осмысленные результаты ($0+\text{Inf}$ равно `Inf`, а $1_000_000_000-\text{Inf}$ ¹² — соответственно, `-Inf`).

3.4. Комплексные числа

В записи комплексных чисел в *Julia* `im` — это обозначение комплексного числа i ; может сопровождаться необходимым числовым коэффициентом (безо всякого промежуточного знака умножения), что даёт удобный синтаксис (если забыть о «двухбуквенности» мнимой единицы) записи комплексных чисел, сходный с традиционным, например: `3+4im`, `1/2im` (в последнем случае надо помнить, что константа «привязывается» до любых операций, поэтому число `1/2im` будет равно `0.0 - 0.5im`, а вовсе не `0.0 + 0.5im!`).

Интересно, что не удастся получить результат вычисления `sqrt(-1)` (возникает ошибка области определения функции), правильный способ получения результата: `sqrt(-1+0im)`.

3.5. Рациональные числа

Рациональные числа (дроби) имеют целые числитель и знаменатель и создаются из такой пары при помощи повторённой дважды наклонной черты ($1/2$) или функции `Rational()`; хранятся в сокращённом виде (без общего делителя). Если в паре будет использовано хотя бы одно вещественное число, результат вместо рационального станет вещественным.

3.6. BigInt, BigFloat

Типы `BigInt` и `BigFloat` — «обёртки» вокруг библиотек *GMP* и *MPFR*, соответственно. Они являются типами с произвольной точностью. Надо иметь в виду, что такие величины не возникают в *Julia*-программе самопроизвольно (как это было бы в случае языка *Python*), поэтому вычисление значения выражения $2^{64}-1$ даст «странный» ответ `-1`; «правильный» ответ получится лишь с выражением `big(2)^64-1` (`18446744073709551615`).

3.7. Иррациональные числа

Пока доступные иррациональные величины¹³ ограничены имеющимися в библиотеке *MPFR*. Введём, например, в *Julia* имя `pi`. В ответ будет выведено значение числа π с некоторой точностью. Неплохо... Это значит, что имя `pi` сопоставлено (без нашего участия) этой константе. Правда, если присмотреться, то окажется, что имя `pi` при выводе не использовалось, вместо него появился значок π ¹⁴. Интересно? Посмотрим, какого типа эта величина, вводя `typeof(pi)`. Получим ответ `Irrational{π}`, что можно понять так: это иррациональное число, изображаемое¹⁵ значком π . Можно даже скопировать этот значок, чтобы вызвать функцию по-другому: `typeof(π)`. Ответ не меняется — имена `pi` и `π` синонимичны.

Удобство иррационального типа (`Irrational`) в том, что величины этого типа автоматически преобразуются в наиболее подходящий по точности вариант в момент «контакта» с другим числом, т. е., либо в `BigFloat`, либо в `Float64`.

¹²Символы подчёркивания в записи числовых констант — это разделители групп десятичных разрядов.

¹³По данным `names(Base.MathConstants)` это: `e` ($e = 2.7182818284590\dots$), `pi` ($\pi = 3.1415926535897\dots$), `golden` ($\varphi = 1.6180339887498\dots$), `eulergamma` ($\gamma = 0.5772156649015\dots$), `catalan` = `0.9159655941772\dots`

¹⁴В какой степени это будет похоже на настоящий знак π , зависит от используемого в консоли шрифта.

¹⁵В языке *Julia* имеется специальная форма строки для представления имён переменных: тип `Symbol`. Создаётся из обычной строки (скажем, "ABC") вызовом `Symbol("ABC")` или при помощи двоеточия — `:ABC`.

4. Функции в языке *Julia*

Предназначение функций в различных языках программирования мало изменилось со времён появления первых языков: их задача — «скрыть» набор (повторно используемых) действий под некоторым именем, часто — сопоставляя ему вычисляемое значение, «замещающее» вызов функции в выражениях. А вот способов определения функций и вариантов передачи им необходимых величин появляется всё больше, да и сами функции теперь из просто фрагментов исполняемого кода превратились в полноценные «объекты».

Вызывается функция всегда с помощью употребления её имени вместе с круглыми скобками со списком значений параметров. Без скобок имя функции — это ссылка на объект функции, которую можно передать другим функциям в качестве значения параметра. Как и в случае с другими именами в *Julia*, символы *Unicode* могут быть использованы в качестве (частей) имён функций. Если есть необходимость, при определении функции можно указать также тип возвращаемого значения с помощью оператора `::`, тогда функция всегда будет возвращать значение такого типа, независимо от того, какие величины использовались для вычисления возвращаемого значения.

Особенностью функций в языке *Julia* является их способность иметь многочисленные конкретные реализации, которые на жаргоне *Julia* называются *методами*. Каждый такой метод специализирован для разных типов параметров функции. А это значит, что функция в *Julia* — это общее название некоторого «действия», метод же определяет «поведение» функции в зависимости от конкретной комбинации типов её параметров. Для определения метода, который должен быть вызван, используются типы **всех** позиционных (см. стр. 13) параметров функции, а сам процесс выбора метода называется *диспетчеризацией*.

Если в определении функции типы параметров не указаны, то получается общий (*generic*) метод, применимый ко всем типам параметров.

Для определения функции со многими методами она просто определяется несколько раз — с разным количеством параметров или с другими типами параметров; первое определение функции/метода создаёт «функциональный» объект, а последующие определения методов добавляют новые методы к этому объекту.

Можно также при определении функции «ограничить» возможные типы входных параметров «указаниями» вида `::<Тип>`; определённая подобным образом функция будет применима лишь тогда, когда её параметры при вызове будут иметь «предусмотренные» типы.

Большинство операторов¹⁶ языка *Julia* — это функции, поддерживающие специальный (инфиксный) синтаксис. Соответственно, они могут быть вызваны и со списком параметров в скобках (т. е., например, возможно и `+(1, 2, 3)`, и традиционное `1+2+3`). Поэтому и узнать о них многое можно с помощью функции `methods()`¹⁷, «полезной» для вариантов вызова функций, и обходиться с операторами можно так же, как и с именами функций, присваивая их или передавая при необходимости другим функциям (см., например, рис. 10 и рис. 11).

4.1. «Традиционное» определение функции

Базовый синтаксис определения функции в *Julia* использует ключевое слово `function` (вместе с завершающим ключевым словом `end`), но в остальном вполне типичен:

```
function <Имя> (<СписокПараметров>)  
    <Действия>  
end
```

Оператор возврата `return` имеется, но писать его явно зачастую нет необходимости: достаточно, чтобы возвращаемое значение было указано в определении функции последним.

¹⁶Исключение составляют операторы со специальной семантикой исполнения вроде `&&` и `||`.

¹⁷Например, в *Julia* версии *1.0.0* команда `methods(+)` выдаёт 163 метода, `methods(-)` — 175 методов, `methods(*)` — 343 метода, `methods(/)` — 104 метода.

4.2. Специальное (однострочное) присваивание

Определение несложной функции можно записать более компактно в виде специального присваивания¹⁸ (тогда ключевые слова `function` и `end` вообще не нужны, но тело функции должно быть одним выражением — возможно, составным); по сути, подобное определение показывает, чему равно возвращаемое значение функции:

```
<Имя> (<СписокПараметров>) = <Выражение>
```

4.3. Безымянные (анонимные) функции

Помимо сказанного выше функции также могут быть созданы вообще без имени, причём сделано это может быть двумя способами: либо можно опустить имя функции в базовом варианте синтаксиса, либо воспользоваться специальным синтаксисом, в котором набору параметров сопоставляется значение некоторого выражения (т. н. *anonymous functions*):

```
(<СписокПараметров>) -> <Выражение>
```

Символом «сопоставления» служит стилизованная «стрелочка» из двух символов (`->`). Круглые скобки списка параметров для единственного параметра можно не употреблять.

Например, анонимная функция возведения в квадрат может быть определена так: `(x)->x*x` (или так: `x->x*x`). Но для того, чтобы потом воспользоваться ею, надо либо ссылку на неё присвоить какой-то переменной (чтобы к функции можно было хоть как-то обратиться), либо тут же передать её как параметр при вызове какой-то другой функции, либо вызвать её сразу после определения, для чего определение заключается в круглые скобки, после которых — в других круглых скобках, — указывается значение параметра¹⁹ для вызова: `((x)->x*x)(-1)`.

Основное применение анонимных функций — при передаче их в качестве параметров.

4.4. Передача параметров функциям

В *Julia* параметры передаются функциям в виде ссылок, поэтому изменения, происходящие с ними в функциях, будут видимы «снаружи». Кроме *позиционных* параметров, имеющих фиксированный порядок следования, функции могут иметь *поименованные* (*keyword arguments*); и те, и другие могут быть *необязательными* (*optional arguments*). Возможны в *Julia* также функции с произвольным числом параметров (*varargs functions*).

Необязательные параметры (или параметры по умолчанию) довольно распространены, поскольку позволяют не передавать при вызовах те величины, которые чаще всего имеют какие-то определённые известные значения. Подобные значения указываются при определении функции — после имён соответствующих параметров и знаков равенства.

Если каким-то функциям необходимо передавать очень много параметров, удобно хотя бы часть из них сделать поименованными (точно так же, как и в языке *Python*, в *Julia* такие параметры возможны): в определении функции они отделяются от позиционных точкой с запятой; они не участвуют во множественной диспетчеризации и порядок их следования при вызове не имеет значения.

«Заголовок» одного из методов функции преобразования некоторого целого числа `n` в массив отдельных цифр по заданному основанию `base`, дополняемый при необходимости до числа элементов `pad`, выглядит в *Julia v1.0.0* так (см. файл `.../share/julia/base/intfuncs.jl` дистрибутива):

```
digits(n::Integer; base::Integer = 10, pad::Integer = 1)
```

¹⁸Далее такое определение будет условно именоваться однострочным, хотя, строго говоря, это не так.

¹⁹Именно такой способ вызова анонимных функций применяется в коде на рис. 5 (стр. 18).

Здесь параметр `n` — единственный позиционный, за ним следуют два (необязательных) поименованных (`base` и `pad`) — каждый со своим значением по умолчанию. В предшествующих версиях *Julia* параметры `base` и `pad` у этой функции были просто позиционными необязательными с такими же значениями по умолчанию. Поэтому ранее функцию можно было вызывать с одним, двумя и тремя параметрами так: `digits(2019)`, `digits(2019,3)`, `digits(2019,3,8)`, а начиная с версии *Julia v1.0.0* следует вызывать уже так: `digits(2019)`, `digits(2019,base=3)`, `digits(2019,base=3,pad=8)` (см. стр. 41), но можно также ещё и так: `digits(2019,pad=8)`.

Функции, которые должны принимать произвольное число позиционных параметров, в своём определении используют троеточие (`...`), называемое оператор *splat*. Его можно употребить после всех позиционных и необязательных параметров или в конце всего списка параметров. Можно использовать *splat* и при вызове, тогда это означает, что содержимое всего предшествующего ему массива (или кортежа) попадёт в набор параметров функции (а не один «сложный» параметр).

Определим простенькую функцию с произвольным числом параметров: `f(x...) = x`. Такое определение означает, что сколько бы этой функции не было передано параметров (их «набор» обозначен здесь именем `x`), эти параметры будут её возвращаемым значением. Поэтому ничего удивительного не должно быть в том, что из этих параметров будет создан кортеж, который в результате и будет возвращён (ниже показаны результаты эксперимента):

```

julia> f(1)           julia> f(1:5)           julia> f([1,2])
(1,)                 (1:5,)                 ([1, 2],)

julia> f(1,2)        julia> f(1:5...)         julia> f([1,2]...)
(1, 2)               (1, 2, 3, 4, 5)       (1, 2)

```

В случае одного объекта возвращается кортеж с одним элементом, а при использовании *splat* при вызове функции передаваемый ей диапазон или вектор «расщепляются» на отдельные элементы, из которых опять формируется кортеж.

4.5. Операции, применимые к функциям

В языке *Julia* к функциям могут быть применены также некоторые операции (они же — и функции) «комбинирования» функций: операции композиции (`∘`) и «конвейеризации» (`|>`). Обе довольно компактно определяются в файле `.../share/julia/base/operators.jl`²⁰.

Первая (т. н. *function composition*) определена как `∘(f,g) = (x...) -> f(g(x...))`, т. е., композиция двух функций — это последовательное (сначала — вторая, затем — первая) их применение к задаваемым аргументам; вторая (т. н. *function pipelining*) применяет функцию, заданную вторым параметром, к значению первого параметра: `|>(x,f) = f(x)`. При многократном применении операция «конвейеризации» `|>` удобна тем, что функции будут записаны в том порядке, в котором они применяются, а не в противоположном — как это было бы в случае вложенных вызовов или операции композиции.

4.6. Макроопределения (макросы)

Макроопределения в *Julia* выглядят почти как функции, но при определении используют ключевое слово `macro` вместо `function`, а их вызовы обязательно предваряются символом `@` и возможны без обязательных круглых скобок. Можно записать вызов макроса в стиле вызова функции (параметры перечисляются в круглых скобках через запятую), а можно просто записать параметры через пробел после имени макроса безо всяких скобок.

Однако — в отличие от функций — макросы вызываются не во время исполнения кода, а чуть раньше (перед компиляцией), поэтому они позволяют *Julia*-программам работать с собственным кодом, который после разбора превращён в *абстрактное синтаксическое дерево* (**AST**, abstract syntax tree). Переменные величины, с которыми предстоит работать программному коду, макросам недоступны (ибо откомпилированной программы ещё нет),

²⁰Путь к этому файлу указан относительно каталога, где установлен дистрибутив *Julia*.

зато они могут работать с фрагментами синтаксического дерева: выражениями (тип `Expr`), именами переменных и функций (тип `Symbol`) и константными величинами.

Так что можно сказать, что макросы — это специальные функции, «имеющие дело» с *AST*. Написание такого кода обработки и изменения самого кода (в отличие от данных) называется *метапрограммированием*; то, что в *Julia* оно возможно, — особенность языка.

4.7. Некоторые часто используемые макросы

@time *<ИсполняемыйКод>*

Макрос `@time` «запускает» таймер перед исполнением кода и «останавливает» его после исполнения, сообщая затраченное на это время и сопутствующий расход памяти.

@elapsed *<ИсполняемыйКод>*

«Упрощённый» вариант предыдущего макроса: формируемое в результате исполнения кода значение игнорируется; выводится лишь затраченное на исполнение время в секундах — в виде вещественного значения.

@which *<ВыражениеВызова>*

Макрос `@which` перед выражением не даёт ему вообще быть выполненным. Вместо этого он сообщает, какой метод будет использован для конкретных заданных значений параметров. Кроме того, указывается исходный файл с определением метода и соответствующая строка.

@show *<Выражение>*

Макрос `@show` обеспечит во время исполнения вывод значения указанного выражения.

@assert *<Выражение-Условие>*

Этот макрос заменяет *<Выражение-Условие>* тернарным условным оператором с данным выражением в качестве условия. Во время исполнения кода *<Выражение-Условие>* будет вычислено; если его значение окажется «ложным», будет возбуждена ошибка, сопровождаемая выводом проблемного выражения; при «истинном» значении ничего не произойдёт.

@debug *<Строка>*

Поведение этого макроса зависит от значения переменной окружения `JULIA_DEBUG`²¹: если это значение пустое или переменная вообще отсутствует, то строка с вызовом макроса игнорируется, если же это значение равно, например, `"all"`, то выводится содержимое строки/переменной, указанной как параметр макроса — вместе с именем файла и номером исходного кода строки (если всё происходит в *REPL*), что позволяет выводить текстовые сообщения или значения ключевых переменных при исполнении строки с макросом.

4.8. Несколько практических примеров функций

Далее приводятся несколько примеров, где определены различные несложные функции: «традиционные», «однотрочные» (рекурсивные и нерекурсивные), вложенные анонимные, функции со «специальными» (включающими символы *Unicode*) именами, которые могут использоваться и как символы инфиксных операций, функции с необязательными параметрами и пр., — отобранные автором из своей повседневной практики.

²¹Изменять значение переменной окружения можно как до запуска *Julia* (системными средствами), так и прямо из кода *Julia* — присваиваниями `ENV["JULIA_DEBUG"] = "all"` или `ENV["JULIA_DEBUG"] = ""`.

4.9. Пример: свойства последовательностей Коллатца

Под преобразованием $C(N)$ (функцией Коллатца) натурального числа N понимается

$$C(N) = \begin{cases} N/2, & \text{если } N - \text{чётное,} \\ 3N + 1, & \text{если } N - \text{нечётное,} \end{cases}$$

т. е., результат деления чётного числа N на два или утроенное значение нечётного N плюс единица. Далее можно рассмотреть многократное применение этого преобразования (тут уже возникает последовательность Коллатца, зависящая от первоначальной величины) и задаться простым вопросом: как ведут себя подобные последовательности при различных начальных значениях? Гипотеза Коллатца (*Collatz*, 1929) состоит в том, что независимо от значения, с которого всё начинается, в конце концов всегда будет достигнута «финальная» величина 1 — хотя промежуточные значения могут оказаться весьма большими.

Для экспериментов с последовательностями Коллатца имеет смысл определить функцию преобразования на каждом шаге (скажем, так²²: $C(n) = \text{iseven}(n) ? n \div 2 : 3n+1$)²³ и на её основе — функцию подсчёта шагов до «финала»:

$$\text{Путь}C(N) = N == 1 ? 0 : 1 + \text{Путь}C(C(N))$$

Количество шагов в последовательности $C(N)$ («путь») определяется здесь рекурсивно: «путь» от значения N до единицы на один шаг длиннее, чем «путь» до неё от следующего значения $C(N)$; рекурсия²⁴ прекращается при достижении²⁵ единичного значения N .

Можно рассматривать последовательности без чётных чисел вообще — как несущественных «промежуточных» состояний (фрагмент графа подобных переходов показан на рис. 4), и ввести соответствующее «одношаговое» преобразование $T(\cdot)$, определив его так: $T(n) = C(n) \mid > \text{ToOdd}$ (здесь употреблена операция $\mid >$ «конвейеризации», это эквивалентно определению $T(n) = \text{ToOdd}(C(n))$); везде используется «приведение» к нечётным значениям: $\text{ToOdd}(N) = \text{isodd}(N) ? N : \text{ToOdd}(N \div 2)$

Функция подсчёта числа шагов для $T(\cdot)$:

$$\text{Путь}T(N) = N == 1 ? 0 : 1 + \text{Путь}T(T(N))$$

Располагая подобными функциями подсчёта шагов, можно попробовать найти нечётные числа из какого-либо заданного диапазона, приводящие к наиболее длинным последовательностям.

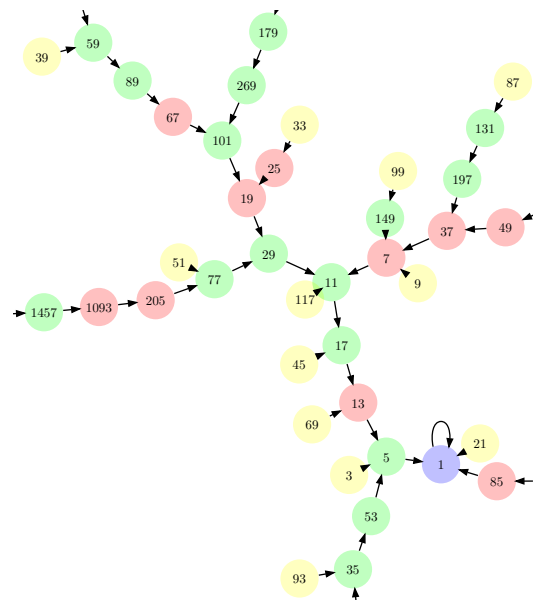


Рис. 4: Фрагмент «нечётного» графа.

²²Операция деления с использованием наклонной черты в *Julia* даже при целочисленных операндах даёт всегда вещественный результат, поэтому для получения гарантированно целого результата в тех случаях, когда деление возможно без остатка, имеется лишь две возможности: либо применить специальную функцию целочисленного деления `div()` (её аналог — операция \div), либо — в случае «удобной» величины делителя — «сдвинуть» двоичное представление числа вправо на нужное число разрядов.

²³Ввести символ операции \div с клавиатуры (в *REPL*) можно при помощи последовательности `\div` `[Tab]`.

²⁴Возможно, не совсем правильно в качестве иллюстрации использовать примеры рекурсивных функций, поскольку их «аккуратное» написание порой трудозатратно, а применение часто сопряжено с «неприятностями», но здесь это оправдано тем, что определения получаются простыми и наглядными, а исполнение — приемлемым по времени и затрачиваемым ресурсам.

²⁵Может показаться, что такое определение слишком «самонадеянно», т. к. не учитывает, что единичное значение может и не встретиться (гипотеза Коллатца ведь никем пока не доказана!), но реальная «опасность» здесь немного другая: промежуточные значения в последовательности могут получаться весьма большими по сравнению с исходным значением величин (см., например, стр. 17), поэтому вполне возможно переполнение в процессе вычисления отдельных промежуточных значений последовательности.


```
julia> maximum([(ПутьC(i),i) for i in 1:2:99])
(118,97)
```

```
julia> maximum([(ПутьC(i),i) for i in 1:2:999])
(178,871)
```

```
julia> maximum([(ПутьC(i),i) for i in 1:2:9_999])
(261,6171)
```

```
julia> maximum([(ПутьC(i),i) for i in 1:2:99_999])
(350,77031)
```

```
julia> maximum([(ПутьT(i),i) for i in 1:2:99])
(43,97)
```

```
julia> maximum([(ПутьT(i),i) for i in 1:2:999])
(65,871)
```

```
julia> maximum([(ПутьT(i),i) for i in 1:2:9_999])
(96,6171)
```

```
julia> maximum([(ПутьT(i),i) for i in 1:2:99_999])
(129,77031)
```

Таким образом, среди нечётных чисел первой сотни самая длинная последовательность Коллатца будет у 97: 118 чисел (43 — для нечётных значений), из первой тысячи — у 871 и т. д.

Можно, кстати, временно переопределить функцию `ToOdd()`, дополняя «финальное» значение числа N предварительным выводом его на печать (с последующей точкой с запятой, играющей роль разделителя нескольких операторов языка) и заключением всего в круглые скобки для образования «составного» выражения со значением N : `ToOdd(N) = isodd(N) ? (print("$N"); N) : ToOdd(N÷2)` — тогда в рабочей области *Julia* определение этой функции станет другим, поэтому результаты всех её вызовов из других функций будут сопровождаться добавленным «промежуточным» выводом. Для того, чтобы «подавить» вывод «окончательного» результата для всех выражений в строке *REPL* (и оставить только «вживлённый» промежуточный вывод), необходимо завершать выражения точкой с запятой, например, для вывода *T*-«траектории» числа 9:

```
julia> ПутьT(9);
7 11 17 13 5 1
```

или числа 27:

```
julia> ПутьT(27);
41 31 47 71 107 161 121 91 137 103 155 233 175 263 395 593 445 167 251 377 283
425 319 479 719 1079 1619 2429 911 1367 2051 3077 577 433 325 61 23 35 53 5 1
```

При этом выражения вроде `maximum([(ПутьT(i),i) for i in 1:2:101])`; будут выводить «склеенные» последовательности для всех промежуточных результатов — до тех пор, пока определение `ToOdd()` снова не изменится...

Если ввести понятие *предшественника* какого-либо значения, понимая под ним такое число, преобразование $T(\cdot)$ которого даёт это значение, то тогда нетрудно доказать следующий факт.

Утверждение. Пусть $k \in \mathbb{N}$ является *предшественником* некоторого значения, тогда и все $Q(Q(\dots Q(k)\dots))$, где $Q(k) = 4k + 1$, тоже являются *предшественниками* этого же значения²⁶.

Например, у 1 (см. фрагмент «нечётного» графа на рис. 4) предшественниками будут 1, 5, 21, 85, 341 и т. д., у 5 — 3, 13, 53, 213 и т. д., у 13 — 17, 69, 277 и т. д.; значения, делящиеся нацело на 3, предшественников не имеют (легко доказывается от противного).

Приводимые рекурсивные варианты функций вполне работоспособны, но увеличивают расход оперативной памяти; если это неприемлемо, лучше использовать нерекурсивные.

Нерекурсивный вариант `ToOdd()` получается менее наглядным, но столь же простым:

```
ToOdd(N) = (while iseven(N) N >>= 1 end; N)
```

Здесь `end` — завершитель цикла `while`, а точка с запятой разделяет цикл и возвращаемый результат (значение N). Круглые скобки «оформляют» всё как единое выражение; без них будет выдано сообщение об ошибке (`"N not defined"`).

Чуть более «тяжеловесным» (в основном, из-за неизбежной локальной переменной) будет выглядеть нерекурсивный вариант функции `ПутьT()`:

```
ПутьT(N) = (k = 0; while N > 1 k += 1; N = ToOdd(C(N)); end; k)
```

²⁶Можно сформулировать это по-другому: все предшественники какого-либо значения имеют в двоичном представлении один и тот же *префикс*; различаются они все лишь *суффиксами*, построенными из разного количества (от нуля и более) фрагментов **01**. Например,

$$1_{10} = 1_2, 5_{10} = 101_2, 21_{10} = 10101_2, 85_{10} = 1010101_2, 341_{10} = 101010101_2, \dots$$

4.10. Пример: реализация классического метода Рунге–Кутты

На материале, который можно считать общеизвестным (классический метод Рунге–Кутты решения задачи Коши для обыкновенных дифференциальных уравнений первого порядка вида $y' = f(x, y)$, $x \in [x_0, b]$ студентам факультета сообщается уже на первом курсе), здесь демонстрируются синтаксические возможности *Julia* в части различных способов определения функций (в том числе и безымянных). Как известно, формулы классического метода Рунге–Кутты — это набор выражений, вычисляемых при $i = 0, \dots, n - 1$, например:

$$\begin{aligned} \mathbf{k}_1 &= f(x_i, y_i), \\ \mathbf{k}_2 &= f(x_i + \frac{h}{2}, y_i + \frac{h}{2}\mathbf{k}_1), \\ \mathbf{k}_3 &= f(x_i + \frac{h}{2}, y_i + \frac{h}{2}\mathbf{k}_2), \\ \mathbf{k}_4 &= f(x_i + h, y_i + h\mathbf{k}_3), \\ y_{i+1} &= y_i + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \end{aligned}$$

С их помощью можно вычислить приближённые значения неизвестной функции $y(x)$ по известной функции $f(x, y)$, заданному значению $y(x_0) = y_0$ и выбранному шагу h , используя приближающие угловые коэффициенты $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3, \mathbf{k}_4$ функции $f(x, y)$ (точнее, угловые коэффициенты четырёх разных интегральных кривых) в трёх точках: $x_i, x_i + \frac{h}{2}, x_{i+1} = x_i + h$.

Поскольку алгоритмы Рунге–Кутты можно записать по-разному (вычисляя либо угловые коэффициенты, либо приращения искомой функции), мы воспользуемся этим, чтобы немного отойти от приводимой на странице сайта *Rosetta Code*²⁷ реализации и написать всё в соответствии с приведёнными выше формулами, но в обозначениях, принятых на сайте (т. е., в качестве независимой переменной используя t , а не x , а шаг h обозначая Δt).

Примерно так же может быть записано всё и в языке *Python*, тоже поддерживающем анонимные функции, но в нём употребляется для этого дополнительное ключевое слово `lambda`, а *Julia* в определении анонимной функции «обходится» лишь скобками и стилизованной стрелочкой, поэтому даже такой сложный (и вложенный) набор функций не очень сильно отличается от математических формул²⁸, а потому легче воспринимается, — хотя и требует пояснений.

```
rk4(f) = (t, y, Δt) ->
  ((k1) ->
    ((k2) ->
      ((k3) ->
        ((k4) -> Δt * (k1 + 2k2 + 2k3 + k4) / 6
          ) (f(t + Δt, y + Δt * k3))
        ) (f(t + Δt / 2, y + Δt * k2 / 2))
      ) (f(t + Δt / 2, y + Δt * k1 / 2))
    ) (f(t, y))
```

Рис. 5: Полный текст функции `rk4()`.

`rk4()` — это (почти) однострочное определение функции *Julia* (зависящей от функции `f()`), для удобства восприятия и разбора записанное в несколько строк (что допустимо).

Начнём разбор с самого «внешнего» определения, временно игнорируя то, что написано «внутри» («внутренность» тут пока заменена многоточием):

```
rk4(f) = (t, y, Δt) -> ( ... ) ( f(t, y) )
```

Параметры

Возвращаемое значение

Рис. 6: «Внешние» определения в функции `rk4()`.

²⁷Этот сайт содержит многочисленные алгоритмы, записанные на различных языках программирования.

²⁸Здесь также «помогает» возможность языка *Julia* использовать «нестандартные» имена переменных.

Итак, мы видим однострочное определение функции `rk4()` с использованием приводимого после символа присваивания выражения; имя функции — `rk4`, её параметр — `f` (то, что это тоже должна быть функция, на время забудем). Само же выражение в свою очередь является определением анонимной функции (об этом нам напоминает стилизованный знак отображения «стрелочка»: данное определение задаёт отображение набора параметров в возвращаемое анонимной функцией значение, которое формируется как результат вызова функции, «спрятанной» за многоточием). Таким образом, функция `rk4()` *возвращает ссылку на определяемую анонимную функцию*, принимающую параметры `t`, `y`, `Δt`.

Теперь уже становится понятно, что не очень простое выражение рис. 5 содержит определения нескольких функций (поименованной функции `rk4()`, анонимной функции с тремя параметрами и ещё четырёх анонимных функций с одним параметром, называемых далее #1, #2, #3, #4, см. рис. 7), а часть определений использует также и вызовы некоторых из этих функций.

```
rk4(f) = (t, y, Δt) -> #1
  ((k1) -> #2
    ((k2) -> #3
      ((k3) -> #4
        ((k4) -> Δt * (k1 + 2k2 + 2k3 + k4) / 6
          ) (f(t + Δt, y + Δt * k3))
        ) (f(t + Δt / 2, y + Δt * k2 / 2))
      ) (f(t + Δt / 2, y + Δt * k1 / 2))
    ) (f(t, y))
```

Рис. 7: «Расшифровка» текста функции `rk4()`.

`rk4(f)` — это определение функции, зависящей от одного параметра `f` и являющейся ссылкой на определение безымянной функции от трёх параметров (`t`, `y`, `Δt`), значения которой формируются вызовом первой безымянной функции (#1) одного параметра `k1` при значении параметра, равном `f(t, y)`. В свою очередь, значение первой безымянной функции является результатом вызова второй (#2) безымянной функции (с одним параметром `k2`) при значении параметра, равном `f(t + Δt / 2, y + Δt * k1 / 2)`. Вторая безымянная функция получает своё значение в результате вызова третьей (#3) безымянной функции (с одним параметром `k3`) при значении этого параметра, равном `f(t + Δt / 2, y + Δt * k2 / 2)`. Третья безымянная функция возвращает значение, получаемое при вызове четвёртой (#4) безымянной функции (с одним параметром `k4`) со значением `f(t + Δt, y + Δt * k3)`. Сама же четвёртая безымянная функция сопоставляет своему параметру значение $\Delta t * (k_1 + 2k_2 + 2k_3 + k_4) / 6$ ²⁹. При вычислении этого выражения доступны как все параметры `k1`, `k2`, `k3`, `k4` (а каждый предшествующий из них доступен в последующих безымянных функциях), так и параметр `Δt` (фиктивный параметр определения безымянной функции с тремя параметрами), — поскольку каждая новая безымянная функция определена в рамках области определения предыдущей, равно как доступна во всех безымянных функциях и передаваемая при вызове `rk4()` функция, обозначенная здесь фиктивным параметром `f`, — поскольку является параметром определения функции `rk4()`, в рамках которого задаются все остальные безымянные функции.

С помощью приведённого выше определения функции `rk4()` можно создать тестовую программку (см. рис. 8) численного решения конкретного дифференциального уравнения (в качестве которого выбрано использованное в примере на сайте уравнение $y' = t\sqrt{y}$ с известным точным решением $y(t) = \frac{1}{16}(t^2 + 4)^2 + C$).

Функция (безымянная) для вычисления приращения искомой функции $y(t)$ возникает при исполнении строки программы `Δy = rk4(f)`, где функции `rk4()` передаётся (в виде ссылки) функция `f()` (равная в данном случае `t*sqrt(y)`). В результате вызова `rk4()` возвращается ссылка на «сконструированную» безымянную функцию и после выполнения присваивания у неё появится имя `Δy`.

Фиксируя в переменных `t0`, `Δt`, `tmax` и `y0` необходимые начальные значения и создавая две новые переменные `t` и `y` (инициализированные в соответствии с начальными условия-

²⁹В этом выражении использована возможность записи в *Julia* числовых коэффициентов при переменных без употребления символа операции умножения.

ми), можно далее осуществлять т. н. численное интегрирование этого дифференциального уравнения. В цикле по времени последовательно вычисляются приращения искомой функции y и момента времени t и модифицируют как y , так и t .

```

using Printf

rk4(f) = (t, y, Δt) ->
    ((k1) ->
        ((k2) ->
            ((k3) ->
                ((k4) -> Δt * (k1 + 2k2 + 2k3 + k4) / 6
                ) (f(t + Δt, y + Δt * k3))
            ) (f(t + Δt / 2, y + Δt * k2 / 2))
        ) (f(t + Δt / 2, y + Δt * k1 / 2))
    ) (f(t, y))

f(t, y) = t * sqrt(y) # Exact solution: (t^2+4.0)^2/16.0

Δy = rk4(f)

t0, Δt, tmax = 0.0, 0.1, 10.0
y0 = 1.0
t, y = t0, y0

while t ≤ tmax
    global t, y
    if t ≈ round(t)
        @printf("y(%4.1f) = %10.6f (error: %12.6e)\n",
            t, y, abs(y - (t^2+4.0)^2/16.0))
    end
    y += Δy(t, y, Δt)
    t += Δt
end

```

Рис. 8: Полный текст программы RK4.jl, использующей rk4().

Понадобится ввести (с завершающим нажатием клавиши `Tab`!): `\DeltaTab` — для символа «дельта», `\leTab` — для символа «меньше или равно», `\approxTab` — для символа приближённого равенства, а также `_0Tab` (или `_1Tab`, `_2Tab`, `_3Tab`, `_4Tab` и т. п.) — для нижних индексов переменных.

Для контроля получаемых значений полезно выводить текущие значения вычисляемых величин или сохранять их в массивах; здесь выбран первый (иллюстративный) способ; он реализован с помощью макроса `@printf`. Выдача значений производится на экран в моменты времени, примерно соответствующие целому числу секунд.

Стоит отметить, что в более ранних версиях *Julia* эта программа работала без использования двух строк: `using Printf` и `global t, y` — потому, что макрос `@printf` ранее не располагался в отдельном подгружаемом модуле, а глобальные переменные были доступными для записи без явного объявления.

```

julia> include("RK4.jl")
y( 0.0) =  1.000000 (error: 0.000000e+00)
y( 1.0) =  1.562500 (error: 1.457219e-07)
y( 2.0) =  3.999999 (error: 9.194792e-07)
y( 3.0) = 10.562497 (error: 2.909562e-06)
y( 4.0) = 24.999994 (error: 6.234909e-06)
y( 5.0) = 52.562489 (error: 1.081970e-05)
y( 6.0) = 99.999983 (error: 1.659460e-05)
y( 7.0) = 175.562476 (error: 2.351773e-05)
y( 8.0) = 288.999968 (error: 3.156520e-05)
y( 9.0) = 451.562459 (error: 4.072316e-05)
y(10.0) = 675.999949 (error: 5.098329e-05)

```

4.11. Пример: оценка значения числа π методом Монте-Карло

Это — один из часто встречающихся примеров в различных языках программирования. Здесь он иллюстрирует возможности определения однострочных и безымянных функций в *Julia*, способ задания массива с помощью закона построения его элементов (*array comprehension*), используемый также в языке *Python*, и (две) функции работы с массивами.

Функции, осуществляющей оценку, передаётся параметр размера, используемый в ней и как размер, и как обозначение границы диапазона (поскольку в *Julia* размер массива совпадает с номером/индексом его последнего элемента); выглядеть она может вот так:

```
Pi(N) = 4sum([1 for _ in filter(_->rand()^2+rand()^2<1, 1:N)])/N
```

Параметр N фактически определяет количество проводимых «испытаний»; для пояснения деталей приведённого фрагмента будем последовательно выделять его «значимые» части.

Стоит пояснить роль применённого в выражении (даже дважды) символа подчёркивания, обозначающего имена двух различных переменных, для которых важно не использование их имён, а просто их наличие по синтаксическим правилам. В первом случае это переменная цикла, во втором случае это фиктивная переменная анонимной функции. Поскольку в обоих случаях переменные эти никак не используются, каждой из них можно дать минимально возможное имя (один допустимый символ, не «нарушающий» восприятие выражения), причём одно и то же, поскольку обе они имеют (непересекающиеся) локальные области действия.

```
Pi(N) = 4sum([1 for _ in filter(_->rand()^2+rand()^2<1, 1:N)])/N
```

Это однострочное определение функции, осуществляющей суммирование отличных от нуля элементов массива и возвращающей учетверённое среднее значение этой суммы (с учётом того, что в *Julia* числовые коэффициенты часто могут быть записаны без символа операции умножения).

```
Pi(N) = 4sum([1 for _ in filter(_->rand()^2+rand()^2<1, 1:N)])/N
```

Массив (содержащий какое-то количество единиц) возникает как результат «фильтрации» передаваемого функции `filter()` массива из N последовательных целых чисел

```
Pi(N) = 4sum([1 for _ in filter(_->rand()^2+rand()^2<1, 1:N)])/N
```

при помощи *предиката* — безымянной функции, вызываемой для каждого из элементов массива, в результате чего в массиве остаются только те элементы, для которых значение предиката было истинным.

```
Pi(N) = 4sum([1 for _ in filter(_->rand()^2+rand()^2<1, 1:N)])/N
```

Каждый вызов этой анонимной функции (с единственным, но неиспользуемым параметром с именем `_`) включает в себя два вызова функции получения вещественной случайной величины `rand()` (распределена равномерно на отрезке $[0, 1)$), по результатам которых формируется сумма квадратов этих величин и сравнивается с 1 (тем самым фиксируя попадание случайной точки из единичного квадрата в «четвертушку» круга с центром в точке $(0, 0)$).

Результаты тестирования предложенной функции показаны ниже; следует иметь в виду, что генерируемые результаты случайны, а потому едва ли совпадут с получаемыми читателями. Обратите также внимание на возможность разделения в языке *Julia* групп цифр «длинной» числовой константы с помощью символа подчёркивания.

```
julia> Pi(N) = 4sum([1 for _ in filter(_->rand()^2+rand()^2<1,1:N)])/N
Pi (generic function with 1 method)

julia> Pi(1000)
3.2

julia> Pi(100_000)
3.13144

julia> Pi(10_000_000)
3.1419388
```

4.12. Пример: умножение матриц в идемпотентных алгебрах

В то время как классическая алгебра построена на тройке $(\mathbb{R}, +, \times)$, где \mathbb{R} — множество вещественных чисел, а $+$ и \times — операции сложения и умножения в нём, имеются и альтернативные варианты алгебр, определяемые на других множествах и с другими операциями «сложения» \oplus и «умножения» \otimes (приоритет «умножения» над «сложением» сохраняется).

Скажем, *max-plus*-алгебра — множество вещественных чисел \mathbb{R} с добавлением значения $-\infty$, операцией $\max()$ в качестве «сложения» и обыкновенным сложением в роли «умножения», что кратко можно записать так: $\mathbb{R}_{\max,+} = (\mathbb{R} \cup \{-\infty\}, \oplus = \max, \otimes = +)$. Присоединение к множеству вещественных чисел отдельного значения $-\infty$ вызвано тем, что для операции «сложения», роль которой в различных *max*-алгебрах исполняет операция $\max()$,

$$x \oplus y \stackrel{\text{def}}{=} \max(x, y)$$

именно $-\infty$ является «нейтральным» («нулевым») элементом (для «умножения» в *max-plus*-алгебре это значение также считается «поглощающим», т. е., $x \otimes -\infty = -\infty \otimes x = -\infty$).

Алгебраическая структура $(\mathcal{B}, \oplus, \otimes)$, где \mathcal{B} — упорядоченное множество с коммутативными и ассоциативными операциями $\oplus = \max$, $\otimes = \min$, именуется *bottleneck*-алгебра (иногда — *max-min*-алгебра). Наиболее распространённые случаи *bottleneck*-алгебр — множество целых чисел \mathbb{Z} и множество вещественных чисел \mathbb{R} с естественным порядком; другие варианты — $([0, 1], \max, \min)$ (*fuzzy*-алгебра), $(\{0, 1\}, \vee, \wedge)$ (*булевская* алгебра).

Под *max*-алгеброй $\mathbb{R}_{\max,\times}$ понимается множество неотрицательных чисел \mathbb{R}_+ с операцией «сложения» $x \oplus y := \max(x, y)$ и «умножения» $x \otimes y := xy$: $\mathbb{R}_{\max,\times} = (\mathbb{R}_+, \oplus = \max, \otimes = \times)$ ³⁰.

В отечественной литературе используется также термин *идемпотентные алгебры*; это связано с тем, что «сложение» в подобных системах (реально — полукольцах) является идемпотентным (т. е., $x \oplus x = x$). Причины интереса к подобным алгебрам в том, что, как оказывается, «многие классические задачи оптимизации... сводятся в терминах идемпотентной алгебры к решению линейных уравнений, нахождению собственных чисел и векторов линейного оператора и тому подобным вычислениям.» (Н.К. Кривулин. *Методы идемпотентной алгебры в задачах моделирования и анализа сложных систем*. СПб. : Изд-во СПбГУ, 2009).

Операции \oplus и \otimes в любой из этих алгебр обычным образом расширяются на векторы (одномерные наборы величин) и матрицы (двумерные наборы величин), причём операция «умножения» матриц тоже имеет «привычный» вид:

$$c_{ij} = \bigoplus_{k=1}^n a_{ik} \otimes b_{kj},$$

но совершенно другой смысл — со своими неочевидными результатами.

Рассмотрим теперь такой прагматический вопрос как выполнение вычислений в этих алгебрах. Операции со скалярными и поэлементные операции с векторными величинами могут быть часто проделаны «в уме» и результаты их интуитивно понятны, чего нельзя сказать об «умножении» матриц друг на друга (в соответствии с приведённой формулой и учётом смысла операций) или просто «умножении» матрицы на вектор. Для экспериментов с такими алгебрами удобно использовать какой-нибудь язык программирования. *Julia* для этой цели вполне подходит, тем более, что векторы и матрицы в языке имеются; остаётся реализовать нужные операции. В качестве иллюстрации возможностей языка остановимся здесь только на одной из них — операции умножения матриц. Разумеется, эти вычисления могут быть записаны практически в любом языке программирования. Но вот сопутствующие дополнительные возможности («математические» названия переменных и функций, применимость «символьных» имён функций как знаков операций) выгодно отличают *Julia* от других языков³¹. Рассмотрим первый вариант подобной функции — конкретно для *bottleneck*-алгебры (см. рис. 9).

³⁰Иногда $\mathbb{R}_{\max,\times}$ называется также *max-prod*-алгеброй; она изоморфна алгебре $\mathbb{R}_{\max,+}$ ($x \mapsto \ln x$).

³¹Справедливости ради следует отметить, что в языке *APL* вообще не понадобилась бы никакая дополнительная функция, поскольку подобные операции были предусмотрены в нём задолго до появления интереса к упомянутым алгебрам; в частности, функция, разбираемая здесь, может быть прямо реализована выражением $A \uparrow \cdot B$ с употреблением символов необходимых операций \uparrow (\max) и \downarrow (\min) в т. н. *внутреннем произведении* языка *APL* (для обычного умножения матриц выражение будет, соответственно, $A + \cdot \times B$).


```

function ⊗(A::Array{Int,2},B::Array{Int,2})
    @assert size(A)[end] == size(B)[1]
    S = Array{Int,2}(undef,size(A)[1],size(B)[end])
    for i = 1:size(A)[1], j = 1:size(B)[end]
        S[i,j] = typemin(Int)
        for k = 1:size(A)[end]
            S[i,j] = max(S[i,j],min(A[i,k],B[k,j]))
        end
    end
    return S
end

```

Рис. 9: Вариант функции «умножения» целочисленных матриц.

Эта функция мало отличается от записи алгоритма обычного перемножения матриц на каком-либо языке программирования — потому что (двойной вложенный) цикл перебора элементов результирующей матрицы тут присутствует (хотя и в специфическом *Julia*-варианте³²) и имеется инициализация вычисляемого значения каждого элемента нейтральным для операции «суммирования» значением с последующим «добавлением» (в смысле операции «суммирования») очередного «произведения» элементов матриц-«сомножителей».

Здесь `typemin(Int)` — это минимально возможная величина используемого реализацией *Julia* целочисленного типа (она зависит от разрядности процессора), играющая в рассматриваемой алгебре роль «нейтрального» элемента («нуля») для операции «сложения», каковой в данном случае является операция взятия максимума `max()`. Этой минимальной величиной инициализируется при вычислении каждый элемент результирующего массива `S` — чтобы в дальнейшем правильно формировались значения этих элементов, т. к. в процессе вычисления текущее значение вычисляемого элемента (`S[i,j]`) является и операндом, и затем следующим промежуточным результатом (`S[i,j] = max(S[i,j], <ОчередноеПроизведение>)`). Роль «произведения» играет операция определения минимума двух значений `min()` (`min(A[i,k],B[k,j])`). Макрос `@assert` осуществляет проверку «совместимости» размеров матриц во время выполнения функции: если размеры матриц при её вызове не согласуются, исполнение функции прекращается с указанием условия, которое не выполняется.

Если есть желание поэкспериментировать сразу с несколькими вариантами алгебр, то для «умножения» матриц можно определить одну максимально общую функцию (рис. 10).

```

function ⊗(A::Matrix{T},B::Matrix{T},⊕=+,⊗=*,∅=0) where T<:Number
    @assert size(A)[end] == size(B)[1]
    S = Matrix{T}(undef,size(A)[1],size(B)[end])
    for i = 1:size(A)[1], j = 1:size(B)[end]
        S[i,j] = ∅
        for k = 1:size(A)[end]
            S[i,j] = ⊕(S[i,j],⊗(A[i,k],B[k,j]))
        end
    end
    return S
end

```

Рис. 10: Функция «обобщённого умножения» матриц.

Операции, используемые в приводимой выше «общей» функции, сделаны её параметрами — чтобы их можно было передавать ей и тем самым осуществлять заданные действия; здесь учтено и то, что имена функций и символы операций часто взаимозаменяемы в *Julia*. Также по сравнению с частным вариантом тип массива `Array{Int,2}` заменён на синонимичный `Matrix{T}`, но с параметром типа `T`, — так, чтобы определяемая функция (операция) работала как с целочисленными, так и вещественными значениями. «Нейтральное» значение для операции «сложения» тоже сделано параметром, поскольку его не столь просто определить программно по переданной операции.

³²В том, что подобный «кратный» цикл исполняется вполне ожидаемо, легко убедиться, выполняя строку вроде такой: `for i=1:4, j=1:3, k=1:2 println(i,j,k) end`.

На основе такой общей функции могут быть созданы нужные варианты (см. рис. 11):

```
⊗1(A::Matrix{T},B::Matrix{T}) where T<:Number = ⊗(A,B,max,min,typemin(T))
⊗2(A::Matrix{T},B::Matrix{T}) where T<:Number = ⊗(A,B,max,+,typemin(T))
⊗3(A::Matrix{T},B::Matrix{T}) where T<:Number = ⊗(A,B,max)
```

Рис. 11: Умножение матриц в *bottleneck*-, *max-plus*- и *max*-алгебрах.

\otimes_1 , \otimes_2 и \otimes_3 — это функции (их имена также могут служить знаками операций) для работы с матрицами из элементов типа T , унаследованного от абстрактного численного типа $Number$, включающего, в частности, все целочисленные и все вещественные типы величин. Первая функция (операция) соответствует перемножению матриц в *bottleneck*-алгебре (она же — *fuzzy*-алгебра, если величины в обеих матрицах — из диапазона $[0, 1]$), вторая — в *max-plus*-алгебре, третья — в *max*-алгебре. В определении третьей функции учтено, что «нейтральное» значение для операции «сложения» (на самом деле — операции максимума) есть 0 (в силу неотрицательности значений в \mathbb{R}_+), что совпадает с нейтральным значением для обычного сложения, а «умножение» в *max*-алгебре является обычным умножением.

Теперь введённые функции-операции \otimes , \otimes_1 , \otimes_2 , \otimes_3 легко проверить «в деле», задавая им для тестирования небольшие целочисленные или вещественные матрицы: $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$, $C = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix}$, $D = \begin{pmatrix} 0.5 & 0.6 \\ 0.7 & 0.8 \end{pmatrix}$.

```
julia> A = [1 2; 3 4];          julia> C = [0.1 0.2; 0.3 0.4];
julia> B = [5 6; 7 8];          julia> D = [0.5 0.6; 0.7 0.8];
julia> A⊗B                       julia> C⊗D
2×2 Array{Int64,2}:             2×2 Array{Float64,2}:
 19  22                          0.19  0.22
 43  50                          0.43  0.5

julia> A*B                       julia> C*D
2×2 Array{Int64,2}:             2×2 Array{Float64,2}:
 19  22                          0.19  0.22
 43  50                          0.43  0.5

julia> A⊗1B                     julia> C⊗1D
2×2 Array{Int64,2}:             2×2 Array{Float64,2}:
 2  2                             0.2  0.2
 4  4                             0.4  0.4

julia> A⊗2B                     julia> C⊗2D
2×2 Array{Int64,2}:             2×2 Array{Float64,2}:
 9  10                            0.9  1.0
11 12                            1.1  1.2

julia> A⊗3B                     julia> C⊗3D
2×2 Array{Int64,2}:             2×2 Array{Float64,2}:
14 16                            0.14  0.16
28 32                            0.28  0.32
```

Нетрудно убедиться, что функции работают правильно, причём «имена» этих функций можно использовать как символы новых операций (заранее это было неочевидно!), что чрезвычайно удобно и наглядно, — хотя и требует дополнительных действий при вводе выражений.

Ввести символы операций можно (в *REPL* или «Блокноте») с помощью соответствующих \LaTeX -последовательностей, завершаемых клавишей Tab : \otimes_1 — $\backslash\otimes\text{Tab}\backslash_1\text{Tab}$, \otimes_2 — $\backslash\otimes\text{Tab}\backslash_2\text{Tab}$, \otimes_3 — $\backslash\otimes\text{Tab}\backslash_3\text{Tab}$, \boxplus — $\backslash\boxplus\text{Tab}$, \boxtimes — $\backslash\boxtimes\text{Tab}$, \emptyset — $\backslash\varnothing\text{Tab}$ и т. д.

Если «экзотические» символы понадобятся повторно, то не обязательно вводить их заново, достаточно просто скопировать с экрана или из текста (в кодировке *UTF-8*).

4.13. Пример: вычисление меры близости между ранжированиями

Ранжирования — отношения линейного порядка (см. Б.Г. Литвак. *Экспертная информация*. М., «РиС», 1982), представляющие, к примеру, экспертную информацию о предпочтительности альтернатив; посмотрим, как работать с этим в *Julia*. В примере (стр. 43) заданы такие ранжирования:

$$P_1 = \begin{pmatrix} a_2 \\ a_5 \\ a_1 \sim a_4 \\ a_3 \end{pmatrix}, \quad P_2 = \begin{pmatrix} a_2 \sim a_3 \\ a_1 \sim a_5 \\ a_4 \end{pmatrix}$$

В первом указано, что альтернатива a_2 предпочтительнее альтернативы a_5 , альтернатива a_5 предпочтительнее равноценных альтернатив a_1 и a_4 и каждая из альтернатив a_1, a_4 предпочтительнее альтернативы a_3 . Во втором ранжировании содержится информация о том, что a_2 и a_3 (равноценные) предпочтительнее a_1 и a_5 (тоже равноценных), а каждая из a_1, a_5 предпочтительнее a_4 .

Представлять ранжирования программно можно по-разному, но в конце концов численно всё будет сводиться к чему-нибудь вроде набора «оценок», являющихся по сути относительными весами альтернатив, поэтому будем считать, что $P_1 = [2, 4, 1, 2, 3]$; $P_2 = [2, 3, 3, 1, 2]$; (альтернативам здесь соответствует индекс массива, а в каждом массиве располагаются их относительные веса, например, для P_1 относительный вес a_2 — наибольший, скажем, 4, далее в ранжировании следует a_5 , пусть её вес будет 3, a_1 и a_4 — вес 2 и, наконец, a_3 — вес 1).

По любому представленному в таком виде ранжированию P можно построить его матрицу отношений $M(P)$:

$$M(P) = [\text{sign}(P[i]-P[j]) \text{ for } i=1:\text{length}(P), j=1:\text{length}(P)]$$

Здесь функция формирования матрицы отношений возвращает последнюю, создавая её при помощи т. н. *array comprehension*.

Для заданных выше ранжирований P_1 и P_2 их матрицы отношений $M(P_1)$ и $M(P_2)$ равны

```
julia> M(P1)
5×5 Array{Int64,2}:
 0  -1  1  0  -1
 1   0  1  1  1
-1  -1  0  -1 -1
 0  -1  1  0  -1
 1  -1  1  1  0

julia> M(P2)
5×5 Array{Int64,2}:
 0  -1  -1  1  0
 1   0  0  1  1
 1   0  0  1  1
-1  -1  -1  0  -1
 0  -1  -1  1  0
```

Мера близости между двумя ранжированиями может быть вычислена через их матрицы отношений A и B с помощью функции (т. н. *Kemeny distance, расстояние Кемени*):

$$d(A, B) = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n |a_{ij} - b_{ij}|$$

и реализована в *Julia* через заданные «оценками» ранжирования P_1 и P_2 , например, так:

```
d(P1, P2) = sum(abs.(M(P1) - M(P2))) ÷ 2
```

Рис. 12: Вычисление расстояния Кемени в *Julia*.

В приводимом на рис. 12 выражении для вычисления расстояния Кемени функция `abs()` употребляется с точкой после имени, чтобы (скалярная) операция взятия абсолютной величины была применена поэлементно; в отличие от неё операция вычитания и функция суммирования элементов `sum()` применимы и к массивам, а потому в дополнительной точке не нуждаются.

Мера близости между P_1 и P_2 , вычисляемая функцией `d()` (рис. 12), равна 9 (т. е., отличается от приведённой в книге). Нетрудно убедиться, что это значение — правильное.

```
julia> d(P1, P2)
9
```

5. Пакеты в языке *Julia*

Программы *Julia* — помимо ввода их в *REPL* (вручную или через буфер обмена) — могут сохраняться в виде файлов (которые имеют расширение `.jl`), а также модулей и пакетов. Код из файла можно подгрузить с помощью `include(<ИмяФайла>)` из текущего каталога; это ничем не отличается от ввода этого кода вручную. Однако, разумнее всего оформлять создаваемый код в виде модулей, а впоследствии — пакетов.

5.1. Модули и пакеты

Модуль — это набор определений, переменных и функций, объединённых под некоторым общим именем в каком-то файле; фактически, он представляет собой новое пространство имён для предотвращения конфликтов имён глобальных переменных и функций с другими глобальными переменными и функциями. В нём указано, какие другие модули он использует, какие имена типов и функций экспортирует, какие исходные файлы в себя включает.

Можно также сказать, что модуль — это «загрузочная» единица, поскольку получение доступа к новым переменным и функциям происходит после подгрузки нужного модуля в «рабочее пространство» *Julia* (модули предварительно должны быть установлены, если их ранее не было; проверить, какие модули уже установлены в используемой версии, можно с помощью команд `status` или `st` из интерфейса менеджера пакетов, см. далее).

Модули `Base` и `Core` всегда доступны в *Julia*. Определённый набор стандартных модулей (`stdlib`) также присутствует после установки, но они не загружаются автоматически в начале сессии. Если какие-то модули из этого набора необходимы (скажем, `Printf` или `LinearAlgebra`), они могут быть загружены обычным образом (через `using` или `import`).

С помощью `using` загружаются экспортированные/указанные функции модуля и они доступны без указания его имени; добавить новые методы к загруженным функциям нельзя. Если же это необходимо, понадобится `import`, но тут для доступа придётся указывать имя модуля.

Пакет (*package*) — это «дистрибутивная» единица, т. е., распространяемый³³ как единое целое набор модулей, реализующий какие-то расширения *Julia*. Для организации пакетов и управления ими *Julia* использует программу управления (и одноимённый протокол) `git`. По соглашению, все пакеты хранятся в `git`-репозиториях и снабжены суффиксом `.jl`.

Встроенный в *Julia* менеджер пакетов `Pkg` позволяет устанавливать их, обновлять или удалять³⁴. У него есть свой собственный интерфейс командной строки, в который можно «войти» из *Julia REPL*, вводя символ `]` (закрывающая квадратная скобка); для возврата оттуда можно использовать клавишу `[BackSpace]` или сочетание `[Ctrl] + [C]`. После перехода в менеджер пакетов *REPL*-«подсказка» сменяется другой: (*<АктивноеОкружение>*) `pkg>`, где *<АктивноеОкружение>* — это именно то окружение, которое будет модифицироваться командами `add`, `rm`, `update` (его можно сменить, если активировать какое-то другое окружение командой `activate <ДругоеОкружение>`). Команда `add <ИмяПакета>` добавляет пакет, команда `rm <ИмяПакета>` удаляет его³⁵, команда `update` (или просто `up`) подгружает обновления для установленных пакетов. Установка «незарегистрированных» пакетов требует указания полного адреса (т. н. *URL*): `add https://github.com/.../<ИмяПакета>.jl`.

Поскольку (по соглашению) все пакеты должны находиться в `git`-репозиториях, а `GitHub` — это бесплатное (для малых проектов) хранилище такого рода, скорее всего, адресом пакета будет какой-то раздел этого сайта, поэтому он и указан в качестве примера.

5.2. «Экосистема» *Julia*: `pkg.julialang.org` и `juliaobserver.com`

Все «одобренные» (зарегистрированные) пакеты языка *Julia* размещены на сайте `GitHub` и часто группируются по областям «интереса» (группы: <https://julialang.org/community/>).

³³Как правило, через `GitHub`, сайт-хранилище исходных кодов с системой контроля версий.

³⁴Список зарегистрированных пакетов языка *Julia* располагается по адресу pkg.julialang.org.

³⁵При установке или удалении имя пакета указывается без кавычек и суффикса.

Пакеты, созданные пользователями, не входящими в перечисляемые ниже группы, располагаются в авторских репозиториях, организованных также на сайте [GitHub](#).

Вычисления: *JuliaArrays* (26) — пакеты различных специальных типов массивов и утилит для построения типов массивов, *JuliaBerry* (6) — пакеты для микрокомпьютера *Raspberry Pi*, *JuliaCI* (24) — инфраструктура поддержки/тестирования пакетов *Julia*, *JuliaGPU* (26) — вычисления на *GPU*, *JuliaInterop* (14) — пакеты для взаимодействия с другими языками (*C++*, *Java*, *R*, *MATLAB*, *Mathematica* и пр.), *JuliaIO* (56) — инфраструктура ввода-вывода в *Julia*, *JuliaParallel* (20) — различные модели параллельного программирования, *JuliaWeb* (25) — пакеты взаимодействия с Интернет, *JuliaCloud* (13) — интерфейсы к «облачным» сервисам.

Математика: *JuliaDiff* (13) — пакеты дифференцирования, работа с **дуальными числами**, *JuliaDiffEq* (86) — дифференциальные уравнения, *JuliaGeometry* (28) — вычислительная геометрия, *JuliaGraphs* (24) — визуализация, анализ графов, *JuliaIntervals* (17) — интервальные вычисления, *JuliaMath* (39) — различные математические пакеты, включая привязки к существующим математическим библиотекам, *JuliaOpt* (68) — математическая оптимизация, *JuliaPolyhedra* (10) — вычисления с многогранниками, *JuliaSparse* (12) — решатели для разреженных матриц.

Другие науки: *BioJulia* (70) — биоинформатика/биология, *EcoJulia* (6) — экология, *JuliaAstro* (26) — астрономия/астрофизика, *JuliaDSP* (7) — цифровая обработка сигналов, *JuliaQuant* (21) — финансы, *JuliaPhysics* (10) — физика, *JuliaDynamics* (16) — нелинейная динамика и хаос, *JuliaGeo* (18) — науки о Земле, *JuliaReach* (19) — вычисления достижимости и верификация для динамических систем.

Работа с данными: *JuliaML* (25) — машинное обучение (*Machine Learning*), средства работы с наборами данных, *JuliaStats* (37) — статистические пакеты, *JuliaImages* (25) — пакеты обработки изображений, *JuliaText* (10) — обработка текста на естественных языках (*Natural Language Processing*), вычислительная лингвистика, *JuliaDatabases* (17) — пакеты доступа к различным базам данных, *JuliaData* (23) — пакеты преобразования, хранения и ввода/вывода данных.

Визуализация: *GiovineItalia* (3) — пакет статистической графики/визуализации данных *Gadfly.jl*, *JuliaPlots* (28) — визуализация с помощью пакета *Plots.jl*, высокоуровневая графика с использованием *GPU* (пакет *Makie.jl*), а также бэкенды для этих пакетов, *JuliaGL* (13) — пакеты, использующие *OpenGL API*, *JuliaGraphics* (29) — вспомогательные пакеты для графики: цвета, шрифты, пользовательские интерфейсы.

Числа в скобках после названий групп «интереса» показывают примерное (поскольку не все репозитории соответствуют именно пакетам) количество существующих и поддерживаемых представителями этих групп пакетов *Julia*. Видимо, их следует воспринимать как предметные области, интересующие тех, кто уже «увлёкся» языком *Julia*: иначе сложно объяснить бурное развитие «биологических» пакетов по сравнению, например, с «физическими» или, скажем, преобладание пакетов для дифференциальных уравнений среди всех «математических».

Сайты pkg.julialang.org и juliaobserver.com представляют пакеты *Julia* с точки зрения популярности у пользователей. В первой восьмёрке — помимо самого языка *Julia* — ядро для работы с *Julia* из «Блокнота»: *IJulia.jl*, два графических пакета: *Gadfly.jl* и *Plots.jl*, два пакета машинного обучения: *Flux.jl* и *Knet.jl*, оптимизационный пакет *JuMP.jl* и пакет решателей дифференциальных уравнений *DifferentialEquations.jl*.

5.3. Куда устанавливаются пакеты

Сразу после установки *Julia* создаёт в «домашнем» пространстве пользователя³⁶ каталог с именем `.julia` (в системах *Linux* он является скрытым), в котором будет размещаться всё, что помимо дистрибутива пользователь установит себе дополнительно (плюс некоторая сопутствующая информация).

³⁶Вызвано это тем, что в «домашнем» каталоге у пользователя точно имеются права записи.

Первоначально там имеется только подкаталог `logs` с двумя файлами: `manifest_usage.toml` и `repl_history.jl`. Первый фиксирует времена обращений к файлам `Manifest.toml`, которые содержат граф взаимозависимостей и версий пакетов, используемых конкретным окружением (изначально — это окружение с именем используемой версии *Julia*), второй — это журнальный файл вводимых пользователем команд в сессии *REPL*.

С началом установки первого же пакета *Julia* создаёт подкаталог `registries`, куда клонирует реестр `General` с сайта github.com; в `registries` возникает подкаталог `General`, в котором располагаются однобуквенные подкаталоги, объединяющие в себе подкаталоги со сведениями об отдельных пакетах на эту букву; в каждом подкаталоге с названием пакета лежат файлы `Compat.toml`, `Deps.toml` (этот файл иногда может отсутствовать, если пакет не имеет зависимостей в виде других пакетов), `Package.toml`, `Versions.toml`.

Исходные файлы пакета располагаются в каталоге с именем пакета в его подкаталоге `src` и чаще всего это просто набор `.jl`-файлов, среди которых обязательно есть один с именем `<ИмяПакета>.jl`. Небольшие пакеты могут содержать вообще только этот файл, пакеты побольше могут включать в себя кроме дополнительных файлов и некоторую структуру подкаталогов с `.jl`-файлами — всё это зависит от авторов пакета.

Практически всегда в каталоге пакета имеется подкаталог `test`, содержащий чаще всего один файл с именем `runtests.jl`. Это набор тестов для проверки функционирования пакета, но в минимальном варианте это просто команда подгрузки пакета (`using`).

В случае, когда пакет представляет собой просто «обёртку» вокруг какого-то существующего стороннего пакета, в каталоге пакета появится подкаталог `deps`, содержащий автоматически сгенерированные вспомогательные файлы `deps.jl` и `build.jl`.

5.4. Пакеты визуализации данных

Язык *Julia* сам по себе не имеет никаких графических средств, поэтому авторы языка, будучи сосредоточенными на его воплощении, вынуждены были, разумеется, рассчитывать на появление сторонних пакетов, реализующих переносимые решения в этой части.

Таких пакетов в настоящее время имеется несколько (приведены в порядке убывания популярности): `Gadfly.jl`, `Plots.jl`, `UnicodePlots.jl`, `PyPlot.jl`, `GR.jl`, `PlotlyJS.jl`, `Winston.jl`, `PGFPlots.jl` и, возможно, какие-то ещё. Для 3D-графики хорош `Makie.jl`.

`PyPlot.jl` (<https://github.com/JuliaPy/PyPlot.jl>) восходит к графике *MATLAB*; она была перевоплощена затем в рамках классического пакета `matplotlib` для языка *Python*; теперь к последнему пакету организован доступ из языка *Julia*.

`Gadfly.jl` (<https://github.com/GiovineItalia/Gadfly.jl>) следует другой философии отображения; его истоки можно отыскать в реализации пакета `ggplot2` (используется в языке *R*).

`Plots.jl` (<https://github.com/JuliaPlots/Plots.jl>) даёт доступ к другим графическим пакетам (бэкендам) через единообразный (переключаемый) программный интерфейс.

`GR.jl` (<https://github.com/jheinen/GR.jl>) — интерфейс к графическому пакету `GR`.

`PlotlyJS.jl` (<https://github.com/sqllyon/PlotlyJS.jl>) — интерфейс к библиотеке `plotly.js`.

`PGFPlots.jl` (<https://github.com/JuliaTeX/PGFPlots.jl>) использует \LaTeX -пакет `pgfplots` для создания графиков; интегрируется с *Julia* для вывода `.svg`-изображений в «Блокноте».

`UnicodePlots.jl` (<https://github.com/Evizero/UnicodePlots.jl>) — графики в терминале.

5.5. Прочие пакеты

Из-за того, что число пользователей языка *Julia* пока увеличивается не слишком быстро, количество официально доступных пакетов ещё не очень велико — чуть более двух тысяч. Сказался здесь и недавний (август 2018) выход первой версии языка, начиная с которой его синтаксис был, наконец, объявлен стабильным: не все созданные до этого пакеты были своевременно обновлены авторами.

Думается, что стоит обратить внимание на многочисленные математические пакеты, пакеты визуализации (кратко разобраны выше), пакеты для работы с другими языками (`Cxx.jl` — обеспечивает экспериментальный вариант *C++ REPL*, `PyCall.jl` — позволяет вызывать функции *Python* из *Julia*, `RCall.jl` и `MATLAB.jl` — дают возможность взаимодействовать с интерпретаторами *R* и *MATLAB*); иногда встречаются интересные по замыслу пакеты, не входящие в официальный список.

5.6. Пример: APL.jl — интерпретатор строк APL-кода

Пакет `APL.jl` случайно обнаружился среди многочисленных проектов *Shashi Howda* и привлёк внимание к себе тем, что это был пакет для языка *Julia*, но позволяющий работать с интереснейшим, хотя и несколько (незаслуженно) подзабытым языком *APL* ³⁷.

Практическая полезность пакета `APL.jl`, конечно, невелика. Он не реализует сколько-нибудь законченный вариант интерпретации строки *APL*-кода, коды символов *APL*, применённые в нём, не совпадают со стандартно используемыми, поэтому его трудно применить даже для проверки уже готовых кодовых фрагментов на языке *APL* (для этой цели гораздо лучше подходит сайт TryAPL.org), но как демонстрация возможностей *Julia* он просто превосходен.

В нём наглядно показано, что в *Julia* можно не просто работать со строками, содержащими разнообразные «экзотические» символы, но и сопоставлять им способ их разбора, что позволяет минимальными средствами создавать интерпретаторы, например, для такого «ёмкого» языка как *APL*, использующего массу нестандартных «закорючек».

```
module APL

export @apl_str, parse_apl, eval_apl

include("parser.jl")
include("eval.jl")
include("show.jl")

macro apl_str(str)
    parse_apl(str) |> eval_apl |> esc
end

end # module
```

Рис. 13: Модуль APL из пакета APL.jl.

Разумеется, эта реализация вполне «игрушечная» — в том смысле, что делать на ней что-либо серьёзное (по размерам) довольно затруднительно, но вот в плане исполнения «замысловатых» выражений языка *APL* она может оказаться весьма кстати, тем более, что в одной строке *APL*-кода часто можно уместить целую программу...

Данный пакет определяет единственный модуль APL (содержимое файла `APL.jl` приводится на рис. 13), экспортирующий имена макроса `@apl_str` и функций `parse_apl`, `eval_apl`; в него входят тексты из четырёх файлов: `parser.jl` (а в этот файл, в свою очередь, включается не упомянутый здесь `defns.jl`), `eval.jl`, `show.jl`.

Определяемый в модуле макрос `apl_str()` демонстрирует уникальную «способность» *Julia* сопоставлять каждому «типу» строки «разбирающую» эту строку процедуру. Дело в том, что строка символов, предваряемая некоторым префиксом (`<Префикс>"<Строка>"`), ведёт себя как вызов макроса `@<Префикс>_str("<Строка>")`, задаваемого пользователем. Здесь это — «конвейерное» исполнение цепочки действий (с передачей промежуточного результата между этапами) по синтаксическому анализу строки, подготовке её к исполнению и преобразованию в объект выражения; оно работает для строк с префиксом `apl`.

Оценить реальные возможности интерпретатора поможет фрагмент с определениями из файла `defns.jl` (см. рис. 14). Массив `prim_fns` — это соответствия между символами различных функций и парами выражений ³⁸: последние показывают, что делают эти функции при вызове их с одним и с двумя параметрами; формируемый по его содержимому массив `function_names` будет содержать просто список символов реализованных функций. В массиве `numeric` перечислены символы, из которых «состоят» числа в *APL* (кроме цифр и точки там имеется специальный символ для отрицательного знака числа — и это не знак минус). Остальные массивы определяют: какие монадические или диадические операторы допустимы в интерпретаторе; какими символами обозначаются аргументы функций.

Пользоваться этим интерпретатором можно (после `using APL`), вводя в командной строке *Julia* (*REPL*) выражение с *APL*-кодом в виде строки с префиксом `apl`, например:

```
julia> apl "+/ι100"
5050
```

³⁷С языком *APL* автора брошюры (в бытность его на первом курсе) познакомил Павел Игоревич Рубан, за что ему хотелось бы выразить особую благодарность.

³⁸Язык *Julia* позволяет работать с ещё не выполненными выражениями, которые могут быть созданы из функций или исполнимых фрагментов при помощи двоеточия или двоеточия с круглыми скобками.

```

const prim_fns = [
  ( '+', (:+, :+)),
  ( '-', (:-, :-)),
  ( '÷', (:(1/ω), :/)),
  ( '×', (:sign, :(α.*ω))),
  ( '⌈', (:ceil, :max)),
  ( '⌊', (:floor, :min)),
  ( '⋈', (:(ω[:]), :vcat)),
  ( '^', (:exp, :(^))),
  ( '!', (:factorial, :(factorial(ω) / (factorial(α) * factorial(ω-α)))), # ...
  ( '|', (:abs, :(ω%α))),
  ( '⊗', (:log, :log)),
  ( '⊙', (:(π*ω), nothing)),
  ( '~', (:!, nothing)),
  ( '⊞', (:inv, :\)),
  ( 'ℓ', (:(1:ω[1]), :(ℓ(α, ω))), # ω[1] is not really right
  ( '?', (:(rand(1:ω[1])), nothing)),
  ( '>', (nothing, :(>))),
  ( '<', (nothing, :(<))),
  ( '=', (nothing, :(&=))),
  ( '≠', (nothing, :(&!=))),
  ( '≤', (nothing, :(&≤))),
  ( '≥', (nothing, :(&≥))),
  ( '^', (nothing, :(&))),
  ( '∨', (nothing, :|)),
  ( ']', (nothing, :(getindex(ω, α))),
  ( 'T', (nothing, :T)),
  ( 'Δ', (:sortperm, nothing)),
  ( '∇', (:(sortperm(ω, rev=true)), nothing)),
  ( 'ρ', (:([size(ω)...]), :(reshape(ω, tuple(α...)))),
  ( '⊘', (:(ω.'), :permutedims)),
  ( 'ϕ', (:(flipdim(ω, 1)), nothing)),
  ( '↓', (nothing, :(ω[α+1:length(ω)]))),
  ( '↑', (nothing, :(ω[1:α]))),
]
const function_names = map(x->x[1], prim_fns)
const numeric = ['-', ' ', '0':'9'..., '.']
const monadic_operators = ['\|', '/', '÷', '×', '!', '~', '⊞', '⊙', '⊗']
const dyadic_operators = ['.', '⋈']
const argnames = ['α', 'ω']

```

Рис. 14: Фрагмент файла `defns.jl` из пакета `APL.jl`.

Небольшая неприятность заключается в том, что символы, использованные при реализации интерпретатора, не соответствуют символам, применяемым в языке *APL*; скажем, греческие символы (см. приводимую ниже таблицу) будут иметь совершенно не тот код, что необходим для «настоящей» *APL*-программы (в языке *APL* используются символы *Unicode* из вполне определённого диапазона: U+2200 .. U+23FF).

Таблица 1: Некоторые используемые в языке *APL* символы.

Символ	Код в <i>APL</i>		Ввод в <i>Julia</i>	Код в <i>Julia</i>	
α	U+237A	E2 8D BA	<code>\alpha</code> <input type="text" value="Tab"/>	U+03B1	CE B1
ω	U+2375	E2 8D B5	<code>\omega</code> <input type="text" value="Tab"/>	U+03C9	CF 89
ι	U+2373	E2 8D B3	<code>\iota</code> <input type="text" value="Tab"/>	U+03B9	CE B9
ρ	U+2374	E2 8D B4	<code>\rho</code> <input type="text" value="Tab"/>	U+03C1	CF 81
	Unicode	UTF-8		Unicode	UTF-8

Можно было избежать этой неприятности, если, к примеру, продублировать операции, реализуемые «неправильными» символами, также и для символов «правильных». Однако тогда всё равно остаётся не устранённым неудобство ввода с клавиатуры всех используемых в интерпретаторе символов, поскольку их *L^AT_EX*-«имена» не предусмотрены в *Julia*.

6. О плохом и хорошем (вместо заключения)

После стольких страниц благожелательного повествования о новом языке имеет смысл остановиться и на некоторых отрицательных моментах, характерных для него. Здесь они изложены в стиле вопрос/замечание – ответ.

- При выполнении пары строк всё «подвисло» на нескольких минут!

Если пользователь никогда ранее не сталкивался с *JIT*-компиляцией большого пакета, то это может стать совершенно неожиданным и труднообъяснимым. Но причина здесь проста: когда пакет импортировался, код пакета ещё не был откомпилирован полностью; компиляция (взаимозависимых функций) началась лишь после первых вызовов функций. Подобное характерно, например, для популярного графического пакета `Gadfly.jl`.

Существует, однако, радикальное решение для подобных ситуаций: откомпилировать некоторый набор часто используемых «тяжеловесных» пакетов и добавить всё это в т. н. **системный образ** *Julia*. Тем самым время на компиляцию «большого» пакета (перед первым использованием функций из него) затрачиваться не будет. Правда, подобная рекомендация хороша лишь для тех, кто имеет опыт подобной перекомпиляции системного образа (или же не боится трудностей).

- Символы *Unicode* слишком сложно вводятся.

Если пользоваться для записи кода набором символов, куда должны быть включены какие-то *Unicode*-символы, то затруднений при вводе программ будет, конечно, больше, поскольку введение каждого «необычного» символа потребует некоторых усилий. С другой стороны, код от такого включения только выигрывает (в читабельности), а принципиальных сложностей здесь нет, поскольку пользователь, скорее всего, уже имеет опыт подобного ввода, т. к. точно такие же последовательности используются при оформлении документов и публикаций в *Л^AT_EX*.

- Индексация с 1 неестественна и непривычна.

Если ранее вы пользовались языками программирования, где индексы отсчитываются с нуля, то работа в *Julia* может показаться возвращением в «страшные» времена Фортрана. Однако, во-первых, к этому вполне легко привыкнуть, а, во-вторых, сейчас существует возможность создавать массивы с **произвольной индексацией**; для работы с ними соответствующий код понадобится написать чуть иначе.

- Нашлись интересные примеры, но почему-то не работают...

Да, действительно, не все найденные в Сети примеры будут работать сейчас, хотя ранее они могли работать. Объясняется это тем, что до появления версии *Julia 1.0.0* в августе 2018 года в языке постоянно происходили различные изменения. Правда, они никогда не были совсем уж радикальными. Сейчас синтаксис языка объявлен стабильным (по крайней мере до версии *2.0...*), а внесение небольших исправлений в неработающую программу будет отличным упражнением для освоения языка.

- Зачем нужны такие огромные сообщения об ошибках?

Может показаться, что иногда сообщения об ошибках (выводимые к тому же в рамках *REPL* красным цветом) слишком многословны и способны испугать неискушённого пользователя. Однако их размер бывает большим по двум причинам: во-первых, тут выводится весь путь в иерархии вызовов, во-вторых, предлагаемые варианты (скажем, функций) могут иметь сложные (с длинным описанием) типы параметров.

Опыт показывает, что внимательное прочтение хотя бы начала подобного сообщения позволяет понять причину его появления и наметить путь устранения «проблем».

ПРИЛОЖЕНИЕ. Поиск вариантов формул Штрассена

Здесь описывается небольшой вычислительный эксперимент, выполненный при помощи языка *Julia* версии 1.0.0 и позволяющий найти альтернативные варианты формул для перемножения матриц 2×2 с использованием семи операций умножения.

В 1969 году в статье Штрассена³⁹ был предложен алгоритм перемножения матриц

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

в котором использовалось только семь операций умножения (*формулировка здесь немного изменена по сравнению с оригиналом*⁴⁰):

$$P_1 = (A_{21} + A_{22})B_{11} \quad (1)$$

$$P_2 = A_{22}(-B_{11} + B_{21}) \quad (2)$$

$$P_3 = (A_{11} + A_{12})B_{22} \quad (3)$$

$$P_4 = A_{11}(B_{12} - B_{22}) \quad (4)$$

$$P_5 = (-A_{11} + A_{21})(B_{11} + B_{12}) \quad (5)$$

$$P_6 = (A_{11} + A_{22})(B_{11} + B_{22}) \quad (6)$$

$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \quad (7)$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} P_2 - P_3 + P_6 + P_7 & P_3 + P_4 \\ P_1 + P_2 & -P_1 + P_4 + P_5 + P_6 \end{pmatrix}, \quad (8)$$

а не восемь, как в стандартной формуле:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \quad (9)$$

Отметим, что формулы верны как для матриц 2×2 , так и для блочных матриц A и B ⁴¹.

Идея сокращения числа операций умножения в пользу операций сложения, лежащая в основе формул (1)–(8), весьма проста. Произведение двух сумм (начиная с минимальных двучленных сумм $(a + b)(c + d) = ac + bc + ad + bd$ и т. п.) — это всего лишь одно умножение, в отличие от «развёрнутого» варианта подобного выражения (суммы произведений), где умножений не менее четырёх. В случае стандартного (блочного) перемножения двух матриц 2×2 каждый элемент результирующей матрицы представляется суммой двух произведений элементов исходных матриц (9). Если попытаться перемножить не просто элементы/блоки, а их некоторые алгебраические суммы, возможно, что, подправляя результат значениями других произведений, удастся (и пример формул Штрассена показывает — это возможно) сформировать выражения для элементов произведения двух матриц с меньшим числом операций умножения в них.

О том, что формулы Штрассена не являются уникальными и обнаружены (скорее всего) путём подбора, косвенно свидетельствует их довольно «беспорядочный» вид и отсутствие какой-либо внутренней симметрии, поэтому можно попробовать найти **все** возможные варианты подобных формул — при некоторых ограничениях на вид сомножителей в произведениях.

Для поисков можно воспользоваться любым языком программирования, но выбор *Julia* оправдан как минимум двумя соображениями. Во-первых, в *Julia* сильно вложенные циклы записываются существенно проще, нежели в традиционных языках программирования, — вместе с более удобными способами пропуска итераций. Во-вторых, в *Julia* разрешены

³⁹V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, **13**: 354–356, 1969.

⁴⁰Выбранные произведения перенумерованы в соответствии с вводимым далее условным порядком.

⁴¹Поскольку нигде не используется коммутативность умножения, естественная для элементов численной матрицы 2×2 , но не имеющая места для блочных матриц.

нестандартные имена переменных, для данного случая полностью совпадающие с математическими обозначениями в рассматриваемых выражениях, — что весьма удобно.

Наиболее общий вид возможных произведений (со скалярными коэффициентами α_i), которые могут участвовать в формировании элементов результирующей матрицы C_{11} , C_{12} , C_{21} , C_{22} , включает какую-то линейную комбинацию элементов матрицы A , умноженную на какую-нибудь линейную комбинацию элементов матрицы B :

$$(\alpha_1 A_{11} + \alpha_2 A_{12} + \alpha_3 A_{21} + \alpha_4 A_{22})(\alpha_5 B_{11} + \alpha_6 B_{12} + \alpha_7 B_{21} + \alpha_8 B_{22}). \quad (10)$$

В результате раскрытия произведения, вообще говоря, может получиться до шестнадцати слагаемых — в зависимости от конкретных значений α_i , $i = 1, \dots, 8$:

$$\begin{aligned} & \alpha_1 \alpha_5 A_{11} B_{11} + \alpha_1 \alpha_6 A_{11} B_{12} + \alpha_1 \alpha_7 A_{11} B_{21} + \alpha_1 \alpha_8 A_{11} B_{22} + \\ & \alpha_2 \alpha_5 A_{12} B_{11} + \alpha_2 \alpha_6 A_{12} B_{12} + \alpha_2 \alpha_7 A_{12} B_{21} + \alpha_2 \alpha_8 A_{12} B_{22} + \\ & \alpha_3 \alpha_5 A_{21} B_{11} + \alpha_3 \alpha_6 A_{21} B_{12} + \alpha_3 \alpha_7 A_{21} B_{21} + \alpha_3 \alpha_8 A_{21} B_{22} + \\ & \alpha_4 \alpha_5 A_{22} B_{11} + \alpha_4 \alpha_6 A_{22} B_{12} + \alpha_4 \alpha_7 A_{22} B_{21} + \alpha_4 \alpha_8 A_{22} B_{22}. \end{aligned}$$

Коэффициенты α_i в каждом из произведений должны быть такими, чтобы далее подобные произведения (возможно, со своими сомножителями) в сумме могли бы сформировать пары «элементарных» произведений $A_{11}B_{11} + A_{12}B_{21}$, $A_{11}B_{12} + A_{12}B_{22}$, $A_{21}B_{11} + A_{22}B_{21}$, $A_{21}B_{12} + A_{22}B_{22}$, а потому интуитивно понятно, что получающимся в результате умножения произведениям $\alpha_i \alpha_j$ нельзя быть слишком разнообразными, иначе при суммировании возникнут проблемы — как с сокращением одних сумм, так и равенством единице других.

Простейший вариант — использовать коэффициенты только из набора $\alpha_i \in \{-1, 0, +1\}$, $i = 1, \dots, 8$ (всего получается $3^8 = 6561$ комбинация), тогда будет достаточно одного суммирования с противоположным знаком для сокращения «лишнего» слагаемого, а «нужные» слагаемые смогут сразу иметь необходимый единичный коэффициент. По этой причине подходящие семь произведений мы будем пытаться искать без числовых сомножителей.

Для определённости имеет смысл зафиксировать порядок следования «элементарных» произведений $A_i B_j$, $i, j = 1, \dots, 4$ (т. е., например, использовать уже приведённый выше), тогда «развёрнутые» произведения (10) однозначно задаются⁴² упорядоченным набором числовых коэффициентов $\alpha_i \alpha_j$, $i, j = 1, \dots, 4$ из вот такой таблицы:

	B_{11}	B_{12}	B_{21}	B_{22}
A_{11}	$\alpha_1 \alpha_5$	$\alpha_1 \alpha_6$	$\alpha_1 \alpha_7$	$\alpha_1 \alpha_8$
A_{12}	$\alpha_2 \alpha_5$	$\alpha_2 \alpha_6$	$\alpha_2 \alpha_7$	$\alpha_2 \alpha_8$
A_{21}	$\alpha_3 \alpha_5$	$\alpha_3 \alpha_6$	$\alpha_3 \alpha_7$	$\alpha_3 \alpha_8$
A_{22}	$\alpha_4 \alpha_5$	$\alpha_4 \alpha_6$	$\alpha_4 \alpha_7$	$\alpha_4 \alpha_8$

Храниться подобные таблицы будут в двумерных «срезах» трёхмерного массива.

```
function TernaryToCoeffs(N, Size)
    Co = digits(N, base=3, pad=Size)
    @. Co[Co==2] = -1
    return Co
end

ProdTermCoeffs = Array{Int, 3}(undef, 4, 4, 3^8);

for I=1:3^8
    alpha = TernaryToCoeffs(I-1, 2^3)
    ProdTermCoeffs[:, :, I] = [alpha[1+k%4]*alpha[5+k÷4] for k=0:15]
end
```

Рис. 15: Код заполнения массива «развёрнутых» произведений.

⁴²А также имеют вполне определённый номер в массиве, соответствующий троичному числу $\alpha_8 \dots \alpha_1$.

Функция `TernaryToCoeffs()` преобразует троичное значение с заданным числом цифр в набор коэффициентов «развёрнутого» произведения, попутно меняя все цифры 2 в нём на коэффициент -1 .

Строку `@. Co[Co==2] = -1` из функции `TernaryToCoeffs()` можно также записать с помощью т. н. поэлементных операций с точкой впереди (`Co[Co.==2] .= -1`), но для случаев, когда выражение содержит много поэлементных операций, в *Julia* предусмотрен специальный макрос с необычным именем `@.` — он автоматически заменяет знаки операций на поэлементные в том выражении, которое предвывает.

Массив `ProdTermCoeffs` — это трёхмерный массив для хранения (в виде двумерных массивов 4×4) коэффициентов $\alpha_i \alpha_j$ «развёрнутых» произведений (и последующего доступа к ним по их условному номеру); его двумерные «срезы» заполняются (перевычисляемым) содержимым одномерного массива, причём здесь учтено, что массивы в *Julia* хранятся по столбцам (коэффициенты следуют в порядке $\alpha_1 \alpha_5, \alpha_2 \alpha_5, \alpha_3 \alpha_5, \alpha_4 \alpha_5, \alpha_1 \alpha_6, \alpha_2 \alpha_6, \alpha_3 \alpha_6, \alpha_4 \alpha_6, \dots$).

Тем самым у каждого «развёрнутого» произведения будет зафиксировано положение коэффициентов «элементарных» произведений $A_i B_j$, а все «развёрнутые» произведения (из-за коэффициентов из множества $\{-1, 0, +1\}$) окажутся «занумерованными» какими-то конкретными троичными значениями.

Например, у произведения $(A_{11} + A_{12})B_{22}$ (см. формулу (3) выше) будет номер 2192 ($\alpha_1 = \alpha_2 = 1, \alpha_3 = \alpha_4 = \alpha_5 = \alpha_6 = \alpha_7 = 0, \alpha_8 = 1; 1 + 1 \cdot 3^7 + 1 \cdot 3^1 + 1 \cdot 3^0 = 2192$), а у произведения $A_{11}(B_{12} - B_{22})$ (формула (4)) — 4619 ($\alpha_1 = 1, \alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = 0, \alpha_6 = 1, \alpha_7 = 0, \alpha_8 = 2; 1 + 2 \cdot 3^7 + 1 \cdot 3^5 + 1 \cdot 3^0 = 4619$). Выделенные в начале сумм для номеров единицы связаны с нумерацией в *Julia* элементов массивов (она начинается с единицы).

В принципе, и коэффициенты $\alpha_i, i = 1, \dots, 8$, и коэффициенты «развёрнутого» произведения имело бы смысл хранить вещественными, но пока ограничимся только целочисленными значениями. После суммирования нескольких произведений (в виде таблиц из их коэффициентов) должна получаться таблица, соответствующая значению одного из элементов результата.

1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
$C_{11} = A_{11}B_{11} + A_{12}B_{21}$				$C_{12} = A_{11}B_{12} + A_{12}B_{22}$				$C_{21} = A_{21}B_{11} + A_{22}B_{21}$				$C_{22} = A_{21}B_{12} + A_{22}B_{22}$			

Рис. 16: Наборы коэффициентов для элементов матрицы результата.

С помощью созданного глобально (скажем, в строке *REPL*) массива `ProdTermCoeffs` уже можно проверять (хотя и не в самой удобной форме) справедливость приводимых в публикациях соотношений (некоторые результаты сообщаются далее — см. стр. 42–45), но «интенсивная» программная работа с глобальным массивом в *Julia* невозможна, поэтому далее код определения и заполнения этого массива используется в рамках функций.

Разность произведений с условными номерами 118 и 1567 даёт элемент C_{21} результирующей матрицы:

```

julia> ProdTermCoeffs[:, :, 118] - ProdTermCoeffs[:, :, 1567]
4×4 Array{Int64, 2}:
 0  0  0  0
 0  0  0  0
 1  0  0  0
 0  0  1  0

```

а сумма произведений с условными номерами 2192 и 4619 — её элемент C_{12} :

```

julia> ProdTermCoeffs[:, :, 2192] + ProdTermCoeffs[:, :, 4619]
4×4 Array{Int64, 2}:
 0  1  0  0
 0  0  0  1
 0  0  0  0
 0  0  0  0

```

Остаётся лишь одно техническое затруднение: каждую формулу-произведение надо уметь переводить в её условный номер в массиве. Но это весьма просто делается в строке `REPL` с помощью выражения вроде `1+[3~i for i=0:7]*[0,0,1,0,1,2,1,2]`; в нём вектор-строка троичных весов разрядов (полученная транспонированием из вектора-столбца) умножается на вектор-столбец троичных цифр, описывающих структуру формулы-произведения. Цифры эти соответствуют Z -упорядоченным⁴³ элементам сначала левой, а затем правой матрицы: 0 означает отсутствие элемента в произведении, 1 — наличие со знаком плюс, 2 — со знаком минус; таким образом, в приводимом примере «закодировано» вычисление номера произведения $A_{21}(B_{11} - B_{12} + B_{21} - B_{22})$.

Хотя подобная форма верификации не отличается особым удобством, она имеет одно несомненное преимущество именно в *Julia*: её можно использовать и с произвольными коэффициентами, просто записывая их перед каждым слагаемым.

Стратегия перебора

Несмотря на вроде бы небольшой размер массива с «развёрнутыми» произведениями (6561), осуществить полный перебор по всем семи произведениям, тем не менее, нереально — из-за семикратной вложенности цикла перебора⁴⁴, — поэтому имеет смысл действовать поэтапно. По аналогии с формулами Штрассена, будем искать два элемента результата (не обязательно внедиагональные, как в оригинале) в виде сумм двух произведений, а для остальных постараемся обойтись ещё тремя произведениями — так, чтобы одно из произведений «пригодилось» для формирования обоих оставшихся элементов.

Самые простые произведения в формулах Штрассена — умножения элемента на сумму/разность двух других элементов — дают два «элементарных» произведения в сумме, но так как в необходимых в результате суммах $A_{11}B_{11} + A_{12}B_{21}$, $A_{11}B_{12} + A_{12}B_{22}$, $A_{21}B_{11} + A_{22}B_{21}$, $A_{21}B_{12} + A_{22}B_{22}$ вообще нет общих сомножителей, становится понятно, что формирование нужной суммы произведений требует добавления как минимум ещё одного произведения (полученного из аналогичного произведения элемента и суммы/разности), чтобы в результате сокращения двух из них (из-за противоположных знаков) остались только требуемые два произведения. Подобным образом в формулах Штрассена получаются внедиагональные элементы результирующей матрицы и на это будет затрачено четыре умножения.

Диагональные элементы после этого так формировать нельзя, иначе суммарное число умножений достигнет восьми. Здесь необходима более сложная сумма из четырёх произведений, по два из которых (максимально «разнесённым» образом) берутся из предыдущего этапа. К ним подыскиваются ещё два выражения-произведения — чтобы завершить формирование первого диагонального элемента; одно из только что найденных произведений далее используется повторно, чтобы после выбора последнего (седьмого по счёту) выражения-произведения получился оставшийся диагональный элемент результирующей матрицы. В последнем случае успешность поиска не совсем очевидна, но по крайней мере стоит ожидать получения уже известных формул Штрассена. Как оказывается, существуют и другие варианты формул.

Важной вспомогательной функцией для поиска пар произведений, алгебраическая сумма которых даёт заданный элемент результирующей матрицы, является `FindExprs2!()`, см. рис. 17. Ей в качестве параметров передаётся массив с «развёрнутыми» произведениями, таблица, соответствующая элементу результата, а также (изначально пустой) массив для «накопления» подходящих пар (номеров произведений). Восклицательный знак в имени функции (по принятому в *Julia* соглашению) показывает, что один из передаваемых ей параметров (здесь это массив накапливаемых пар) будет изменён в процессе работы. Поскольку типы параметров этой функции в её заголовке не указаны, в самом начале делаются проверки (происходящие во время исполнения), необходимые для её нормальной работы: что массив с произведениями является трёхмерным и размеры по первым двум измерениям у него равны четырём; что таблица, описывающая матричный элемент результата, двумерна с такими же размерами; что тип параметра для хранения пар — это одномерный массив двоек целых чисел (полный листинг функции показан на рис. 17).

⁴³Имеется в виду порядок элементов матрицы 2×2 в виде буквы Z , т. е., A_{11} , A_{12} , A_{21} , A_{22} .

⁴⁴И это при том, что мы ограничились лишь коэффициентами из набора $-1, 0, +1$; поиски более сложных соотношений перебором (скажем, с вещественными коэффициентами) будут гораздо более трудоёмкими.

```

IsTermEmpty(Idx) = Idx÷3^4 == 0 || Idx%3^4 == 0

function FindExprs2!(ProdArray, Coeffs, Pairs)
    @assert length(size(ProdArray)) == 3
    @assert size(ProdArray)[1] == 4 && size(ProdArray)[2] == 4
    @assert length(size(Coeffs)) == 2
    @assert size(Coeffs)[1] == 4 && size(Coeffs)[2] == 4
    @assert typeof(Pairs) == Array{Tuple{Int,Int},1}
    Signs = "+-"
    Ln = size(ProdArray)[3]
    for I = 1:Ln, II = I+1:Ln
        IsTermEmpty(I-1) && continue
        IsTermEmpty(II-1) && continue
        Term1 = ProdArray[:, :, I]
        Term2 = ProdArray[:, :, II]
        for Sign1 in Signs, Sign2 in Signs
            Result = zeros{Int, 4, 4}
            Result = eval(Expr(:call, Symbol(Sign1), Result, Term1))
            Result = eval(Expr(:call, Symbol(Sign2), Result, Term2))
            if Result == Coeffs
                if I == Canonize(I)[1] && II == Canonize(II)[1]
                    push!(Pairs, (I, II))
                    print("($I,$II): $Sign1 $(ShowAsProduct(I))")
                    println(" $Sign2 $(ShowAsProduct(II))")
                    @debug println("$Sign1$Term1\n$Sign2$Term2")
                else
                    @debug println("($I,$II)->($(Canonize(I)),$(Canonize(II)))")
                end
            end
        end
    end
end
end
end
end

```

Рис. 17: Код функции FindExprs2!() из файла FindExprs2.jl.

Основа функции — двойной цикл по «невырожденным»⁴⁵ «развёрнутым» произведениям, составляющим сразу упорядоченную пару (условный номер I первого произведения — меньше, чем номер II второго). Для каждой пары произведений формируются все возможные алгебраические выражения с использованием знаков `Signs` (ещё один небольшой двойной цикл), причём происходит это «на лету» (применяются: `Symbol()` — для создания символа нужной операции, `Expr(:call, ...)` — для получения соответствующего выражения и `eval()` — для его исполнения). В каждой операции участвуют двумерные массивы (матрицы) размера 4×4 и результаты (включая промежуточные) тоже являются таковыми. Если получаемый массив `Result` поэлементно равен содержимому таблицы `Coeffs`, значит, получена подходящая пара произведений; она запоминается и выводится (с помощью вспомогательной функции `ShowAsProduct()`, см. рис. 19), правда, лишь при условии, что оба эти произведения — т. н. «канонизированные»⁴⁶ (подробности см. далее на стр. 40): это эффективно избавляет выводимые соотношения от дублирования произведений, отличающихся лишь различными комбинациями знаков сомножителей.

Программное формирование выражений, применённое здесь, позволяет записать алгоритм подбора компактнее, однако отрицательно сказывается на быстродействии: дело в том, что в цикле всякий раз формируется новое выражение, которое перед исполнением должно быть откомпилировано, а это фактически превращает код в интерпретируемый. На следующем этапе подбора, где циклы становятся четырёхкратными, от такого подхода пришлось отказаться.

Часть вывода носит отладочный характер и при обычном исполнении не воспроизводится (макрос `@debug <Код>` позволяет коду быть исполненным только при определённом значении переменной окружения `JULIA_DEBUG`, см. стр. 15).

⁴⁵Произведения с нулевыми коэффициентами одного из сомножителей из рассмотрения исключаются в типичном стиле *Julia*: если справедлива проверка (с помощью вспомогательной однострочной функции `IsTermEmpty()`), тело этих циклов не выполняется: конструкция `<Условие> && continue` эквивалентна коду `if <Условие> continue end`.

⁴⁶Если никак не ограничивать получаемые соотношения, то в результате для выражения какого-либо элемента в виде суммы двух произведений можно получить по 80 вариантов представления, включая «тривиальные» из формулы (9) (и это с учётом просмотра лишь упорядоченных пар!), — поскольку там будут все варианты сомножителей с различными знаками и разные знаки комбинирования произведений.

Вызывается функция `FindExprs2!()` в двух других функциях, `FindC11C222Terms!()` и `FindC12C212Terms!()`; код первой из них приводится на рис. 18, вторая — аналогична ей.

```
function FindC11C222Terms!(PairsC11, PairsC22)
  @assert typeof(PairsC11) == Array{Tuple{Int,Int},1}
  @assert typeof(PairsC22) == Array{Tuple{Int,Int},1}
  ProdTermCoeffs = Array{Int,3}(undef,4,4,3^8);
  C11Coeffs = Array{Int,2}(undef,4,4);
  C11Coeffs[:,:] = [1,0,0,0, 0,0,0,0, 0,1,0,0, 0,0,0,0];
  C22Coeffs = Array{Int,2}(undef,4,4);
  C22Coeffs[:,:] = [0,0,0,0, 0,0,1,0, 0,0,0,0, 0,0,0,1];
  for I=1:3^8
    α = TernaryToCoeffs(I-1,2^3)
    ProdTermCoeffs[:,:,I] = [α[1+k%4]*α[5+k÷4] for k=0:15]
  end
  println("\n\tC11\n")
  FindExprs2!(ProdTermCoeffs, C11Coeffs, PairsC11)
  println("\n\tC22\n")
  FindExprs2!(ProdTermCoeffs, C22Coeffs, PairsC22)
end
```

Рис. 18: Код функции `FindC11C222Terms!()` из файла `FindExprs2.jl`.

Отличия функции `FindC12C212Terms!()` от приведённой функции `FindC11C222Terms!()` таковы: передаются как параметры массивы пар `PairsC12` и `PairsC21`; для сравнения необходимы (и создаются) таблицы `C12Coeffs` и `C21Coeffs`; вызовами функции `FindExprs2!()` отыскиваются двучленные выражения для `C12` и `C21`. Результаты поисков будут выведены в привычном виде с помощью функции `ShowAsProduct()` (рис. 19).

```
ZPM(D) = (D == 0) ? "" : ((D == 1) ? "+" : "-")

function AlgSum(Num,Elements)
  S = ""
  for E in Elements
    R = Num % 3
    if R != 0 S *= ZPM(R) * E end
    Num ÷= 3
  end
  S
end

function ShowAsProduct(No)
  ElemA = ["A11", "A12", "A21", "A22"]
  ElemB = ["B11", "B12", "B21", "B22"]
  "($ (AlgSum((No-1)%3^4,ElemA)) ) ($ (AlgSum((No-1)÷3^4,ElemB)) )"
end
```

Рис. 19: Код функции `ShowAsProduct()` (и необходимых ей вспомогательных).

Строка выводимого произведения формируется с помощью т. н. «интерполяции», когда вместо выражения (записанного после символа доллара в последующих круглых скобках) будет подставлен результат его вычисления. В данном случае так оформлены вызовы вспомогательной функции `AlgSum(Num,Elements)` с троичными номерами сомножителей `Num` и наборами символьных имён `Elements` для их записи. Однострочная функция `ZPM(D)` преобразует передаваемое ей троичное значение `D` в сопоставленный этому значению символ знака. Вызывать функцию `ShowAsProduct()` можно как для одного номера произведения, так и для массива номеров (например, с помощью макроса `@.`):

```
julia> @. ShowAsProduct([118,1567,2192,4619,344,2297,2974])
7-element Array{String,1}:
 "(+A21+A22)(+B11)"
 "(+A22)(+B11-B21)"
 "(+A11+A12)(+B22)"
 "(+A11)(+B12-B22)"
 "(+A11-A21)(+B11+B12)"
 "(+A11+A22)(+B11+B22)"
 "(+A12-A22)(+B21+B22)"
```

Результат исполнения (понадобилось несколько часов!) функций `FindC11C222Terms!()` и `FindC12C212Terms!()` содержал по 10 пар; в каждой пятёрке одна пара — «тривиальная», т. е., состоящая из «элементарных» произведений (как в стандартных формулах)⁴⁷.

```
julia> PairsForC11 = Array{Tuple{Int,Int},1}();
julia> PairsForC22 = Array{Tuple{Int,Int},1}();
julia> FindC11C222Terms!(PairsForC11, PairsForC22)
```

C₁₁

```
(83,733): + (+A11)(+B11) + (+A12)(+B21)
(86,1543): + (+A11+A12)(+B11) - (+A12)(+B11-B21)
(89,814): + (+A11-A12)(+B11) + (+A12)(+B11+B21)
(734,1541): + (+A11+A12)(+B21) + (+A11)(+B11-B21)
(737,812): - (+A11-A12)(+B21) + (+A11)(+B11+B21)
```

C₂₂

```
(253,2215): + (+A21)(+B12) + (+A22)(+B22)
(280,4645): + (+A21+A22)(+B12) - (+A22)(+B12-B22)
(307,2458): + (+A21-A22)(+B12) + (+A22)(+B12+B22)
(2224,4627): + (+A21+A22)(+B22) + (+A21)(+B12-B22)
(2251,2440): - (+A21-A22)(+B22) + (+A21)(+B12+B22)
```

```
julia> PairsForC12 = Array{Tuple{Int,Int},1}();
julia> PairsForC21 = Array{Tuple{Int,Int},1}();
julia> FindC12C212Terms!(PairsForC12, PairsForC21)
```

C₁₂

```
(245,2191): + (+A11)(+B12) + (+A12)(+B22)
(248,4621): + (+A11+A12)(+B12) - (+A12)(+B12-B22)
(251,2434): + (+A11-A12)(+B12) + (+A12)(+B12+B22)
(2192,4619): + (+A11+A12)(+B22) + (+A11)(+B12-B22)
(2195,2432): - (+A11-A12)(+B22) + (+A11)(+B12+B22)
```

C₂₁

```
(91,757): + (+A21)(+B11) + (+A22)(+B21)
(118,1567): + (+A21+A22)(+B11) - (+A22)(+B11-B21)
(145,838): + (+A21-A22)(+B11) + (+A22)(+B11+B21)
(766,1549): + (+A21+A22)(+B21) + (+A21)(+B11-B21)
(793,820): - (+A21-A22)(+B21) + (+A21)(+B11+B21)
```

На следующем этапе найденные пары⁴⁸ для диагональных (или же внедиагональных) элементов комбинировались «перекрёстным» образом друг с другом⁴⁹, чтобы найти способы представления одного из оставшихся элементов результата в виде алгебраической суммы четырёх произведений; для формирования последнего элемента использовалось одно из только что найденных произведений (т. н. «общее») и подходящее произвольное.

Таблица полученных соотношений⁵⁰ приводится на стр. 39; все используемые произведения в ней — «канонизированные» (см. стр. 40). Эти 32 соотношения разбиты на 8 групп сообразно «подходящим» друг к другу парам произведений из найденных ранее; в каждой группе всегда присутствуют одни и те же произведения: **335** и **2947**, **344** и **2974**, **578** и **5134**, **587** и **5161**. Упорядочены соотношения в таблице по группам — в порядке их «появления», а в каждой группе — по возрастанию номера «общего» произведения.

⁴⁷В массивах `PairsForC...` найденные пары сохранялись для последующей работы с ними.

⁴⁸Без «тривиальных» пар, которые почему-то (ошибочно!) были сочтены малоперспективными...

⁴⁹Код этого этапа опущен, поскольку, во-первых, принципиально не отличается от уже приведённого, и, во-вторых, слишком объёмен — из-за развёрнутого (для устранения постоянной перекомпиляции очередного выражения) цикла перебора всех сочетаний знаков четырёхчленных алгебраических выражений.

⁵⁰В соотношениях вместо самих произведений указаны их условные номера в таблице `ProdTermCoeffs`.

Таблица 2: Некоторые формулы для умножения матриц 2×2 с помощью семи произведений.

C_{11}	C_{12}	C_{21}	C_{22}
$+(86) - (1543)$	$-(86) + (4627) + (344) + (2281)$	$-(1543) - (2224) + (2281) - (2974)$	$+(2224) + (4627)$
	$-(86) - (4627) + (335) + (2290)$	$+(1543) - (2224) - (2290) + (2947)$	
	$+(86) + (4627) - (587) - (4468)$	$-(1543) + (2224) + (4468) - (5161)$	
	$+(86) - (4627) - (578) - (4477)$	$+(1543) + (2224) - (4477) + (5134)$	
$+(89) + (814)$	$+(89) - (2440) - (578) + (2281)$	$-(814) - (2251) + (2281) + (5134)$	$-(2251) + (2440)$
	$+(89) + (2440) - (587) + (2290)$	$+(814) - (2251) - (2290) - (5161)$	
	$-(89) - (2440) + (335) - (4468)$	$-(814) + (2251) + (4468) + (2947)$	
	$-(89) + (2440) + (344) - (4477)$	$+(814) + (2251) - (4477) - (2974)$	
$+(734) + (1541)$	$-(734) - (4645) + (1001) + (2974)$	$+(1541) - (280) - (344) + (1001)$	$+(280) - (4645)$
	$-(734) + (4645) + (1028) + (2947)$	$-(1541) - (280) + (335) - (1028)$	
	$+(734) - (4645) + (1730) - (5161)$	$+(1541) + (280) - (587) - (1730)$	
	$+(734) + (4645) + (1757) - (5134)$	$-(1541) + (280) + (578) + (1757)$	
$-(737) + (812)$	$-(737) - (2458) + (1001) - (5134)$	$-(812) + (307) + (578) + (1001)$	$+(307) + (2458)$
	$-(737) + (2458) + (1028) - (5161)$	$+(812) + (307) - (587) - (1028)$	
	$+(737) - (2458) + (1730) + (2947)$	$-(812) - (307) + (335) - (1730)$	
	$+(737) + (2458) + (1757) + (2974)$	$+(812) - (307) - (344) + (1757)$	
$-(248) + (1549) + (344) + (985)$	$+(248) - (4621)$	$+(766) + (1549)$	$-(4621) - (766) + (985) - (2974)$
$-(248) - (1549) + (335) + (994)$			$+(4621) - (766) - (994) + (2947)$
$+(248) + (1549) + (587) - (1714)$			$-(4621) + (766) + (1714) + (5161)$
$+(248) - (1549) + (578) - (1723)$			$+(4621) + (766) - (1723) - (5134)$
$+(251) - (820) + (578) + (985)$	$+(251) + (2434)$	$-(793) + (820)$	$-(2434) - (793) + (985) - (5134)$
$+(251) + (820) + (587) + (994)$			$+(2434) - (793) - (994) + (5161)$
$-(251) - (820) + (335) - (1714)$			$-(2434) + (793) + (1714) + (2947)$
$-(251) + (820) + (344) - (1723)$			$+(2434) + (793) - (1723) - (2974)$
$-(2192) - (1567) + (2297) + (2974)$	$+(2192) + (4619)$	$+(118) - (1567)$	$+(4619) - (118) - (344) + (2297)$
$-(2192) + (1567) + (2324) + (2947)$			$-(4619) - (118) + (335) - (2324)$
$+(2192) - (1567) + (4484) + (5161)$			$+(4619) + (118) + (587) - (4484)$
$+(2192) + (1567) + (4511) + (5134)$			$-(4619) + (118) - (578) + (4511)$
$-(2195) - (838) + (2297) + (5134)$	$-(2195) + (2432)$	$+(145) + (838)$	$-(2432) + (145) - (578) + (2297)$
$-(2195) + (838) + (2324) + (5161)$			$+(2432) + (145) + (587) - (2324)$
$+(2195) - (838) + (2947) + (4484)$			$-(2432) - (145) + (335) - (4484)$
$+(2195) + (838) + (2974) + (4511)$			$+(2432) - (145) - (344) + (4511)$

«Канонизация» формул

Канонизация необходима для возможности сравнения различных соотношений — с целью исключения «несущественных» повторений. «Существенными» различиями будем считать: 1) различный набор элементов в произведениях; 2) различные комбинации знаков у сомножителей⁵¹; при этом, пусть (для определённости) знак первого слагаемого у каждого сомножителя всегда будет положительным (это уменьшает «разнообразие» используемых произведений; поскольку порядок следования слагаемых в сомножителях был ранее уже выбран, подобное соглашение может привести лишь к необходимости изменения знака произведения в алгебраической сумме произведений).

```
function Canonize(Half::Array{Int,1})
# . . .
end

function Canonize(No::Int)
  Lo = (No-1)%3^4
  Hi = (No-1)÷3^4
  @assert Hi < 3^4
  (CanonicLo,Flip1) = Canonize(digits(Lo,base=3,pad=4))
  (CanonicHi,Flip2) = Canonize(digits(Hi,base=3,pad=4))
  return 1+CanonicLo+(CanonicHi*3^4), xor(Flip1,Flip2)
end
```

Рис. 20: Код «канонизации» (фрагмент); файл `Canonize.jl`.

Функция канонизации каждого из сомножителей (`Canonize(Half::Array{Int,1})`) включает в себя преобразование троичного представления «половинки» (путём взаимозамены «двоек» и «единиц»), чтобы её младший ненулевой разряд был единичным (если это не так), а также формирование признака совершённого преобразования. Код этой функции тут не приводится, поскольку, будучи тривиальным и прямолинейным, он выглядел бы слишком «громоздко» для столь «скромного» действия; возможно, проще всего реализовать это преобразование таблицей.

Функция канонизации всего произведения `Canonize(No::Int)` работает с передаваемым ей условным номером произведения в общей таблице произведений, «расщепляя» его (после приведения к «индексной» форме `No-1`) на две «половинки» и канонизируя по отдельности каждую из них, а затем превращая полученные результаты в номер с указанием необходимости перемены знака произведения, если знак изменился лишь у одной из «половинок».

«Семейства» формул

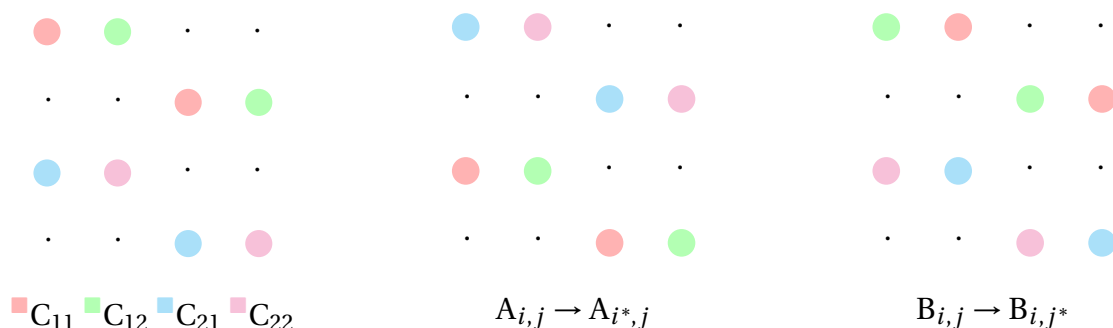


Рис. 21: «Конфигурация» произведений результата и её допустимые преобразования.

Нетрудно заметить: «конфигурация» «элементарных» произведений результата (рис. 21) может быть подвергнута некоторым преобразованиям (реально — перестановкам индексов матричных элементов матриц-сомножителей), оставляющим её неизменной. Подобными

⁵¹Например, из четырёх комбинаций знаков двучлена ($++$, $+-$, $-+$, $--$) существенно отличающимися будут первые две; оставшиеся (путем изменения знаков) приводятся к первым двум.

преобразованиями будут, например, $A_{i,j} \rightarrow A_{i^*,j}$ и $B_{i,j} \rightarrow B_{i,j^*}$ (где i^* и j^* означают «противоположные» значения индексов i и j , соответственно), а также композиция этих двух независимых преобразований. Это означает, что по уже имеющимся вариантам формул Штрассена можно получить другие⁵², используя функции преобразования с именами $T1()$, $T2()$, $T3()$, где троичные цифры «индексов» массива переставляются нужным образом.

```
include("Canonize.jl")

function T1(Num::Int)
    K = digits(Num-1,base=3,pad=8)
    K[1],K[2],K[3],K[4], K[5],K[6],K[7],K[8] =
    K[3],K[4],K[1],K[2], K[5],K[6],K[7],K[8]
    Canonize(1+[3^i for i=0:7]'*K)[1]
end

function T2(Num::Int)
    K = digits(Num-1,base=3,pad=8)
    K[1],K[2],K[3],K[4], K[5],K[6],K[7],K[8] =
    K[1],K[2],K[3],K[4], K[6],K[5],K[8],K[7]
    Canonize(1+[3^i for i=0:7]'*K)[1]
end

function T3(Num::Int)
    K = digits(Num-1,base=3,pad=8)
    K[1],K[2],K[3],K[4], K[5],K[6],K[7],K[8] =
    K[3],K[4],K[1],K[2], K[6],K[5],K[8],K[7]
    Canonize(1+[3^i for i=0:7]'*K)[1]
end
```

Все функции преобразований построены однотипно: условный номер произведения (уменьшаемый предварительно на единицу, поскольку нумерация индексов массива в *Julia* начинается с единицы) превращается в восемь цифр троичного представления. Далее троичные цифры переставляются нужным образом (через присваивание величинам одного кортежа значений другого кортежа; подобная запись возникла ещё в языке *Python* и теперь оказалась поддержанной и в *Julia*). В код функций, реализующих все преобразования, включается также финальное приведение любого номера произведения к «каноническому» значению (т. е., с первым положительным слагаемым в каждом сомножителе произведения). Но так как функция «канонизации» возвращает два значения: «канонизированный» условный номер произведения и (логический) признак изменения знака произведения при канонизации, для получения только номера надо выделять первое (целочисленное) значение (с индексом 1) в этом кортеже.

Верификация формул

Полезно, получив (или увидев) какие-то формулы, попытаться их проверить. Поскольку подавляющее большинство известных формул — это алгебраические суммы, имеет смысл использовать небольшую функцию для их вычисления (или её краткий синоним $VF()$).

```
function VerifyFormula(Numbers,Ops)
    @assert length(Numbers) == length(Ops)
    Result = zeros(Int,4,4)
    for (Op,ProdNo) in zip(Ops,Numbers)
        Result = eval(Expr(:call,Symbol(Op),Result,ProdTermCoeffs[:, :, ProdNo]))
    end
    Result
end

VF(Numbers,Ops) = VerifyFormula(Numbers,Ops)
```

Рис. 22: Функция верификации алгебраических сумм произведений (без коэффициентов).

⁵²Известно, что все возможные варианты этих формул «эквивалентны» уже открытой Штрассеном, но нигде явно (или, по крайней мере, в простом виде) не указано, каким образом получить их, причём **все**.

В цикле одновременного просмотра двух последовательностей: из знаков $+/-$ и (условных) номеров произведений — выполняется (при помощи функции `eval()`) «сконструированное» выражение (`Expr(:call, Symbol(Op), <Операнды>)`), состоящее из применения очередной указанной операции `Op` к операнду (матрице) `Result` (изначально заполненному нулями) и очередному набору (матрице) коэффициентов произведения с номером `ProdNo`; результат выполнения операции снова записывается в (матрицу) `Result`.

Функция `VF()` — это синоним функции `VerifyFormula()` с более коротким именем.

Пользуясь этой функцией, можно либо наглядно убедиться, что какое-то соотношение действительно порождает таблицу коэффициентов матричного элемента C_{ij} результата (аналогично выводам проверок на стр. 34), либо, предварительно определив эти таблицы:

```
C11Coeffs = Array{Int,2}(undef,4,4);
C11Coeffs[:,:] = [1,0,0,0, 0,0,0,0, 0,1,0,0, 0,0,0,0];
C22Coeffs = Array{Int,2}(undef,4,4);
C22Coeffs[:,:] = [0,0,0,0, 0,0,1,0, 0,0,0,0, 0,0,0,1];
C12Coeffs = Array{Int,2}(undef,4,4);
C12Coeffs[:,:] = [0,0,0,0, 1,0,0,0, 0,0,0,0, 0,1,0,0];
C21Coeffs = Array{Int,2}(undef,4,4);
C21Coeffs[:,:] = [0,0,1,0, 0,0,0,0, 0,0,0,1, 0,0,0,0];
```

Рис. 23: Таблицы элементов результата $C_{11}, C_{12}, C_{21}, C_{22}$ (файл `ResultCoeffs.jl`).

сравнивать (`@assert CijCoeffs == VerifyFormula([<Номера>], "<Знаки>")`) формулы и ожидаемые результаты, фиксируя (с помощью `@assert`⁵³) лишь несовпадения.

Paterson (1974)

В статье⁵⁴ приводится (в виде довольно оригинальной иллюстрации) вариант, вроде бы полученный из формул Штрассена с помощью “простого линейного преобразования”, однако само преобразование опущено; как можно выяснить из иллюстрации, имеется в виду такой набор соотношений (здесь он записан способом, отмеченным в работе⁵⁵; в круглых скобках над обозначениями произведений указаны введённые здесь их условные номера):

$$(83) \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + (733) \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + (2793) \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} + (442) \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} + (2264) \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + (4629) \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} + (5536) \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

Кстати, в приводимой там иллюстрации используется нестандартный порядок матричных элементов перемножаемых матриц ($A_{12}, A_{11}, A_{21}, A_{12}, B_{21}, B_{11}, B_{12}, B_{22}$), из которого можно заключить, что существует ещё одно преобразование, сохраняющее неизменной «конфигурацию» произведений результата: $A_{i,j} \rightarrow B_{j,i}, B_{i,j} \rightarrow A_{j,i}$ (вместо величин $A_{i,j}$ использовать $B_{j,i}$ и наоборот — при сохранении *выбранного ранее* порядка элементов в произведении); это позволяет дополнить набор «элементарных» преобразований $T1()$, $T2()$ (с их композицией $T3()$) ещё одним «элементарным» преобразованием $T4()$,

```
function T4(Num::Int)
    K = digits(Num-1, base=3, pad=8)
    K[1], K[2], K[3], K[4], K[5], K[6], K[7], K[8] =
    K[5], K[7], K[6], K[8], K[1], K[3], K[2], K[4]
    Canonicalize(1+[3^i for i=0:7]'*K)[1]
end
```

в результате чего будут возможны и различные варианты их композиций ($T5()$, $T6()$, $T7()$).

Для верификации указанного набора соотношений можно использовать такой код (см. сл. стр.):

⁵³Применяя вместо макроса `@assert` другой, можно преобразовывать такие «проверочные» фрагменты в стандартную запись соответствующих формул, так что подобный способ записи годится и как «конструктивный» способ их хранения. Приводимые здесь результаты верификации найденных в литературе вариантов соотношений даны именно в таком виде.

⁵⁴M.S. Paterson. Complexity of Product and Closure Algorithms for Matrices. *Proceedings of the International Congress of Mathematicians* (1974), 483–489.

⁵⁵V.P. Burichenko. On symmetries of the Strassen algorithm. [arXiv:1408.6273v1](https://arxiv.org/abs/1408.6273v1) [cs.CC], 2014.

```
@assert C11Coeffs == VerifyFormula([83, 733], "+")
@assert C12Coeffs == VerifyFormula([83, 2793, 442, 2264], "++")
@assert C21Coeffs == VerifyFormula([83, 2793, 4629, 5536], "+++")
@assert C22Coeffs == VerifyFormula([83, 2793, 4629, 442], "++++")
```

Рис. 24: Код верификации для Paterson (1974).

Kolen, Bruce (2001)

Публикация⁵⁶ использует стандартный способ нумерации элементов матриц, при этом формулы (1) содержат (ни на что не влияющие) опечатки (в третьей формуле написано $X_{21}Y_{21}$ вместо $X_{21}Y_{11}$, а в четвёртой — $X_{21}Y_{22}$ вместо $X_{21}Y_{12}$). Приведены формулы Штрассена (стр. 162) и ещё один (полученный экспериментально с помощью эволюционного поиска) вариант формул (кстати, со всеми «канонизированными» — в принятом здесь смысле — произведениями; стр. 164), имеющий небольшую неточность: слагаемое q_1 (произведение с условным номером 2440) в формуле (10) для Z_{12} должно быть со знаком плюс, а не минус. В таблице 2 с обнаруженными формулами (стр. 39) этот последний вариант располагается в строке 8; (канонизированный вариант) формулы Штрассена — в строке 25.

```
# Kolen, Bruce (2001), p.162
@assert C11Coeffs == VerifyFormula([2297, 919, 2192, 2974], "+--+")
@assert C12Coeffs == VerifyFormula([4619, 2192], "+")
@assert C21Coeffs == VerifyFormula([118, 919], "+")
@assert C22Coeffs == VerifyFormula([2297, 4619, 118, 336], "+--+")
# Kolen, Bruce (2001), p.164
@assert C11Coeffs == VerifyFormula([89, 814], "+")
@assert C12Coeffs == VerifyFormula([2440, 89, 4477, 344], "+--+")
@assert C21Coeffs == VerifyFormula([4477, 2251, 2974, 814], "-+-")
@assert C22Coeffs == VerifyFormula([2440, 2251], "+-")
```

Рис. 25: Код верификации для Kolen, Bruce (2001).

Oh, Moon (2010)

В публикации⁵⁷ матричные элементы матрицы B занумерованы чуть иначе (в I -порядке), нежели элементы матриц A и C (в Z -порядке), и все — с помощью одного индекса; это следует иметь в виду при использовании приводимых там соотношений. При этом, надо сказать, произведения из формул Штрассена выглядят гораздо «лучше» — в смысле симметричного вхождения индексов элементов в различные произведения, — но в остальных соотношениях подобный эффект не наблюдается.

Для поисков авторы использовали генетический алгоритм; в результате его работы были получены 608 наборов соотношений (в т. ч. и формулы Штрассена), что дало авторам возможность объявить свою публикацию “первой работой, где алгоритм Штрассена был воспроизведён автоматически”. Все полученные соотношения условно разбиты ими на группы — по количеству ненулевых элементов в каждом произведении⁵⁸ (в принятых здесь обозначениях это соответствует числу ненулевых α_i , $i = 1, \dots, 8$, в выражении (10)) — и представлено по одному соотношению из каждой группы⁵⁹. В некоторых группах найдено довольно мало соотношений, что оставляет определённый простор для обнаружения ещё неизвестных формул.

⁵⁶J.F. Kolen, P. Bruce. Evolutionary Search for Matrix Multiplication Algorithms. *Proceedings of the Fourteenth International Florida AI Research Society Conference (FLAIRS-01)* (2001), 161–165.

⁵⁷S. Oh, B. Moon. Automatic Reproduction of a Genius Algorithm: Strassen’s Algorithm Revisited by Genetic Search. *IEEE Trans. Evol. Comput.*, 14 (2): 246–250, 2010.

⁵⁸Эта схема не представляется удовлетворительной, поскольку в ней будут неразличимы, например, произведения вида $3+2$ (перемножаются 3 и 2 слагаемых) и $4+1$ (перемножаются 4 слагаемых и одно), — хотя они дают в «развёрнутом» виде совершенно разное число компонент (6 и 4 соответственно).

⁵⁹Стоит отметить: в части соотношений в публикации (группы 3–7) присутствуют вещественные коэффициенты (0.5), что было для авторов некоторой неожиданностью. При этом произведения по-прежнему имеют вид, соответствующий формуле (10) с целыми коэффициентами $\alpha_i \in \{-1, 0, +1\}$, $i = 1, \dots, 8$.

В представленных соотношениях имеются опечатки. Так, в группе 3 в выражении для C_1 (он же — C_{11} в принятых здесь обозначениях) знак произведения P_3 — не минус, а плюс. В группе 7 выражение для C_2 (это C_{12} в принятых здесь обозначениях) — неправильное (и вообще указанные произведения не могут сформировать этот элемент, т. к. ни у одного из них нет «элементарного» произведения A_1B_3), а в выражении для C_3 (элемент C_{21}) знак при произведении P_6 (**6448**) — минус, а не плюс. В последней группе 9 в произведении P_6 два раза употреблён элемент B_2 , но исправление первого из них на B_1 не приводит к заявляемому результату; не являются приводимые соотношения и т. н. вариантом Винограда.

Код верификации (с учётом исправления опечаток и без проблемных соотношений):

```
#      Group 1 (Strassen)
@assert C11Coeffs == VerifyFormula([2192, 919, 2297, 2974], "--++")
@assert C12Coeffs == VerifyFormula([4619, 2192], "--")
@assert C21Coeffs == VerifyFormula([118, 919], "--")
@assert C22Coeffs == VerifyFormula([4619, 118, 2297, 336], "--++")
#      Group 2
@assert C11Coeffs == VerifyFormula([2192, 2944, 1570, 2300], "--++")
@assert C12Coeffs == VerifyFormula([4619, 2192], "--")
@assert C21Coeffs == VerifyFormula([2192, 2944, 2300, 122], "--++")
@assert C22Coeffs == VerifyFormula([2192, 334, 2300, 122], "--++")
#      Group 3
@assert C11Coeffs == VF([4619, 793], "--") + 0.5VF([5477, 6227], "--")
@assert C12Coeffs == VerifyFormula([4619, 2192], "--")
@assert C21Coeffs == VerifyFormula([793, 820], "--")
@assert C22Coeffs == VF([2192, 820], "--") + 0.5VF([3281, 5477], "--")
#      Group 4
@assert C11Coeffs == 0.5VF([3592, 4214, 3998], "--") + VF([5108, 2245], "--")
@assert C12Coeffs == 0.5VF([3592, 4214, 3998], "--") + VF([2245], "--")
@assert C21Coeffs == 0.5VF([3592, 4214, 3998], "--") + VF([263, 361], "--")
@assert C22Coeffs == 0.5VF([3592, 4214, 3998], "--") + VF([263], "--")
#      Group 5
@assert C11Coeffs == 0.5VF([329, 1265, 613, 1592], "--")
@assert C12Coeffs == 0.5VF([329, 1265, 613, 1592], "--") + VF([6160], "--")
@assert C21Coeffs == 0.5VF([329, 1265, 613, 887], "--")
@assert C22Coeffs == 0.5VF([329, 1265, 613, 887], "--") + VF([5698], "--")
#      Group 6
@assert C11Coeffs == 0.5VF([2435, 4625, 4270, 3271], "--") + VF([2324], "--")
@assert C12Coeffs == 0.5VF([2435, 4625], "--")
@assert C21Coeffs == 0.5VF([2435, 4625, 4270, 3271, 644, 395], "--")
@assert C22Coeffs == 0.5VF([2435, 4625, 644, 395], "--") + VF([2324], "--")
#      Group 7
@assert C11Coeffs == VF([83, 733], "--")
#@assert C12Coeffs == ???
@assert C21Coeffs == VF([733], "--") + 0.5VF([3250, 5701, 6448], "--")
@assert C22Coeffs == VF([733, 2980], "--") + 0.5VF([3250, 5701, 6448], "--")
#      Group 8
@assert C11Coeffs == VF([83, 733], "--")
@assert C12Coeffs == VF([83, 2251, 2560, 395], "--")
@assert C21Coeffs == VF([733, 2440, 3271, 2560], "--")
@assert C22Coeffs == VF([2440, 2251], "--")
#      Group 9
#      ???
```

Рис. 26: Код верификации для **Oh, Moon (2010)**.

Deng, Zhou, Min, Zhu (2010)

В статье⁶⁰ для обнаружения вариантов формул Штрассена использовался т. н. алгоритм случайного поиска, характеризуемый в сравнении с генетическими алгоритмами большей скоростью работы. Были получены многочисленные соотношения, дающие в результате

⁶⁰S. Deng, Y. Zhou, H. Min, J. Zhu. Random Search Algorithm for 2x2 Matrices Multiplication Problem. *Third International Workshop on Advanced Computational Intelligence* (2010), 409–413.

более полутысячи уникальных вариантов, разнесённых авторами в 10 различных групп (аналогично тому, как это было сделано в предыдущей публикации). Для записи примеров из каждой группы применена уже упомянутая выше одноиндексная нумерация элементов матриц с нестандартным порядком следования элементов матрицы B .

Замечены небольшие неточности: в группе 4 произведение P_1 (с условным номером **253**) в выражении для элемента результата C_1 (он же — C_{11}) и в выражении для элемента C_3 (он же — C_{21}) должно иметь знак минус, а не плюс.

Код верификации⁶¹ (с учётом исправленной опечатки):

```
#      Group 1
@assert C11Coeffs == VF ([83, 733], "+")
@assert C12Coeffs == VF ([733, 280, 2957, 3440], "--+-")
@assert C21Coeffs == VF ([83, 4645, 3440, 6167], "-+++")
@assert C22Coeffs == VF ([280, 4645], "+-")
#      Group 2
@assert C12Coeffs == VF ([83, 388, 2258, 2576], "---+")
@assert C21Coeffs == VF ([83, 2441, 3268, 2576], "--+-")
@assert C22Coeffs == VF ([83, 2441, 388, 2576], "+++-")
#      Group 3
@assert C12Coeffs == VF ([83], "-")+0.5VF ([2258, 377, 2579], "-+++")
@assert C21Coeffs == VF ([83, 733, 2441, 3298, 2579], "-+---")
@assert C22Coeffs == VF ([83, 2441], "+")+0.5VF ([2258, 377, 2579], "---")
#      Group 4
@assert C11Coeffs == VF ([253, 2215], "--")+0.5VF ([365, 5180, 4232, 6187], "+++-")
@assert C12Coeffs == VF ([253, 2215], "+-")+0.5VF ([4232, 6187, 4019], "---")
@assert C21Coeffs == VF ([253, 2215], "-")+0.5VF ([365, 5180, 4019], "+---")
@assert C22Coeffs == VF ([253, 2215], "+")
#      Group 5
@assert C11Coeffs == VF ([86, 1543], "+-")
@assert C12Coeffs == VF ([86, 4627, 335, 2290], "--+-")
@assert C21Coeffs == VF ([1543, 2224, 2290, 2947], "+---")
@assert C22Coeffs == VF ([2224, 4627], "+")
#      Group 6
@assert C12Coeffs == VF ([254, 1543, 5935, 6166], "+++-")
@assert C21Coeffs == VF ([2218, 1543, 2953, 5935], "-+--")
@assert C22Coeffs == VF ([2218, 1543, 5935, 6166], "+---")
#      Group 7
@assert C12Coeffs == VF ([1543, 307], "+-")+0.5VF ([6467, 6227], "-+")
@assert C21Coeffs == VF ([86, 2458], "--")+0.5VF ([3281, 6467], "+")
@assert C22Coeffs == VF ([307, 2458], "+")
#      Group 8
@assert C11Coeffs == VF ([92], "+")+0.5VF ([6160, 4222, 5449], "--+")
@assert C12Coeffs == VF ([92, 329], "-")+0.5VF ([6160, 4222, 5449], "-+-")
@assert C21Coeffs == VF ([760], "+")+0.5VF ([6160, 4222, 5449], "+++-")
@assert C22Coeffs == VF ([760, 5140], "+-")+0.5VF ([6160, 4222, 5449], "+---")
#      Group 9
@assert C11Coeffs == 0.5VF ([815, 1547], "+")
@assert C12Coeffs == VF ([1001], "+")+0.5VF ([815, 1547, 4243, 3298], "-+++")
@assert C21Coeffs == VF ([1001], "+")+0.5VF ([815, 1547, 608, 377], "-+---")
@assert C22Coeffs == 0.5VF ([815, 1547, 608, 377, 4243, 3298], "++---")
#      Group 10
@assert C11Coeffs == 0.5VF ([329, 631, 1610, 1283], "+---")
@assert C12Coeffs == VF ([6160], "-")+0.5VF ([329, 631, 1610, 1283], "++---")
@assert C21Coeffs == 0.5VF ([329, 631, 851, 1283], "-+---")
@assert C22Coeffs == VF ([5698], "-")+0.5VF ([329, 631, 851, 1283], "--+-")
```

Рис. 27: Код верификации для Deng, Zhou, Min, Zhu (2010).

Что интересно: соотношения для матричных элементов в некоторых группах содержат пары (**253**, **2215** — в группе 4) или даже тройки (**4222**, **5449**, **6160** — в группе 8, **329**, **631**, **1283** — в группе 10), входящие в выражения для всех матричных элементов. При этом в группе 4 остальные произведения (если входят) имеют числовой коэффициент 0.5, а тройки в группах 8 и 10 сами обладают этим коэффициентом, причём в случае группы 10 тройка — почти что четвёрка: в двух случаях она дополнена произведением **1610**, а в двух — произведением **851**.

⁶¹Поскольку выражения для C_{11} в группах 1–3 (а также 5–7) одинаковы, в коде они не дублируются.

Zhou, Lai, Li, Dong (2013)

Все приводимые здесь⁶² формулы (10 типов) содержат «канонизированные» произведения в принятом выше понимании. В произведении p_4 из типа 10 имеется опечатка: слагаемое y_{22} дано со знаком плюс, а должно быть со знаком минус (давая произведение с условным номером **5680**). Код верификации приводимых формул (и неявно — они сами):

```
# Type 1
@assert C11Coeffs == VF ([2191, 145, 5249, 5174], "+--")
@assert C12Coeffs == VF ([245, 2191], "+")
@assert C21Coeffs == VF ([145, 838], "+")
@assert C22Coeffs == VF ([245, 838, 5681, 5249], "--+")
# Type 2
@assert C11Coeffs == VF ([91, 2441, 3244, 2528], "--+")
@assert C12Coeffs == VF ([91, 2441, 332, 2528], "++-")
@assert C21Coeffs == VF ([91, 757], "+")
@assert C22Coeffs == VF ([91, 332, 2258, 2528], "---")
# Type 3
@assert C11Coeffs == VF ([2215, 2921], "-") + 0.5VF ([6158, 4270, 6458], "+")
@assert C12Coeffs == VF ([2215], "+") + 0.5VF ([6158, 4270, 6458], "+")
@assert C21Coeffs == VF ([253, 2215, 2921, 644, 6458], "+--")
@assert C22Coeffs == VF ([253, 2215], "+")
# Type 4
@assert C11Coeffs == VF ([83, 733], "+")
@assert C12Coeffs == VF ([83, 733], "-") + 0.5VF ([2987, 620, 6467], "---")
@assert C21Coeffs == VF ([83, 733], "-") + 0.5VF ([3251, 5728, 6467], "+")
@assert C22Coeffs == VF ([83, 733], "--") + 0.5VF ([3251, 2987, 620, 5728], "+--")
# Type 5
@assert C11Coeffs == VF ([734, 1541], "+")
@assert C12Coeffs == VF ([734, 4645, 1757, 5134], "++-")
@assert C21Coeffs == VF ([1541, 280, 578, 1757], "--")
@assert C22Coeffs == VF ([280, 4645], "+")
# Type 6
@assert C11Coeffs == VF ([812, 737], "+")
@assert C12Coeffs == VF ([812, 2218, 1325, 5698], "+--")
@assert C21Coeffs == VF ([254, 812, 631, 1325], "+--")
@assert C22Coeffs == VF ([254, 812, 1325, 5698], "+--")
# Type 7
@assert C11Coeffs == VF ([89, 814], "+")
@assert C12Coeffs == VF ([814, 280], "-") + 0.5VF ([5711, 5471], "-")
@assert C21Coeffs == VF ([89, 4645], "--") + 0.5VF ([5471, 4283], "+")
@assert C22Coeffs == VF ([280, 4645], "+")
# Type 8
@assert C11Coeffs == VF ([2920], "+") + 0.5VF ([2258, 134, 4523], "++")
@assert C12Coeffs == VF ([326], "+") + 0.5VF ([2258, 134, 4523], "---")
@assert C21Coeffs == VF ([2920, 868], "-") + 0.5VF ([2258, 134, 4523], "+")
@assert C22Coeffs == VF ([326, 2441], "-") + 0.5VF ([2258, 134, 4523], "+")
# Type 9
@assert C11Coeffs == 0.5VF ([815, 1547], "+")
@assert C12Coeffs == VF ([1730], "+") + 0.5VF ([815, 1547, 6187, 5728], "+--")
@assert C21Coeffs == VF ([1730], "-") + 0.5VF ([815, 1547, 365, 620], "--")
@assert C22Coeffs == 0.5VF ([815, 1547, 365, 620, 6187, 5728], "--++")
# Type 10
@assert C11Coeffs == 0.5VF ([2921, 5167, 4505, 1610], "++")
@assert C12Coeffs == VF ([6158], "+") + 0.5VF ([2921, 5167, 4505, 1610], "+--")
@assert C21Coeffs == 0.5VF ([2921, 5167, 851, 4505], "--")
@assert C22Coeffs == VF ([5680], "-") + 0.5VF ([2921, 5167, 851, 4505], "--")
```

Рис. 28: Код верификации для Zhou, Lai, Li, Dong (2013).

Точно так же, как и в соотношениях предыдущей публикации, в некоторых группах в соотношениях для всех элементов участвуют пары (815, 1547 для типа 9) или тройки произведений (134, 2258, 4523 — для типа 8, 2921, 4505, 5167 — для типа 10), а в части групп (типы 9 и 10) половина выражений для матричных элементов снова содержит общий вещественный коэффициент 0.5. Кроме того, у соотношений типа 10 один из элементов представлен алгебраической суммой целых 6 произведений (все — с коэффициентом 0.5).

⁶²Y. Zhou, X. Lai, Y. Li, W. Dong. Ant Colony Optimization With Combining Gaussian Eliminations for Matrix Multiplication. *IEEE Trans. Cybernetics*, **43** (1): 347–357, 2013.

Обсуждение результатов

Итак, с помощью (не совсем уж «прямолинейного») перебора найдены (все?) соотношения (включая и формулы Штрассена⁶³), где два элемента матрицы результата (C_{11} , C_{22} или C_{12} , C_{21}) представлены алгебраическими суммами двух произведений (каждое из которых не является «элементарным»), а остальные два — суммами четырёх. Эти соотношения — вероятно, самые простые (в смысле «сложности» используемых произведений); их строение можно охарактеризовать описанием вида $(1)(2):(2)(1)$, $(1)(2):(2)(1):(2)(2):(2)(2)$, $(1)(2):(2)(1):(2)(2):(2)(2)$, $(1)(2):(2)(1)$ для последовательности элементов результата, где цифры в скобках означают число слагаемых в присутствующих произведениях (они перепорядочены по возрастанию количества используемых элементов первой матрицы).

Обнаружены также (не приведённые здесь) 32 соотношения, где один из элементов матрицы результата выражен «традиционно»: с помощью «элементарных» произведений (описание: $(1)(1):(1)(1)$, $(1)(1):(2)(1):(3)(3):(4)(2)$, $(1)(1):(1)(2):(2)(4):(3)(3)$, $(1)(2):(2)(1)$). Они содержат приводимые в публикациях *Deng, Zhou, Min, Zhu (2010)* (группа 1) и *Zhou, Lai, Li, Dong (2013)* (тип 1) примеры, но почему-то не включают пример из *Oh, Moon (2010)* (группа 8) — вероятно, из-за пока ещё не обнаруженной ошибки в коде. . .

Замечено, что имеются преобразования (перестановки индексов матричных элементов), позволяющие получить другие соотношения по уже имеющимся.

Все введённые выше функции преобразования ($T1()$, $T2()$, $T3()$, $T4()$, а также упомянутые и легко получаемые $T5()$, $T6()$, $T7()$) отображают целое значение в целое. Тем не менее, их можно применить и к последовательностям, причём в *Julia* для этого есть как минимум четыре способа. В трёх из них появляется символ «точка», введённый для поэлементных операций.

Поэлементное применение какой-либо функции $T()$ возможно с помощью добавления точки после имени функции: $T.<Последовательность>$. Используя операцию «конвейеризации» ($|>$), можно получить эквивалентную предыдущей записи применения функции, но здесь поэлементно будет применяться операция (для них точка как признак поэлементности располагается впереди): $<Последовательность> .|> T$. Для случаев массового употребления поэлементных операций имеется, как уже упоминалось ранее, специальный макрос $@.$, поэтому можно также воспользоваться им: $@.<Последовательность> |> T$. И, наконец, всегда остаётся возможность (она есть не только в *Julia*), предоставляемая функцией $map()$, специально созданной для подобных действий: $map(T, <Последовательность>)$.

Упомянутые выше преобразования ($A_{i,j} \rightarrow A_{i^*,j}$; $B_{i,j} \rightarrow B_{i,j^*}$; $A_{i,j} \rightarrow B_{j,i}$, $B_{i,j} \rightarrow A_{j,i}$) никак не изменяют количество слагаемых в произведениях и не выводят за пределы групп, введённых в ряде публикаций для классификации формул. С помощью этих преобразований (дополненных финальной канонизацией) соотношения⁶⁴, приведённые в таблице, могут быть «превращены» друг в друга (для примера указаны «образы» первого соотношения).

```
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T1)
[118, 1567, 2192, 4619, 344, 2297, 2974]
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T2)
[248, 4621, 766, 1549, 344, 985, 2974]
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T3)
[280, 4645, 734, 1541, 344, 1001, 2974]
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T4)
[812, 737, 2458, 307, 578, 1001, 5134]
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T5)
[820, 793, 2434, 251, 578, 985, 5134]
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T6)
[2432, 2195, 838, 145, 578, 2297, 5134]
julia> print([86, 1543, 2224, 4627, 344, 2281, 2974] .|> T7)
[2440, 2251, 814, 89, 578, 2281, 5134]
julia>
```

⁶³Вследствие «канонизации» произведений формулы Штрассена представляются набором произведений **118, 1567, 2192, 4619, 344, 2297, 2974** (в оригинальных формулах использованы **336** и **919** вместо **344** и **1567**, соответственно), в результате чего изменились знаки произведений P_2 и P_5 (строка 25 таблицы 2).

⁶⁴Точнее — наборы условных номеров, т. к. из-за применяемой финальной канонизации могут изменяться знаки произведений в соотношениях, в результате чего сами соотношения становятся немного другими.

Видно, что эти преобразования позволяют перейти от соотношений из одной группы таблицы 2 к какому-то из соотношений любой другой её группы; правда, пока неясно, как можно осуществить «переход» в пределах какой-либо группы.

Хотя опубликованные варианты формул используют произведения из введённой выше таблицы `ProdTermCoeffs`, т. е., имеют коэффициенты α_i , $i = 1, \dots, 8$ из набора $\{-1, 0, +1\}$ в сомножителях произведений, однако, возможны и многочисленные менее тривиальные соотношения, вероятно, не столь интересные с практической точки зрения, но всё-таки. . .

Оказывается⁶⁵, в соотношения (1)–(7) можно добавить параметр \mathbf{R} , считая его матрицей такого же размера, что и все элементы A_{ij} , B_{ij} (или скалярным в случае скалярных величин A_{ij} , B_{ij}), так что требуемые семь произведений будут выглядеть так:

$$\begin{aligned} P_1 &= (A_{21} + A_{22}\mathbf{R}^{-1})B_{11}, \\ P_2 &= A_{22}(-\mathbf{R}^{-1}B_{11} + B_{21}), \\ P_3 &= (A_{11}\mathbf{R} + A_{12})B_{22}, \\ P_4 &= A_{11}(B_{12} - \mathbf{R}B_{22}), \\ P_5 &= (-A_{11} + A_{21})(B_{11} + B_{12}), \\ P_6 &= (A_{11} + A_{22}\mathbf{R}^{-1})(B_{11} + \mathbf{R}B_{22}), \\ P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Нетрудно проверить, что для набора этих произведений формулы (8) тоже справедливы, — если произведение $\mathbf{R}^{-1}\mathbf{R}$ даёт единичную матрицу (или единицу — в скалярном случае).

Это порождает (в данном случае — из формул Штрассена) семейство аналогичных формул. Вот, например, два набора произведений, использующих простейшие коэффициенты:

$$\begin{array}{ll} P_1 = (A_{21} + 2A_{22})B_{11}, & P_1 = (A_{21} + 0.5A_{22})B_{11}, \\ P_2 = A_{22}(-2B_{11} + B_{21}), & P_2 = A_{22}(-0.5B_{11} + B_{21}), \\ P_3 = (0.5A_{11} + A_{12})B_{22}, & P_3 = (2A_{11} + A_{12})B_{22}, \\ P_4 = A_{11}(B_{12} - 0.5B_{22}), & P_4 = A_{11}(B_{12} - 2B_{22}), \\ P_5 = (-A_{11} + A_{21})(B_{11} + B_{12}), & P_5 = (-A_{11} + A_{21})(B_{11} + B_{12}), \\ P_6 = (A_{11} + 2A_{22})(B_{11} + 0.5B_{22}), & P_6 = (A_{11} + 0.5A_{22})(B_{11} + 2B_{22}), \\ P_7 = (A_{12} - A_{22})(B_{21} + B_{22}). & P_7 = (A_{12} - A_{22})(B_{21} + B_{22}). \end{array}$$

Вероятно, подобным же образом можно строить (изменяемые непрерывно!) семейства соотношений для любого из уже известных вариантов формул (включая и те, что имеют дробные множители при отдельных произведениях).

Уже довольно давно известно, что любые соотношения для умножения матриц 2×2 , использующие семь произведений, эквивалентны формулам, найденным Штрассеном⁶⁶⁶⁷. В последнее время определены возможные группы симметрии для близких к оптимальным разложениям тензора умножения матриц⁶⁸ (поскольку алгоритм Штрассена — разложение его в сумму семи одноранговых тензоров); отмечено, что некоторые симметрии проявляют себя только в трilinearном представлении алгоритма, которое инвариантно относительно циклической перестановки трёх матриц этого представления⁶⁹; решаются вопросы эквивалентности соотношений путём вычисления инвариантов действия групп симметрий⁷⁰.

Вероятно, скоро удастся явно указать такой набор преобразований, с помощью которых подобные соотношения трансформируются из какого-либо одного во все остальные.

Отыскание подобных преобразований автор оставляет заинтересованному читателю.

⁶⁵N. Gastinel. Sur le calcul des produits de matrices. *Numer.Math.*, **17**: 222–229, 1971.

⁶⁶H.F. de Groote. On Varieties of Optimal Algorithms for the Computation of Bilinear Mappings I. The Isotropy Group of a Bilinear Mapping. *Theoretical Computer Science*, **7** (1): 1–24, 1978.

⁶⁷H.F. de Groote. On Varieties of Optimal Algorithms for the Computation of Bilinear Mappings II. Optimal Algorithms for 2×2 -Matrix Multiplication. *Theoretical Computer Science*, **7** (2): 127–148, 1978.

⁶⁸L. Chiantini, C. Ikenmeyer, J.M. Landsberg, G. Ottaviani. The Geometry of Rank Decompositions of Matrix Multiplication I: 2×2 Matrices. [arXiv:1610.08364](https://arxiv.org/abs/1610.08364), 2016.

⁶⁹C. Ikenmeyer, V. Lysikov. Strassen’s 2×2 Matrix Multiplication Algorithm: A Conceptual Perspective. [arXiv:1708.08083](https://arxiv.org/abs/1708.08083), 2019.

⁷⁰M.J.H. Heule, M. Kauers, M. Seidl. New Ways to Multiply 3×3 -Matrices. [arXiv:1905.10192](https://arxiv.org/abs/1905.10192), 2019.