DOI: 10.32517/2221-1993-2023-22-6-18-30

Е. А. Еремин
Пермский государственный гуманитарно-педагогический университет, г. Пермь, Россия

ТРИ ЭТЮДА ПО КОМПИЛЯЦИИ, ИЛИ КАК ИСПОЛЬЗОВАТЬ ЯЗЫК JULIA ДЛЯ ИЗУЧЕНИЯ ГЕНЕРИРУЕМОГО МАШИННОГО КОДА

Аннотация

Статья демонстрирует интересную для образования возможность языка программирования Julia выводить ассемблерный текст для любой написанной на Julia функции. Подробно рассмотрен исполняемый код для трех наглядных примеров: вычисление по формуле, разветвляющийся и циклический фрагменты. Показано, что компилятор Julia действительно генерирует тщательно оптимизированную программу, стараясь использовать выполняемые максимально быстро инструкции процессора Intel. Он успешно пытается обойтись без трудоемких арифметических операций (если удается, то значения вычисляются заранее, умножение заменяется сдвигом и т. п.). Особенно впечатляют результаты для целочисленных данных. Удается увидеть целый ряд оптимизационных приемов, в том числе учитывающих особенности самого алгоритма: например, компилятор способен предсказать результаты простейшего цикла и исключить этот цикл из итоговой программы. Обсуждается также проблема контроля переполнения в программе.

Проведенное рассмотрение убедительно подтверждает полезность теоретических знаний для разработки эффективно работающей программы. Описанная методика анализа представляет интерес для образования, поскольку важность глубокого понимания современными специалистами сути процесса программирования трудно переоценить. Материал может быть полезен при обучении идеям программирования на разных уровнях. Детальность изложения делает статью доступной для чтения без специальной подготовки.

Ключевые слова: программирование, язык Julia, компиляция, оптимизация, эффективность, исполняемый код, машинный код, обучение программированию, методики обучения.

Контактная информация

Еремин Евгений Александрович, канд. физ.-мат. наук, доцент, доцент факультета информатики и экономики, Пермский государственный гуманитарно-педагогический университет, г. Пермь, Россия; *адрес:* 614990, Россия, г. Пермь, ул. Сибирская, д. 24; *e-mail:* evg_eremin@mail.ru

E. A. Eremin
Perm State Humanitarian Pedagogical University, Perm, Russia

THREE ETUDES ON COMPILING, OR HOW TO USE THE JULIA LANGUAGE TO STUDY GENERATED MACHINE CODE

Abstract

The article demonstrates the ability of the Julia programming language to output an assembly text for any function written in Julia, which is interesting for education. The executable code for three illustrative examples is examined in detail: calculation by formula, branching and cyclic fragments. It is shown that Julia compiler really generates a carefully optimized program, trying to use the fastest executing instructions of the Intel processor. It successfully tries to do without time-consuming arithmetic operations (if possible, the values are calculated in advance, multiplication is replaced by shift and so on). The results for integer data are especially impressive. It is possible to see a whole range of optimization techniques, including those that take into account the features of the algorithm itself: for example, the compiler is able to predict the results of the simplest loop and exclude this loop from the final program. The problem of overflow control in a program is also discussed.

The consideration carried out convincingly confirms the usefulness of theoretical knowledge for developing an effectively working program. The described analysis technique is interesting for education, because the importance of a deep understanding by modern specialists of the programming process essence cannot be overestimated. The material can be useful in teaching programming ideas at different levels. The detailed presentation makes the article readable without special preparation.

Keywords: programming, Julia language, compilation, optimization, efficiency, executable code, machine code, programming training, teaching methods.

> И только потом я понял, что отличает Julia от других языков. Julia разрушает барьер между высокоуровневым и ассемблерным кодом. Julia не просто позволяет писать код, который работает так же быстро, как код на C, но и дает возможность посмотреть на LLVM-представление функций и их сгенерированный ассемблерный код. И все это прямо в интерактивной среде.
>
> *Э. Миллер [35]*

1. Julia — новый язык программирования

Язык программирования Julia [39] появился относительно недавно: он стал общедоступным в феврале 2012 года [20]. Его создатели [25, 26] из Массачусетского технологического института (Massachusetts Institute of Technology — MIT) первоначально планировали применить язык для целей научного программирования (в первую очередь как быстродействующую замену пакета MATLAB). Однако результат превзошел ожидания создателей: родился не просто эффективный язык для вычислений и построения графиков, но общецелевой и многоплатформенный язык программирования. Автор недавно вышедшей книги [22] Э. Энгхейм, будучи профессиональным разработчиком ПО, написал об этом так: «Для того чтобы использовать Julia, вовсе не нужно быть гением или ученым. Julia — замечательный язык общецелевого программирования для всех! Я не ученый, и мне нравится использовать его уже более девяти лет. Вы обнаружите, что с Julia вы сможете решать задачи быстрее и элегантнее, чем раньше. И вишенкой на торте является то, что вычислительно емкий исходный код будет исполняться невероятно быстро. <...> С тех пор я испробовал много других языков программирования, но так и не приблизился к тому, чего добился на языке Julia. <...> Язык Julia помог мне вернуть радость от программирования» [22], Немалое число программистов разделяют такую эмоциональную симпатию к языку. Сошлемся, в частности, на результаты опроса 2021 года, в котором приняли участие свыше 83 тысяч разработчиков из 181 страны мира. Чтобы определить «самый любимый» язык, специалисты известного сайта Stack Overflow спросили у разработчиков, какой язык они использовали в прошлом году и на каком языке они хотели бы писать в следующем. Язык Julia по результатам опроса занял почетное пятое место (данные взяты из [10]).

Автор убедительно просит читателей не считать вводный абзац восхвалением нового «суперъязыка» программирования! Хотя и во многом согласен с приведенной выше оценкой.

Знакомство с литературой на английском языке подтверждает внимание мировой аудитории к языку. Приведем несколько примеров книг по Julia [36–38], в названиях которых обращает на себя внимание упоминание наиболее передовых областей современной информатики. Из публикаций на русском языке хотелось бы отметить статью обзорного характера [3], где помимо общей характеристики языка особое внимание уделено возможностям уже написанных математических библиотек. Подчеркнем, что, согласно базовым идеям создателей Julia [25], язык должен работать настолько быстро, чтобы позволять писать все библиотеки непосредственно на нем, а не подключать результаты трансляции с другого языка, например, с C или FORTRAN.

В работах [14, 19] дается характеристика нового языка, причем в статье [14] приводятся примеры решения на Julia простых практических задач. Статьи [11, 31] посвящены применению Julia в математике, а [4, 17, 30] демонстрируют полезность языка в других областях (от машинного обучения до биологии).

Особо хотелось бы отметить, что наши ведущие вузы уже обучают своих студентов языку Julia. И, что еще приятнее, делятся в Сети своими учебно-методическими пособиями по курсам данной тематики (см., например, [1, 2, 21]).

Таким образом, по имеющимся источникам можно сделать вывод, что язык Julia эффективно применяется в самых разнообразных областях, в том числе в образовании. Об особенностях и достоинствах языка можно написать отдельную статью. Но в данной публикации речь пойдет только об одном аспекте Julia — о возможности очень легко посмотреть, как выглядит машинный код для несложных фрагментов на языке программирования высокого уровня. По мнению автора, изучение данного вопроса полезно по нескольким причинам.

Во-первых, можно на собственном опыте проверить, действительно ли Julia генерирует эффективный код, как это везде утверждается.

Во-вторых, некоторое представление о том, как выглядит одна и та же программа для человека и для компьютера, весьма полезно и должно быть хотя бы раз продемонстрировано при обучении. Julia позволяет обойтись без вспомогательного программного обеспечения, которое приходилось для этих целей специально разрабатывать [6, 29].

Примечание. Возможно, данный тезис требует обоснования. Сошлемся как на образец на систематический курс [16], показывающий тесную взаимосвязь всех уровней работы компьютера. А в известных статьях [23, 28] утверждается, что необходимо рассматривать материал минимум на один уровень глубже, чем это требуется в данном учебном курсе.

Наконец, в-третьих, изучать поставленную проблему средствами Julia можно без особой предварительной подготовки, так что автор не может не поделиться с окружающими своим мнением о том, насколько это легко и приятно.

Завершая вводную часть, поясним, что многозначное слово ***этюд*** (от *франц.* étude, буквально — изучение, упражнение) в статье понимается следующим образом. Во-первых, как «*небольшое по объему произведение (научное, критическое), посвященное частному вопросу*» [13]. Во-вторых, как «*упражнение, служащее для развития и совершенствования техники какого-либо искусства*» [8]. И, наконец, в-третьих (что, возможно, не очень скромно), известны случаи, когда удачный этюд представляет самостоятельный интерес.

2. Как Julia оптимизирует программу

Одна из целей, поставленных при разработке языка Julia, заключалась в генерации эффективного машинного кода в момент выполнения программы. Для решения поставленной задачи была предложена следующая схема

трансляции [24], в ходе которой оптимизация возможна на нескольких этапах.

Сначала *исходный текст* на Julia анализируется и превращается в *абстрактные синтаксические деревья* (Abstract Syntax Tree — AST). Последние далее преобразуются в некоторое *внутреннее представление*, что позволяет проводить оптимизацию на уровне языка Julia.

Полученный результат транслируется в некоторый *промежуточный код* под названием LLVM IR [33, 34]. Расшифровывается это сочетание букв так. Речь идет о промежуточном представлении (**I**ntermediate **R**epresentation; первоначально также использовался термин *битовый код* [12], чтобы не путать с байткодом Java). Код предназначен для LLVM; сначала это называлось виртуальной машиной низкого уровня — **L**ow **L**evel **V**irtual **M**achine, но позднее от идеи виртуальной машины проект отказался, оставив только старое сокращенное название. На уровне LLVM IR также производится оптимизация, уже не зависящая от языка высокого уровня. Заметим, что текстовое представление LLVM-кода Julia может показать при вызове функции **code_llvm()**.

Наконец, на завершающем этапе LLVM IR стандартным образом преобразуется в *машинный код* конкретного процессора.

В Julia компиляция выполняется при вызове функции, причем результат запоминается и повторная компиляция уже не требуется.

Познакомившись в общих чертах с идеями компиляции, перейдем к экспериментам. Дальнейшее изложение предполагает, что язык Julia уже установлен на вашем компьютере. С этим не должно возникать проблем, поскольку установка среды и базовые принципы общения с ней неоднократно описаны (см., например, [1]). В ОС Windows (32-битная версия) автору даже не потребовалось проводить инсталляцию — оказалось достаточно разархивировать скачанный с официального сайта **julialang.org** дистрибутив (версия 1.9.2) и в традиционном каталоге **bin** найти исполняемый файл **julia.exe**. Для проведения описанных в данной публикации экспериментов можно больше никакое дополнительное программное обеспечение не использовать.

Итак, полагаем, что вы запустили указанный выше файл и получили приглашение к диалогу в виде текста «**julia>**» в начале строки. Можно набирать команду! Что именно вводить, описывается далее в этюдах 1–3. Отметим, что количество этюдов выбрано не случайно: существуют три базовые алгоритмические структуры (следование, ветвление (развилка), цикл), и каждую из них мы опробуем.

3. Этюд 1. Оптимальное вычисление арифметических выражений

Рассмотрим для начала, как компилятор Julia оптимизирует вычислительные формулы. При анализе нам могут потребоваться некоторые дополнительные сведения, которые, чтобы не отвлекаться от основного изложения, имеет смысл обсудить предварительно. Подготовленные читатели могут пропустить знакомый материал.

3.1. Этюд 1: предварительные сведения

3.1.1. Функция code_native()

Для просмотра машиной программы в Julia предусмотрена замечательная функция code_native(), которая выводит на экран текст исполняемого («родного») кода процессора. При обращении надо всегда указывать имя интересующей нас функции и тип аргумента (аргументов), для которого следует построить ее код. А далее в качестве последнего параметра приходится явно задавать синтаксис ассемблера: хочется видеть общепринятый синтаксис Intel, а тестируемая версия Julia по умолчанию устанавливает ассемблер AT&T (American Telephone and Telegraph; компания известна своей связью с разработкой ОС Unix).

Примечание. Между прочим, такой же выбор синтаксиса необходимо делать и при использовании встроенного ассемблера в среде Free Pascal.

Приведем пример. Пусть у нас имеется функция одной переменной $f(x)$ и мы хотим посмотреть ее код для аргумента типа Int32 (можно задать и другой, в том числе вещественный). Тогда необходимо ввести следующую команду:

```
code_native(f, (Int32,); syntax=:intel)
```

Поясним дополнительно, что у функции *f* мог быть не один аргумент; тогда второй параметр code_native() должен вмещать в себя несколько значений (такой набор значений называют *кортежем*). Запись (Int32,), которая выглядит несколько непривычно, как раз и описывает кортеж из одного значения.

3.1.2. Некоторые необходимые для анализа команды

Каждая программа для компьютера — это последовательность машинных инструкций (команд), которая представляется в двоичном (*машинном*) коде. Именно в таком виде машина, а точнее, ее процессор, выполняет программу. Чтобы облегчить программирование на таком низком уровне, используется специальный язык — *ассемблер*, где для удобства человека инструкции записываются в текстовой форме.

Любая инструкция ассемблера состоит из кода операции (его часто называют *мнемоникой*), который показывает, какое действие требуется выполнить, а также описания, где взять исходные данные (*операнды*) и куда поместить результат.

Рассмотрим несколько примеров, которые пригодятся нам при изучении кода этюда 1.

Пример 1.

```
mov ebp, esp
```

Приведенная инструкция **копирует** (переносит — move) содержимое из регистра **esp** в регистр **ebp** (по сути, это низкоуровневый эквивалент оператора присвоения **ebp** = **esp**).

Если операнд заключен в квадратные скобки, то он является **адресом памяти**, откуда необходимо прочитать данные.

Пример 2.

```
mov eax, dword ptr [ebp + 8]
```

Здесь в регистр **eax** копируется число из памяти по адресу, равному содержимому регистра **ebp** плюс 8. Запись задает только начальный адрес памяти, и по нему невозможно догадаться, сколько байт надо прочитать. Поэтому в команду приходится добавлять текст ***dword ptr***, который задает размер данных. Сокращение **dword** (double word — двойное слово) указывает на четырехбайтовые данные, потому что обычное слово (word) занимает два байта.

Пример 3.

```
shl eax, 5
```

А эта инструкция сдвигает (sh — начало слова shift) влево (буква «l» в конце мнемоники) на 5 двоичных разрядов содержимое **eax**. Результат заносится в тот же самый регистр. В итоге получается результат, эквивалентный умножению на $2^5 = 32$ (см. п. 3.1.4 ниже).

Пример 4.

```
or eax, 1
```

Данная инструкция выполняет логическую операцию ИЛИ (**or**) между содержимым регистра **eax** и константой 1. Правила этой операции хорошо известны. Отметим здесь лишь то, что младший бит регистра в результате операции всегда будет установлен в единицу, в то время как все остальные останутся без изменения.

3.1.3. Команды SSE

Технология *SSE* (***S**treaming **S**IMD **E**xtensions* — потоковое SIMD-расширение процессора; SIMD — **S**ingle **I**nstruction, **M**ultiple **D**ata, т. е. одна инструкция — множество данных) впервые была представлена в процессорах Pentium III. Ее основное назначение состоит в организации параллельного выполнения операций над несколькими вещественными числами [18]. В процессор добавлен набор специализированных регистров, которые обозначаются **xmm**. Каждый регистр имеет емкость 128 бит, что одновременно позволяет обрабатывать четыре 32-битных значения с плавающей запятой одинарной точности (*single*) или два 64-битных значения двойной точности (*double*). Параллельно обрабатываемые данные называются *упакованными* (packed, к мнемонике команды добавляется буква «p»). Допускается действие только над одним вещественным числом — такая инструкция называется *скалярной* (scalar, добавляется «s»). В зависимости от того, какая точность чисел используется — одинарная или двойная, — к обозначению команды приписывается еще одна буква — «s» или «d» соответственно. Например, сочетание «sd» говорит о скалярной обработке одного вещественного числа двойной точности.

При изучении машинного кода этюда 1 нам встретятся следующие **команды SSE:**

- `movss xmm0, dword ptr [ebp + 8]`
  Эта инструкция загружает одно скалярное число одинарной точности в регистр **xmm0**. Исходное число берется из памяти, о чем говорят квадратные скобки в записи команды, из адреса, вычисляемого как содержимое регистра **ebp** плюс 8. Запись ***dword ptr***, как мы уже знаем из раздела 3.1.2, указывает в явном виде, что размер числа равен 4 байтам.
- `addss xmm0, dword ptr [.LCPI0_1]`
  Здесь к такому же по точности числу в **xmm0** прибавляется (**add**) из памяти «техническая» константа с именем **LCPI0_1**. Аналогичным образом следующая команда описывает операцию умножения (**mul**).
- `mulss xmm0, dword ptr [.LCPI0_0]`

3.1.4. Умножение с помощью сдвига

Пусть у нас есть число 12 в привычной десятичной системе счисления. Для умножения его на 10 достаточно формально приписать к нему справа ноль — получится 120. Для умножения на $10^2 = 100$ нужно приписать два нуля (1200), и т. д. Если внимательно посмотреть, то в данной операции исходное число фактически сдвигается влево на количество разрядов, равное показателю степени, а «освободившиеся» при этом младшие разряды заполняются нулями.

В двоичной системе все работает точно так же, только множители надо брать кратными не 10, а 2 (т. е. кратными основанию системы счисления). Например, для «быстрого» умножения числа $101_2 = 5_{10}$ на $32 = 2^5$ достаточно сдвинуть код на 5 разрядов влево: $10100000_2 = 160_{10}$ (см. пример 3 в разделе 3.1.2).

Подчеркнем, что все сказанное относится только к целым числам. Для умножения вещественного числа на степень двойки (фактически для этого достаточно просто увеличить порядок числа) существует специальная инструкция **fscale**. Но из-за хранения вещественных чисел в виде двух компонентов (порядок и значащая часть) к указанной инструкции потребуется добавить несколько дополнительных команд. Как (длинно) это будет выглядеть, можно посмотреть на с. 92—93 методического пособия [9].

3.2. Этюд 1: постановка задачи

Перейдем, наконец, непосредственно к содержанию этюда.

Пусть мы хотим посмотреть, как компилятор Julia раскрывает арифметические выражения. Для этого возьмем простую функцию

$$f(x) = 3 + 32x - 2.$$

Легко заметить, что в целях проверки способностей компилятора в ней заложены два «подводных камня». Во-первых, можно заранее вычислить $3 - 2 = 1$ и исключить одну из арифметических операций. Во-вторых, переменная x умножается на степень двойки, что, как мы видели в разделе 3.1.4, позволяет заменить длинную операцию умножения быстрой инструкцией сдвига. Посмотрим, сумеет ли компилятор преодолеть искусственно созданные нами трудности.

3.3. Этюд 1: решение задачи

Начинаем работу. Запускаем Julia и набираем следующие интуитивно легко понятные команды (листинг 1).

Тут все ясно без особых пояснений. Обратим только внимание на два обстоятельства. Во-первых, Julia, стремясь обеспечить максимальный комфорт математикам, позволяет опускать знак умножения в записи $32x$ (языки программирования этого обычно не разрешают). Во-вторых, функция прекрасно работает и для целых, и для вещественных чисел.

команда 1 ответ (готово)	`julia> f(x)=3+32x-2` `f (generic function with 1 method)`
команда 2 ответ (Int)	`julia> f(2)` `65`
команда 3 ответ (Float)	`julia> f(2.0)` `65.0`

Листинг 1. Описание и вызов функции f(x)

А теперь попытаемся заглянуть внутрь нашей подопытной функции. Для этого аккуратно наберем команду вызова функции **code_native()**, описанную в разделе 3.1.1, и увидим на экране следующее (см. третий столбец таблицы в листинге 2; первые два — это вспомогательные комментарии).

Важное примечание. Код для 64-битной версии ОС будет несколько отличаться. 32-битная архитектура IA32 была выбрана автором, так как по ней больше литературы, чем по Intel 64.

На первый взгляд, сложно, но давайте не спеша разберемся.

Разобьем полученный текст программы на четыре фрагмента F0 — F3. (Сразу о хорошем: собственно программа есть самый короткий фрагмент F2.)

Фрагмент F0 содержит некоторый стандартный заголовок. Он практически неизменен и не содержит особо интересной для нас сейчас информации.

Входной фрагмент F1 служит для подготовки извлечения значения аргумента *x*. Параметры в машинных программах, как правило, передаются через стековую память, доступ к которой осуществляется через специальный регистр процессора **esp**. Как следует из строки с номером 2, содержимое этого регистра копируется

Команда		`julia> code_native(f, (Int32,);syntax=:intel)`

F0 — заголовок		`.text`
		`.file "f"`
		`.globl julia_f_152 # -- Begin function julia_f_152`
		`.p2align 4, 0x90`
		`.type julia_f_152,@function`

F1 — вход		`julia_f_152: # @julia_f_152`
		`; ┌ @ REPL[1]:1 within `f``
		`.cfi_startproc`
		`# %bb.0: # %top`

	1	`push ebp`
		`.cfi_def_cfa_offset 8`
		`.cfi_offset ebp, -8`
	2	`mov ebp, esp`
		`.cfi_def_cfa_register ebp`
	3	`and esp, -16`
	4	`sub esp, 16`
F2 — **код программы**	5	`mov eax, dword ptr [ebp + 8]`
		`; │┌ @ int.jl:88 within `*``
	6	`shl eax, 5`
		`; │└`
		`; │┌ @ int.jl:86 within `-``
	7	`or eax, 1`
		`; │└`
F3 — выход	8	`mov esp, ebp`
	9	`pop ebp`
	10	`ret`
		`.Lfunc_end0:`
		`.size julia_f_152, .Lfunc_end0-julia_f_152`
		`.cfi_endproc`
		`; └`
		`# -- End function`
		`.section ".note.GNU-stack","",@progbits`

Листинг 2. Этюд 1, аргумент — Int32

в регистр **ebp**; с его помощью позднее в F2 будет прочитано значение аргумента.

Фрагмент F2 наиболее интересен для нас, так что подробно обсудим его чуть позднее.

И, наконец, последний фрагмент F3 восстанавливает первоначальное состояние регистров и обеспечивает выход из функции по команде **ret**. Заметим, что в F3 нет команд возврата результата функции через стек; это значит, что ответ возвращается через регистр, где был сосчитан, т. е., как мы увидим, через **eax**.

А теперь вернемся к центральному фрагменту F2. Если не обращать внимания на комментарии, которые пишутся после знака «;», остается всего три строки, и в каждой из них — по одной машинной команде. При их расшифровке полезно заглядывать в раздел 3.1.2.

В строке под номером 5 значение аргумента копируется в 32-битный регистр **eax**. Далее с ним будут производиться вычисления по нашей тестовой формуле. Возможно, у вас возник вопрос, зачем к адресу из регистра прибавляется 8. Все дело в том, что в первых 8 байтах стека лежит информация для возврата из функции по команде **ret**, так что их надо пропустить.

Строка 6 свидетельствует о «грамотности» компилятора: он «заметил», что множитель есть степень двойки и заменил операцию умножения командой сдвига (см. разделы 3.1.2 и 3.1.4). Последняя команда находится в строке 7. Здесь тоже видна оптимальность реализации: вместо сложения и вычитания будет выполняться только одна операция. Но почему мы не видим ни сложения (**add**), ни вычитания (**sub**)? Дело в том, что прибавить надо 1, а текущее значение самого младшего бита, полученное в результате сдвига, всегда 0. 0 + 1 = 1 (перенос отсутствует). Очевидно, что выполнять сложение не требуется, а достаточно просто занести в младший бит 1. Что и делает команда логического ИЛИ (**or**). Подчеркнем, что логические операции всегда выполняются чуть быстрее, чем арифметические, поскольку в них не требуется обеспечивать перенос между разрядами.

Итак, мы имеем замечательный результат! Оба «подводных камня» компилятор успешно обошел, а всю формулу сумел вычислить всего за две машинные команды, причем даже не арифметические. Браво, Julia!!!

Попробуем посмотреть, как обстоит дело для других типов данных. Для этого при вызове функции **code_native()** вместо Int32 достаточно написать любой интересующий вас тип данных (Int8-Int128, BigInt, UInt8-UInt128, Float16-Float64 или BigFloat).

Начнем с Int8 и Int16, разрядность которых меньше базовой. Сравнивая вывод с листингом 2, с удивлением обнаружим, что все отличия сосредоточены в строке с номером 5. Напомним, что там извлекается значение аргумента. Сравним команды для Int32 и Int8:

```
mov eax, dword ptr [ebp + 8]
```

и

```
movsx eax, byte ptr [ebp + 8]
```

Ожидаемо различается размер числа (dword и byte). Но еще вместо **mov** используется команда **movsx**. Посмотрев документацию [32] и литературу (например, [5, 15, 18]), понимаем, что в последнем случае операнд со знаком меньшего размера (у нас байт) копируется в регистр большего (двойное слово). При этом старшие 24 бита формально не заданы. Поскольку типы Int могут содержать отрицательные значения, то эти неопределенные биты команда **movsx** устанавливает по следующему правилу: они обнуляются для положительных значений операнда и устанавливаются в единицу для отрицательных. По сути, процессор копирует во все старшие биты знаковый разряд исходного числа. Данный процесс принято называть *расширением знака*.

В результате вместо 8- или 16-битного аргумента получаем 32-разрядный. Все дальнейшие вычисления и результат тоже будут 32-разрядными.

Обратим здесь внимание на еще одно важное обстоятельство. Как мы видим, компилятор по умолчанию не принимает специальных мер для анализа *переполнения* (т. е. получения слишком большого результата, выходящего за пределы отведенного количества битов). Скажем, для нашей функции при Int8, когда $x = 10$, должно получаться $y = 321$, что явно выходит за границы байта. Julia здесь предотвращает ошибку путем возвращения 32-битного результата. (Напомним, что в экспериментах использовалась 32-разрядная ОС.) С практической точки зрения, это, возможно, и выход, но с академической — выглядит несколько неестественно.

Если мы теперь протестируем беззнаковые целочисленные типы UInt8 и UInt16, то увидим в обсуждаемой команде вместо **movsx** мнемонику **movzx**: все значения положительны, а значит, старшие биты всегда устанавливаются нулевыми.

Подчеркнем, что арифметика для чисел со знаком и без знака не отличается.

Завершая обсуждение ситуации с целыми числами, проверим бóльшую разрядность Int64. Здесь (в случае 32-битной ОС) мы обнаружим, что при считывании числа его младшая часть помещается в регистр **ecx**, а старшая — в **edx**. Для совместного сдвига обеих «половинок» используется пара инструкций **shld edx, ecx, 5** и **shl ecx, 5**. Детали для краткости здесь обсуждать не будем.

Если еще нарастить разрядность (Int128 или BigInt), то компилятор ради экономии кода начнет использовать служебные подпрограммы. Их вызывают командой **call**. Анализ такого кода усложняется, так что для первого знакомства он не очень подходит.

Итак, для целочисленных типов компилятор Julia продемонстрировал очень даже оптимальные результаты. Посмотрим теперь, как обстоят дела для вещественных чисел Float32 (листинг 3; фрагменты F1 и F3 сокращены).

Во фрагменте F0 мы замечаем вещественные константы 32, 3 и –2 (строки 1–6). Настроение сразу ухудшается, поскольку вместо двух последних значений хотелось бы видеть их сумму.

Кодовый фрагмент F2 подтверждает пессимистические ожидания. Заглянув в раздел 3.1.3, расшифруем команды 7–12. В строке 7 аргумент загружается в регистр **xmm0** математического сопроцессора. Далее в строках 8–10 выполняются арифметические действия в строгом соответствии с исходной формулой: значение x умножается на 32, а затем тупо выполняются два сложения (вычитание заменено сложением с отрицательной константой –2).

F0 — заголовок		`.text`
		`.file "f"`
		`.section .rodata.cst4,"aM",@progbits,4`
		`.p2align 2 # -- Begin function julia_f_117`

	1	`.LCPI0_0:`
	2	`.long 0x42000000 # float 32`
	3	`.LCPI0_1:`
	4	`.long 0x40400000 # float 3`
	5	`.LCPI0_2:`

	6	`.long 0xc0000000 # float -2`
		`.text`
		`.globl julia_f_117`
		`.p2align 4, 0x90`
		`.type julia_f_117,@function`

F1 — вход		`julia_f_117: # @julia_f_117` …
F2 — **код программы**	7	`movss xmm0, dword ptr [ebp + 8] #xmm0 = mem[0],zero,zero,zero`
		`; ┌ @ promotion.jl:411 within `*` @ float.jl:410`
	8	`mulss xmm0, dword ptr [.LCPI0_0]`
		`; └`
		`; ┌ @ promotion.jl:410 within `+` @ float.jl:408`
	9	`addss xmm0, dword ptr [.LCPI0_1]`
		`; └`
		`; ┌ @ promotion.jl:412 within `-` @ float.jl:409`
	10	`addss xmm0, dword ptr [.LCPI0_2]`
		`; └`
	11	`movss dword ptr [esp + 4], xmm0`
	12	`fld dword ptr [esp + 4]`

F3 — выход		…
		`# -- End function`
		`.section ".note.GNU-stack","",@progbits`

Листинг 3. Этюд 1, аргумент — Float32

Таким образом, в случае вещественного аргумента оптимальность кода, увы, теряется. Во всяком случае, два соседних сложения с константами так и просятся на объединение!

4. Этюд 2. Реализация ветвлений

Подавляющее большинство программ не просто производят последовательные вычисления, но в зависимости от получаемых результатов могут выбирать, какие действия выполнять, а какие нет. Вы, конечно, догадались, что сейчас речь пойдет о ветвлениях и о том, как они реализуются в Julia.

Нам снова потребуются некоторые дополнительные сведения, которые мы предварительно обсудим.

4.1. Этюд 2: предварительные сведения

4.1.1. Анализ условий в процессоре

При построении разветвляющихся и циклических программ используются так называемые *условные* команды, которые в зависимости от результата проверки указанного в них условия либо выполняются, либо нет. Чаще всего это условные переходы, которые всегда работают по следующему алгоритму: если анализируемое условие истинно, то переход происходит; в противном случае он игнорируется и выбирается следующая команда.

Чтобы определить, справедливо ли требуемое условие, любая условная команда обращается к специальному регистру процессора, который называется *регистром флагов*. *Флаги* — это биты, которые устанавливаются в зависимости от результата выполнения инструкции. Например, один из битов, отвечающий за анализ знака числа, устанавливается в 1, если результат отрицателен, и сбрасывается в 0 в противном случае (проще говоря, в него копируется знаковый бит числа).

Установка флагов в процессоре Intel — это тонкий вопрос. Некоторые инструкции на флаги не воздействуют совсем (например, постоянно используемая команда **mov**), а некоторые изменяют только часть флагов; наконец, есть арифметические и некоторые другие операции (в том числе специализированная команда сравнения двух чисел), которые влияют практически на все флаги. Чтобы не ошибиться, полезно проверять сведения о воздействии на флаги по справочнику.

Таким образом, вырисовывается следующая схема реализации ветвления в программе. В ходе вычислений инструкции влияют на состояние регистра флагов. В том месте программы, где нужно произвести ветвление, ставится условная команда, которая анализирует текущее состояние. Рассмотрим пример.

Пример 5.

```
cmp ax, 2
jne label
```

Инструкция **cmp** сравнивает два своих операнда путем вычитания. Полученная разность не сохраняется, но будет проанализирована с целью установки флагов. В нашем примере надо проследить за тем из них, который фиксирует факт равенства/неравенства чисел (а более точно, равенства их разности нулю). Следующая далее инструкция условного перехода по неравенству **jne** (**j**ump if **n**ot **e**qual) анализирует указанный флаг и, если его состояние соответствует неравенству, передает управление на участок программы, помеченный меткой **label**; в противном случае никаких действий не производится и выполнение программы продолжается.

Примечание. Обсуждение логики анализа условий возможно и без упоминания флагов (сравниваем содержимое регистра **ax** с константой 2 и в случае неравенства результата переходим...). Тем не менее понимание механизма работы флагов иметь полезно. Достаточно, например, сказать, что у процессора есть некоторые специальные команды сброса или установки флагов.

По аналогичной логике работают и другие условные команды. Проанализируем такой фрагмент программы:

Пример 6.

```
xor ecx, ecx
cmp eax, 2
sete cl
```

Первая инструкция — исключающее ИЛИ (**xor**) — имеет неочевидное, на первый взгляд, назначение. Как известно, эта логическая операция, по своей сути, позволяет сравнивать двоичные коды: если соответствующие биты операндов совпадают, то результирующий бит сбрасывается в 0, а в противном случае устанавливается в 1. Поскольку двоичное содержимое регистра не может не совпасть «с самим собой» (оба операнда в рассматриваемой команде одинаковы), то результат *всегда* будет нулем и, таким образом, в регистр будет занесен ноль.

Примечание. Можно применить для обнуления регистра инструкцию **mov ecx, 0**. Но, поскольку нулевая константа будет включена в код команды, полученная инструкция будет занимать в памяти больше места и дольше выполняться. Именно поэтому профессиональные программисты (а тем более оптимизирующие компиляторы!) предпочитают вариант с **xor**.

Итак, в начале рассматриваемого фрагмента регистр **ecx** будет обнулен. Далее следует обычная инструкция сравнения значения **eax** с константой 2, которая устанавливает флаги для последующего анализа. Наконец, инструкция в конце фрагмента проверяет флаг, ответственный за равенство (последняя буква в мнемонике операции «e», что означает if **e**qual). Если состояние флага оказалось подходящим, то инструкция выполняется, иначе — игнорируется. А выполнение инструкции состоит в установке (**set**) младшего бита регистра **cl** (причем **cl** — это младшая часть регистра **ecx**). Если же команда «не сработала», то там сохранится прежнее нулевое значение.

В итоге в результате работы рассмотренного фрагмента при **eax** = 2 получается **ecx** = 1; во всех остальных случаях устанавливается **ecx** = 0.

4.1.2. Вызов подпрограмм и передача им параметров

Для вызова подпрограммы используется команда **call**. В отличие от обычного перехода, по команде **call** аппаратным образом сохраняется текущий адрес для той точки основной программы, в которую необходимо вернуться после завершения подпрограммы. Адрес возврата заносится в специально организованную память, которая называется *стеком*. Доступ в стек, как мы уже видели в листинге 2, регулируется регистром **esp** — его еще называют *указателем стека*.

Команда **ret** обеспечивает возврат из подпрограммы по адресу, занесенному в стек командой **call**.

Перед вызовом подпрограммы в стек часто записывают значения входных параметров. Для выходных параметров стек также можно использовать, но компилятор Julia, если удается, применяет более простой способ — результат возвращается через определенный регистр.

При занесении данных в стек по инструкции **push** значение **esp** автоматически уменьшается, а при извлечении (команда **pop**) — растет. Допускается (аккуратное и продуманное) изменение указателя стека **esp**: этим приемом компилятор Julia часто пользуется (см., например, строку 4 в листинге 2).

Более подробно о работе стека (в том числе и при обслуживании подпрограмм) можно почитать в публикации автора [7].

4.1.3. Некоторые необходимые для анализа команды

В программах, которые будут рассмотрены в данном этюде, появится еще несколько новых (для нашего обсуждения) инструкций.

- `jmp label`
  Кроме описанных в разделе 4.1.1 условных переходов существует еще безусловный, который всегда (без всякого анализа флагов) передает управление на указанную метку. Интересно, что в языках высокого уровня безусловный переход принято считать вредным. Так, классик теории программирования Э. В. Дейкстра писал: «Я убежден в том, что оператор go to должен быть отменен в языках программирования “высокого уровня” (т. е. отовсюду, кроме, возможно, простого машинного кода)» [27]. Зато написать полную развилку на машинном языке без **jmp** не удается!
- `inc eax`
  Эта инструкция прибавляет к операнду 1 (инкрементирует его). По сравнению с обычным сложением **add eax, 1**, инструкция **inc** сохраняет значения некоторых флагов. В частности, флага переноса, что упрощает организацию арифметики для «длинных» чисел.
  Для уменьшения значения на 1 предусмотрена инструкция **dec**.
- `cwde`
  Данная экзотическая инструкция позволяет преобразовать 16-битное (w) число со знаком, находящееся в регистре **ax**, в 32-битное (d). Результат помещается в регистре **eax**, младшие байты которого есть не что иное, как **ax**. По сути дела, число в **eax** расширяет свою разрядность

с учетом знака. Очень похожа на нее описанная в этюде 1 инструкция **movsx**, с той лишь разницей, что в **cwde** нельзя изменить операнды.

- `lea eax, [ecx + eax + 1]`
  Инструкция с мнемоникой **lea** (**l**oad **e**ffective **a**ddress) предназначена для определения адреса данных в ОЗУ. Вычисления ведутся по формуле:

  адрес = база + M ∗ индекс + C,

  где база и индекс — это значения в указанных регистрах, C — константа (возможно, отрицательная), а M — множитель, описывающий размер данных (1, 2, 4 или 8 байт). Попутно заметим, что такой способ адресации в процессоре Intel принято называть *масштабируемым*. С помощью описываемой инструкции можно вычислить адрес переменной; особенно это ценно при работе с массивами или строками.
  Можно смотреть на **lea** как на «незаконченную» команду **mov**: когда адрес сосчитан, она выдает **его** в качестве результата вместо того, чтобы прочитать данные из ОЗУ.
  Компиляторы (в том числе и языка Julia) часто используют **lea** как комплексную вычислительную операцию, которая к тому же не портит флаги. Указанная выше команда прибавит к содержимому **eax** значение из регистра **ecx** и к полученной сумме еще добавит 1. И все это одной командой.

4.2. Этюд 2: постановка задачи

Представим себе, что на пустую шахматную доску в горизонтальный ряд с номером i ставится белая пешка. Для простоты будем считать, что $1 < i < 9$, т. е. имеет корректное значение. Программа должна вывести все возможные ходы пешки.

Для тех, кто не силен в шахматных правилах, напомним, что белая пешка двигается по доске на одну клетку вверх, т. е. номер горизонтали увеличивается на единицу. Единственным исключением служит случай начального положения пешки ($i = 2$), когда ее можно двинуть не только на одну, но и при желании на две клетки вверх.

Таким образом, мы имеем дело с простой задачей на ветвление.

4.3. Этюд 2: решение задачи

Обратимся к листингу 4, где приведено решение нашего несложного этюда на языке Julia.

В функции **pawn1()** использована естественная логика: если $i = 2$, то возможны два хода, во всех остальных случаях (**else**) — один. Алгоритмически не слишком красиво то, что один ход будет всегда, независимо от выполнения условия. А это значит, что лучше было бы написать по-другому: если $i = 2$, то возможен дополнительный ход, а далее во всех случаях (вне оператора **if**) один ход реализуется стандартным (безусловным!) образом. Видно, что это хорошо хотя бы потому, что отпадает необходимость в ветви **else**. Но для наших экспериментов интереснее именно приведенная выше версия.

Теперь посмотрим, как устроена функция **pawn1()** для типа Int16. В листинге 5 приводится только собственно программный код, а все вспомогательные фрагменты (они практически не отличаются от показанных ранее в листинге 2) опущены.

Перед нами — классическая реализация развилки.

В строке 1 из стека в 32-битный регистр **eax** считывается 16-битное число. Заметим, что в следующей команде использоваться будет только младшая часть регистра, которую в ассемблере принято маркировать **ax**. В строке 2 полученное значение сравнивается с 2, и если нет совпадения (т. е. $i \neq 2$), то в строке 3 следует переход на метку **LBB0_2** (строка 9) к ветви **else**. Подчеркнем, что в программе было написано $i == 2$, а переход используется противоположный — jne (if **n**ot **e**qual). В случае равенства переход «не срабатывает» и программа выполняется дальше.

В строке 4 в стек заносится аргумент для вывода на экран (см. раздел 4.1.2), равный константе 3. Вспомним, что для этой ветви равенство выполнилось (т. е. значение i равнялось 2). Мы видим, что «мудрый» компилятор, стараясь уменьшить число арифметических действий в программе, заранее сосчитал 2 + 1 и подставил в программу ответ 3. Далее в строке 5 вызывается служебная подпрограмма Julia для вывода целого числа на экран.

Строки 6 и 7 устроены аналогично, только выводимая константа теперь равна 4 (компилятор опять предусмотрительно вычислил следующее значение).

команда 1
(ввод функции)
 	
ответ (готово)
	julia> function pawn1(i::Integer)
	  if i==2
	  i=i+1; println(i)
	  i=i+1; println(i)
	  else
	  i=i+1; println(i)
	  end
	  end
	pawn1 (generic function with 1 method)

команда 2 (тест 1) ответы команда 3 (тест 2) ответ	julia> pawn1(2) 3 4 julia> pawn1(4) 5

*Листинг 4. Описание **полного** варианта функции и ее вызов*

F2 — **код программы**

```
1           movzx   eax, word ptr [ebp + 8]
            ; | @ REPL[2]:2 within `pawn1`
            ; |┌ @ promotion.jl:449 within `==` @ promotion.jl:499
2           cmp     ax, 2
            ; |└
3           jne     .LBB0_2
            # %bb.1:                                # %L4
            ; | @ REPL[2]:3 within `pawn1`
4           mov     dword ptr [esp], 3
5           call    j_println_108@PLT
            ; | @ REPL[2]:4 within `pawn1`
6           mov     dword ptr [esp], 4
7           call    j_println_108@PLT
8           jmp     .LBB0_3
9           .LBB0_2:                                # %L10
            ; | @ REPL[2]:6 within `pawn1`
            ; |┌ @ int.jl:1040 within `+`
            ; ||┌ @ int.jl:523 within `rem`
10          cwde
            ; ||└
            ; || @ int.jl:1042 within `+` @ int.jl:87
11          inc     eax
            ; |└
12          mov     dword ptr [esp], eax
13          call    j_println_108@PLT
14          .LBB0_3:                                # %common.ret
```

*Листинг 5. Этюд 2, **полный** вариант, аргумент — Int16*

После этого в строке 8 следует безусловный переход на метку **LBB0_3** (строка 14), что обеспечивает обход альтернативной ветви **else**. Подчеркнем, что именно здесь то самое место, где безусловный переход незаменим.

Переходим к анализу кода ветви **else**. В строке 10 аргумент *i* из 16-битного регистра **ax** преобразуется в 32-битный и результат заносится в **eax** (см. раздел 4.1.3). В этом месте, как нам кажется, компилятор недоработал: если бы в строке 1 вместо **movzx** стояло **movsx** (команды обсуждались ранее в этюде 1), то строка 10 не потребовалась бы!

В строке 11 текущее значение **eax** (т. е. *i*) увеличивается на 1. Результат в строке 12 заносится в стек в качестве аргумента, и в следующей строке следует вызов служебной подпрограммы вывода.

Строка 14 служит выходом из развилки: в нее мы попадаем как при выполнении, так и при невыполнении условия $i = 2$ (вспомним про переход в строке 3).

А теперь, с точки зрения изучения оптимальности кода, интересно проделать еще один эксперимент. Удалим из функции **pawn1()** все операторы вывода **println(i)**. Чтобы как-то увидеть результат, добавим в конец функции (перед последним **end**) оператор **return i**, который будет возвращать значение *i* в качестве ответа. Автор надеется, что читатели без труда самостоятельно внесут перечисленные изменения в листинг 4.

Машинный код для сокращенной программы получится совсем коротким и очень необычным (листинг 6). Бросается в глаза полное отсутствие переходов. Как же это работает?

Строка 1 считывает из стека в 32-битный регистр **eax** 16-битный аргумент, расширяя при этом его знак инструкцией **movsx**. Фрагмент в строках 2–4 вырабатывает в **ecx** значение либо 1, когда **eax** (т. е. *i*) = 2, либо 0 в противном случае; работа кода подробно описана в разделе 4.1.1. Именно здесь и сосредоточена развилка, которая реализована за счет условной инструкции **sete**.

Наконец, в строке 5 результат вычисляется по формуле

$$eax = eax + 1 + ecx,$$

причем последнее слагаемое равно 1 при $i = 2$ и 0 в остальных случаях. Иными словами, результирующее значение переменной *i* увеличивается либо на 2, либо на 1.

Заметим, что отсутствие какого-либо дополнительного кода, связанного с реализацией оператора **return i**, косвенно свидетельствует о том, что результат вычислений будет взят непосредственно из регистра **eax**.

F2 — **код программы**

```
1           movsx   eax, word ptr [ebp + 8]
2           xor     ecx, ecx
3           cmp     eax, 2
4           sete    cl
            ; | @ REPL[1]:2 within `w_pawn`
5           lea     eax, [ecx + eax + 1]
```

*Листинг 6. Этюд 2, **сокращенный** вариант, аргумент — Int16*

5. Этюд 3. Реализация циклов

Чтобы наши упражнения-этюды охватили все базовые алгоритмические конструкции, осталось посмотреть, как реализуются циклы. Тут нас ждет приятный сюрприз: оказывается, организация кода для циклов во многом похожа на то, как это делалось для развилки. Если вдуматься, то логика построения неполной (без **else**) развилки действительно имеет много общего со структурой цикла с предусловием **while**. В связи с этим автор во многом надеется на самостоятельность читателей при разборе данного этюда.

5.1. Этюд 3: постановка задачи

Изучим реализацию программы, которая выводит на экран «обратный отсчет»: последовательность уменьшающихся положительных чисел, которая начинается с заданного значения *k* и заканчивается 1.

5.2. Этюд 3: решение задачи

Решение нашего несложного этюда на языке Julia приведено в листинге 7.

Фрагмент F2 эквивалентного машинного кода, составленного компилятором, размещен в листинге 8. Кратко прокомментируем его.

Строка 1 считывает 32-битное (dword) значение аргумента в регистр **esi**. В строке 2 полученное значение тестируется (инструкция **test** эквивалентна **and** без записи результата; ее цель — установить флаги в соответствии с текущим значением операнда). По результатам анализа в случае отрицательного или нулевого аргумента по команде **jle** (**j**ump if **l**ess or **e**qual — переход по меньше или равно) происходит обход цикла, так как предусловие сразу оказывается не выполнено, и цикл игнорируется.

Строки 5–6 узнаваемо выводят текущее значение переменной *k* из **esi**, а в строке 7 переменная уменьшается на 1.

А далее ради оптимизации кода компилятор «совершает подлог»: вместо того чтобы сделать безусловный переход на проверку *предусловия* (строки 2–3), в строке 8 он вставляет *постусловие*. Оптимизация здесь заключается в том, что нет необходимости специально проверять на равенство нулю текущее значение переменной, поскольку инструкция **dec** это и так уже сделала! В итоге, если счетчик не дошел до 0, цикл повторяется, в противном случае перехода нет и цикл завершается.

Примечание. Попутно заметим, что цикл с постусловием всегда имеет более короткий код, чем цикл с предусловием. Печально, что в Julia такой цикл, похоже, вообще не предусмотрен.

команда 1
(ввод функции)
 	
ответ (готово)
	`julia> function countdown(k)`
	`  while k>0`
	`  println(k)`
	`  k=k-1`
	`  end`
	`  end`
	`countdown (generic function with 1 method)`

команда 2 (тест)
результаты
	`julia> countdown(3)`
	`3`
	`2`
	`1`

Листинг 7. Описание и вызов функции

F2 — **код программы**		
	1	`  mov  esi, dword ptr [ebp + 8]`
		`; │ @ REPL[19]:2 within `countdown``
		`; │┌ @ operators.jl:369 within `>``
		`; ││┌ @ int.jl:83 within `<``
	2	`  test  esi, esi`
		`; │└└`
	3	`  jle  .LBB0_2`
		`  .p2align  4, 0x90`
	4	`.LBB0_1:  # %L5`
		`  # =>This Inner Loop Header: Depth=1`
		`; │ @ REPL[19]:3 within `countdown``
	5	`  mov  dword ptr [esp], esi`
	6	`  call  j_println_172@PLT`
		`; │ @ REPL[19]:2 within `countdown``
		`; │┌ @ operators.jl:369 within `>``
		`; ││┌ @ int.jl:83 within `<``
	7	`  dec  esi`
		`; │└└`
	8	`  jne  .LBB0_1`
	9	`.LBB0_2:  # %L8`

Листинг 8. Этюд 3, аргумент — Int32

Те, кто хочет заглянуть поглубже, могут в листинге 7 удалить **println** (подобно тому как мы делали в этюде 2) и, добавив еще строку **return k**, посмотреть генерируемый код. Вероятно, вы удивитесь, но цикла для целого аргумента в этом случае совсем не будет! Будет только ветвление: если аргумент $k \le 0$, то он и является ответом (моделируется игнорирование цикла), либо сразу 0 в противном случае (зачем «прокручивать» цикл, если его результат заранее известен?!).

Любопытно, что для вещественных аргументов такой оптимизации не будет и цикл сохранится.

6. Заключение

Надеюсь, предложенные вашему вниманию эксперименты вызвали интерес. Какие из них следует сделать выводы?

- Действительно, с помощью Julia легко «увидеть программу изнутри», т. е. анализировать эквивалентный ей исполняемый машинный код (на наш взгляд, анализ простейших случаев доступен даже школьнику). Возможно, потребуются материалы по системе команд процессора, но для семейства Intel (невзирая даже на санкционные ограничения, принятые сейчас на официальном сайте компании) это не проблема.
- Проведенные эксперименты подтверждают, что генерируемый компилятором Julia код вполне эффективен.
- Оптимизация для целочисленных данных заметно лучше, чем для вещественных. Для целых чисел компилятор, как мы увидели, очень успешно старается обойтись без трудоемких арифметических операций (если удается, то значения вычисляются заранее, умножение заменяется сдвигом и т. п.). Компилятор даже способен предсказать результаты простейшего цикла и исключить сам цикл из программы.
- Для вещественных чисел компилятор активно использует инструкции новой технологии SSE. Поэтому интересно в будущих экспериментах посмотреть на организацию параллельных вычислений для векторных операций.
- Ради краткости кода компилятор по умолчанию не принимает мер по фиксации переполнения. Перед нами еще один аргумент в пользу важности понимания того, как компьютер считает и когда у него могут возникать вычислительные трудности.
- Проведенное рассмотрение кода учит, что тщательный анализ алгоритма и знания в области теории программирования могут помочь существенно ускорить работу программы. На наш взгляд, в подготовке современных специалистов значение подобного опыта трудно переоценить.
- К языку Julia стоит присмотреться повнимательнее!

Список источников

1. *Антонюк В. А.* Язык Julia как инструмент исследователя. М.: Физ. фак. МГУ имени М. В. Ломоносова, 2019. 48 с. https://cmp.phys.msu.ru/sites/default/files/VA_Antonyk_Julia_2019.pdf

2. *Белов Г. В.* Краткое описание языка программирования Julia и примеры его использования. М.: МГТУ им. Н. Э. Баумана, 2023. 105 с. http://ihed.ras.ru/~thermo/Julia/Brief description of Julia language.pdf

3. *Белов Г. В., Аристова Н. М.* О возможностях использования языка программирования Julia для решения научных и технических задач // Вестник МГТУ им. Н. Э. Баумана. Серия «Приборостроение». 2020. № 2 (131). С. 27–43. EDN: XODPOB. DOI: 10.18698/0236-3933-2020-2-27-43

4. *Белов Г. В., Аристова Н. М., Мальцев М. А.* Использование языка программирования Julia для расчета равновесного состава многокомпонентной газофазной системы // Вестник Объединенного института высоких температур. 2022. Т. 7. № 1. С. 52–55. EDN: OIBSEI.

5. *Гук М. Ю.* Процессоры Intel: от 8086 до Pentium II. СПб.: Питер, 1997. 224 с.

6. *Еремин Е. А.* Компилятор? Это довольно просто! Пермь: Изд-во ПРИПИТ, 1998. 120 с.

7. *Еремин Е. А.* Стек // Информатика. 2008. № 2. С. 37–38; 2008. № 3. С. 42–44; 2008. № 4. С. 41–44; 2008. № 6. С. 43–45; 2008. № 8. С. 43–45; 2008. № 9. С. 39–42. http://emc.orgfree.com/theory/raznoe/d4.html

8. *Ефремова Т. Ф.* Самый полный толковый словарь русского языка. В 3 т. М.: АСТ, 2015. 3312 с.

9. *Иванова Е. М.* Программная модель процессора и базовые инструкции процессоров Intel и MIPS. М.: НИУ ВШЭ, 2019. 139 с. https://publications.hse.ru/pubs/share/direct/317732672.pdf

10. ИТ-шники всего мира назвали самый любимый и самый ненавистный язык программирования // CNews, 4 августа 2021. https://www.cnews.ru/news/top/2021-08-03_programmisty_vsego_mira

11. *Кулябов Д. С., Королькова А. В.* Компьютерная алгебра на Julia // Программирование. 2021. № 2. С. 44–50. EDN: HETTIQ.

12. *Лопес Б. К., Аулер Р.* LLVM: инфраструктура для разработки компиляторов. М.: ДМК Пресс, 2015. 342 с.

13. *Ожегов С. И., Шведова Н. Ю.* Толковый словарь русского языка. М.: Азбуковник, 1997. 944 с.

14. *Оконешникова Е. А.* Новый язык программирования в учебном процессе и научных исследованиях // Наука. Информатизация. Технологии. Образование. Материалы XIII международной научно-практической конференции (Екатеринбург, 24–28 февраля 2020 г.). Екатеринбург: Российский государственный профессионально-педагогический университет, 2020. С. 123–129. EDN: WVFCHN.

15. *Смит Б. Э., Джонсон М. Т.* Архитектура и программирование микропроцессора INTEL 80386. М.: Конкорд, 1992. 334 с.

16. *Таненбаум Э. С.* Архитектура компьютера. СПб.: Питер, 2003. 704 с.

17. *Устинин Д. М.* Использование языка научных вычислений Julia для моделирования внутриклеточных процессов методом броуновской динамики // Доклады Международной конференции «Математическая биология и биоинформатика». 2018. № 7. С. e29.1–e29.4. EDN: YPPYWL. DOI: 10.17537/icmbb18.93

18. *Шагурин И. И., Бердышев Е. М.* Процессоры семейства Intel P6. Архитектура, программирование, интерфейс. М.: Горячая линия — Телеком, 2000. 248 с.

19. *Шадрин Ю. А., Грюмова Е. А., Гумирова А. И.* Обзор языка программирования Julia // Технологии инженерных и информационных систем. 2019. № 3. С. 43–52. EDN: IIUXHP.

20. *Шеррингтон М.* Осваиваем язык Julia. Совершенствование мастерства в области аналитики и программирования. М: ДМК-Пресс, 2017. 416 с.

21. *Шиндин А. В.* Язык программирования математических вычислений Julia. Базовое руководство. Нижний Новгород: ННГУ, 2016. 24 с. http://www.unn.ru/books/met_files/JULIA_tutorial.pdf

22. *Энгхейм Э.* Julia в качестве второго языка. М.: ДМК Пресс, 2023. 446 с.

23. *Ben-Ari M.* Constructivism in computer science education // Journal of Computers in Mathematics and Science Teaching. 2001. Vol. 20. No. 1. P. 45–73.

24. *Bezanson J., Chen J., Chung B., Karpinski S., Shah V. B., Vitek J., Zoubritzky L.* Julia: Dynamism and performance reconciled by design // Proceedings of the ACM on Programming Languages. 2018. Vol. 2 (OOPSLA). P. 120.1–120.23. DOI: 10.1145/3276490

25. *Bezanson J., Edelman A., Karpinski S., Shah V. B.* Julia: A fresh approach to numerical computing // SIAM Review. 2017. Vol. 59. No. 1. P. 65–98. DOI: 10.1137/141000671

26. *Bezanson J., Karpinski S., Shah V. B., Edelman A.* Julia: A fast dynamic language for technical computing // arXiv preprint. arXiv:1209.5145. 2012. DOI: 10.48550/arXiv.1209.5145

27. *Dijkstra E. W.* Go to statement considered harmful // Communications of the ACM. 1968. Vol. 11. No. 3. P. 147–148. DOI: 10.1145/362929.362947. (Русский перевод: http://hosting.vspu.ac.ru/~chul/dijkstra/goto/goto.htm)

28. *Du Boulay B.* Some difficulties of learning to program // Journal of Educational Computing Research. 1986. Vol. 2. No. 1. P. 57–73. DOI: 10.2190/3LFX-9RRF-67T8-UVK9

29. *Eremin E. A.* Educational Pascal compiler into MMIX code // Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling. 2007. P. 143–144. EDN: MWQNJR.

30. *Gao K., Mei G., Piccialli F., Cuomo S., Tu J., Huo Z.* Julia language in machine learning: Algorithms, applications, and open issues // Computer Science Review. 2020. Vol. 37. P. 100254.1–100254.13. DOI: 10.1016/j.cosrev.2020.100254

31. *Gevorkyan M. N., Korolkova A. V., Kulyabov D. S.* Julia language features for processing statistical data // Discrete and Continuous Models and Applied Computational Science. 2023. Vol. 31. No. 1. P. 5–26. EDN: VNJCSU. DOI:10.22363/2658-4670-2023-31-1-5-26

32. Intel 64 and IA-32 Architectures Software Developer's Manual. 2023. 5066 p. http://asmcourse.cs.msu.ru/wp-content/uploads/2022/01/325383-sdm-vol-2abcd.pdf

33. *Lattner C.* LLVM // The Architecture of Open Source Applications. 2011. Vol. 1. P. 155–170. (Русский перевод: https://rus-linux.net/MyLDP/BOOKS/Architecture-Open-Source-Applications/Vol-1/llvm-1.html)

34. *Lattner C., Adve V.* LLVM: a compilation framework for lifelong program analysis & transformation // Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004). San Jose, CA, USA, 2004. P. 75–86. DOI: 10.1109/CGO.2004.1281665

35. *Miller E.* Why I'm betting on Julia. https://www.evanmiller.org/why-im-betting-on-julia.html. (Русский перевод: https://habr.com/ru/articles/210298/)

36. *Nazarathy Y., Klok H.* Statistics with Julia: Fundamentals for data science, machine learning and artificial intelligence. Springer Series in the Data Sciences. Springer Nature, 2021. 527 p. DOI: 10.1007/978-3-030-70901-3

37. *Salceanu A.* Julia Programming Projects: Learn Julia 1.x by building apps for data analysis, visualization, machine learning, and the web. Birmingham: Packt Publishing Ltd, 2018. 500 p.

38. *Sengupta A., Edelman A.* Julia High Performance: Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond. 2nd Edition. Birmingham: Packt Publishing Ltd, 2019. 218 p.

39. The Julia Language. V.1.9.2. The Julia Project. July 6, 2023. 1644 p. https://docs.julialang.org/en/v1/