Сказ о том, чем компилятор от интерпретатора отличается :)

(в вольном изложении братца)
((слегка почитанный и подправленный СергейАнатольичем))

май - август 2014

Што это такое

Программирование сильно смахивает на героическую борьбу человека с компьютером. Человек пишет программу. Он пишет её на странном, нелогичном языке, с трудом выражая свои мысли. Язык больше мешает, чем помогает: нужной функции нет, зато есть куча каких-то дурацких никому не нужных штук.

Вот человек написал программу. Посидел, порассматривал своё кривоватое детище. С виду — странная смесь английских слов и бессмысленных крикозябликов. Но цепкие глаза программиста смотрят сквозь буквы, въедаются в неуловимую суть. Наконец человек устаёт пыриться в экран и отдаёт программу на суд компьютера. Начинается долгое, утомительное противостояние: человек то букву добавит, то две уберёт, то вообще начинает удалять текст большими кусками.

И вот наступил великий момент: программа запустилась. На чёрном фоне высветились белые буквы “Segmentation fault”.

Что я хочу сказать этой болтовнёй, так это то, что языки программирования малость не идеальны. :D Я хотела рассказать тебе, что я о них знаю, что меня вдохновляет и позволяет худо-бедно продираться через сотни костылей и заведомую плохость результата.

План у меня примерно такой. Я начну с мутных рассуждений о том, что такое язык программирования. Потом поговорю о более конкретных вещах: что такое машинный язык и как от него перейти к языкам высокого уровня. Наконец, верхом конкретности будет пример построения нескольких языковых процессоров для очень простого языка.

Я много где могу привирать или говорить не всю правду, и наверняка даже не про все такие случаи догадываюсь. Зато я пыталась взять шириной охвата. :D Скажем так, я пыталась разбросать щупальца во все стороны, но при этом в паре мест въедливо ковыряюсь в мелочах (чтобы показать, что магии нет, и в любом программистском вопросе можно докопаться до сути).

Поковыряй пример, пойми, как он работает. :)

Есть много книжек, которые стоит прочитать. Не для того, чтобы стать очень умной — это мелкий побочный эффект. Просто программирование без идеи — на редкость нудное дело, а идея как фонарь — то потухнет, то погаснет. Книжки, то есть мысли других людей, не дадут твоей идее погаснуть. Я их люблю за это, а умность — хоть бы её и совсем не было.

Из совсем уж классных книжек: Petzold, “Code” (про устройство компьютера); Tanenbaum (он много про что написал: архитектура компьютеров, компьютерные сети); SICP (Structure and Interpretation of Computer Program, я лично не читала, но собираюсь, хорошие люди советуют); Jacobs, Grune: “Parsing Techniques. A Practical Guide”, 2nd edition (моя лично горячо любимая книжка про синтаксический анализ); Kline: “Mathematics: the Loss of Certainty” (про историю и смысл математики). И другие...

Оглавление

Глава 1

Малость о языках программирования

Язык программрования определяется двумя вещами: синтаксисом и семантикой. Синтаксис — это что на языке можно сказать. Семантика — это как следует понимать сказанное. Синтаксис подобен форме, семантика — сути.

По-хорошему, и синтаксис, и семантика должны быть *формально* определены. Формально определённый язык представляет из себя чистую абстракцию, он не привязан к конкретной реализации. Это позволяет любому желающему написать свою реализацию языка, что хорошо как для программистов (есть выбор), так и для разработчиков языка (необходимость удовлетворять формальному определению неслабо дисциплинирует, а здоровая конкуренция реализаций постоянно подстёгивает).

На практике встречаются разные языки. У некоторых вообще нет формального определения — только мутное описание и единственная полусгнившая реализация. Другие худо-бедно дают формальное определение синтаксиса, обычно вперемешку с замечаниями и комментариями. С семантикой дело обстоит хуже: хорошо, если она просто описана словами и обрывками псевдокода.

Попытка формально описать язык называется *стандартом* языка. Название стандарта часто отражает год его принятия (C99, C++11, Haskell98) или стандартизирующую организацию (ECMA-262 — стандарт JavaScript, ECMA-408 — стандарт Dart).

1.1 Синтаксис

Хотя синтаксис — всего лишь форма, он очень важен как для программистов (им в этой форме выражать свои мысли), так и для разработчиков языкового процессора (им писать синтаксический анализатор).

С точки зрения программиста в синтаксисе важны:

- Простота — она позволяет программисту сосредоточиться на решаемой задаче. Кроме того, простой синтаксис легко изучить, и язык будет привлекать, а не отпугивать новых программистов.

- Минималистичность — чем меньше кода требуется для решения задачи, тем легче охватить его одним глазом и понять.
- Ортогональность — хорошо, когда семантически разные конструкции описываются визуально непохожими закорючками.
- Однозначность — плохо, когда есть много семантически эквивалентных способов выразить мысль, это приводит к путанице и разнобою (тут, впрочем, многие любители разнообразия со мной не согласятся).
- Красота — работа программистов нематериальна, а результат далёк и отвязан от жизни, поэтому важно видеть в ней хоть какую-то простую человеческую ценность.

С точки зрения разработчика языкового процессора важно:

- Синтаксическая простота — то есть возможность алгоритмически легко распознавать синтаксис на компьютере.
- Расширяемость — возможность легко добавлять в язык новое или изменять старое. В этом смысле разработчики синтаксиса должны предвидеть будущее, оставлять свободу для расширения.

Дальше мы будем говорить о синтаксисе с точки зрения разработчика языка, то есть обсуждать подходы к формальному определению и алгоритмам распознавания синтаксиса.

1.1.1 Формальные грамматики

Где-то в районе 2-й половины XX века (в связи с распространением компьютеров и рождением языков программирования) человечество придумало универсальный математический аппарат для описания синтаксиса языка: *формальные грамматики*.

Формальная грамматика — это четвёрка $G =< \Sigma_T, \Sigma_N, P, S >$, где:

1. Σ_T — множество терминалов (алфавит);
2. Σ_N — множество нетерминалов;
3. P — множество правил вывода вида $\alpha A \beta \to \gamma$, где $\alpha, \beta, \gamma \in (\Sigma_T \cup \Sigma_N)^*$ (Σ^* означает множество всех цепочек в алфавите Σ), а $A \in \Sigma_N$;
4. $S \in \Sigma_N$ — стартовый нетерминал.

Множество всех цепочек, выводимых из стартового нетерминала и состоящих только из терминалов, называется *языком*, порождаемым грамматикой. (Говоря о формальных грамматиках, язык отождествляют с его синтаксисом). Грамматика однозначно определяет язык, но один и тот же язык может быть порождён разными грамматиками.

Классификацию грамматик (и языков соответственно) обычно проводят по виду правил вывода. Самая известная классификация — классификация Ноама Хомского (1956 год):

1. Type 0 (unrestricted, неограниченные): правила вида $\alpha A \beta \to \gamma$

2. Type 1 (context-sensitive, контекстно-зависимые): правила вида $\alpha A\beta \to \alpha\gamma\beta$ (γ — не пустая строка)
3. Type 2 (context-free, контекстно-свободные): правила вида $A \to \gamma$
4. Type 3 (regular, регулярные): правила вида $A \to \alpha B$ или $A \to \alpha$

Каждый следующий класс, как ты могла заметить по виду правил, — сужение предыдущего, (то есть более простой, ограниченный). Обсудим эти четыре класса, начиная с самого простого и заканчивая самым сложным.

Регулярные языки и регулярные выражения

Регулярные языки — самые простые. Проще них может быть только язык, задаваемый простым перечислением всех слов этого языка. Типичный пример регулярной грамматики — грамматика для описания натуральных десятичных чисел (| — это метасимвол, обозначающий "или": он разделяет альтернативные правые части правила):

$$\Sigma_T = \{0,1,2,3,4,5,6,7,8,9\}$$

$$\Sigma_N = \{digit, nonzero\_digit, number\}$$

$$P = \begin{cases} digit & \to 0|1|2|3|4|5|6|7|8|9 \\ nonzero\_digit & \to 1|2|3|4|5|6|7|8|9 \\ number & \to number\ digit \mid nonzero\_digit \end{cases}$$

$$S = number$$

Регулярные выражения — это другой подход к описанию того же самого понятия. Регулярные выражения над алфавитом Σ определяются так:

- $\emptyset$ (пустое множество) — регулярное выражение;
- ϵ (пустая строка) — регулярное выражение;
- $\alpha \in \Sigma$ (символ из алфавита) — регулярное выражение;
- если e_1 и e_2 — регулярные выражения, то $e_1\ e_2$ (конкатенация) — регулярное выражение;
- если e_1 и e_2 — регулярные выражения, то $e_1|e_2$ (альтернатива) — регулярное выражение;
- если e — регулярное выражение, то e^* (итерация, или звезда Клини) — регулярное выражение;

С помощью регулярных выражений натуральные десятичные числа можно описать так:

$$number = (1|2|3|4|5|6|7|8|9)\ (0|1|2|3|4|5|6|7|8|9)*$$

Регулярные выражения и регулярные грамматики эквивалентны по выразительной силе: они описывают одни и те же языки (регулярные). Есть алгоритм, позволяющий для любой регулярной грамматики построить эквивалентное регулярное выражение, и наоборот.

Регулярные выражения — очень полезная в хозяйстве вещь. Представь, например, ситуацию: в большом текстовом файле нужно найти все строки, начинающиеся с определённого слова, или все слова, в которых N букв, причём половина букв — согласные, или ещё какое-нибудь дурацкое ограничение. В программах часто возникают подобные задачи, поэтому почти у каждого языка программирования есть библиотека регулярных выражений (а во многие языки они попросту встроены). Но ещё чаще регулярные выражения пригождаются в жизни: например, когда на файловой системе нужно быстренько найти все файлы с именем определённого вида, или все файлы, в которых есть строка определённого вида, или отфильтровать вывод слишком болтливой программы. В линуксе есть много удобных и жутко полезных консольных программ для этого (посмотри хотя бы на `grep`, `sed` и `F7` или `F4` в `mc` — зуб даю, не пожалеешь!).

Ввиду своей невероятной полезности регулярные выражения обросли целым рядом довесков и наворотов. Большинство из этих наворотов — синтаксический сахар, то есть просто более краткая и удобная запись базового синтаксиса. Вот некоторые из них: e^+ (одно или более повторений), $e?$ (ноль или одно повторение), $[a_i - a_j]$ (класс символов от a_i до a_j), $e\{n\}$ (ровно n повторений), $e\{n, m\}$ (от n до m повторений). Все эти конструкции выражаются через базовые (например, e^+ — это ee^*). Но есть и другие, принципиальные навороты, из-за которых регулярные выражения перестают быть регулярными (они начинают описывать куда более широкий класс языков). Самый главный такой наворот — обратные ссылки (backreferences). Это конструкция, позволяющая выражению ссылаться на часть самого себя: например, $(cat|dog)\backslash 1$ описывает язык $\{catcat, dogdog\}$. Подлинные регулярные выражения отличаются от таких "расширенных" не-регулярных выражений тем, что первые можно обработать эффективно (алгоритмическая сложность — $O(n)$, линейная), а вторые — нельзя (алгоритмическая сложность — $O(2^n)$, экспоненциальная). И те, и другие бывают полезны, но в разных задачах.

В последнее время возникла большая путаница, что считать регулярными выражениями. Одни подразумевают регулярность в математическом смысле (допускают только синтаксический сахар). Другие подразумевают наличие обратных ссылок и прочих не-регулярных наворотов. Эта путаница осложняется наличием множества мутных "стандартов" типа "Basic Regular Expressions", "Extended Regular Expressions", "Perl Regular Expressions", "Posix Regular Expressions" и т.д. Большинство людей путает эти понятия, и в каждом конкретном случае нужно выяснять, что имеется в виду.

Другое важное (и даже основное) применение регулярных языков — *лексический анализ* (так называется синтаксический анализ регулярных языков). Часто грамматику языка бывает удобно разбить на два уровня: лексическая грамматика и синтаксическая грамматика. Такое разделение возникает, когда текст (поток терминальных символов) распадается на чёткие отдельные группы символов — лексемы. В естественных языках это могут быть слова и пунктуация, в языках программирования — идентификаторы, литералы, операторы и прочее. Сам текст при этом удобно воспринимать не как поток символов, а как поток лексем. Структура лексем описывается лексической грамматикой языка, а взаимосвязи между лексемами — синтаксической грамматикой. Обычно лексическая грамматика простая (регулярная), а синтаксическая — как повезёт (хорошо, если контекстно-свободная).

Регуляные языки отлично подходят для описания простых вещей: чисел, имён, IP-шников, URL-ов, ассемблерных инструкций. Они прекрасно выражают последовательность, альтернативу и повторение. Но для описания сложных связей и закономерностей они не подходят. Самое яркое, бросающееся в глаза ограничение регулярных языков — их неспособность выражать вложенные структуры (без которых не обходится большинство высокоуровневых языков программирования). Простейший пример — правильные скобочные выражения.

Как уже говорилось, алгоритмическая сложность лексического анализа — $O(n)$, где n — длина строки. Математический аппарат для лексического анализа — детерминированные конечные автоматы (Deterministic Finite Automata, DFA).

Контекстно-свободные языки и нотация Бэкуса-Наура

Следующий по сложности класс языков — контекстно-свободные. Это очень важный класс: он используется для синтаксического анализа большинства языков программирования. Например, пресловутые правильные скобочные выражения описываются так:

$$\Sigma_T = \{(,)\}$$
$$\Sigma_N = \{A\}$$
$$P = \{\ A \ \rightarrow (A) \mid AA \mid ()$$
$$S = A$$

Для описания контекстно-свободных грамматик часто используется нотация Бэкуса-Наура (BNF), или расширенная нотация Бэкуса-Наура (Extended BNF, EBNF). Она мало отличается от обычных правил вывода.

Алгоритмическая сложность синтаксического анализа контекстно-свободных языков — $O(n^3)$, детерминированных контекстно-свободных языков — $O(n)$. Математический аппарат — автоматы с стековой памятью (PushDown Automata, PDA).

Как следует из названия (и из вида правил вывода), контекстно-свободные языки не позволяют выражать зависимость от контекста. Простейший пример не контекстно-свободного языка — язык $L = \{a^n b^n c^n, n >= 1\}$. Более близкий к жизни пример — фрагмент программы на языке C: “`a * b;`”. В зависимости от того, что есть `a` и `b`, это может означать объявление указателя или мультипликативное выражение.

В интернедах часто возникают споры о том, является ли грамматика того или иного языка программирования контекстно-свободной. В этих спорах обычно доказывается, что мол, нет, не является, и зря бедным школьникам из года в год втирают о полезности контекстно-свободных грамматик. Формально эти люди правы: если рассматривать полную, исчерпывающую грамматику языка программирования, то почти наверняка она будет не контекстно-свободной. Но важно другое: часто грамматику можно *приблизить* с помощью контекстно-свободной и использовать эффективный алгоритм анализа. Чтобы выразить не контекстно-свободные детали, используются стандартные ухищрения: грамматику делят на лексическую и синтаксическую,

используют глобальную таблицу символов (в примере "`a * b;`" это позволяет установить тип `a` и `b`). В общем, хотя языки программирования не контекстно-свободные, анализируют их контекстно-свободными методами.

Контекстно-зависимые языки

Следующий класс языков — контекстно-зависимые. Эти языки уже настолько сложные, что в общем случае алгоритмическая сложность их синтаксического анализа — $O(2^n)$ (экспоненциальная). Экспоненциальная сложность — это очень плохо. Это значит, что удлиннение программы на один символ может в разы увеличить время её анализа. Из-за такой чудовищной сложности никто не станет работать с контекстно-зависимыми языками в чистом виде: обычно работают с более простыми контекстно-зависимыми подклассами.

Математический аппарат для анализа контекстно-зависимых языков — машина Тьюринга с ограниченной памятью (Linear Bounded Automaton, LBA). (О машинах Тьюринга я поговорю чуть позже.)

Неограниченные языки

Ну и, наконец, самый широкий класс языков — неограниченные. В общем случае задача синтаксического анализа этих языков алгоритмически неразрешима (undecidable). Вот такие они сложные, хо-хо! На самом деле, в математике же как: поставил задачу в самом общем виде, а она возьми да и окажись эквивалентной задаче останова (Halting Problem). Всё упирается в один и тот же потолок. Математический аппарат для анализа неограниченных языков — не кто иной, как сама великая и ужасная машина Тьюринга.

Любопытно, что долгое время шли споры о принадлежности естественных языков (в частности, английского) к конкретному классу. Лично мне интуиция говорит, что английский конечно же не контекстно-свободный, и скорее всего даже не контекстно-зависимый, а неограниченный. Было много "наивных" доказательств того, что английский не контекстно-свободный, но строгое формальное доказательство было получено сравнительно недавно. Дальше дело вроде бы не пошло: никто ещё не доказал неограниченности грамматики английского (но и не построил контекстно-зависимую грамматику).

Ещё любопытнее попытаться выйти за пределы логики первого порядка и посмотреть на какие-нибудь более широкие классы языков. Здесь, впрочем, кончается школьная математика и начинаются куда более странные и философские вещи. О них написано в книге Морриса Клайна "Mathematics: The Loss of Certainty". Эту книжку прочитай — хоть убейся, мало я *таких* книжек знаю. Можешь, правда, не сразу, а как время буит. ;D

Моя любимая книжка о синтаксическом анализе — Jacobs, Grune: "Parsing Techniques. A Practical Guide" (2nd edition). Это книжка невероятной широты: она охватывает сотни алгоритмов, затонувших статей в интернете, покрытых пылью диссертаций и горящих прорывов. С другой стороны, это книжка философская, и рассчитана на широкий круг читателей: лингвистов, математиков, программистов. Прочитай там первые пару глав (это быстро). :)

Другая книжка о синтаксическом анализе и компиляции в целом — Aho, Sethi, Lam, Ullman: "Compilers: Principles, Techniques, and Tools". Она считается классической книжкой для разработчиков компиляторов. Просмотри там первые главы (это быстро). :)

1.2 Семантика

Семантика языка определяет смысл синтаксиса. Задана она может быть по-разному: от простого словесного описания до чёткого алгоритма какой-нибудь абстрактной математической машины. В любом случае семантика определяет *выразительную силу* языка, его способность решать разные задачи.

Влияние семантики на жизнь программиста не такое явное, как влияние синтаксиса, но куда более серьёзное. Программисту важны следующие качества семантики:

- Выразительность — если семантика не позволяет решить задачу, или позволяет со скрипом, придётся поменять язык.
- Простота — чтобы писать программы, достаточно знания синтаксиса, а вот семантику программистам изучать обычно лень (но приходится, когда в программе что-то идёт не так). Поэтому очень важно, когда взмыленный, доведённый до отчаяния программист таки полезет читать ман, чтобы в этом мане всё было просто, логично и без подковёрных мелких деталей и оговорок.
- Надёжность — семантика не должна меняться вообще никогда (или крайне редко), иначе программы будут ломаться самым худшим образом (хитро и незаметно).

Разработчику языка важно:

- Соответствие семантики чёткой математической модели вычислений — разработчик должен хорошо представлять вычислительные возможности языка, иначе он не сможет реализовать язык вообще, или реализует его неэффективно.
- Расширяемость — часто, чтобы расширить выразительную силу языка, к нему прикручивают какие-то сбоку-бантики, которые ломают вычислительную модель языка и сбивают с толку программистов. Разработчик языка должен предвидеть эти сбоку-бантики и заранее заложить достаточно выразительную модель вычислений.

1.2.1 Виды семантики

Я выделю два основных вида семантики:

- Денотационная семантика определяет, *что* означает синтаксическая конструкция на данном языке в терминах другого языка (обычно какой-нибудь абстрактной математической системы), то есть *транслирует* программу.
- Операционная семантика говорит, *как* вычислить значение синтаксической конструкции на данном языке, то есть *интерпретирует* программу. Алгоритм вычисления подразумевает исполнителя, в качестве которого может выступать человек, компьютер, абстрактная машина или ещё кто-то.

Обычно выделяют ещё аксиоматическую семантику: она определяет смысл синтаксической конструкции языка косвенно, через набор логических аксиом и утверждений, истинность или ложность которого надо установить. Но я почему-то не могу выделить её в отдельный вид: мне она кажется подвидом денотационной семантики, просто таким вот "особенным". А ты — смотри сама, лучше почитай умные книжки. ;)

То, что денотационная семантика транслирует, а операционная — интерпретирует, не означает, что в первом случае у языка должен быть транслятор, а во втором — интерпретатор. Определение любого вида семантики апеллирует к каким-то аксиомам: денотационной — к другому языку; операционной — к командам исполнителя; аксиоматической — к законам логики. Семантика может быть смешанной.

1.2.2 Выразительная сила языка

Разные языки предназначены для разных задач. Грубо языки можно поделить на *специализированные* и *общего назначения*, но это разделение скорее относится с сфере использования языка, а не к его выразительной силе. Выразительная сила — это то, насколько широкий круг задач можно решать на этом языке.

Когда начинаешь об этом думать, возникает много вопросов:

- Насколько сложные бывают задачи?
- Все ли задачи можно в принципе решить?
- Если нет, можно ли по задаче хотя бы понять, решаемая она или нет?
- Можно ли быть уверенным, что задача решаема, не зная её решения?

Вопросы непростые, и ответы на них вросли корнями в историю математики. Про всё это хорошо написано в книге Морриса Клайна "Mathematics: The Loss of Certainty", и кое-какие мысли из этой книги (видимо, грязно перевранные) я щас приведу.

Первыми по-настоящему заинтересовались математикой древние греки. Математика существовала и раньше, но египтяне использовли в основном отдельные её элементы и в чисто практических целях. Греки увидели в математике внутреннюю силу и красоту, средство идеально описывать законы реального мира. Они придумали аксиоматический подход и основали на нём геометрию. С алгеброй было сложнее: почти сразу вылезли иррациональные и отрицательные числа, для которых не было аналогии в реальном мире, но математические законы почему-то выполнялись. Возникали горячие споры, что делать с такими "искусственными" областями математики: объяснить не получалось, а выкинуть не хотелось (слишком уж полезные, тем более что среди математиков было немало физиков, астрономов и инженеров).

Следом за греками были индусы — их математика не очень интересовала с философской точки зрения, больше с практической, поэтому они смело и решительно использовали "искусственные" области математики. Индусы придумали много интересных вещей (в том числе — ноль), но ничего толком не объяснили.

Потом были арабы. Они внимательно изучили геометрию древних греков (и исправили некоторые неточности и ошибки в доказательствах теорем) и сильно продвинули алгебру, которая у древних греков считалась второстепенной по сравнению с геометрией. Но арабы тоже не очень интересовались основами математики и связанными с ними философскими проблемами.

Зато эти проблемы заинтересовали европейцев. Математика дошла до них в интересном виде: от древних греков осталась евклидова геометрия — свод незыблемых истин о природе, основанных на малом числе аксиом; от индусов и арабов — ни на чём не основанная, но крайне полезная алгебра. Математиков беспокоило, что необъяснёнными остаются самые простые и фундаментальные вещи: числа. А тем временем математика развивалась стремительными темпами: новые теории были позарез нужны физикам и астрономам. Кроме иррациональных и отрицательных чисел, появилось много новых "искусственных" понятий: дифференциалы, интегралы, комплексные числа и матрицы. Всё чаще математики замечали, что основная беда кроется в понятии бесконечности: как только начинаешь работать с бесконечностью как с числом, законы ломаются. Они даже стали различать два понятия бесконечности: *потенциальная бесконечность* — это когда рассматриваются произвольно маленькие или большие величины; и *реальная бесконечность* — это когда бесконечность воспринимается как нечто целое, единое.

Между тем, как алгебра считалась "искусственной" (или как минимум необоснованной), в абсолютности геометрии не было сомнений. Долгое время люди верили, что геометрия — это язык Бога, с помощью которого он идеально точно описывает законы природы. Аксиомы Евклида абсолютны, потому что они следуют из законов реального мира. Геометрия, построенная на этих аксиомах — единственно возможная геометрия нашего мира. В её доказательствах могут быть мелкие ошибки, но все они исправимы, потому что основа незыблема, основа — это сама природа.

Но ведь если не верить в Бога, то нет особых оснований верить ни во что абсолютное, в том числе в аксиомы Евклида. Когда люди задумались об этом, они поняли, что геометрия не основана ни на чём, кроме чистой веры. Эта мысль не давала им покоя. Математики и философы пытались доказать, что аксиомы следуют из чего-то "более объективного": законов природы, интуиции, чувств, каких-то человеческих особенностей восприятия реальности. Ни одно из этих объяснений не было удовлетворительным. Хуже того: оказалось, что вместо евклидовых аксиом можно взять другие и построить геометрию, которая тоже будет описывать реальный мир.

Пока математики свыкались с мыслью о неабсолютности евклидовой геометрии, похожие открытия продолжались в других областях. Неабсолютность обнаруживалась во всём: аксиомы, методы доказательства, даже предназначение математики было предметом горячих споров. Оказалось, что математика никак не следует из природы, скорее из человеческой головы. А голова — она у всех разная.

Математики раскололись на несколько школ: логики, интуиницонисты, формалисты, теоретико-множественники. Каждая школа предлагала свои основы и свои методы доказательства. Математики разных школ предприняли героические усилия в попытках построить математику заново, и во многом преуспели: дыры были залатаны, основы положены. Но как раз когда

почти всё уже было готово, на них обрушились настоящие трагедии.

В математике обнаружились странные противоречия — парадоксы. Непонятно было, откуда они берутся, и главное — можно ли перестроить математику так, чтобы их не было. В 1930 был получен результат, навсегда разрушивший веру в абсолютность. Это теорема Гёделя о неполноте: если формальная система, достаточно выразительная для целых чисел, непротиворечива — она неполна. Это грубая, неточная формулировка, но идею она передают: какой набор аксиом ни выбери, как ни построй на них математику, всё равно в ней останутся неразрешимые задачи или противоречия.

В свете этого немного по другому выглядит вопрос о разрешимости задачи. Чтобы задачу можно было решать, она должна быть формально сформулирована. Формулировка привязывает задачу к какой-то формальной системе, и решение задачи должно выражаться в терминах этой же системы. Но мы знаем, что в любой непротиворечивой формальной системе есть неразрешимые задачи. Значит, нет гарантии, что наша задача не окажется одной из них. Более того, нельзя заранее выяснить вопрос о разрешимости: если повезёт, мы решим задачу; если не очень повезёт — докажем неразрешимость; а если совсем не повезёт — вообще ничего не докажем.

У математика-теоретика после этого опускаются руки. К счастью, (как сказал какой-то хороший человек) “в реальности всё не так, как на самом деле”. Да, никакая математическая система не может похвастаться разрешимостью любой задачи. Но это не значит, что все они одинаково плохи: одни позволяют решить меньше задач, другие — больше. Даже если плюнуть на неразрешимые задачи, для многих разрешимых нету достаточно эффективного алгоритма. Если задача гарантированно решается, но на решение уходит сто лет — для человека она всё равно что неразрешима. В этом смысле неразрешимость — просто высшая степень недосягаемости решения.

Ну хорошо, идеальной формальной системы, в которой решались бы все задачи, нет и быть не может. Но среди неидеальных, какая самая лучшая? Какая позволяет решать наиболее широкий класс задач?

Вскоре после теоремы Гёделя на этот вопрос независимо ответили сразу несколько математиков. Каждый из них предложил свою модель вычислений и на основе этой модели определил понятие *вычислимости* функции: Алан Тьюринг предложил машину Тьюринга, Алонсо Чёрч — λ-исчисление, а Курт Гёдель — рекурсивные функции. Оказалось, что все предложенные модели, хотя и совершенно разные по форме, по сути совпадают: они описывают один и тот же класс функций. Гипотеза Чёрча-Тьюринга предполагает, что этот класс совпадает с классом всех *вычислимых на компьютере* функций.

Заметь: *вычислимых на компьютере*. Незаметно математики перешли от теории к практике. Вычислимость на компьютере — понятие глубоко физическое. Оно привязывает абстрактную математику к вычислительным возможностям этого мира. А на что способен наш мир — никто не знает. На что способен человеческий мозг, к примеру? Что если он может решать более широкий класс задач, чем компьютер? Даже кроме мозга, есть всякие неизведанные штуки:

квантовая физика, чёрные дыры. Наконец, ничто не мешает миру качественно измениться.

В этом смысле, гипотеза Чёрча-Тьюринга предполагает, что вычислительные возможности нашего мира совпадают с вычислительными возможностями машины Тьюринга. Что же это за штука — машина Тьюринга? Это очень простая абстрактная машина. Вот её формальное определение: машина Тьюринга — это семёрка $M =< Q, \Gamma, b, \Sigma, \delta, q_0, F >$, где

$$Q \text{ — конечное непустое множество состояний}$$

$$\Gamma \text{ — конечное непустое множество символов (алфавит)}$$

$$b \in \Gamma \text{ — пустой символ}$$

$$\Sigma \subseteq (\Gamma \backslash \{b\}) \text{ — множество начальных символов}$$

$$q_0 \in Q \text{ — начальное состояние}$$

$$F \subseteq Q \text{ — множество конечных состояний}$$

$$\delta : (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{L, R\} \text{ — частично-определённая функция перехода}$$

Проще говоря, машина Тьюринга состоит из беконечной ленты-памяти, считывающей головки и функции переходов. Головка считывает символ на ленте и в зависимости от текущего состояния записывает на ленту новый символ, перемещается влево/вправо и переводит машину в новое состояние.

Если гипотеза Чёрча-Тьюринга — правда, это значит, что люди никогда не построят более вычислительно-сильного компьютера. Соответственно они не смогут решать задачи, неразрешимые на машине Тьюринга.

Если эта гипотеза — неправда, то возможно людям удастся построить компьютеры, которые будут решать задачи, неразрешимые на машине Тьюринга — *гиперкомпьютеры*. Важно понимать, что есть большая разница между тем, чтобы *придумать* гиперкомпьютер и *построить* гиперкомпьютер. Придумали их довольном много, в том числе сам Тьюринг: он предложил машину Тьюринга с оракулом (и показал, что даже для машины с оракулом всё равно остаются неразрешимые задачи). Другие гиперкомпьютеры основаны на таких же нереалистичных предположениях (в духе "машина делает бесконечное число операций за конечное время"). Были и более физически-обоснованные попытки (в том числе что-то там с чёрными дырами), но в целом физический гиперкомпьютер вроде бы ещё никто не построил. Одно могу сказать точно: если будут построены гиперкомпьютеры, люди тут же зададутся вопросом, а нельзя ли построить гипергиперкомпьютеры. :D

Если вернуться к разрешимым на машине Тьюринга задачам, то даже они сильно различаются по сложности. Теоретически для решения любой из них достаточно машины Тьюринга. Но это всё равно что микроскопом клопов давить: можно, но не самое простое и эффективное средство. Поэтому задачи разбиваются на классы сложности: все задачи из одного класса имеют примерно одинаковую сложность. В каждом классе выбирается "самая сложная" задача, к которой сводятся все остальные (то есть являются её частным случаем). Выбор задачи-представителя — простой и очень сильный приём. Он позволяет делать выводы о целом классе задач на основании одной-единственной задачи-представителя: например, можно доказать

равенство двух классов, доказав сводимость их представителей друг к другу, или выяснить алгоритмическую сложность неизвестной задачи, сведя её к какой-нибудь известной.

Для каждого класса задач люди придумали модель вычислений, позволяющую эффективно решать задачи из этого класса. Я не буду приводить иерархию этих классов (и соответствующих моделей вычислений): во-первых, я в этой теме слаба, а во-вторых — существует множество подклассов со своими моделями вычислений, и эти подклассы находятся в странных (не всегда даже известных) взаимоотношениях. Некоторые примеры моделей вычислений я упоминала, когда говорила про формальные грамматики.

В математике для описания сложности алгоритма принята нотация "О большое" ("Big O" notation):

$$f(x) = O(g(x))$$

$$\Leftrightarrow$$

$$\exists M > 0, x_0 \in \mathbb{R} \text{ такие, что } \forall x > x_0 : |f(x)| \leq M|g(x)|$$

Функция $f(x)$ может быть функцией времени, памяти или чего угодно (обычно времени).

Но вернёмся к баранам, то есть языкам программирования. Тьюринг-полнота — то есть способность решать все задачи, которые решает машина Тьюринга — важное свойство для языков общего назначения. Специализированные языки могут не быть Тьюринг-полными — они предназначены для узкого круга задач (например, язык работы с базами данных SQL не Тьюринг-полный, регулярные выражения не Тьюринг-полны).

Выразительную силу языка можно понимать и в другом, практическом смысле: насколько хорошо язык позволяет работать с разными штуками в реальной жизни (файловая система, сетевые взаимодействия и т.д). Поскольку программы обычно исполняются в операционной системе, язык должен предоставлять удобные средства работы с операционной системой.

1.3 Структура языка

Итак, язык программирования — это синтаксис и семантика. Программист, однако, смотрит на язык скорее как на набор инструментов, каждый из которых нужен для выражения какой-то конкретной идеи. Разные языки предоставляют разный набор инструментов. Я перечислю некоторые важные составляющие языка:

- переменные
- управление памятью
- система типов
- поток управления
- абстракция
- области видимости

- ввод/вывод
- доступ к кишкам

И немного поговорю про каждую из них.

Переменные

Переменная — это способ обозначить какую-то изменяющуюся величину. В императивных языках программирования переменная — это обозначение куска памяти (l-value). В функциональных языках переменная не привязана к памяти, это просто обозначение выражения (binding). В некоторых языках переменные не могут изменяться (immutable), то есть они скорее "постоянные". Но в целом идея та же: переменная — это обозначение.

Управление памятью

Данные, с которыми программа работает (и саму программу) надо хранить в памяти. Различают *статическую* и *динамическую* память.

В статической памяти хранятся те данные, размер которых известен при старте программы и не изменяется во время исполнения. Сами данные при этом могут меняться, но остаются в той же памяти. Статическая память выделяется один раз, при запуске программы, и освобождается при завершении.

В динамической памяти хранятся данные, которые возникают и исчезают по ходу исполнения прогрммы. Количество и размер этих данных не известны заранее, они зависят от входных данных (и, возможно, фазы луны).

Одни языки отдают управление динамической памятью на откуп программисту: он сам должен следить за выделением новой памяти и освобождением старой. При таком подходе программисты делают много ошибок: забывают освободить память, или освобождают, а потом обращаются к уже затёртым данным. Языки с ручным управлением памятью жертвуют безопасностью ради эффективности. Программы на них могут быть быстрыми, но их надо дополнительно проверять на отсутствие ошибок памяти. Примеры языков с ручным управлением памятью: Assembler, Algol, C, C++, Cobol, Forth, Fortran, Pascal.

Другие языки осуществляют автоматическое управления памятью — *сборку мусора* (garbage collection, GC). Программист в таких языках почти не имеет доступа к упавлению памятью. Сборка мусора — это безопасность, купленная ценой чудовищной неэффективности: сборщик мусора должен во время исполнения программы периодически сканировать всю память, искать ненужные куски и освобождать их. Причём, в отличие от программиста, сборщик мусора мало что знает о времени жизни объектов в памяти: ему приходится отслеживать все ссылки на объект и удалять его только когда не осталось ссылок. Это очень тормозит программу: нередки случаи, когда GC съедает 90% времени. Примеры языков со сборщиком мусора: Lisp, Java, Javascript, Haskell, Perl, Prolog, Ruby.

Наконец, в последнее время появляются языки, которые пытаются отказаться от сборки мусора, но в то же время не дать программисту наделать ошибок в управлении памятью. Сборщик мусора в таком языке может присутствовать, но быть необязательным, нежелательным, выключенным по умолчанию. Это, конечно, самый лучший вариант: он сочетает эффективность и безопасность. Я пока знаю один пример — Rust, но язык — экспериментальный и у него много проблем (прим. Сергея Анатольевича).

Система типов

Тип — это множество значений. Изначально типы выдумали математики, чтобы избавиться от противоречий вроде парадокса Рассела: "Пусть K — множество всех множеств, которые не содержат себя в качестве своего элемента. Содержит ли K само себя в качестве элемента?". Если потребовать, чтобы множество состояло из элементов одного типа, то это множество не может содержать само себя, а вопрос некорректен. Типы спасли математиков от парадоксов, но оказалось, что они накладывают на математику ограничения, не позволяющие выразить некоторые простые и важные вещи. Пришлось добавлять некую сомнительную "аксиому сводимости" (axiom of reducibility) и прочие костыли.

В языке программирования типы — одна из светлых и прекрасных идей, они вносят в прорамму ясность. Типы могут быть *простыми* (число, строка, etc.), а могут быть *составными*, то есть построенными из других типов (структуры, классы, записи, перечисления — всё это примеры составных типов данных в разных языках). Отношения между типами данных могут быть самыми разными: непересекающиеся множества, подмножества и т.д. В некоторых языках можно выстраивать сложные иерархии типов.

У системы типов есть следующие ортогональные характеристики:

- **статическая/динамическая**
  Если типизация статическая, все типы проверяются на этапе компиляции, во время исполнения программы типов уже нет. Если типизация динамическая, типы проверяются во время исполнения программы. Динамическая типизация имеет два чудовищных недостатка: во-первых — программа становится намного медленнее (из-за проверок типов в рантайме и невозможности целого ряда оптимизаций), во-торых — все ошибки типов выясняются только во время исполнения. Многие люди любят динамическую типизацию за её "естественность", но лично я её злостно недолюбливаю за медленность. Примеры: статическая: C, Haskell, Java, C#; динамическая: Python, JavaScript, Ruby.

- **явная/неявная**
  Иногда программисту не надо указывать тип — языковой процессор может вывести тип самостоятельно, на основании уже известных типов в программе. Если информации недостаточно, чтобы вывести тип, языковой процессор может выдать ошибку или выбрать дефолтный вариант. Неявная типизация — это когда типы можно не указывать (или даже нельзя указывать), явная — когда обязательно указывать. В общем, это дело вкуса: я предпочитаю видеть типы, но иногда они занимают очень много места. Примеры: явная — C++, D, C#; неявная: PHP, Lua, JavaScript.

- **строгая/нестрогая (сильная/слабая)**

Языки со строгой типизацией не позволяют смешивать разные типы. Обычно есть возможность *привести* один тип к другому, но такое приведение должно быть явным. Языки с нестрогой типизацией автоматически (неявно) выполняют приведение типов по мере надобности Иногда это бывает удобно (например, складывать целые числа с действительными), но в большинстве случаев приводит к ошибкам. В сущности, пропадает главное достоинство типизации: защита от дурака (то есть самого программиста). Примеры: строгая — Java, Python, Haskell, Lisp; нестрогая: C, JavaScript, Visual Basic, PHP.

Очень важно не смешивать эти понятия: не думать, что динамическая типизация обязательно нестрогая, или что неявная типизация — это когда есть неявное приведение типов. Люди постоянно путают и мешают всё в кучу, да это и понятно: они судят о типизации по какому-нибудь знакомому языку. Например, типизация Javascript'а — динамическая, неявная и нестрогая, поэтому для джаваскриптовиков понятие динамичности прочно увязывается с понятием нестрогости и даже какой-то расхлябанности: делай что хочешь, интерпретатор съест. В Haskell'е типизация статическая, наполовину неявная и строгая — поэтому у хаскеллистов статичность увязывается со строгостью (если программа на хаскеле компилируется, значит в ней отсутствует целый класс потенциальных ошибок).

Бывают и языки совсем без типов (например, Assembler).

Поток управления

Программа на любом языке подразумевает выполнение *действий* для достижения *результата*. Форма, в которой записана программа, может быть очень разной, но два основных подхода такие:

- **императивный**: программа — это список инструкций исполнителю
- **декларативный**: программа — это описание результата, который нужно получить

Обычно эту форму называют *парадигмой программирования* и выделяют ещё много других парадигм: объектно-ориенированную, функциональную, логическую и т.д. Однако около всех этих парадигм возникла неслабая путаница, в результате которой мало кто понимает, что означает, к примеру, функциональное программирование: то ли это языки, в которых есть функции высших порядков (high-order functions, HOF), то ли языки в которых функции являются чистыми функциями своих аргументов (нет side-effect'ов), то ли просто языки, которые хорошо выполняют свою функцию. :D Кроме того, ничто не мешает объектно-ориентированному языку быть при этом функциональным. Лучше вообще не употреблять слова "парадигма", потому что непонятно, к чему оно относится. Вот "поток управления" — это понятно: это порядок выполнения действий для достижения результата.

Ты наверное заметила, что выделенные мной формы задания потока управления (императивный/декларативный) сильно смахивают на виды семантики (операционная/денотационная). Как и с семантикой, подчёркиваю, упреждаю, предостерегаю, наконец, попросту говорю: это мне внутреннее чувство не даёт выделить другие виды как что-то принципиально новое, а в миру бытуют разные мнения. :D

В императивных языках порядок, в котором записана программа, отражает порядок выполнения действий. Если два действия записаны последовательно друг за другом, они так же последовательно будут выпоплнены. Есть разные способы передачи управления: безусловные переходы, условные переходы, циклы, процедуры и функции. Программист управляет тем, *как* вычисляется результат. Примеры императивных языков: Assembler, C, C++, Java, Pascal.

В декларативных языках порядок, в котором записана программа, не важен: программа — это описание конечного результата, и порядок действий определяется зависимостью промежуточных результатов друг от друга. Если конечный результат зависит от промежуточного — промежуточный должен быть вычислен раньше. Программист не управляет напрямую тем, *как* вычисляется результат — он только говорит, *что* надо получить. Поэтому в декларативных языках нет способов передачи управления, вместо них есть способы описания зависимостей. Примеры декларативных языков: SQL, HTML, Prolog, регулярные выражения, с большего Haskell.

Есть ещё одна важная штука: параллелизм. Часто какие-то независимые действия в программе можно выполнять параллельно. Если процессор многоядерный, речь идёт о настоящем параллелизме: одновременно выполняются несколько потоков управления. Настоящий параллелизм может давать неслабый прирост производительности (почитай про закон Амдала). Есть ещё псевдопараллелизм, когда потоки управления выполняются на одном и том же процессоре, просто по очереди: немного повыполнялся один, потом следующий и т.д. Обычно потоков управления больше, чем ядер процессора (особенно если одновременно выполняется много программ), поэтому псевдопараллелизм надстраивается поверх настоящего параллелизма: все потоки управления выстраиваются в очередь, а из очереди распределяются на разные ядра.

Одни языки лучше приспособлены для параллелизма (легко отслеживать зависимости между данными), другие — хуже (много запутанных зависимостей). Между тем роль параллелизма растёт: в последнее время ускорять процессоры стало почти некуда, и мощности стали наращивать вширь: делать многоядерные процессоры и многопроцессорные архитектуры. Поэтому для языка программирования крайне важно предоставлять простые и эффективные средства распараллелить программу.

Абстракция

Иногда есть много похожих вещей, объединённых какой-то общей чертой. Это могут быть похожие структуры данных, похожие функции, похожие типы или что угодно ещё. Если эта похожесть — не случайное, а неотъемлемое качество, её обязательно надо зафиксировать (как инвариант, который не должен нарушаться). Если одна и та же идея реализована несколько раз, приходится постоянно следить, чтобы все реализации делали одно и тоже, и почти неизбежно вкрадутся различия и ошибки. Наконец, просто написать много одинакового кода — уже неприятно. Поэтому у языка должны быть средства выразить похожесть, то есть абстрагироваться от мелких отличий и выразить общую идею — средства абстракции. Вот некоторые из них:

- **структурное программирование**
  Это разбиение программы на подпрограммы: модули, процедуры, функции и т.д.

- **параметирический полиморфизм**
  Параметрический полиморфизм — это когда есть одна общая для всех возможных параметров реализация идеи. Эта реализация не делает никаких предположений о передаваемом в неё параметре, она максимально общая и универсальная. Это как бы аналитически заданная всюду определённая функция параметра.

- **ad-hoc полиморфизм**
  Ad-hoc, или "абы-какой" полифорфизм — это когда идея имеет несколько разнородных реализаций для отдельных параметров. Каждая реализация особенная, подходит только для этого конкретного параметра. Параметры, для которых идея не реализована — в пролёте. Это как бы таблично заданная частично определённая функция параметра.

- **полиморфизм подтипов**
  Полиморфизм подтипов, или полиморфизм включения (subtyping) — это когда есть иерархия, основанная на включении: потомки наследуют свойства предков. При этом потомки могут менять унаследованные свойства. Получается, что идея реализована для всей иерархии (у одних потомков переопределённая реализация, у других — унаследованная), но вне иерархии не существует.

- **DSL**
  DSL — Domain Specific Language. Некоторые языки позволяют легко писать на них маленькие специализированные языки — DSL'и. Такой DSL затачивается под определённый круг задач, и эти задачи решаются на нём в несколько строчек. Конечно, любой DSL можно написать с нуля хотя бы и на ассемблере, то есть он ничем принципиальнвм не отличается от обычного языка. Но с реализацией обычного языка долго мучаются, а DSL'и пишут быстро и легко за счёт средств исходного языка.

Абстракция может относиться к потоку управления (структурное программирование), данным (полифморфизмы) или целому языку (DSL'и). Вообще, язык высокого уровня — сам по себе абстракция над языками низкого уровня.

Хорошие абстракции — это такие, которые не ухудшают эффективность программы, то есть программист не стоит перед мучительным выбором: стройная, но медленная программа или распухшая, но быстрая. Про такие абстракции говорят ещё "нулевая абстракция". Но довольно часто абстракция вынуждает жертвовать эффективностью: например, один общий сложный алгоритм для решения многих задач (некоторые из которых можно решить проще). Особенно хорошо это чувствуется в достаточно низкоуровневых языках (типа C/C++): часто какая-то библиотечная функция слишком медленная, потому что рассчитана на общий случай и делает много лишнего.

Области видимости

Область видимости (scope) — это часть программы, в пределах которой идентификатор продолжает быть связанным со своим значением. Есть два главных вида областей видимости:

- **лексическая (статическая)**, или *раннее связывание* — под "областью программы" понимается часть исходного кода, в которой объявлен идентификатор.

- **динамическая**, или *позднее связывание* — под “областью программы” понимается состояние программы во время исполнения, когда встретился идентификатор.

Приведу для понятности хороший пример с википедии. Это программа на bash’e:

```
x=1
function g () { echo $x ; x=2 ; }
function f () { local x=3 ; g ; }
f
echo $x
```

У bash’a динамическая область видимости, поэтому программа выведет 3 и 1. Точно такая же программа на C:

```
#include <stdio.h>

int x = 1;

void g () {
    printf ("%d", x);
    x = 2;
}

void f () {
    int x = 3;
    g ();
}

int main () {
    f ();
    printf ("%d", x);
    return 0;
}
```

выведет 1 и 2, потому что в C лексическая область видимости.

Области видимости можно классифицировать по их “ширине охвата”:

- глобальная
- модуль
- единица трансляции
- функция
- блок
- выражение

Ввод/вывод

Полезный язык предоставляет возможности для взаимодействия с внешним миром: файловой системой, сетью, другими программами, устройствами ввода/вывода и прочим. Обычно это просто обёртки над системными вызовами, но, поскольку хороший язык должен быть портабельным, он должен учитывать разные операционные системы. Например, хорошо бы, чтобы одна и та же программа читала файлы и на линуксе, и на винде.

Такая портабельность даётся непросто (особенно если все разработчики языка живут на какой-то одной системе), но она очень важна: это одно из свойств, за которые любят высокоуровневые языки. Никому не хочется мучиться с разными архитектурами, хочется получить портабельность за бесплатно.

Доступ к кишкам

Некоторые языки позволяют прямо во время исполнения программы узнавать что-то о программе (introspection), менять саму программу (reflection, self-modifying code) или просто динамически генерировать код (dynamic code generation).

Например, C++ позволяет узнать тип объекта во время исполнения (`typeid` и `dynamic_cast`).

А в Javascript'е есть функция `eval`, принимающая строчку, которая интерпретируется как код программы. Функция `eval` прямо на ходу парсит и исполняет эту строчку. Например, такая программа:

```
eval("2 + 3")
```

выведет 5. А вот такой пример:

```
a = 1
s = "print(a++); if (a <= 10) eval(s)"
eval(s)
```

выведет 1, 2, 3, 4, 5, 6, 7, 8, 9 и 10. Можно на ходу создавать функции или менять уже существующие:

```
eval = function () { print("no eval!") }
eval("2 + 3")
```

выведет, уже не 5, а “no eval!”.

Некоторые языки позволяют динамически генерировать и исполнять не код, а какое-нибудь промежуточное представление: байткод или AST. В примере из последней главы мы будем строить маленький JIT-компилятор и тоже займёмся динамической генерацией кода (машинного).

Всё это интересные возможности, но довольно часто они приводят к нехорошим последствием: программист начинает писать запутанные программы, по которым непонятно, что они делают, пока их не начнёшь выполнять. Во время исполнения начинаются какие-то ковыряния в

кишках интерпретатора и прочие странные махинации. Обычно это говорит либо о том, что программист не осилил что-то сделать нормальными средствами, либо что он намеренно хочет запутать программу (недаром на Javascript'е пишут столько вирусов и троянов). Но есть и другие примеры использования: например, ранние компиляторы C были ограничены таким маленьким объёмом памяти (размер самой программы был ограничен), что им приходилось на ходу перетирать свой же код более новым. Да и упомянутые JIT-компиляторы должны на ходу генерировать и исполнять код.

1.4 Итоги

Язык программирования состоит из синтаксиса и семантики.

Синтаксис языка может быть определён с помощью формальных грамматик. Иерархия Хомского — это классификация формальных грамматик по правилам вывода: неорганиченные, контекстно-зависимые, контекстно-свободные, регулярные. Для языков программирования наиболее важны два класса грамматик: регулярные (они используются для описания лексем) и контекстно-свободные (они используются для описания связей между лексемами).

Семантика языка определяет его выразительную силу, то есть круг задач, которые можно решать на этом языке. На сегодняшний день выразительная сила всех языков ограничена выразительной силой машины Тьюринга.

Синтаксис и семантика до определённой степени взаимозаменяемы: сематические особенности можно выразить с помощью более сложного синтаксиса, а синтаксически трудновыразимые тонкости можно взвалить на семантику.

В языке программирования очень важны: выразительность, эффективность, защита от дурака, взаимодействие с внешним миром.

Глава 2

Машинный язык

Компьютер — это программируемое вычислительное устройство. Он состоит из четырёх основных компонент:

- процессор
- память
- устройство ввода
- устройство вывода

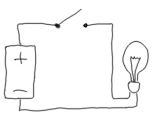
В этой главе я приведу грязно перевранную выжимку из книги Чарльза Петцольда “Code”: попытаюсь кратко объяснить, что такое машинный язык.

2.1 Электричество и логика

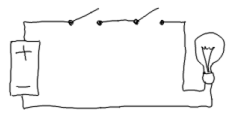
Компьютеры придумали для автоматизации вычислений. Первый компьютер, придуманный Чарльзом Бэббиджем, был механическим и основанным на десятичной системе счисления. Из-за своей сложности он так и не был построен. Понадобилось сделать два серьёзных упрощения, чтобы построение компьютеров стало возможным: от механики перейти к электричеству, а от десятичной системы счисления — к двоичной.

В принципе, компьютеры не обязательно строить на основе электрических схем. Более того, не исключено, что есть какие-то другие физические процессы, которые позволят построить более вычислительно-мощные компьютеры. Но чтобы привязаться хоть к какой-то реальности, я рассмотрю несколько простых электрических схем, на основе которых можно собрать схемы любой сложности.

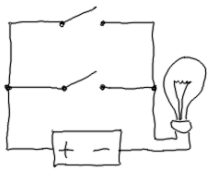
Начнём с простой схемы:



В этой цепи не много элементов: лампочка, батарея, ключ. Если ключ разомкнут, ток не идёт, и лампочка не горит. Если ключ замкнут, ток идёт, и лампочка горит. (Кстати, лампочка нас интересует только как индикатор, есть ток или нет.) Добавим ещё один ключ в схему (последовательно):

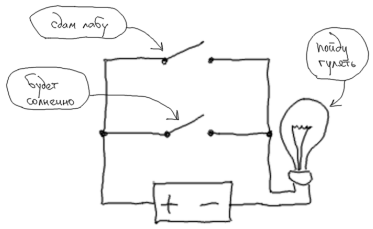


Теперь лампочка зависит от двух ключей: она горит, только если первый *и* второй ключ замкнут. Если бы мы добавили второй ключ параллельно, схема бы получилась такой:

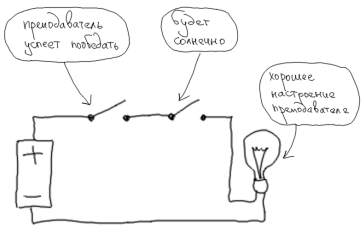


В этой схеме лампочка горит, только если первый *или* второй ключ замкнут.

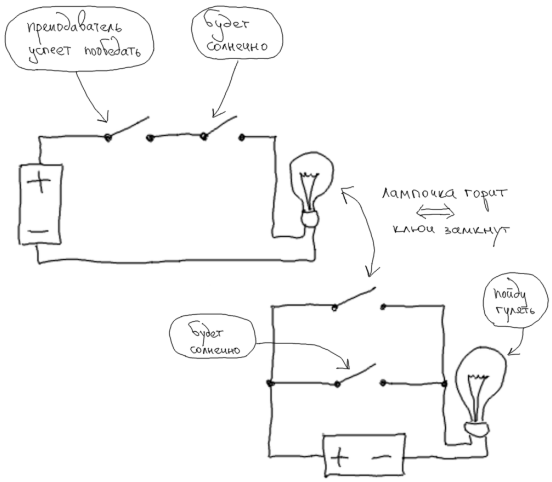
Мы получили две интересных схемы: первая отражает логическое И (AND), вторая — логическое ИЛИ (OR). На основе этих схем можно собирать другие, более сложные. Например, есть у тебя условие: “если я сдам лабу или будет солнечно, то я пойду гулять”. Это условие описывается схемой ИЛИ:



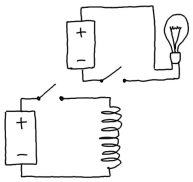
Преподаватель, которому ты собралась сдавать лабу, знает, что если у него будет хорошее настроение, то ты сдашь лабу. Его настроение зависит от двух вещей: во-первых, будет ли солнечно, а во вторых, успеет ли он пообедать. Он может описать это с помощью схемы И:



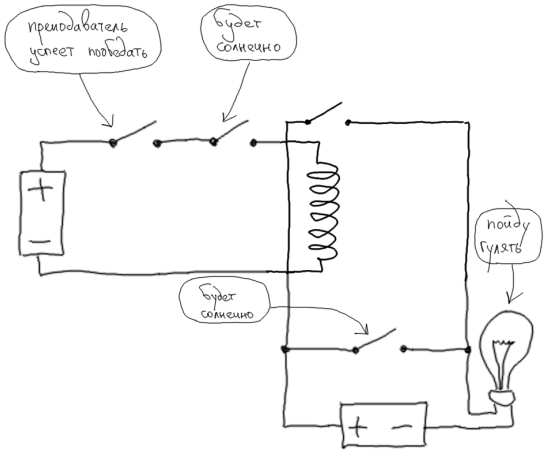
Допустим, преподаватель хочет узнать, пойдёшь ли ты завтра гулять, но не очень хорошо разбирается в хитросплетениях твоей логики. Зато у него есть твоя схема. Он берёт её и пытается присобачить к ней свою. Логически это очень просто: нужно связать выход “хорошее настроение” со входом “сдам лабу”: если на выходе “хорошее настроение” есть ток, ключ “сдам лабу” замкнут, если тока нет – разомкнут:



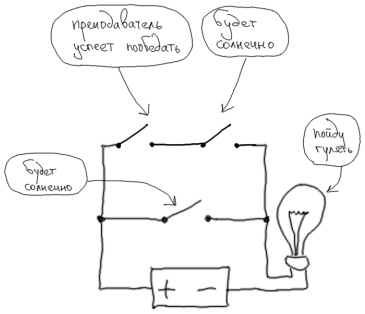
Нужно что-то, что автоматически замыкало и размыкало бы ключ в зависимости от тока. Одним из таких устройств является реле:



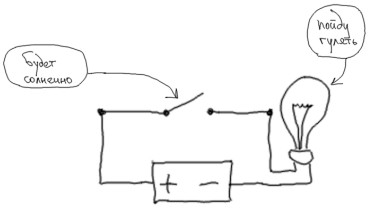
Если в первой цепи течёт ток, то катушка намагничивается и начинает притягивать ключ во второй цепи. Ключ замыкается. Если в первой цепи ток пропадает, то катушка размагничивается, перестаёт притягивать ключ и он размыкается. Реле позволяет связать вход “сдам лабу” с выходом “хорошее настроение”:



Конечно, эту конкретную схему можно упростить и выкинуть реле, просто встроив преподавательскую схему в твою вместо ключа “сдам лабу”:



И даже ещё сильнее упростить:

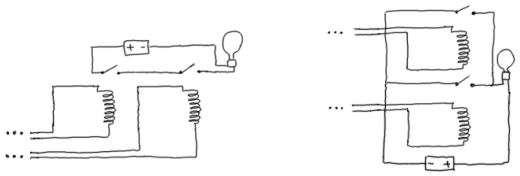


Но в общем случае реле позволяет сделать очень важное предположение: что выход любой схемы совместим со входом любой другой схемы, то есть можно собирать из простых схем более сложные. (В телеграфах реле использовалось для усиления слабого тока при передаче сигнала на большие растояния.)

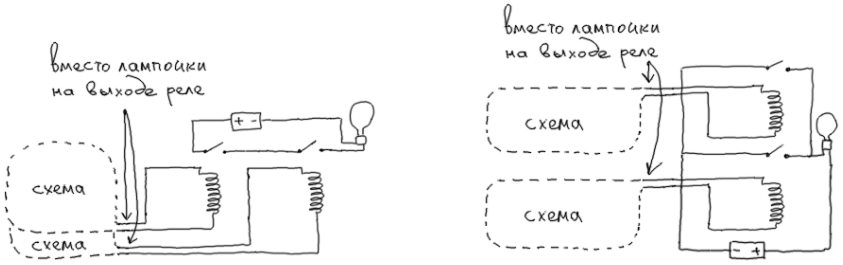
Реле — не единственное устройство, позволяющее управлять током с помощью тока. Сейчас

используются транзисторы, а до этого были вакуумные лампы.

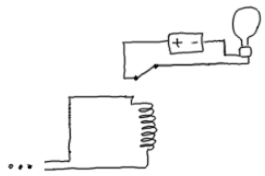
Вот как выглядят схемы И и ИЛИ с использованием реле:



Строго говоря, это уже не схемы, а фрагменты схем. Но, с другой стороны, “выход” схемы — тоже не выход, а кусок цепи, по которому идёт или не идёт ток. Чтобы соединить выход и фрагмент, нужно разомкнуть цепь в районе выхода и получившиеся два конца провода примотать к концам провода, торчащим из фрагмента:



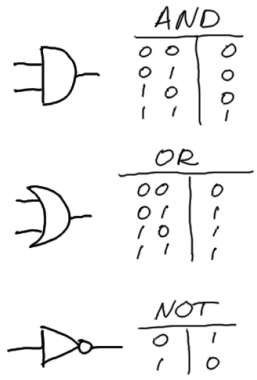
Дальше я всё буду называть схемой. Вот ещё одна важная схема, отражающая логическое НЕ (NOT):



В предыдущих схемах реле повторяло сигнал, а здесь оно его инвертирует: если в первой цепи идёт ток, то во второй тока нет, и наоборот. Поэтому эта схема называется *инвертором*.

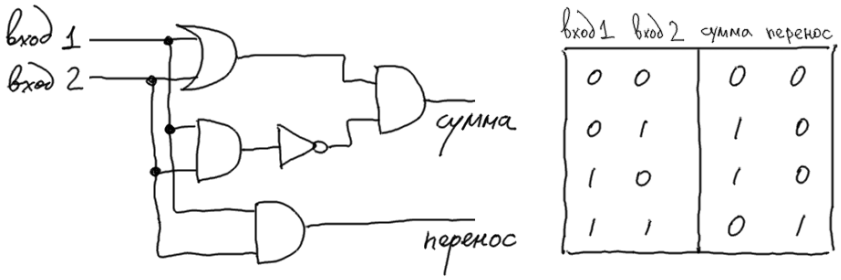
Логические функции И, ИЛИ, НЕ образуют *полную систему булевых функций* — через них можно выразить любую логическую функцию. (Вообще, это не минимальная полная система: с помощью законов де Моргана ИЛИ выражается через И и НЕ, а И — через ИЛИ и НЕ, но мне удобнее иметь под рукой три схемы.)

Для этих трёх схем — И, ИЛИ, НЕ — я введу (общепринятые) условные обозначения:



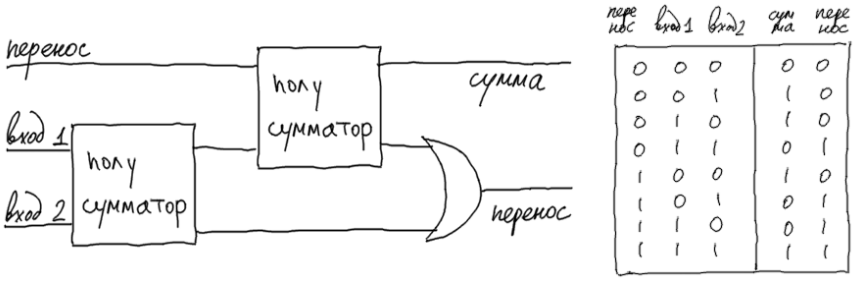
2.2 Сумматор

Вот устройство, способное сложить два бита — полусумматор (half-adder):



У полусумматора два входа (для двух однобитных слагаемых) и два выхода (для однобитной суммы и переноса на следующий разряд, поскольку сумма может быть двухбитной).

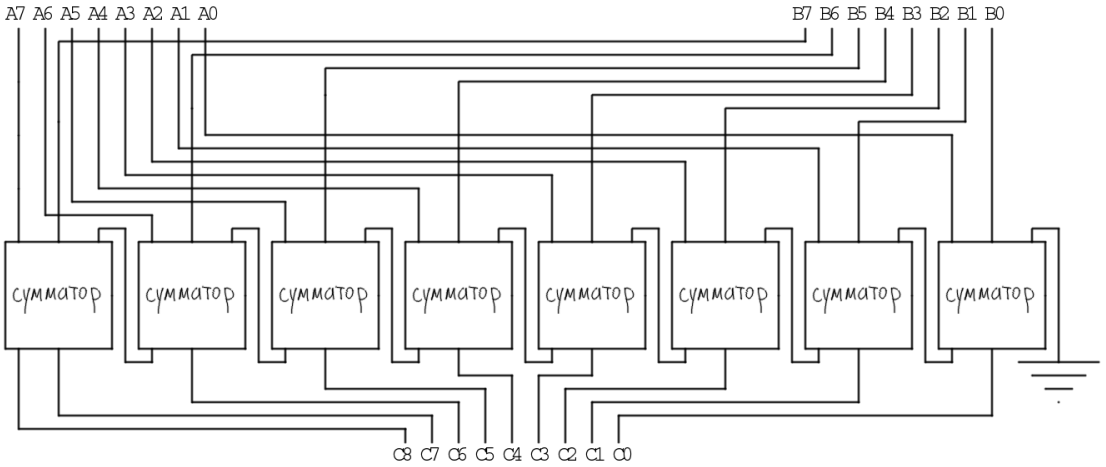
Чтобы складывать многобитные числа, в каждом разряде кроме двух слагаемых нужно учитывать ещё и перенос с предыдущего разряда. Вот схема, способная сложить три бита — сумматор (adder):



У сумматора три входа (для двух однобитных слагаемых и переноса с предыдущего разряда) и два выхода (для однобитной суммы и переноса на следующий разряд).

Соединяя сумматоры в цепочку, мы можем собрать сумматор для N-битных чисел. Вот, на-

пример, 8-битный сумматор:



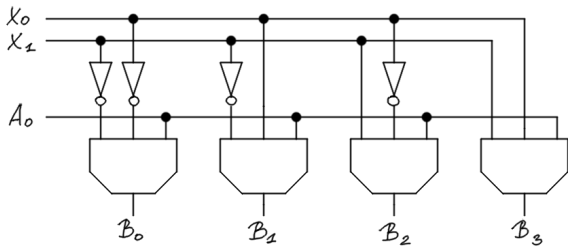
У 8-битного сумматора два 8-битных входа $A_0 - A_7$ и $B_0 - B_7$ (для двух 8-битных слагаемых) и один 9-битный выход $C_0 - C_8$ (сумма двух 8-битных чисел может быть 9-битным числом).

2.3 Команды

Примерно так же, как N-битный сумматор, можно собрать N-битный вычитататель, умножатель, делитель и т.д. Допустим, мы собрали много разных N-битных схем и хотим объединить их в один большой N-битный калькулятор. У калькулятора должно быть два N-битных входа для данных и ещё какой-то вход для выбора нужного действия.

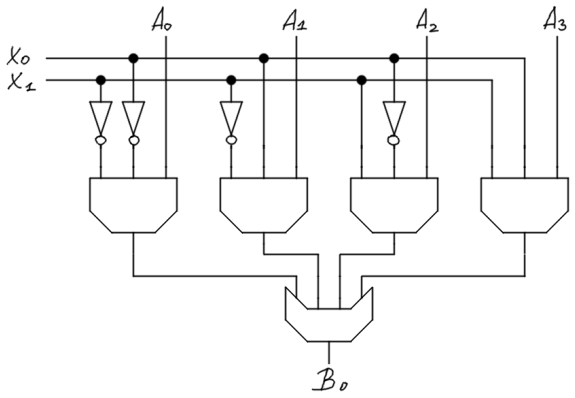
Для выбора действия нам понадобятся два устройства.

Первое устройство — дешифратор — отвечает за подачу входных данных. Оно должно перенапрвалять операнды на вход выбранной схемы, а на вход остальных схем подавать ноль. Вот пример дешифратора 2 на 4:



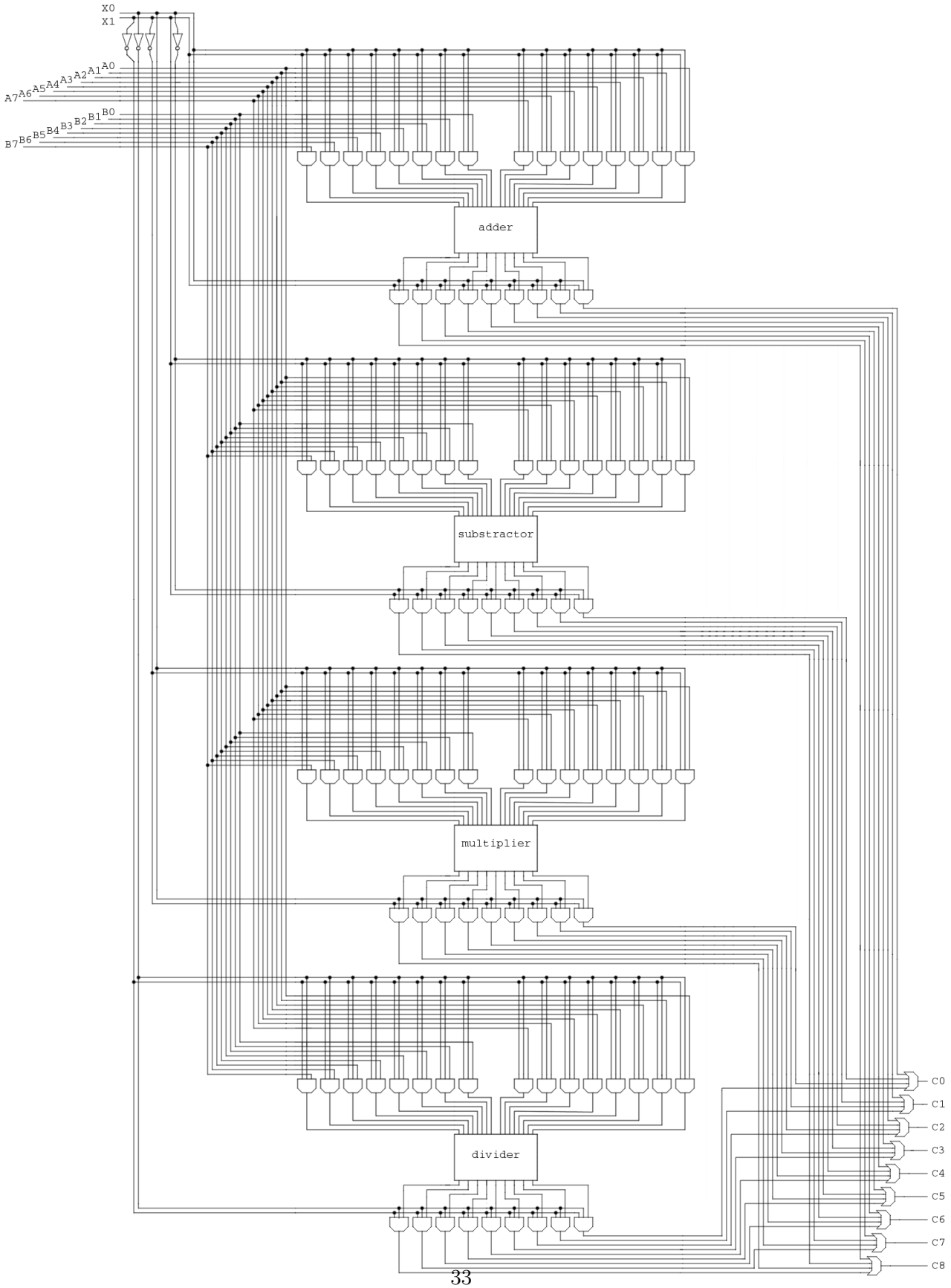
Входы X_0 и X_1 — управляющие. Меняя комбинацию битов на этих входах, можно управлять тем, на какой из выходов B_0, B_1, B_2 или B_3 попадёт бит на входе A_0: значение пары (X_0, X_1), равное $(0, 0)$ кодирует выход B_0, $(0, 1)$ — B_1, $(1, 0)$ — B_2 и $(1, 1)$ — B_3. На остальных выходах сигнал равен нулю.

Второе устройство — селектор — отвечает за считывание результата. Оно должно считывать результат с выхода выбранной схемы, а выходы остальных схем игнороровать. Вот пример селектора 4 на 1:



Входы X_0 и X_1 — управляющие. Меняя комбинацию битов на этих входах, можно управлять тем, какой из входов A_0, A_1, A_2 или A_3 попадёт на выход B_0: значение пары (X_0, X_1), равное $(0, 0)$ кодирует вход A_0, $(0, 1)$ — A_1, $(1, 0)$ — A_2 и $(1, 1)$ — A_3. Сигнал на остальных выходах игнорируется.

Чтобы применить это к N-битным схемам, нужно на каждый бит входа повесить по дешифратору, а на каждый выход — по селектору. Например, для четырёх схем с двумя 8-битными входами и 9-битным выходом (например, сложение, вычитание, умножение и деление) схема была бы такой:


X0
X1
A7 A6 A5 A4 A3 A2 A1 A0
B7 B6 B5 B4 B3 B2 B1 B0
adder
substractor
multiplier
divider
C0
C1
C2
C3
C4
C5
C6
C7
C8

В зависимости от комбинации бит на входе селектор активирует одну из четырёх схем: 11 — сложение, 10 — вычитание, 01 — умножение, 00 — деление. Схема, конечно, не без недостатков: результат любой операции 9-битный, как у сложения, хотя результат вычитания и деления — 8-битный, а умножения — 16-битный. Зато схема на всю страницу. :D

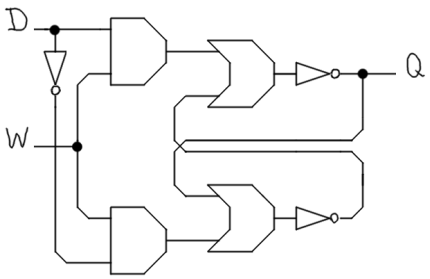
Получился этакий простецкий калькулятор. Мы управляем калькулятором, подавая на вход какую-то комбинацию бит. Эта комбинация бит — это *команда* на *машинном языке*.

В общем-то, на этом можно было бы остановиться: уже должно быть понятно, что такое машинный язык. Но на пути от калькулятора к компьютеру осталось ещё два серьёзных прорыва.

2.4 Память

Что можно сделать с результатом вычислений нашего калькулятора? Можно ли его использовать для дальнейших вычислений в самом калькуляторе? Результат представляет собой набор электрических сигналов на выходе схемы, который напрямую зависит от сигналов на входе: поменяешь вход — тут же поменяется выход. Поэтому, если мы попытаемся подать результат на вход схемы для дальшейшего использования, ничего не получится: схема тут же среагирует и поменяет результат на выходе, что повлечёт изменение входа и т.д. Нужен какой-то способ отвязать результат от входа калькулятора, сохранить его независимо от изменений на входе.

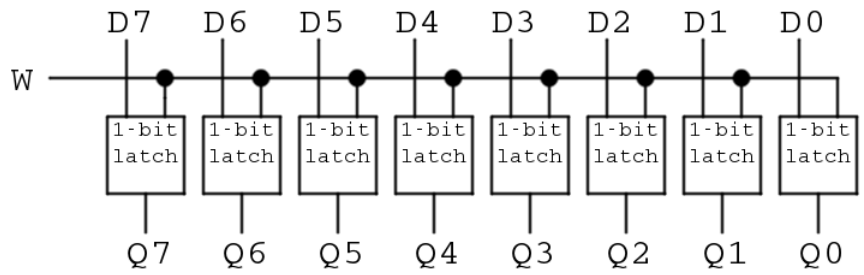
Нам нужна схема, обладающая способностью *запоминать* сигнал. В этом нам поможет однобитная защёлка:



Эта схема принципиально сложнее предыдущих, потому что в ней используется обратная связь. У схемы два входа: вход D для однобитовых данных (data) и вход W для управления записью (write). Когда вход $W = 1$, выход Q копирует значение на входе D. Когда вход $W = 0$, никакие изменения на входе D не влияют на выход Q — он сохраняет последнее значение входа D при $W = 1$.

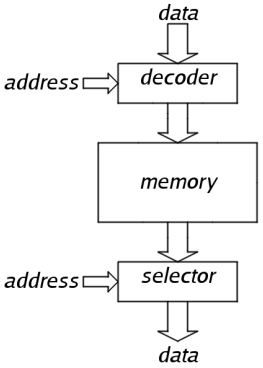
Схема, лежащая в основе защёлки (из двух ИЛИ и двух НЕ с обратной связью) называется RS-триггером (Reset - Set). Именно RS-триггер реализует запомининие, а остальные элементы схемы (два И и одно НЕ) реализуют удобный переключатель для записи/хранения бита.

Из однобитных защёлок очень просто собрать многобитную: надо просто объединить их входы W. Например, 8-битная защёлка выглядит так:

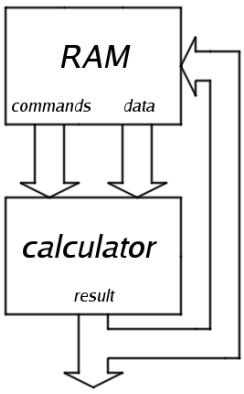


С помощью таких защёлок можно сохранять результат вычислений калькулятора, а потом, когда будет надо, подавать его на вход для дальнейшего использования.

Более того, можно собрать сразу много N-битных защёлок и выбирать, в какую из них записывать или считывать данные (с помощью дешифратора и селектора, как и с выбором команд). Это уже получится память с произвольным доступом (Random Access Memory, RAM).



С помощью M-битного дешифратора и селектора можно адресовать 2^M N-битных ячеек памяти.

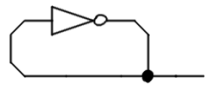


Изначально мы собирали память для того, чтобы хранить в ней промежуточные результаты вычислений на калькуляторе. Но ведь в памяти можно хранить не только данные, но и команды калькулятора — они, как и данные, кодируются битовыми последовательностями. Вместо того, чтобы задавать команды вручную, можно считывать их из памяти и подавать на вход калькулятора.

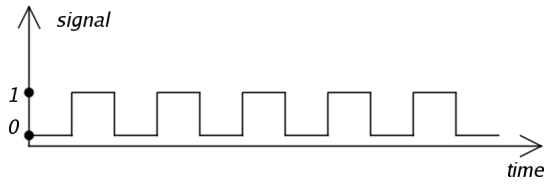
2.5 Автоматизация

Итак, у нас есть программируемый калькулятор. Не хватает одной мелочи: оживить его, чтобы он как-то сам по себе читал команды из памяти и исполнял их. Для этого нам понадобится счётчик. Счётчик состоит из двух частей: осциллятора и делителя частоты.

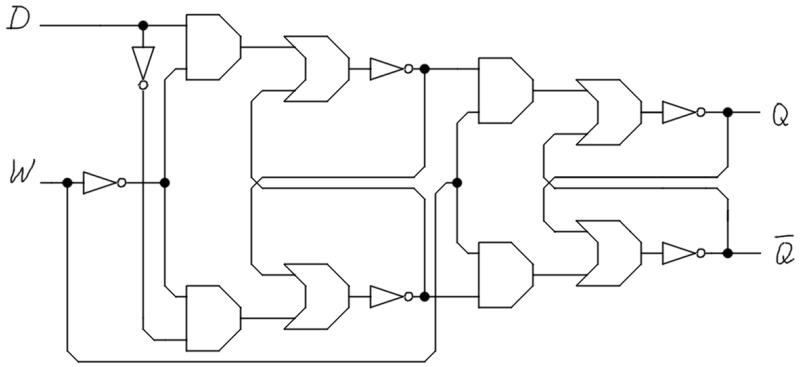
Осциллятор прост:



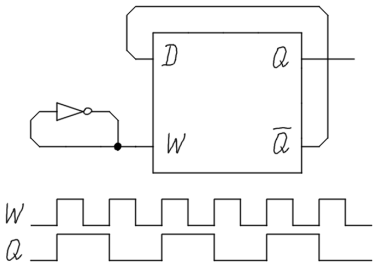
Выход осциллятора колкблется между 0 и 1:



Делитель частоты не так прост. Он построен на основе D-триггера. D-триггер похож на RS-триггер. У него тоже два входа: D — для данных, и W — для управления записью в триггер. Ключевое отличие D-триггера от RS-триггера заключается в условии, при котором происходит запись (то есть сигнал на входе D копируется на выход триггера): если в RS-триггере это условие $W = 1$, то в D-триггере это условие $W = \uparrow$, т.е. изменение сигнала на входе W с 0 на 1. D-триггер состоит из двух RS-триггеров:

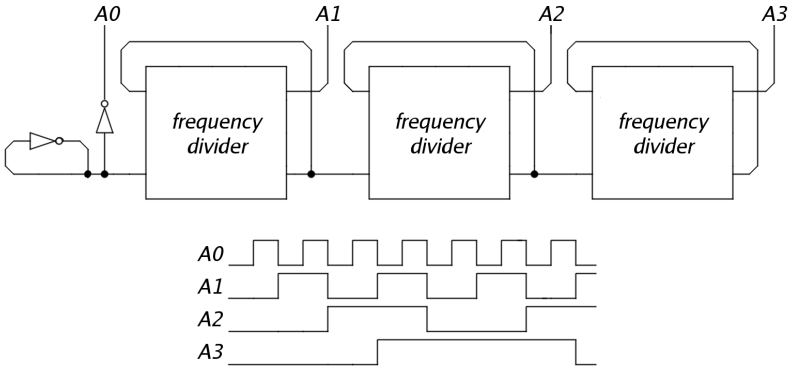


Если соединить выход $\overline{Q}$ D-триггера со входом D, то получится делитель частоты — схема, которая вдвое уменьшает частоту входного сигнала. Например, если ко входу W присоединить осциллятор, выходной сигнал которого колеблется с частотой t, то на выходе Q мы получим сигнал, колеблющийся с частотой $t/2$:



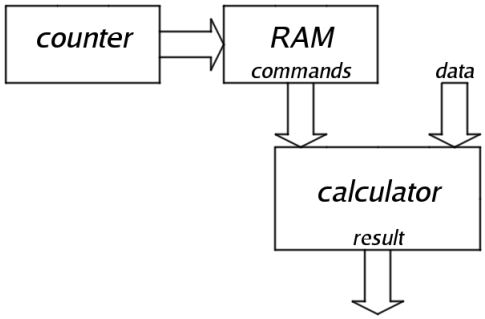
Если последовательно соединить осциллятор и несколько делителей частоты, то получится

счётчик:



Выходы A_0, A_1, A_2, A_3 образуют двоичное число, которое увеличивается на единицу кажую единицу времени (если $A_0 = 1$, $A_1 = 1$, $A_2 = 1$, $A_3 = 1$, то в следующий момент времени счётчик обнуляется).

Если присоединить выход счётчика ко входу селектора памяти, то значение счётчика будет интерпретироваться как адрес в памяти. Получится, что счётчик последовательно перебирает все ячейки памяти. Если при этом выход селектора памяти соединить со входом для команд калькулятора, а в памяти последовательно хранить команды, то счётчик будет раз в единицу времени подавать новую команду на вход калькулятора.



Примерно так можно автоматизировать выборку команд из памяти. Остаётся вопрос, где хранить данные и как синхронизировать подачу команд и данных на вход калькулятора.

Самый простой, негибкий вариант — хранить данные в отдельном массиве, в порядке, строго соответствующем порядку команд: чтобы по сигналу счётчика одновременно выбиралась команда и данные для неё.

Более гибкий вариант — кодировать адрес данных в самой команде: тогда сначала придётся загрузить из памяти команду, а схема, отвечающая за исполнение этой команды, должна выковырять из команды адрес данных и загрузить их. Этот вариант медленнее, и команды становятся более длинными, зато данные можно хранить в любом порядке и в любом уголке памяти.

Чтобы калькулятор стал полноценным компьютером, в него ещё нужно добавить команду условного перехода: возможность по какому-то условию программно изменять значение счётчика. Добавить её не сложно: просто надо на какой-то свободный выход селектора команд повесить схему, записывающую новое значение в счётчик. Я не буду рассматривать, как именно это сделать — и условный переход, и всё остальное намного чётче и подробнее описано

в книжке Петцзольда “Code”.

2.6 Ассемблер

Человеку неудобно работать с битами: трудно запоминать и различать длинные последовательности нулей и единиц. Куда проще придумать короткое, запоминающееся название для каждой машинной команды. Язык таких обозначений называется языком ассемблера (сокращённо просто “ассемблер”).

Программа на ассемблере — это текст, то есть последовательность символов. Чтобы исполнить, её нужно ассемблировать — перевести на машинный язык. Ассемблирование заключается в тупом сопоставлении каждой текстовой команде её бинарного представления.

Полученная машинная программа — это последовательность битов. Это исполняемая программа: её можно загрузить в память, передать на неё управление, и процессор начнёт декодировать биты и исполнять соответствующие команды.

Можно также провести обратное преобразование — дизассемблирование, то есть перевод машинной программы в программу на ассемблере.

2.7 Итоги

Машинная команда — это набор битов, которым обозначается то или иное действие компьютера. Когда этот набор битов попадает в виде электрических сигналов на вход декодера команд, он активирует схему, отвечающую за выполнение этой команды. Множество машинных команд образуют машинный язык.

Ассемблер — это язык символьных обозначений машинных команд.

Глава 3

Переход к языкам высокого уровня

Зачем вообще нужны языки высокого уровня? Люди плохо приспособлены для машинного языка Когда им приходится одновременно держать в голове много мелочей, они:

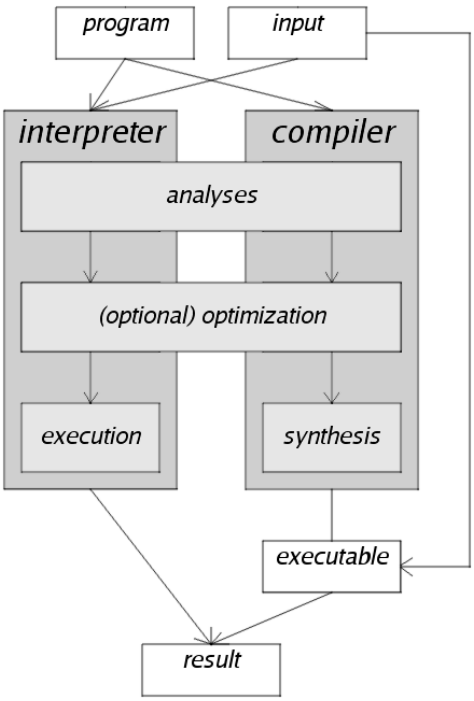
- делают много ошибок
- пишут неэффективные программы (упускают из виду высокоуровневые оптимизации)
- не могут быстро читать машинный код (и соответственно вносить изменения)

Конечно, есть исключения — это маленькие, критичные ко времени исполнения куски программы (например, тело цикла, или отдельная функция). Компилятор не может оптимизировать такой кусок лучше, чем человек. Такие куски программы иногда пишут в виде ассемблерных вставок.

Но писать большую программу на ассемблере — значит обрекать её на непортабельность, неподдерживаемость, и скорее всего некорректность и неэффективность.

Итак, языки высокого уровня нужны, а значит, нужно что-то, что умело бы исполнять программы на этих языках. Можно было бы сделать специальный процессор для высокоуровневого языка, но такой процессор был бы одновременно очень сложным и узкоспециализированным, его трудно было бы менять под нужды программистов (а нужды постоянно меняются). Поэтому высокоуровневые языки реализуются не в железе, а программно.

Есть два основных подхода к реализации языка программирования: интерпретация и компиляция.



Интерпретация состоит из анализа, (возможно) оптимизации и исполнения. Интерпретатор принимает на вход программу и входные данные для этой программы, а на выходе выдаёт результат исполнения программы.

Компиляция состоит из анализа, (возможно) оптимизации и синтеза (то есть генерации кода). Компилятор принимает на вход программу на исходном языке, а выдаёт программу на целевом языке. Целевой язык совсем не обязан быть машинным, но нас интересует именно этот случай (иначе компилятор — только звено в цепи преобразований программы на пути к её исполнению), поэтому дальше я буду подразумевать, что целевой язык — машинный.

Начальные фазы (анализ и оптимизация) — общие для интерпретаторов и компиляторов (хотя есть оптимизации, специфичные для одного из двух подходов).

Все остальные подходы — это в том или ином виде комбинация этих двух. Общего красивого названия для интерпретаторов и компиляторов я не знаю, буду называть их *языковыми процессорами*.

3.1 Анализ

Анализ программы заключается в том, что языковой процессор читает исходный код и восстанавливает по нему структуру программы (как её представляет себе человек). Получается, исходный код — это общий язык программиста и компьютера. Если бы компьютер понимал мысли программиста, анализ был бы не нужен.

Анализ программы состоит из двух этапов: синтаксический анализ и семантический анализ.

Синтаксический анализ

Синтаксический анализ — это преобразование программы-строки в программу-структуру в соответствии с синтаксической грамматикой языка.

Часто грамматика языка разделена на лексическую и синтаксическую. В этом случае лексический анализ выделяется в отдельную фазу: лексический анализатор преобразует исходный код в последовательность *токенов* — пар <тип лексемы, значение лексемы>. Синтаксический анализатор работает уже не с отдельными символами, а с токенами.

Синтаксический анализатор напрямую зависит от лексического анализатора. Обратная зависимость в некоторых случаях тоже имеет место: некоторые лексемы могут означать совер-

шенно разные вещи в зависимости от синтаксического контекста.

Лексическая грамматика обычно регулярная, синтаксическая — контекстно-свободная.

Семантический анализ

Семантический анализатор проверяет структуру, распознанную синтаксическим анализатором, на предмет семантических ошибок. По-хорошему, синтаксическая грамматика языка не должна порождать противоречивые или бессмысленные конструкции. Более того, если она их порождает, то всегда можно составить более точную грамматику. Беда в том, что точная грамматика будет слишком сложной: если язык Тьюринг-полный, то его бесконечно точная грамматика, полностью выражающая семантику, будет Тьюринг-полной (то есть неограниченной). От такой грамматики компьютер может треснуть, поэтому синтаксис приходится умышленно огрублять, а всякие тонкости спихивать на семантику.

Семантический анализ обычно размазан по разным фазам языкового процессора. Хороший пример семантического анализа — это проверка типов: в статически типизированных языках типы проверяются на этапе анализа, в динамически типизированных — на этапе исполнения.

3.2 Оптимизация

Оптимизация — это преобразование программы, которое семантически не изменяет программу, но уменьшает количество потребляемых ресурсов (время, память и т.д.). Оптимизация не делает программу оптимальной, просто делает её лучше.

Оптимизацию можно проводить на разных уровнях.

- Оптимизации программиста:
  - на уровне архитектуры программы (т.е. взаимодействия разных её компонент)
  - на уровне алгоритмов решения отдельных подзадач
  - на уровне исходного кода
- Оптимизации языкового процессора:
  - на уровне промежуточного представления
  - на уровне машинных инструкций (компиляторы)
  - на уровне входных данных (интерпретаторы)

Кроме того, оптимизации можно поделить по ширина охвата:

- глобальные — касаются всей программы
- локальные — касаются отдельной части программы

Чем оптимизация глобальнее и высокоуровневее, тем сильнее она влияет на программу.

С точки зрения програмиста, крайне важно писать программы от большого к малому: тщательно продумать архитектуру и создать рабочий *прототип* программы, потом перейти к отдельным подзадачам, и только в самом конце (когда программа уже работает) заняться мелкими оптимизациями. Прежде чем внедрять оптимизацию, нужно сначала найти проблемное место, а потом обязательно убедиться, что соответствующая оптимизация делает лучше. **Это очень важно.** Бесполезные оптимизации запутывают исходный код и вносят ошибки.

Языковой процессор знает о программе куда меньше программиста, поэтому он вообще не способен проводить высокоуровневые оптимизации. Зато он куда лучше программиста умеет ковыряться в деталях и проверять множество случаев, поэтому низкоуровневые оптимизации — его удел, и программисту стоит соваться в них только в крайнем случае. Некоторые языковые процессоры могут проводить и довольно сильные оптимизации на уровне всей программы (параллелизм, анализ потока данных).

Я не буду подробно рассказывать об оптимизациях — это отдельная серьёзная и интересная тема, и я в ней не шарю. Скажу только, что их очень много и они очень разные.

3.3 Промежуточные представления

И компиляторы, и интерпретаторы могут использовать ноль или более *промежуточных представлений*. Промежуточное представление — это некое эквивалентное представление исходной программы, в котором языковой процессор хранит программу. Промежуточные представления могут быть очень разными: абстракное синтаксическое дерево, байткод какой-нибудь виртуальной машины, более простой язык — что угодно.

В совсем простых случаях можно вообще обойтись без промежуточного представления: читать кусок за куском код программы, анализировать его и тут же исполнять или генерировать инструкции. Такой подход возможен, если в каждой конкретной точке программа зависит только от того, что было, последующие части программы не влияют на предыдущие.

В более сложных случаях промежуточное представление необходимо: исполнение/генерация кода и некоторые фазы анализа требуют информации обо всей программе. Например, во многих языках программирования объявление функции может быть отвязано от её определения, поэтому вызов функции может встретиться в программе после объявления, но до определения. Другой пример: вывод типов на основании всех известных в программе типов.

Даже в простых случаях промежуточное представление может быть очень удобно, в частности для проведения глобальных оптимизаций.

3.4 Интерпретаторы

Поскольку результат работы интерпретатора — это исполнение программы для конкретных входных данных, то результат этот “одноразовый” — его невозможно сохранить для последующего использования. Программы на интерпретируемых языках хранятся и распространяются в виде исходного кода. Каждый раз интерпретатор заново читает исходный код, анализирует

его, (возможно) проводит оптимизации и исполняет.

Большой недостаток интерпретации — медленность исполнения программы. Время исполнения программы включает время интерпретации: анализ и оптимизация каждый раз повторяются заново. Кроме того, непонятно, сколько времени надо тратить на оптимизацию: некоторые программы плохо оптимизируются — их быстрее исполнить "как есть", а иногда стоит помучиться и оптимизировать какой-нибудь горячий кусок — время, затраченное на оптимизацию, с лихвой окупится за счёт быстрого исполнения. Оптимальные затраты на оптимизацию зависят от конкретной программы, их невозможно вычислить заранее. Можно только попытаться угадать (что и делают интерпретаторы).

Большое достоинство интерпретации — портабельность. Чтобы портировать интерпретатор на какую-то архитектуру, не нужно менять ничего в самом интерпретаторе. Достаточно, чтобы язык, на котором написан интерпретатор, был портирован на эту архитектуру. Как только интерпретатор запустился на новой архитектуре, любая программа может быть исполнена на нём.

Другое достоинство интерпретации — гибкость. Во-первых, оптимизации можно проводить с учётом конкретных входных данных. Во-вторых, появляется "доступ к кишкам" интерпретатора: поскольку программу невозможно исполнить без интерпретатора, прямо из программы можно дёргать интерпретатор. Можно на ходу состряпать кусок кода на исходном языке и тут же его исполнить, то есть можно во время исполнения менять саму программу.

Наконец, ещё одно достоинство интерпретации — простота: не нужно ничего знать о целевой архитектуре.

3.5 Компиляторы

Компилятор транслирует программу на исходном языке в программу на целевом языке. (Я уже говорила, что целевой язык не обязательно машинный, но я рассматриваю компилятор как единственное и самодостаточное средство преобразования программы). Программы на компилируемых языках могут храниться и распространяться как в виде исходного кода, так и в виде целевого кода (исполняемых файлов).

Большое достоинство компиляции — быстрота исполнения программы. Исполнение отвязано от компиляции: один раз скомпилировал — исполняй сколько угодно раз, поэтому время исполнение не включает время компиляции. При компиляции можно тратить много времени на оптимизацию. Когда-нибудь потом готовая, оптимизированная до зубов программа быстренько исполнится. Такой подход порождает чёткое разделение статического и динамического в программе: всё статическое вычисляется во время компиляции и не влияет на время исполнения.

Большой недостаток компиляции — непортабельность. В отличие от интерпретатора, компилятор много возится с особенностями целевой архитектуры. Поэтому чтобы портировать его на новую архитектуру, нужно полностью переделать фазу синтеза и связанные с ней оп-

тимизации. Сколько архитектур — столько разных фаз синтеза, и за всеми нужно следить, чтобы они генерировали правильный и быстрый код.

Другой недостаток компиляции — негибкость. Один раз скомпилированная программа используется сразу для всех входных данных, не оптимизируется для каждого конкретного случая. В компилируемых языках обычно отсутствуют `eval`-оподобные возможности: исполнение программы не требует компилятора, поэтому у программы нет доступа к нему (ну разве что включать компилятор в виде библиотеки, от чего теряется весь смысл компиляции).

Ну и ещё один недостаток компиляции — сложность (связанная с поддержкой разных архитектур).

3.6 Bootstrap

И компилятор, и интерпретатор — это просто программа. Как и любая программа, она написана на каком-то языке программирования. На каком?

Если представить, что в мире нет ни одного высокоуровневого языка — то вариантов нет, любой языковой процессор должен быть написан на машинном языке. Однако как только появился хоть один высокоуровневый язык, на нём можно писать языковые процессоры для других языков. (Разумеется, язык должен быть достаточно выразительным — некоторые языки специального назначения плохо подходят для таких задач.)

Когда у языка есть хотя бы одна реализация, можно писать вторую — но уже на этом же самом языке. Язык, у которого языковой процессор написан на нём самом, называется *self-hosting*. Поначалу это сбивает с толку, но нужно помнить, что никакой магии нет: у истоков любого высокоуровневого языка стоял другой язык, а у истоков всех языков стоял машинный. Бытует мнение, что self-hosting — своего рода “тест на вшивость” языка: если язык позволяет реализовать самого себя — значит, он не так плох. Но я думаю, что дело тут в другом: разработчикам self-hosting языка постоянно приходится чувствовать себя в шкуре программистов, и они лучше видят проблемы. Сам процесс создания self-hosting языкового процессора называется *bootstrap*’ом.

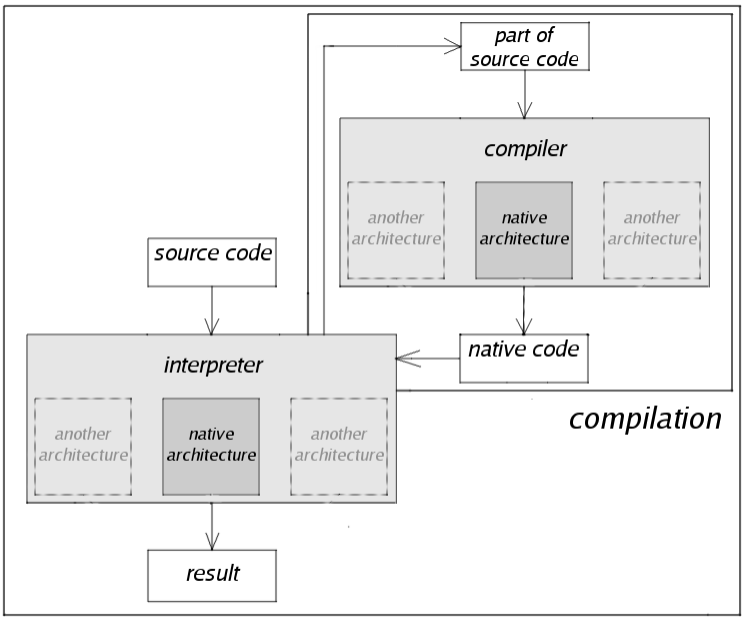
С компиляторами всё просто. Первая версия компилятора пишется на каком-то уже реализованном языке. Эта версия обычно ограниченная, поддерживает только подмножество языка, достаточное для создания второй версии компилятора. Вторая версия пишется уже на новом языке и компилируется при помощи первой. После этого первую версию можно забыть и выбросить, а компилятор с этого момента действительно самодостаточен. Примеры языков, у которых есть self-hosting компилятор: C/C++, Haskell, Rust.

С интерпретаторами всё не так. Когда написана первая версия на чужом языке, можно написать вторую версию на интерпретируемом языке. Но вот выкинуть первую версию нельзя — вторую будет просто негде запустить. Поэтому интерпретатор вынужден всегда таскать за собой самую первую версию. Можно представить себе стек версий: первая запускает вторую, вторая — третью и так далее, пока не запустится самая новая версия, которая наконец со-

изволит запустить программу. Это “малость” неэффективно и очень хрупко — наслаиваются ошибки из разных версий. Поэтому на практике изворачиваются кто как может (например, пишут компилятор для маленького подмножества языка, а интерпретатор пишут на этом подмножестве и компилируют). Примеры языков, у которых есть self-hosting интерпретатор: Basic, Lisp, Prolog, Python.

3.7 JIT-компиляторы

Грубо говоря, компиляция — быстрый, но непортабельный и негибкий подход, а интерпретация — портабельный и гибкий, но медленный. Некоторые люди готовы всем пожертвовать ради скорости (например я), некоторые предпочитают гибкость или портабельность, а некоторые попытались найти компромисс — JIT-компиляторы.



JIT (Just In Time) компиляторы — это языковые процессоры, которые компилируют программу на ходу и тут же исполняют. Обычно JIT-компилятор встроен в интерпретатор: интерпретатор на ходу определяет “горячие” куски программы, компилирует их JIT-ом и тут же исполняют. Правильнее было бы разграничивать интерпретатор и встроенный в него JIT, но применяются JIT-ы в основном как навороченная оптимизация в интерпретаторах, поэтому я под JIT-компиляцией буду подразумевать “интерпретатор + JIT”.

Основная цель JIT-компиляторов — разогнать медленный интерпретатор. Скорость исполнения программ сильно зависит от конкретной программы: если в ней много длинных циклов, JIT-компилятор может быть даже быстрее компилятора (за счёт оптимизаций, специфичных для конкретных входных данных). Если в программе нет “горячих” кусков (время исполнения равномерно размазано по всей программе) или если программа сильно меняется на ходу, то JIT-компилятор будет даже медленнее интерпретатора (за счёт дополнительных затрат на компиляцию). В целом JIT может на порядки разогнать интерпретатор для некоторых типич-

ных программ.

С портабельностью та же беда, что и с компиляторами: тяжело даётся поддержка каждой новой архитектуры. Правда, если JIT не поддерживает какую-то архитектуру, есть запасной вариант — всё исполнять на интерпретаторе. Такая, неэффективная, портабельность есть. Сами программы не нужно перекомпилировать — они распространяются в виде исходного кода и компилируются на ходу.

Гибкость интерпретаторов остаётся, но добавляется новая головная боль: если программа на ходу изменилась, то скомпилированный код мог устареть, и нужно перекомпилировать его заново.

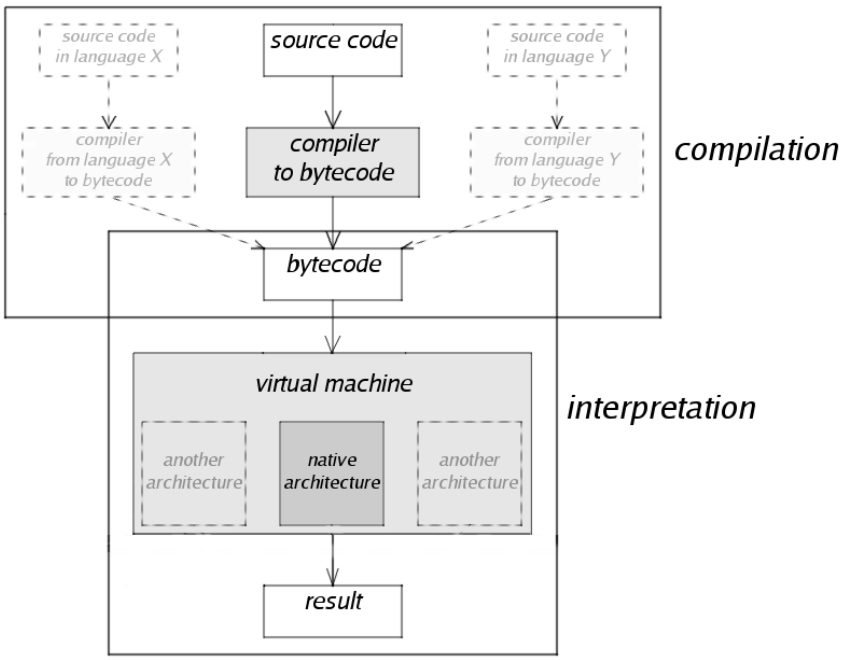
Чего точно нет в JIT-компиляторах — так это простоты. К сложности компиляции добавляется ещё и необходимость эвристически определять, какие куски программы нужно на ходу скомпилировать, и следить за изменениями программы.

Примеры JIT-компиляторов: V8 (Javascript), PyPy (Python), LuaJIT (Lua).

3.8 Виртуальные машины

Другая попытка компромисса между компиляторами и интерпретаторами — виртуальные машины. Виртуальная машина — это эмулятор одного исполнителя на другом исполнителе.

Языковой процессор разбивается на два этапа. Первый этап — это компиляция исходного кода в язык, понимаемый виртуальной машиной — *байткод*. Второй этап — это интерпретация байткода виртуальной машиной (это может быть простая интерпретация, JIT-компиляция или любая цепочка преобразований байткода, которая в конечном счёте приведёт к исполнению программы).



Во-первых, такой подход позволяет на первом этапе сделать значительную часть работы — анализ и оптимизацию исходного кода. Время исполнения программы включает только второй этап: интерпретацию байткода, а это значительно проще, чем интерпретация исходного кода.

Во-вторых, байткод виртуальной машины платформенно-независим: его можно исполнять на любой платформе, где есть эта виртуальная машина. Поэтому программы в виде байткода портабельны.

В-третьих, виртуальная машина по-прежнему может оптимизировать байткод для конкретных входных данных. С изменениями программы во время исполнения уже не так хорошо: виртуальная машина не понимает исходный код, она понимает только байткод. Поэтому программы можно менять, но на уровне байткода.

Виртуальная машина не привязана к исходному языку: она вообще ничего о нём не знает. Поэтому ничто не мешает любому языку компилироваться в байткод виртуальной машины и спокойно паразитировать на ней. Это очень полезное свойство, как для пользоваетелей виртуальной машины (они могут сэкономить усилия при создании языковых процессоров), так и для разработчиков виртуальной машины (много кто заинтересован в её качестве). Главное при этом, чтобы байткод виртуальной машины был стабильным и имел чёткую спецификацю.

Канонический пример виртуальной машины — JVM (Java Virtual Machine).

3.9 Суперкомпиляторы

Суперкомпиляция, несмотря на многообещающее название, — это вовсе не архи-хорошая-компиляция, а просто один из способов оптимизации. Суперкомпиляцию придумал В.Ф. Турчин, и вот как объясняет это слово один из его последователей, Сергей Романенко:

Сам термин "суперкомпиляция", может быть, и не очень хорош в силу своей двусмысленности. "Супер" может означать "крутой и могучий" ("супермен" = "сверхчеловек"), а может означать "тот, кто находится сверху и присматривает" ("супервизор" = "надсмотрщик"). Когда придумывался термин "суперкомпилятор" имелась в виду не "могучесть", а "присмотр"...

Я упоминаю суперкомпиляцию в основном чтобы убрать путаницу, связнную с этим понятием, хотя и сама техника стоит того, чтобы о ней знать.

Суперкомпиляция заключается примерно в следующем. Делается попытка выполнить программу, но не для конкретных данных, а для любых, абстрактных данных (или данных, удовлетворяющих специальным ограничениям). При этом строится *дерево конфигураций*, где конфигурации — это состояния программы на разных этапах исполнения. Если программа содержит циклы или рекурсию, то дерево конфигураций может получиться бесконечным. Чтобы этого избежать, в дереве ищутся повторяющиеся конфигурации и дерево свёртываются в *граф конфигураций*. По графу конфигураций восстанавливается *остаточная* программа — образ исходной программы с учётом входных данных. В остаточной программе отстутствуют бесполезные части исходной программы (недостижимые для входных данных).

Идея суперкомпиляции очень сильная, но подход не лишён практических трудностей (самая большая проблема — остаточная программа может быть слишком большой). Поэтому пока что суперкомпиляция используется редко. Были попытки писать суперкомпиляторы для разных языков: Рефал (язык, придуманный Турчином), Haskell, Scala, даже Java.

3.10 Итоги

Есть два основных подхода к реализации высокоуровневых языков программирования: интерпретация и компиляция. Основной недостаток интерпретации — медленность, компиляции — трудность портирования на разные архитектуры и невозможность менять программу на ходу (на уровне исходного кода). Есть две основных попытки компромисса между компиляцией и интерпретацией: JIT-компиляция и виртуальные машины.

В реальной жизни вид языкового процессора определяется языком. Если главная цель языка — скорость исполнения программ, то почти наверняка у него будет компилятор. Если цели другие (гибкость, возможность изменять программы на ходу) — то скорее всего понадобится интерпретатор. Часто какой-то язык начинает с медленного интерпретатора, который потом обрастает оптимизациями и превращается в JIT. Иногда встречаются настоящие кадавры, состоящие из нескольких уровней интерпретаторов, JIT-компиляторов и виртуальных машин.

Прочитай короткую захватывающую статью с картинками "The Three Projections of Doctor

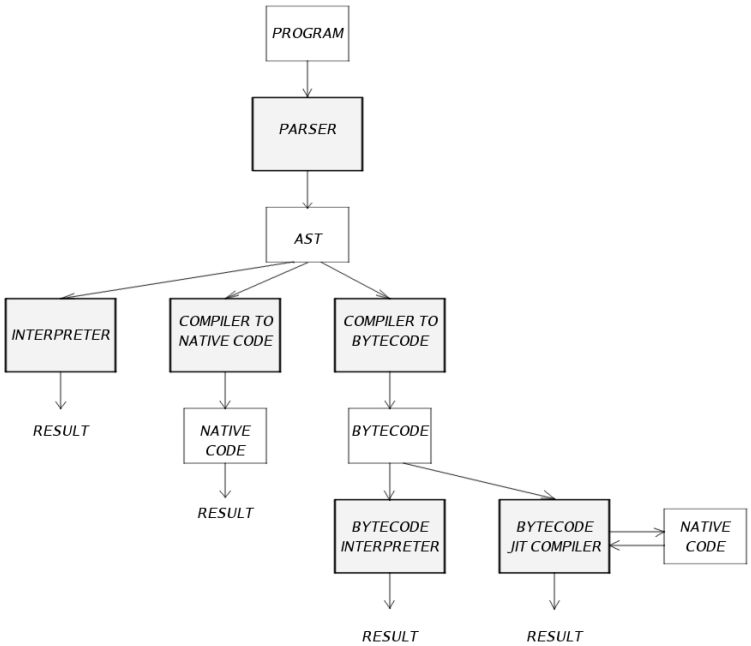
Futamura". :)

Глава 4

Детсадовский пример

Возьмём игрушечный язык и построим для него несколько примеров языковых процессоров:

1. Интерпретатор
2. Компилятор
3. Виртуальную машину
4. Интерпретатор байткода виртуальной машины
5. JIT-компилятор байткода виртуальной машины



Все языковые процессоры будут использовать общий парсер. Парсер будет генерировать AST. Каждый языковой процессор будет рекурсивно обходить AST и делать что-то своё: выполнять программу, генерировать машинные инструкции, генерировать байткод виртуальной машины. Чего в нашем примере не будет — так это оптимизаций (слишком простой язык).

4.1 Язык

Язык я выберу очень простой: арифметические операции над целыми числами в десятичном представлении. Синтаксис нашего языка можно описать такой грамматикой $G =< \Sigma_T, \Sigma_N, P, S >$:

$$\Sigma_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)\}$$

$$\Sigma_N = \{digit, number, factor, term, expr\}$$

$$P = \begin{cases} digit & \to\ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ number & \to\ digit\ number \mid digit \\ factor & \to\ number \mid (expr) \\ term & \to\ term * factor \mid term/factor \mid factor \\ expr & \to\ expr + term \mid expr - term \mid term \end{cases}$$

$$S = expr$$

Эта грамматика может показаться сложноватой. Во-первых, вместо трёх правил для нетерминалов $factor$, $term$ и $expr$ можно обойтись одним:

$$expr\ \to\ expr + expr \mid expr - expr \mid expr * expr \mid expr/expr \mid (expr) \mid number$$

Это правило позволяет описывать те же самые языковые конструкции, что и три правила $factor$, $term$ и $expr$ вместе взятые. Кроме того, оно проще и очевиднее. Зачем разбивать его на три? А вот зачем: такое разделение позволяет прямо в грамматике закодировать *приоритеты* арифметических операций. У скобок самый большой приоритет (правило $factor$); затем идут мультипликативные операции с меньшим приоритетом (правило $term$); затем аддитивные с ещё меньшим приоритетом (правило $expr$). Получается, что трёхуровневые правила описывают тот же *синтаксис*, что и одноуровневое, но при этом описывают часть *семантики* языка (приоритет арифметических операций).

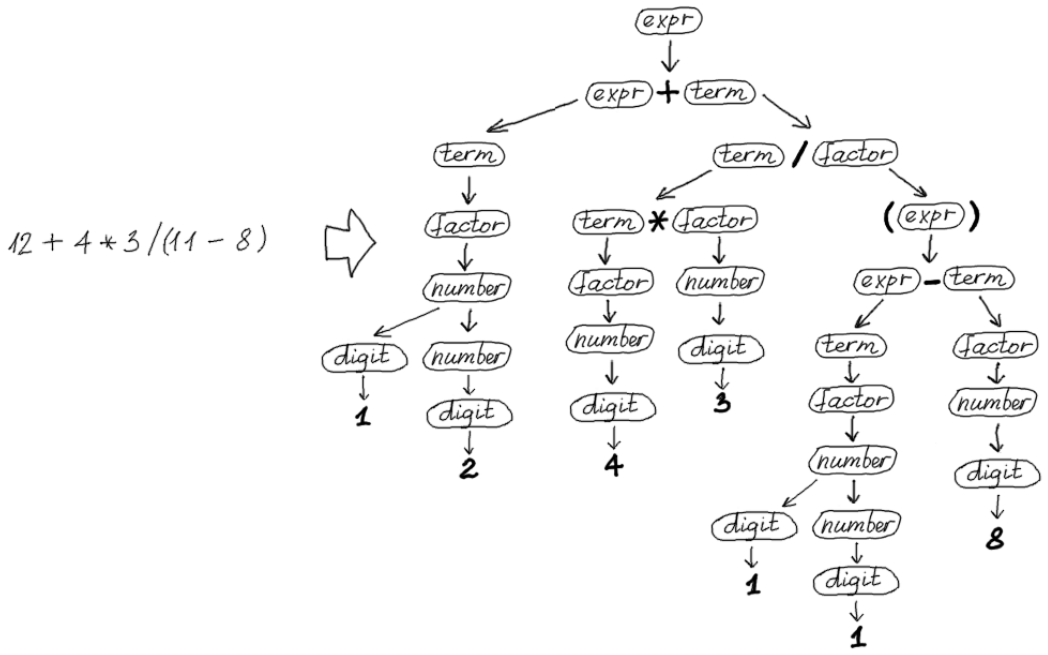
Вторая странность нашей грамматики, это то, что правила для арифметических операций имеют вид $a \to a \circ b$. Возьмём, например, правило для сложения: $expr \to expr + term$. Почему не $expr \to term + expr$? Или самый очевидный вариант: $expr \to expr + expr$? Все три варианта описывают синтаксически эквивалентные языковые конструкции. Чем первый вариант лучше? Дело тут в другой семантической особенности нашего языка: *левоассоциативность* бинарных операторов вычитания и деления. Выражение "$1 - 2 - 3$", к примеру, следует понимать как "$(1 - 2) - 3$", а не "$1 - (2 - 3)$". Рассмотрим повнимательнее правило для вычитания: $expr \to expr - term$. Слева от "$-$" стоит $expr$, т.е. любое выражение. А вот справа — $term$, т.е. выражение, допускающее только мультипликативные операции над числами и выражениями в скобках. Получается, такое правило заставляет нас однозначно понимать "$1 - 2 - 3$" как "$(1 - 2) - 3$". Если бы мы выбрали правило $expr \to term - expr$, мы бы получили *правоассоциативный* оператор вычитания (и такие операторы иногда нужны). А вот третье правило, $expr \to expr - expr$, вообще *неоднозначное*: оно позволяет понимать выражение "$1 - 2 - 3$" двумя способами. Неоднозначность в грамматиках — источних бед. Её следует допускать только тогда, когда сам по себе язык неоднозначен (как, например, русская фраза "он умеет заставить

себя слушать"). Кстати, правила для сложения и умножения могут быть как левоассоциативными, так и правоассоциативными, лишь бы не неоднозначными.

Семантика нашего языка очень проста: это семантика арифметических выражений. По-хорошему, я должна была бы формально определить её, но сделаю вид, что и так понятно. Кстати, из-за этого потом начнутся беды: я втихаря ограничу размер чисел 32-мя битами, а ты ничего не заметишь. ;)

4.2 Парсер

Раз уж мы собрались строить языковые процессоры для языка арифметических выражений, то начинать нужно с парсера. Парсер должен структурировать входную строку в соответствии с нашей грамматикой:



Дерево на картинке — это *дерево разбора*. Вообще, правильнее было бы говорить *граф разбора*, потому что этот граф является деревом только для некоторых классов грамматик. В нашем случае мы с полным правом можем говорить "дерево". Возникает два вопроса:

1. Как заставить парсер догадаться, какая стуктура соответствует строке? Явно придётся выковыривать символы, искать среди них плюсы, минусы и прочее. Но как сделать это грамотно?

2. Допустим, парсеру каким-то чудом удалось распознать в строке структуру. Что он должен делать с этим тайным знанием? Немедленно использовать и тут же забыть? Или как-то сохранить для последующего использования?

Разберёмся сначала с первым.

Один из самых простых подходов при написании парсера — это так называемый *метод рекурсивного спуска* (recursive descent). Суть метода такая: для каждого нетерминала пишется отдельная маленькая функция-парсер. Эта функция вызывает функции-парсеры для других нетерминалов (возможно, саму себя) — точно так же, как правила грамматики ссылаются друг на друга. В итоге парсер структурно повторяет грамматику. Примерно так:

```
1  import Text.Parsec ((<|>), char, parse)
2
3  expr =  (expr >> symbol '+' >> term)
4      <|> (expr >> symbol '-' >> term)
5      <|> term
6
7  term =  (term >> symbol '*' >> factor)
8      <|> (term >> symbol '/' >> factor)
9      <|> factor
10
11 factor = (symbol '(' >> expr >> symbol ')')
12     <|> number
13
14 number = (digit >> number)
15     <|> digit
16
17 digit = symbol '0'
18     <|> symbol '1'
19     <|> symbol '2'
20     <|> symbol '3'
21     <|> symbol '4'
22     <|> symbol '5'
23     <|> symbol '6'
24     <|> symbol '7'
25     <|> symbol '8'
26     <|> symbol '9'
27
28 symbol c = char c >> return ()
29
30 main = case parse expr "" "11+2*3" of
31     Left err -> print err
32     Right _  -> print "ok:)"
```

Парсер написан на языке Haskell с использованием библиотеки Parsec. Как видно из 1-й строки, мы взяли всего три библиотечных функции: `<|>`, `char` и `parse`. Остальное — встроенные хаскельные функции и операторы. Оператор `<|>` работает как логическое *или*: запускает левый парсер, и если он отработал неудачно, запускает правый парсер. Оператор `>>` работает как логическое *и*: запускает левый парсер, и если он отработал удачно, запускает правый парсер. Поскольку наш парсер пока не возвращает никакого результата (только проверяет синтаксическую корректность), то все функции-парсеры должны возвращать значение типа `()` (аналог типа `void` в C/C++). Чтобы возвращать `()`, функция должна конце вызывать `return ()` или другую функцию, которая возвращает `()`.

Метод рекурсивного спуска — это очень простой метод построения парсеров, наиболее *естественный* для человека: он позволяет писать парсер почти не задумываясь, просто глядя на грамматику. На практике у этого метода есть две основные проблемы.

Первая проблема — корректный перебор нескольких альтернатив. Если не сработал первый парсер, надо не просто запустить второй, но ещё и перед этим аккуратно откатить назад все изменения, сделанные первым (backtrack). В частности, если первый парсер продвинулся на несколько символов во входной строке, то надо вернуться на прежнюю позицию. В библиотеке Parsec для этого есть функция `try`: она запоминает состояние, к которому нужно откатиться. С учётом функции `try` парсер выглядит так:

```
import Text.Parsec ((<|>), char, parse, try)

expr =  try (expr >> symbol '+' >> term)
    <|> try (expr >> symbol '-' >> term)
    <|> term

term =  try (term >> symbol '*' >> factor)
    <|> try (term >> symbol '/' >> factor)
    <|> factor

factor = try (symbol '(' >> expr >> symbol ')')
    <|> number

number = try (digit >> number)
    <|> digit

digit = symbol '0'
    <|> symbol '1'
    <|> symbol '2'
    <|> symbol '3'
    <|> symbol '4'
    <|> symbol '5'
    <|> symbol '6'
    <|> symbol '7'
    <|> symbol '8'
    <|> symbol '9'

symbol c = char c >> return ()

main = case parse expr "" "11+2*3" of
    Left err -> print err
    Right _  -> print "ok:)"
```

Обрати внимание, что `try` нужно вставлять перед всеми альтернативами, кроме последней: если она не сработает, весь парсер не сработает. Кроме того, `try` не надо вставлять, если альтернатива проверяет не больше одного символа.

Вторая проблема — левая рекурсия в грамматике. Например, правило $expr \to expr - term$ леворекурсивное, потому что крайний левый символ правой части — это сам определяемый нетерминал. Понятно, что если функция-парсер `expr` первым делом начнёт вызывать саму себя (что она и делает), то программа в лучшем случае зациклится (а в худшем сожрёт память и упадёт). Есть два пути решения этой проблемы:

1. Перестраивать грамматику при помощи формального алгоритма избавления от левой рекурсии. Этот подход гарантирует, что порождаемый грамматикой язык не изменится,

и что левой рекурсии не будет. Из недостатков — грамматика станет не такой красивой и удобной. Зато этот алгоритм универсальный.

2. Вставлять костыли в парсер. В простых случаях удаётся “на глаз” переписать функцию-парсер так, чтобы она по смыслу делала то же самое, но не зацикливалась. В общем случае есть формальные алгоритмы, позволяющие грубой силой или хитростью ограничивать левую рекурсию, но эти алгоритмы довольно сложные и кривые.

Пойдём первым путём: применим к грамматике формальный алгоритм устранения левой рекурсии. Наш случай простой: мы имеем дело с *непосредственной* левой рекурсией, т.е. правилом вида $A \to A\alpha$. (Бывает ещё *косвенная* левая рекурсия, образованная не одним, а несколькими правилами: например $A \to B\alpha$, $B \to A\beta$.) Алгоритм избавления от непосредственной левой рекурсии очень простой: каждое леворекурсивное правило $A \to A\alpha_1|...|A\alpha_n|\beta_1|...|\beta_m$ заменяется парой правил $A \to \beta_1 A_1|...|\beta_m A_1$, $A_1 \to \alpha_1 A_1|...|\alpha_n A_1|\epsilon$, где A_1 — новый нетерминал. В нашей грамматике леворекурсивных правила два: $term$ и $expr$. Изменённая грамматика имеет вид:

$$\Sigma_T = \{0,1,2,3,4,5,6,7,8,9,+,-,*,/,(,)\}$$

$$\Sigma_N = \{digit, number, factor, term, term_1, expr, expr_1\}$$

$$P = \begin{cases} digit & \to\ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ number & \to\ digit\ number \mid digit \\ factor & \to\ number \mid (expr) \\ term & \to\ factor\ term_1 \\ term_1 & \to\ *factor\ term_1 \mid /factor\ term_1 \mid \epsilon \\ expr & \to\ term\ expr_1 \\ expr_1 & \to\ +term\ expr_1 \mid\ -term\ expr_1 \mid \epsilon \end{cases}$$

$$S = expr$$

Алгоритм гарантирует, что новая грамматика порождает в точности тот же язык, что и старая (в том числе сохранилась семантика приоритетов и ассоциативности). Парсер для новой грамматики отличается от старого парсера *точно так же*, как новая грамматика отличается от старой:

```
1  import Text.Parsec ((<|>), char, parse, try)
2
3  expr = term >> expr1
4
5  expr1 = try (symbol '+' >> term >> expr1)
6      <|> try (symbol '-' >> term >> expr1)
7      <|> return ()
8
9  term = factor >> term1
10
11 term1 = try (symbol '*' >> factor >> term1)
12     <|> try (symbol '/' >> factor >> term1)
13     <|> return ()
14
15 factor = try (symbol '(' >> expr >> symbol ')')
```

```
16      <|> number
17
18  number = try (digit >> number)
19      <|> digit
20
21  digit = symbol '0'
22      <|> symbol '1'
23      <|> symbol '2'
24      <|> symbol '3'
25      <|> symbol '4'
26      <|> symbol '5'
27      <|> symbol '6'
28      <|> symbol '7'
29      <|> symbol '8'
30      <|> symbol '9'
31
32  symbol c = char c >> return ()
33
34  main = case parse expr "" "11+2*3" of
35      Left err -> print err
36      Right _  -> print "ok:)"
```

Как хорошо! Все изменения сделаны автоматически, поэтому ошибки маловероятны. Если б мы попытались переписать парсер руками, пришлось бы думать и хитрить, и не было бы этой замечательной уверенности, что всё сделано правильно. Мы сохранили главное: близость парсера к грамматике. До тех пор, пока парсер соответствует грамматике, он остаётся *понятным* и его *легко менять*, изменяя грамматику. Но как только парсер отдалится от грамматики, он навсегда погрязнет в болоте сомнительных костылей и чудом работающих хаков.

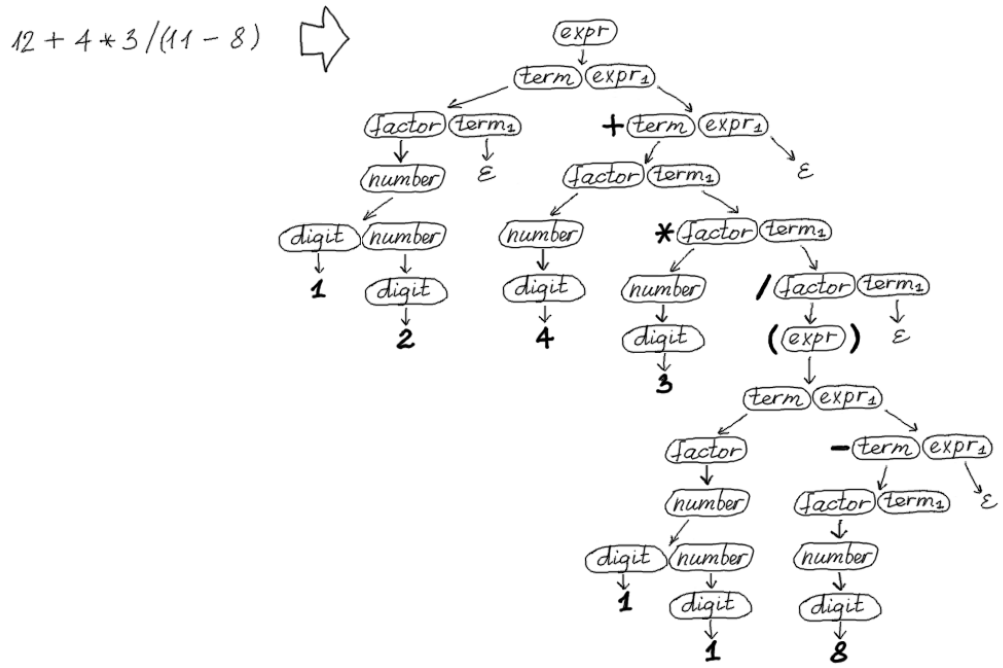
На этом с синтаксическим разбором всё: у нас есть алгоритм написания парсера (метод рекурсивного спуска) и грамматика, хорошо подходящая для этого алгоритма. Дальше по этой теме читай в книжке Grune, Jacobs: "Parsing Technique. A Practical Guide" — там приводится полная классификация алгоритмов синтаксического разбора со всеми их слабыми и сильными местами.

Теперь вернёмся ко второму вопросу: в каком виде парсер должен возвращать результат? Очевидно, это зависит от того, что мы понимаем под результатом. Если нужно просто проверить синтаксическую корректность выражения, можно ничего не возвращать. Если мы хотим *вычислить* значение выражения, то придётся следить за частично вычисленным значением. Если мы хотим *скомпилировать* выражение в программу на другом языке (или в байткод виртуальной машины), нужно таскать за собой буфер и дописывать туда новые инструкции. В общем, в каждом конкретном случае нужно что-то своё.

Можно написать по отдельному парсеру на каждый случай. Но это немного обидно: самая сложная часть парсера (синтаксический разбор) будет везде повторяться. Чтобы этого избежать, мы введём промежуточное представление: AST. Парсер будет строить AST, а потом уже с этим AST можно делать всё, что угодно: вычислять значение выражения, генерировать инструкции, проводить сложные оптимизации и т.д. (Важное замечание: поскольку наш язык очень простой, он допускает *прямую* интерпретацию и компиляцию: прямо по ходу синтаксического разбора, из парсера, можно сделать всё, что надо. Промежуточное представление

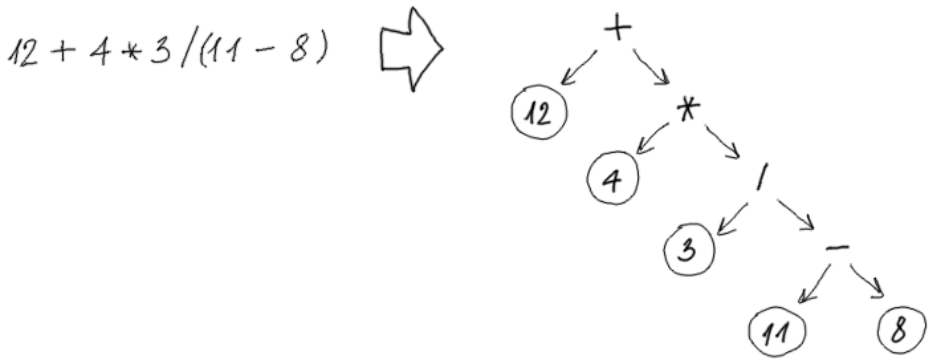
для нас — просто удобный способ не дублировать код. В более сложных случаях без него не обойтись.)

Какой должна быть структура AST для нашего языка? Самый простой вариант — дерево разбора. В начале главы мы уже смотрели на дерево разбора для выражения $12 + 4 * 3/(11 - 8)$. Вот как оно выглядит после избавления от левой рекурсии в грамматике:



Дерево разбора представляет структуру арифметического выражения в точном соответствии с грамматикой. Это дерево хорошее и правильное, но какое-то оно *слишком детальное*. Например, чтобы добраться до числа 12, надо обойти вершины *expr*, *term* $expr_1$, $factor\ term_1$ и *number*, а потом ещё по кускам собирать само число из цифр, сидящих в вершинах *digit*. Зачем нам хранить в AST информацию, что число 12 получилось таким хитрым путём? Ненужная детализация — это источник бед. Во-первых, она дурит голову (например, я размазываю простенький пример на десять страниц, и ты не улавливаешь сути). Во-вторых, все эти детали надо где-то хранить (для больших выражений потребуется уйма памяти). В-третьих, обходя огромное дерево, больше шансов где-то сделать ошибку.

Человек представляет себе выражение $12 + 4 * 3/(11 - 8)$ вот так:



Это дерево отражает структуру выражения ничуть не хуже, чем дерево разбора, но при этом оно намного проще и компактнее. Почему мы не можем сделать дерево разбора таким же простым? Дерево разбора зависит только от грамматики. Чтобы дерево разбора было красивым, нужна другая, красивая грамматика. К несчастью, красивые грамматики обычно напичканы неоднозначностями, и парсер для них написать очень трудно или вообще невозможно (кроме того, такие парсеры работают очень медленно). Наша грамматика некрасивая, потому что она приспособлена для компьютера, а не для человека. Но никто не мешает нам сделать AST не таким, как дерево разбора.

Итак, в нашем AST будут узлы пяти типов: “число”, “плюс”, “минус”, “умножить”, “поделить”. Узел типа “число” всегда будет листом, т.е. у него не будет детей. У остальных типов узлов будет ровно по два ребёнка: левый и правый операнды. Узлы-операнды могут быть числом (если тип узла - “число”) или корнем поддерева (если тип узла — “плюс”, “минус”, “умножить” или “поделить”). На хаскеле такой тип данных выражается проще некуда:

```
data AST
    = Number Int
    | Add AST AST
    | Sub AST AST
    | Mul AST AST
    | Div AST AST
```

Здесь `AST` — тип данных, который мы определяем; `Int` — встроенный хаскельный тип; `Number`, `Add`, `Sub`, `Mul` и `Div` — функции-конструкторы нашего типа `AST`. Конструктор может принимать аргументы, как и любая функция: в нашем случае конструктор `Number` — это функция, принимающая один аргумент типа `Int`, а конструкторы `Add`, `Sub`, `Mul` и `Div` — функции, принимающие по два аргумента типа `AST`. Определения для этих функций писать не надо: они настолько очевидные, что компилятор сгенерирует их сам. Теперь сделаем так, чтоб наш парсер сохранял результаты разбора в структуру AST:

```
1  import Text.Parsec ((<|>), char, parse, try)
2
3  data AST
4      = Number Int
5      | Add AST AST
6      | Sub AST AST
7      | Mul AST AST
```

```
8       | Div AST AST
9       deriving (Show)
10
11  expr = term >>= expr1
12
13  expr1 n1 =
14          try (char '+' >> term >>= \ n2 -> expr1 (Add n1 n2))
15      <|> try (char '-' >> term >>= \ n2 -> expr1 (Sub n1 n2))
16      <|> return n1
17
18  term = factor >>= term1
19
20  term1 n1 =
21          try (char '*' >> factor >>= \ n2 -> term1 (Mul n1 n2))
22      <|> try (char '/' >> factor >>= \ n2 -> term1 (Div n1 n2))
23      <|> return n1
24
25  factor = try (char '(' >> expr >>= \ n -> char ')' >> return n)
26      <|> (number 0 >>= \ i -> return (Number i))
27
28  number i = try (digit >>= \ d -> number (i * 10 + d))
29      <|> (digit >>= \ d -> return (i * 10 + d))
30
31  digit = (char '0' >> return 0)
32      <|> (char '1' >> return 1)
33      <|> (char '2' >> return 2)
34      <|> (char '3' >> return 3)
35      <|> (char '4' >> return 4)
36      <|> (char '5' >> return 5)
37      <|> (char '6' >> return 6)
38      <|> (char '7' >> return 7)
39      <|> (char '8' >> return 8)
40      <|> (char '9' >> return 9)
41
42  main = case parse expr "" "11+2*3/(4-567)" of
43      Left err  -> print err
44      Right ast -> print ast
```

Раньше нас не интересовал результат работы парсера: всё наши функции-парсеры возвращали значение типа `()`. Теперь каждая функция-парсер возвращает результат своей работы: функции `digit` и `number` возвращают результат типа `Int`, а остальные — результат типа `AST`. Функции `expr1` и `term1` теперь принимают параметр типа `AST` — левый операнд. Затем они пытаются распарсить правый операнд, и если это удаётся, то из двух операндов строится новый `AST`, а если нет — возвращается левый операнд. Функция `number` теперь принимает один аргумент типа `Int` — частично вычисленное число.

В тех местах, где возвращаемое из парсера значение важно, вместо оператора `>>` используется очень похожий оператор `>>=`. Отличие этих двух операторов в том, что `>>` игнорирует значение, возвращаемое первым парсером, а `>>=` передаёт это значение второму парсеру в качестве аргумента. Иногда надо не просто передать значение второму парсеру, а как-то более хитро его использовать. Чтобы как-то назвать, *обозначить* это возвращаемое значение, используется синтаксис *лямбда-функции*: `\<arguments> -> <calculations with arguments>`.

`deriving (Show)` (строка 9) говорит компилятору, что надо сгенерировать функции для вывода значений типа `AST` на экран. Без этого мы не смогли бы распечатать полученный `AST` (строка 44).

Если что-то непонятно в этом исходнике — не переживай. Здесь уже слишком много хаскельного сисинтакса, чтобы всё сразу было понятно. Я привожу примеры на хаскеле по двум причинам: во-первых, потому что на хаскеле они короткие и способствуют пониманию алгоритма; во-вторых, чтоб заинтересовать тебя.

Теперь напишем тот же самый парсер на C++. Вот как выглядит структура `AST` (файл ast.h):

```
1  #ifndef __AST__
2  #define __AST__
3
4  struct AST
5  {
6      enum ASTType
7      {
8          NUMBER,
9          ADD,
10         SUB,
11         MUL,
12         DIV
13     } type;
14     union
15     {
16         int number;
17         struct
18         {
19             AST * left;
20             AST * right;
21         } operands;
22     } value;
23
24     AST (ASTType t, AST * l, AST * r);
25     AST (int n);
26     ~AST ();
27 };
28
29 #endif // __AST__
```

В файле ast.cpp находится реализация конструкторов и деструкторов `AST`:

```
1  #include "ast.h"
2  #include "parser.h"
3
4  AST::AST (ASTType t, AST * l, AST * r)
5      : type (t)
6      , value ()
7  {
8      value.operands.left = l;
9      value.operands.right = r;
10 }
11
12 AST::AST (int n)
13     : type (NUMBER)
```

```
    , value ()
{
    value.number = n;
}

AST::~AST ()
{
    if (type != NUMBER)
    {
        delete value.operands.left;
        delete value.operands.right;
    }
}
```

Теперь перейдём к самому парсеру. По сути он мало отличается от хаскельного парсера, хотя и не такой элегантный. На C++ нам надо вручную следить за продвижением указателя во входной строке, поэтому мы передаём его первым аргументом во все функции (в хаскеле это неявно делала библиотека Parsec). Обрати внимание, что указатель везде передаётся *по ссылке*, поэтому все изменения, которые делает с ним *вызываемая* функция, становятся видны *вызывающей* функции. Вот прототипы функций-парсеров (parser.h):

```
#ifndef __PARSER__
#define __PARSER__

#include "ast.h"

AST * expr (const char * & s);
AST * expr1 (const char * & s, AST * l);
AST * term (const char * & s);
AST * term1 (const char * & s, AST * l);
AST * factor (const char * & s);
unsigned int number (const char * & s, unsigned int n);
unsigned char digit (const char * & s);

#endif // __PARSER__
```

А вот их реализации (parser.cpp):

```
#include "parser.h"

AST * expr (const char * & s)
{
    AST * l = term (s);
    return expr1 (s, l);
}

AST * expr1 (const char * & s, AST * l)
{
    switch (*s)
    {
        case '+':
        {
            AST * r = term (++s);
            l = new AST (AST::ADD, l, r);
            return expr1 (s, l);
        }
        case '-':
```

```
        {
            AST * r = term (++s);
            l = new AST (AST::SUB, l, r);
            return expr1 (s, l);
        }
        default:
            return l;
    }
}

AST * term (const char * & s)
{
    AST * l = factor (s);
    return term1 (s, l);
}

AST * term1 (const char * & s, AST * l)
{
    switch (*s)
    {
        case '*':
        {
            AST * r = factor (++s);
            l = new AST (AST::MUL, l, r);
            return term1 (s, l);
        }
        case '/':
        {
            AST * r = factor (++s);
            l = new AST (AST::DIV, l, r);
            return term1 (s, l);
        }
        default:
            return l;
    }
}

AST * factor (const char * & s)
{
    switch (*s)
    {
        case '(':
        {
            AST * e = expr (++s);
            ++s; // skip ')'
            return e;
        }
        default:
            return new AST (number (s, 0));
    }
}

unsigned int number (const char * & s, unsigned int n)
{
    unsigned char d = digit (s);
    if (d == 0xFF)
```

```
76          return n;
77      else
78          return number (s, n * 10 + d);
79  }
80
81  unsigned char digit (const char * & s)
82  {
83      switch (*s)
84      {
85          case '0': ++s; return 0;
86          case '1': ++s; return 1;
87          case '2': ++s; return 2;
88          case '3': ++s; return 3;
89          case '4': ++s; return 4;
90          case '5': ++s; return 5;
91          case '6': ++s; return 6;
92          case '7': ++s; return 7;
93          case '8': ++s; return 8;
94          case '9': ++s; return 9;
95          default:  return 0xFF;
96      }
97  }
```

Вот и всё! Парсер готов. Он, конечно, далеко не идеален (в частности, плохо обрабатывает ошибки). Но больше мы его менять не будем.

4.3 Интерпретатор

Интерпретатор выглядит до смешного просто. Вот его прототип (файл interpreter.h):

```
1  #include "parser.h"
2
3  int interpret (AST * ast);
```

А вот реализация (файл interpreter.cpp):

```
1  #include "interpreter.h"
2
3  int interpret (AST * ast)
4  {
5      switch (ast->type)
6      {
7          case AST::NUMBER:
8              return ast->value.number;
9          case AST::ADD:
10         {
11             int l = interpret (ast->value.operands.left);
12             int r = interpret (ast->value.operands.right);
13             return l + r;
14         }
15         case AST::SUB:
16         {
17             int l = interpret (ast->value.operands.left);
18             int r = interpret (ast->value.operands.right);
19             return l - r;
20         }
```

```
21          case AST::MUL:
22          {
23              int l = interpret (ast->value.operands.left);
24              int r = interpret (ast->value.operands.right);
25              return l * r;
26          }
27          case AST::DIV:
28          {
29              int l = interpret (ast->value.operands.left);
30              int r = interpret (ast->value.operands.right);
31              return l / r;
32          }
33      }
34  }
```

Функция `interpret` определяет тип узла: если это `NUMBER`, то нужно просто вернуть соответствующее число; иначе нужно рекурсивно вычислить значение левого и правого операндов и вернуть результат операции над ними.

4.4 Компилятор

Компилятор структурно не сложнее интерпретатора: он точно так же рекурсивно обходит `AST`, только вместо немедленного вычисления генерирует машинные инструкции. Первым делом надо решить важный вопрос: для какой архитектуры?

Архитектура компьютера — понятие широкое. Нас интересует *архитектура как набор команд процессора* (Instruction Set Architecture, ISA). Можно сделать два совершенно разных процессора, которые будут понимать один и тот же набор команд. Компилятор не увидит разницы меду ними: ему всё равно, как процессор устроен физически. Единственное, что компилятору надо знать — это какие команды понимает процессор и насколько эффективно он выполняет конкретную команду. Поэтому, говоря об архитектуре, я имею в виду набор команд процессора.

Какую архитектуру нам выбрать? В принципе, *любую*. Можно, к примеру, выбрать исторически первый компьютер и компилировать в машинный код для него. Только вот запустить такую программу будет негде, поэтому лучше выбрать архитектуру твоего компьютера. Кстати, что это за архитектура?

Это можно узнать несколькими способами. Например, поискать в интернетах спецификацию этой модели (Lenovo ThinkPad X220i). Привожу её тут полностью (вдруг что полезное узнаешь про свой комп), но нас интересует только секция “Processor / Chipset”:

```
Lenovo ThinkPad X220i

    Processor / Chipset
        CPU                             Intel Core i3 (2nd Gen) 2310M / 2.1 GHz
        Number of Cores                 Dual-Core
        Cache                           L3 - 3 MB
        64-bit Computing                Yes
        Chipset                         Mobile Intel QM67 Express
        Features                        Intel Turbo Boost Technology 2.0

    Memory
```

Max RAM Supported	8 GB
Technology	DDR3 SDRAM
Speed	1333 MHz / PC3-10600
Form Factor	SO DIMM 204-pin
Slots Qty	2
Empty Slots	1
Storage	
Interface	Serial ATA-300
Display	
LCD Backlight Technology	LED backlight
Widescreen	Yes
Image Aspect Ratio	16:9
Features	anti-glare
Audio & Video	
Graphics Processor	Intel HD Graphics 3000
Memory Allocation Technology	Dynamic Video Memory Technology
Camera	Yes
Resolution	0.92 Megapixel
Capture Resolutions	1280 x 720
Sound	Stereo speakers , stereo microphone
Codec	CX20672
Compliant Standards	High Definition Audio
Input	
Type	touch-screen
Features	spill-resistant
Communications	
Wireless	Bluetooth 3.0
Wireless Controller	Intel Centrino Wireless-N 1000
Network Interface	Gigabit Ethernet
Wireless Broadband (WWAN)	
Generation	3G upgradable
Battery	
Installed Qty	1
Max Supported Qty	2
Run Time	8.8 sec
AC Adapter	
Input	AC 120/230 V (50/60 Hz)
Output	65 Watt
Connections & Expansion	
Slots	1 x ExpressCard/54 (1 free)
Interfaces	2 x USB 2.0 PoweredUSB VGA DisplayPort LAN Headphone/microphone combo jack Dock
Memory Card Reader	3 in 1 (SD Card, SDHC Card, SDXC Card)
Software	
Software Included	ThinkVantage Toolbox
Microsoft Office Preloaded	Includes a pre-loaded image of select Microsoft Office suites. Purchase an Office 2010 Product Key Card or disc to activate preloaded software on this PC.
Miscellaneous	
Security	Trusted Platform Module (TPM 1.2) Security Chip, fingerprint reader
Features	Intel Active Management Technology (iAMT)
Compliant Standards	RoHS
Manufacturer Selling Program	TopSeller
Dimensions & Weight	
Width	12 in
Depth	9 in
Height	1.2 in
Weight	3.7 lbs
Environmental Standards	
ENERGY STAR Qualified	Yes
Sustainability	
ENERGY STAR Qualified	Yes
Greenpeace policy rating (Nov 2011)	3.8
General	
Manufacturer	Lenovo

Видно, что процессор у тебя — Intel Core i3, двухъядерный и 64-разрядный. Описание архитектуры этого процессора найти несложно.

Другой, линуксоспецифичный, способ — выполнить одну из команд `arch`, `uname -m`, `lscpu`, `cat /proc/cpuinfo` или ещё какую-нибудь. Команды `arch` и `uname -m` просто бесхитростно пишут название архитектуры (эти команды идентичны). Команда `lscpu` выводит больше информации о процессоре, а команда `cat /proc/cpuinfo` выводит информацию про каждое ядро в отдельности.

Но что, если ты не знаешь модель (нашла компьютер в канаве), и операционная система на нём очень глупая? Можно попробовать расковырять исполняемый файл какой-нибудь программы, которая запускается на этом компьютере. Попытаться понять, что там за инструкции (не самый лёгкий способ). Можно пытаться запустить программы для разных архитектур (пока компьютер не взорвётся). Если компьютер после этих экспериментов больше не включается, можно расковырять его внутренности. В общем, есть способы.

У меня есть основания предполагать, что архитектура твоего компьютера — x86. Как и моего, и вообще, большинства ноутбуков и десктопов, которые ты видишь. Про эту архитектуру написано много, и я только вкратце упомяну только самое основное.

Программа — это последовательность команд процессору. Команды в основном работают с какими-то данными: они могут принимать на вход *операнды* и возвращать *результат*. Все эти данные надо где-то хранить. С точки зрения программы есть три вида памяти:

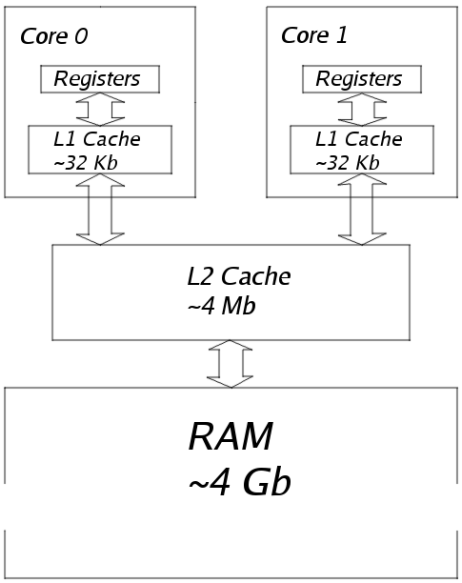
1. **Регистры**
   Это маленький кусок сверхбыстрой памяти, зашитой внутри процессора. Регистры нужны, чтобы не тратить время на доступ к оперативной памяти: загрузка/выгрузка данных выполняется значительно дольше, чем сама команда. Регистры активно используются для хранения операндов и результатов команд. Данные в них не задерживаются: они храняться там временно, чтобы вскоре быть использованными и затёртыми новыми данными. Поскольку регистров мало, у каждого из них есть своё имя, и в командах регистры адресуются всего несколькими битами.

2. **Оперативная память**
   Оперативная память — это большой кусок медленной памяти, внешней по отношению к процессору. В оперативной памяти хранится сама программа и все данные, с которыми она работает. Когда очередной кусок данных нужен для вычислений на процессоре, этот кусок загружается из оперативной памяти в регистры. Окончательные результаты вычислений выгружаются назад в оперативную память. Промежуточные результаты, если они не помещаются в регистры, тоже выгружаются в оперативную память. Оперативная память — это память с *произвольным доступом* (Random Access Memory, RAM). Это значит, что одновременно доступны все байты в этой памяти. Адресом байта служит его порядковый номер. Поскольку команда может обратиться к любому байту, адрес должен быть целиком зашит в команду.

3. **Стек**
   Стек — это кусок оперативной памяти со специальными правилами доступа. Как и опе-

ративная память, стек медленный. Размер стека может быть разным: программа может ограничить себе стек или вообще отказаться от него. Обычно максимальный размер стека ограничен операционной системой. Доступ к стеку осуществляется по правилу LIFO (Last In, First Out): одновременно доступно только одно значение (вершина стека). Адрес вершины стека хранится в специальном регистре, и команды, работающие со стеком, используют этот регистр. Стек нужен для удобства программистов.

Физически иерархия памяти устроена более сложно: между регистровой и оперативной памятью есть ещё несколько уровней **кэш-памяти**. В кэш загружаются те данные, которые не попали в регистры, но предположительно скоро понадобятся. Когда нужны какие-то данные из оперативной памяти, первым делом они ищутся в кэше: если они там есть (cache hit) — отлично, если нет (cache miss) — плохо, придётся тратить время на доступ к памяти. Программист не управляет кэшами напрямую, как регистрами и оперативной памятью (и может вообще не знать об их существовании). Есть команды, позволяющие влиять на кэширование данных, но в общем процессор оставляет за собой свободу действий.



Картинка примерно соответствуют семейству процессоров Intel Core (подробнее о кэшах в мануале “Intel 64 and IA-32 Architectures Optimization Reference Manual”).

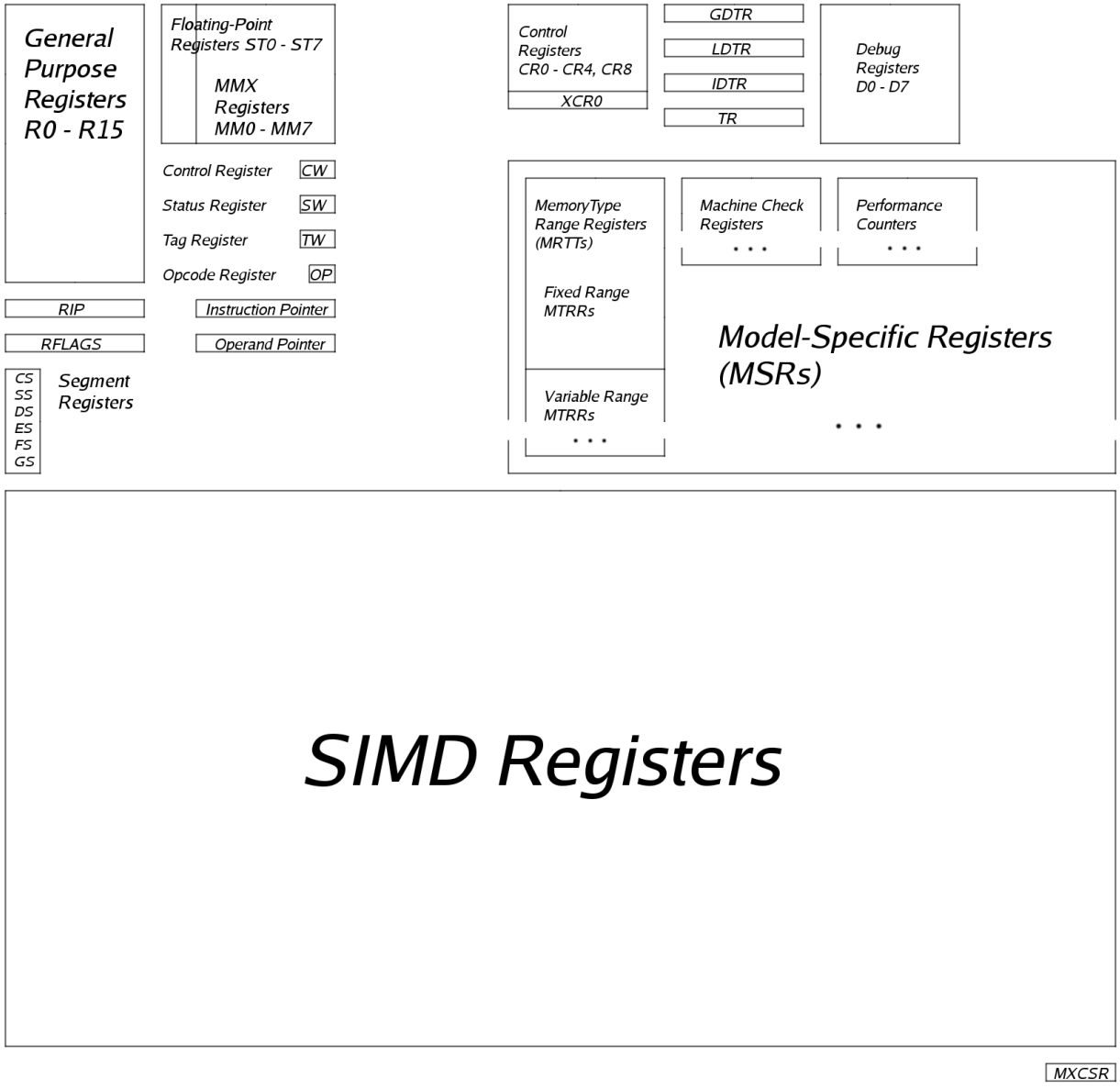
Регистры не безразмерные: каждый регистр рассчитан на хранение определённого числа битов. В N-битном регистре можно сохранить 2^N различных значений. Это могут быть целые беззнаковые числа в диапазоне $[0, 2^N - 1]$, или целые числа со знаком в диапазоне $[-2^{N-1}, 2^{N-1} - 1]$, или адреса байтов в памяти объёмом 2^N байт, или что угодно ещё. Важно, как понимает эти биты команда, которая работает с регистром. Размер регистров ограничивает диапазон данных, с которыми работает процессор: данные, которые не влазят в этот диапазон, приходится обрабатывать по кускам. Число битов в единице данных, с которой работает процессор, определяет его **разрядность**.

Как и с памятью, физическое устройство регистров может сильно отличаться от логического: один логический регистр может состоять из нескольких физических или занимать только часть физического. Некоторые логические регистры вообще не являются регистрами в физическом смысле (скорее набором управляющих сигналов). В этом смысле разрядность процессора — понятие сложное и неоднозначное. Команды работают с логическими регистрами, поэтому нам, как программистам, важно логическое устройство процессора.

Архитектура x86 начиналась с 16-разрядного процессора 8086, сделанного компанией Intel в 1978 году. Постепенно в неё добавлялись новые возможности: увеличивался размер регистров, появлялись новые регистры, расширялся набор команд. При этом сохранялось важное

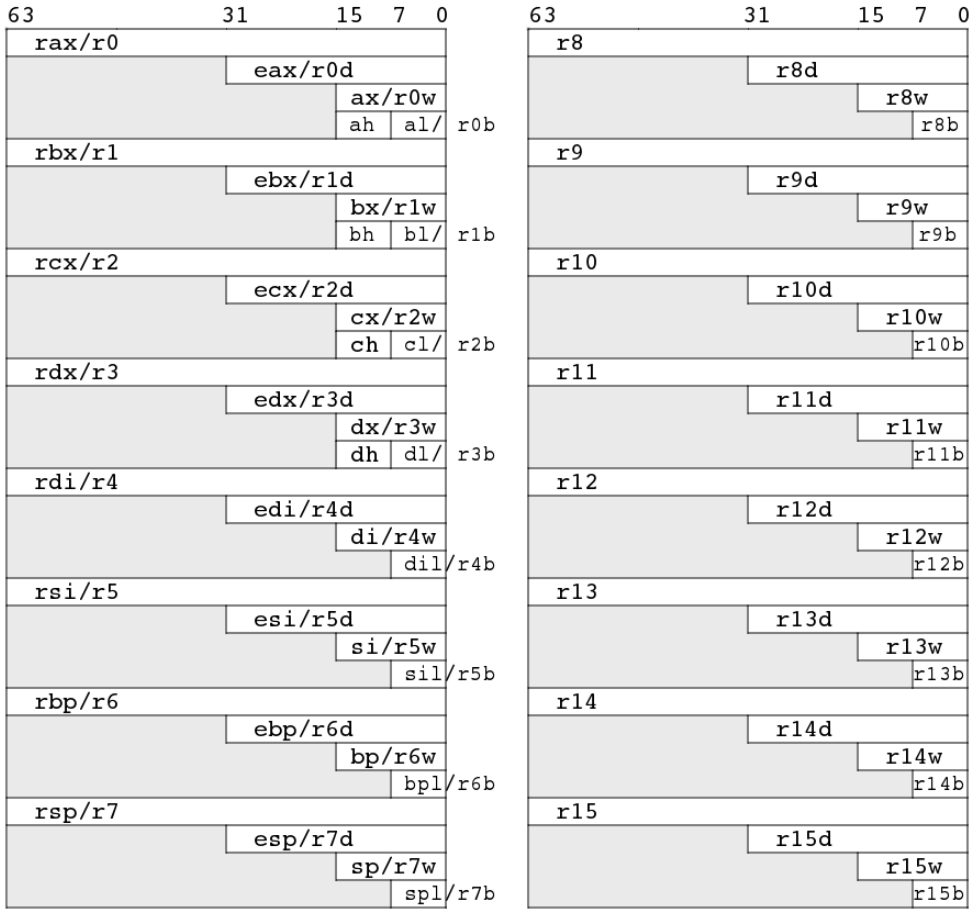
свойство: более новые архитектуры x86 были *обратно совместимы* с предыдущими (то есть понимали программы для предыдущих). Обратная совместимость — очень полезное свойство: не нужно переписывать старые программы, чтобы запустить их на новом процессоре. С другой стороны, из-за обратной совместимости приходится таскать за собой много ненужного устаревшего хлама, что значительно усложняет и замедляет процессор. У твоего ноутбука 64-разрядная архитектура x86, сокращённо x86-64.

Примерно так выглядят регистры процессора x86-64:



На этой картинке сохранён относительный размер регистров (кроме MSR-ов, размер и число которых зависит от конкретной модели процессора). Итого есть следующие важные группы регистров:

- Регистры общего назначения (General Purpose Registers, GPRs). Это основные регистры, с которыми работает программист: в них хранятся операнды, результаты команд и вообще всё, что угодно. Их шестнадцать штук, каждый по 64 бита:

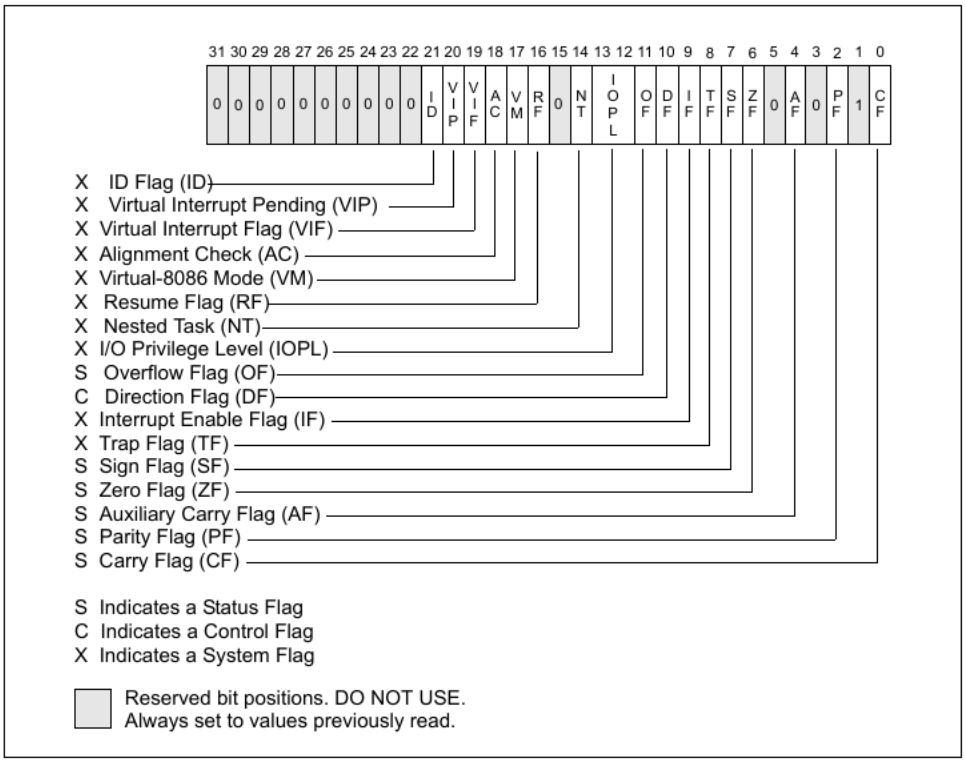


i-й регистр называется R_i, его младшая 32-битная половина — R_iD (Double Word). младшая 16-битная четверть — R_iW (Word), младшая 8-битная восьмушка — R_iB (Byte). Исторически сложилось так, что у первых восьми регистров есть личные имена: RAX, RBX, RCX, RDX, RDI, RSI, RBP и RSP (32-битные половины называются EAX, EBX, ECX, EDX, EDI, ESI, EBP и ESP, 16-битные четверти — AX, BX, CX, DX, DI, SI, BP и SP, а 8-битные восьмушки — AL, BL, CL, DL, DIL, SIL, BPL и SPL). Кроме того, старшие байты регистров AX, BX, CX и DX тоже имеют личные имены: AH, BH, CH и DH. Имена отражают специфику использования каждого из регистров: 'A' — accumulator, 'B' — base, 'C' — counter, 'D' — data, 'DI' — destination index, 'SI' — source index, 'BP' — base pointer, 'SP' — stack pointer. Это не значит, что эти регистры обязательно использовать таким образом, просто некоторые команды придают им специальный смысл.

- Регистр, хранящий адрес следующей команды (Instruction Pointer, RIP). Это 64-битный регистр. Хранящийся в нём 64-битный адрес — это номер байта в оперативной памяти, с которого начинается следующая инструкция. Процессор исполняет инструкцию и продвигает указатель в регистре RIP. Программист не может изменять значение регистра

RIP напрямую: он должен вызвать специальную инструкцию передачи управления (`JMP`, `Jcc`, `CALL`, `RET` и `IRET`). Исполняя эту инструкцию, процессор сам передвинет указатель куда надо.

- Регистр флагов (RFLAGS) — специальный регистр, содержащий набор флагов. Каждый флаг занимает один-два бита и имеет какой-то специальный смысл. Старшая половина битов зарезервирована (всегда нулевая). Младшая половина (EFLAGS) выглядит так:



- Сегментные регистры (CS, DS, SS, ES, FS, GS) — регистры, содержащие адреса *сегментов* памяти. Физически оперативная память — это просто массив байтов, и адрес в памяти — это номер байта. Логически может быть удобно представлять память как-то по-другому (или вообще использовать разные логические представления в зависимости от случая). Одно из таких логических представлений — *сегментная модель памяти*. В этой модели адресное пространство процесса состоит из кусков — сегментов. Разные сегменты соответствуют разным областям физической памяти (которые могут перекрываться и совпадать). Логический адрес, с которым работает программа, состоит из двух частей: *селектора* и *смещения*. Селектор отвечает за сегмент, к которому относится адрес, а смещение — за конкретный байт внутри сегмента. Логический адрес преобразуется в физический по хитрым правилам.

  Сегментная модель позволяет много вариаций в зависимости от числа и взаимного расположения сегментов. Самая простая модель — плоская (flat memory model): адресное пространство процесса расположено в одном сегменте. Плоская модель позволяет забыть о сегментах и работать так, как будто их вообще нет. Это вырожденный случай сегмен-

тирования, но сейчас в основном используется именно плоская модель. Сегментирование было очень кстати в 16-разрядных процессорах, поскольку в помощью 16-битного адреса можно адресовать всего лишь 2^{16} байта (то есть объём памяти, доступной программе — всего 64 Кб). Использование разных сегментов позволило расширить объём доступной памяти. С переходом на 64-разрядную архитектуру необходимость в разных сегментах отпала: 64 бита позволяют адресовать 2^{64} байта, и пока что используется только часть этих битов.

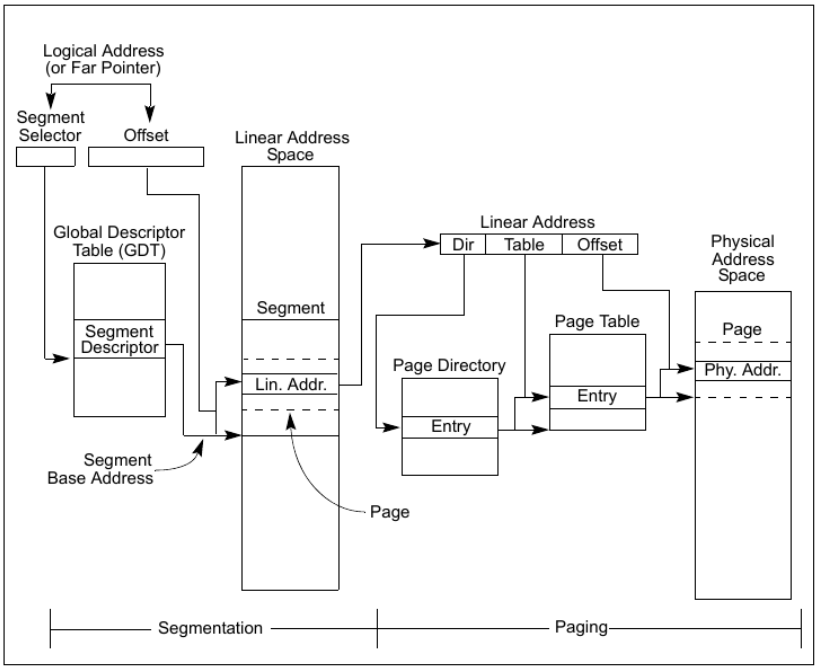
У процессора есть разные режимы, и в этих режимах используется разное логическое представление памяти. Основой режим — защищённый (protected mode) — требует использования сегментной модели памяти. Другое дело, что обычно все сегменты слиты в один, поэтому сегментирование незаметно. Подробнее о режимах процессора и моделях памяти читай в мануале "Intel 64 and IA-32 Architectures Software Developer's Manual", том 3, глава 2.

- Регистры для работы с числами с плавающей запятой (Floating Point Registers). Это восемь 80-битных регистров, организованных в виде стека (ST0 - ST7). К ним добавляются контрольные регистры: 16-битные CW (Control Word), SW (Status Word) и TW (Tag Word), 11-битный FP_OPC (Opcode) и 64-битные FP_IP (Instruction Pointer) и FP_DP (Data Pointer). Всё вместе устройство называется *сопроцессором FPU* (Floating Point Unit), или расширением x87. Подробнее о работе с FPU читай в мануале "Intel 64 and IA-32 Architectures Software Developer's Manual", том 1, глава 8.

- Регистры для работы с векторными инструкциями (Scalar Instruction Multiple Data, SIMD). Это регистры для проведения векторных операций: например, для одновременного сложения не одной, а четырёх пар чисел. К ним относятся регистры MMX, физически совмещённые с регистрами FPU; регистры SSE, AVX и т.д., а также контрольный регистр MXCSR (Control/Status Register). Подробнее о векторных инструкциях и расширениях для работы с ними читай в мануале "Intel 64 and IA-32 Architectures Software Developer's Manual", том 1, главы 9 - 14.

- Системные регистры. Это регистры, с которыми возятся в основном разработчики операционной системы. Большинство из этих регистров с горем пополам описаны в мануале "Intel 64 and IA-32 Architectures Software Developer's Manual", том 3 (так что тебе придётся прочитать его, когда надумаешь писать операционную систему для x86). Вот основные группы системных регистров:

  - Контрольные регистры (Control Registers) CR0 - CR4, CR8, XCR0. Они определяют режим работы процессора и характеристики текущей исполняемой задачи.
  - Регистры управления памятью (Descriptor Table Registers) GDTR, IDTR, LDTR, TR. Регистр IDTR хранит указатель на таблицу дескрипторов обработчиков прерываний IDT (Interrupt Descriptor Table). Регистр TR хранит информацию о текущей выполняемой задаче. Регистры GDTR и LDTR нужны при использовании сегментной модели памяти: они хранят указатели на таблицы сегментных дескрипторов GDT (Global Descriptor Table) и LDT (Local Descriptor Table). Эти таблицы участвуют в процессе преобразования логического адреса в физический:



1. Логический адрес преобразуется в *линейный* адрес. Процессор использует первую часть логического адреса — селектор — для чтения сегментного дескриптора из таблицы. Сегментный дескриптор содержит адрес начала сегмента в памяти (*базовый* адрес) и атрибуты сегмента (размер, права доступа и т.д.). Процессор проверяет, попадает ли смещение (вторая половина логического адреса) внутрь сегмента и не нарушены ли права доступа к сегменту. Сумма базового адреса и смещения даёт линейный адрес.
2. Линейный адрес преобразуется в физический адрес. Этот этап зависит от того, используется ли *страничный механизм* управления памятью (paging). Страничный механизм — это ещё одно логическое представление поверх сегментной модели памяти. Оно нужно для того, чтобы создавать у программ иллюзию очень большого адресного пространства — виртуальной памяти. Виртуальная память разбита на куски (станицы), и в физической памяти присуствует только часть этих страниц (те, которые нужны программе прямо сейчас). Если программе понадобится страница, отсутствующая в физической памяти, придётся подгрузить её. Это займёт некоторое время и, возможно, приведёт к вытеснению других страниц, но в общем ничего страшного не случится (это событие называется “page fault”). Сама программа ничего не знает о страницах и page fault'ах — управлением занимается операционная система. Страничный механизм похож на кэш процессора, а page fault похож на cache miss: если данных нет в кэше, их приходится подгружать из оперативной памяти; если данных нет в оперативной памяти, их приходится подгружать с внешней памяти.

Страничный механизм использовать не обязательно, но обычно он используется в мультизадачных операционных системах. В линуксе есть программа `top` и её более красивый вариант `htop` — они показывают текущие выполняемые задачи. Там хоро-

шо видна разница между потреблением виртуальной и физической памяти (колонки `VIRT` и `RES`): Программа может есть очень много виртуальной памяти и сравнительно немного физической.

Подробнее об управлении памятью в защищённом режиме читай в мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 3, главы 3 - 4.

– Регистры для дебага (Debug Registers) DR0 - DR15. Подробнее в мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 3, глава 17.

– Регистры, зависящие от конкретной модели процессора (Model-Specific Registers, MSRs). Это огромное множество регистров для самых разных нужд. Часть из них присутствует во всех новых моделях — это подмножество называется Architectural MSRs. С большего MSR’ы описаны в мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 3, глава 35. Упомяну наиболее интересные группы MSR’ов:

  * Регистры для управления кэшированием данных (Memory Type Range Registers, MTRRs). Есть разные стратегии кэширования: они определяют, кэшировать ли данные в принципе, когда записывать данные в кэш, когда записывать данные из кэша в оперативную память, когда синхронизировать кэши между собой и т.д. Некотороые области памяти вообще нельзя кэшировать (например, область памяти, через которую какое-то устройство общается с процессором (memory-mapped I/O)). Иногда при записи данных из кэша в память можно не соблюдать порядок изменений (например, при записи в память видеокарты: не важно, в каком порядке меняются отдельные пиксели — главное, чтобы картинка на экране менялась). Иногда надо синхронизировать кэш с памятью каждый раз, когда данные в кэше меняются. Регистры MTRR позволяют разным областям оперативной памяти сопоставить разные стратегии кэширования. Подробнее об управлении кэшами в мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 3, глава 11.
  * Регистры для контроля аппаратных ошибок (Machine Check Registers). Подробнее об обнаружении и исправлении аппаратных ошибок в мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 3, глава 15.
  * Регистры для замера производительности (Performance Counters). Это счётчики всяких событий типа процессорного времени, затраченного на задачу (task-clock) тактов процессора (cycles), переключений контекста на процессоре (context switch), неудачный предсказаний условных переходов (branch miss), отсутствия нужных данных в кэше (cache miss), отсутствия нужной страницы в памяти (page fault) и т.д. Аппаратная поддержка замера производительности — большое дело, она позволяет в точности выяснить, где и почему программа тормозит и как это исправить. Иногда достаточно чуть-чуть поменять порядок вычислений, чтобы получить большой прирост производительности.

Архитектуры процессоров меняются со свистом, поэтому уже через несколько лет всё будет слегка (или сильно) по-другому. Я опиралась на последнюю версию интеловского мануала (он лежит на сайте и время от времени обновляется). В мануале описание скорее всего не точное, наверняка неполное и скоро устареет. Важно примерно представлять себе архитектуру.

Программы, сгенерированные нашим компилятором, будут использовать память, стек и несколько регистров общего назначения (их младшие 32 бита). В нашем языке размер чисел не ограничен — это просто любые целые числа. В компиляторе мы ограничимся 32-битными целыми числами со знаком (то есть числами в диапазоне $[-2^{31}, 2^{31} - 1]$). Я выбрала 32, а не 64 бита потому, что это упростит кодирование команд. Кстати, в парсере и интерпретаторе тоже зашито ограничение на 32 бита — для хранения чисел используется тип `int`.

Теперь, когда ясно, *с чем* работают команды, пора поговорить собственно о командах: какие они бывают и как кодируются. Понятно, что без некоторых команд не обойтись. Необходимый минимум включает команды пересылки данных между памятью и регистрами, арифметико-логические команды и команды передачи управления (как минимум условного перехода). Можно на этом остановиться — ограничиться только самыми простыми командами (и некоторые архитектуры так и делают). А можно добавить ещё много полезных (или не совсем) команд. Как бы там ни было, команды неравноценны по времени выполнения: команды, работающие с памятью, выполняются здорово дольше других (на кэш можно надеяться, но не стоит рассчитывать). От этой неравноценности никуда не денешься, и она порождает два принципиальных подхода к составлению набора команд: **CISC** (который только усиливает неравноценность) и **RISC** (который пытается сгладить неравноценность).

Подход CISC (Complex Instruction Set Computer) возник из-за неудобства программирования на ассемблере: очень уж большой разрыв между высокоуровневыми идеями в человеческой голве и низкоуровневыми машинными инструкциями. На каждый чих приходится писать по маленькой программе. Часто приходится повторять одни и те же рутинные последовательности действий, одни и те же цепочки команд. Естественная мысль — заменить их одной сложной командой. Понятно, что такие сложные команды приведут к ещё большему расслоению команд по времени выполнения, особенно если появится много команд, работающих с памятью. Зато сложные команды можно оптимизировать на уровне микросхем.

Второй подход — RISC (Reduced Instruction Set Computer) — возник, когда стало ясно, что часто ассемблерная программа из простых команд более эффективна, чем сложная команда. Между простыми командами легче отслеживать *зависимости*, поэтому часто их удаётся переставлять местами и выполнять параллельно. Основная идея RISC-архитектуры — сделать так, чтобы почти все команды выполнялись быстро (за один или несколько тактов процессора). Поскольку доступ к памяти занимает много времени, оставляют только две команды работы с памятью: загузка и выгрузка.

CISC-архитектуру можно реализовать по-разному. Первый, очевидный, способ — усложнять процессор, добавляя в него всё новые и новые куски микросхем. Этот способ имеет два недостатка: процессор становится очень сложным, а набор команд — узкоспециализированным или очень большим. Поэтому обычно идут другим путём: делают препроцессор, который раскладывает сложные команды на более простые. Этот препроцессор называется *интерпретатором микрокода*, а сами сложные команды — *микропрограммами*. Микропрограммы хранятся в специальной памяти. Чтобы добавить или поменять команду, нужно просто сохранить её микрокод в эту память. При таком подходе внутренний процессор остаётся простым (причём это вполне может быть RISC-, а не CISC-процессор), а в выборе команд появляется невероят-

ная гибкость: теоретически кто угодно может составить удобный для себя набор команд. На практике далеко не все производители процессоров открывают доступ к микрокоду.

Неправильно думать, что CISC-архитектуры сложные, а RISC-архитектуры простые. Слова “complex” и “reduced” относятся к скорости выполения команд, а не к их многообразию. Есть очень простые CISC-ахитектуры и очень сложные RISC-архитектуры. Многие архитектуры смешивают в себе RISC и CISC черты. Об этом всём хорошо написано в статье “Processor Architectures. RISC-CISC-Harvard-Von Neumann”. :)

x86 — пример очень сложной CISC-архитектуры. Её сложность ты можешь на глаз оценить по размеру мануала “Intel 64 and IA-32 Architectures Software Developer’s Manual”.

Команды в x86 кодируются байтовыми последовательностями разной длины (от 1 до 15 байт). Каждя команда начинается с *опкода*, который занимает от одного до трёх байт. У каждой команды свой, уникальный опкод (поэтому её нельзя спутать с другой командой). После опкода могут быть другие, командо-специфичные байты. В них закодированы операнды: номера регистров, адреса в памяти или *непосредственные операнды* (т.е. константы, зашитые прямо в инструкцию). Детальное описание всех команд x86 ты найдёшь в мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 2.

Ну что ж, немного обсудили целевую архитектуру — пора переходить к генерации кода. Чтобы понять, как компилятор генерирует машинный код, возьмём какую-нибудь программу и скомпилируем её вручную. Для начала возьмём совсем простую программу: 1 + 2. Команда для сложения — `ADD`. Откроем мануал “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 2, на описании команды `ADD`:

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*

00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /r	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32*.
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64*.
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /r	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.
03 /r	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 /r	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

О-хо-хо! На одно только описание команды `ADD` ушла целая страница. Что мы здесь видим?

Команда `ADD` описывается таблицей. Строки таблицы — это разные варианты команды. Колонки отвечают за разные характеристики команды: “Opcode” — схема кодирования команды, “Instruction” — общий вид команды, “Op/En” (Operand Encoding) — тип кодирования операндов, “64 Bit Mode” и “Compat/Leg Mode” (Compatibility/Legacy Mode) — работает ли команда в этих режимах процессора (если помнишь, у процессоров x86 есть разные режимы), “Description” — описание команды на человеческом языке.

У команды `ADD` есть четыре основных варианта, различающихся по типу операндов. Это хорошо видно в колонке “Op/En”: там встречаются четыре разных значения I (immediate), MI (register/memory - immediate), RM (register - register/memory), MR (register/memory - register). Каждый из этих четырёх вариантов дальше разветвляется по размеру операндов.

Какой вариант подходит нам? В нашей программе числа 1 и 2 — это константы (immediate operand). Мы договорились, что компилятор работает с 32-битными числами, поэтому это 32-битные константы. (Из таблицы должно быть понятнее, почему я выбрала 32, а не 64 бита: для 64-битных команд впереди добавляется некий REX-префикс, с которым возиться неохота.) Нам подошла бы команда `ADD imm32, imm32`, но такой нет. Зато есть очень похожие команды `ADD EAX, imm32` и `ADD r/m32, imm32`. Команда `ADD EAX, imm32` берёт первое слагаемое из регистра EAX. Команда `ADD r/m32, imm32` берёт первое слагаемое из любого регистра общего назначения или из памяти. Понятно, что обе команды не идеальны: всё равно придётся загрузить первое слагаемое в память или в регистр. Для этого нам понадобится другая команда — `MOV`:

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8****,*r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.

8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16*.
REX.W + 8C /r	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r/m64*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8*.
REX + B0+ *rb ib*	MOV *r8***, imm8*	OI	Valid	N.E.	Move *imm8* to *r8*.
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16*.
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32*.
REX.W + B8+ *rd io*	MOV *r64, imm64*	OI	Valid	N.E.	Move *imm64* to *r64*.
C6 */0 ib*	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8*.
REX + C6 */0 ib*	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8*.
C7 */0 iw*	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16*.
C7 */0 id*	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32*.
REX.W + C7 */0 io*	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64*.

Час от часу не легче: вариантов ещё больше, чем у команды `ADD`. И это только `MOV` для регистров общего назначения, а есть ещё `MOV`ы для другиих групп регистров. Режимов кодирования операндов тоже прибавилось: кроме уже известных MI, RM и MR появились ещё FD, TD и OI. Нам подходят варианты `MOV r32, imm32` и `MOV r/m32, imm32`.

Пора определиться: в регистр или в память мы будем записывать первое слагаемое, и если в регистр, то в EAX или нет (команда `ADD` предусматривает отдельный вариант для EAX). В архитектуре x86 часто возникают такие ситуации, когда одно и то же можно сделать несколькими разными способами. Хороший компилятор старается всегда выбирать самый быстрый вариант. В нашем случае, во-первых, регистры быстрее памяти — поэтму следует хранить первое слагаемое в регистре. Во-вторых, команда `ADD` явно оптимизирована для регистра EAX, поэтому следует хранить первое слагаемое в регистре EAX. С учётом этого у нас получается

такая ассемблерная программа:

```
mov eax, 1
add eax, 2
```

Неплохо! Осталось перевести эту программу в машинные коды, то есть ассемблировать. Как это сделать? Самый правильный (но не самый простой) способ — прибегнуть к старому другу, интеловскому мануалу. Там подробно объясняется кодирование всех команд. Я разберу только две команды из нашего примера — полную схему кодирования ты всегда найдёшь в мануале.

Начнём с команды `mov eax, 1`. В таблице ей соответствует вариант:

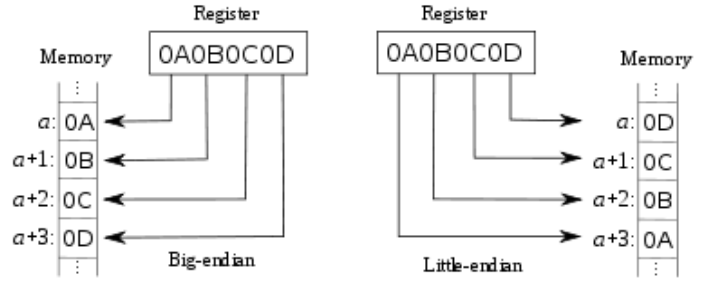
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32.*

В первой колонке указана схема кодирования: `B8 + rd id`. В мануале “Intel 64 and IA-32 Architectures Software Developer’s Manual”, том 2, глава 3.1.1, поясняются эти обозначения:

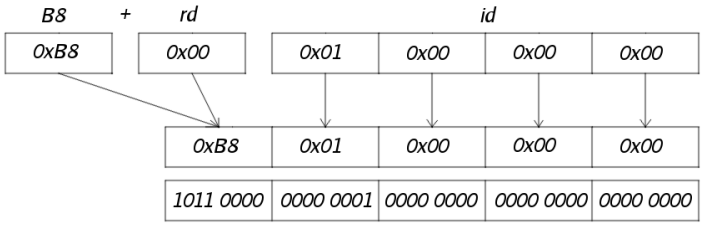
- `B8` — однобайтный шестнадцатеричный опкод.
- `rd` (register dword) — однобайтный номер регистра. Регистру EAX соответствует номер 0, поэтому байт равен `0x00`.
- `B8 + rd` означает, что эти два байта надо сложить: `0xB8 + 0x00 = 0xB8`.
- `id` (immediate dword) — это 4-байтный непосредственный операнд. У нас непосредственный операнд — это константа 1, в шестнадцатеричном представлении — `0x1`. На её кодирование отводится целых четыре байта, в то время как сама константа занимает один бит. Все остальные биты придётся забить нулями: `0x00 0x00 0x00 0x01`. Это кажется напрасной тратой битов, но ведь вместо 1 могло бы быть любое число в диапазоне $[-2^{N-1}, 2^{N-1} - 1]$.

Итого, получаем команду — `0xB8 0x00 0x00 0x00 0x01`, так?

Так, да не совсем. Есть одна гадкая мелочь: почему мы решили, что 4-байтное число 1 должно кодироваться именно как `0x00 0x00 0x00 0x01`? Потому что это *естественно*: мы привыкли представлять числа от старших разрядов к младшим. Однако число ничуть не изменится, если его представлять как-то по-другому: например, от младших разрядов к старшим. Это дело договорённости: важно, чтобы все понимали представление одинаково. Так вот архитектура x86 представляет себе числа от младших *байтов* к старшим: например, 4-байтное число `0x0A0B0C0D` предстaлено в виде `0x0D 0x0C 0x0B 0x0A`. Такой порядок байт называется *little-endian*. Другой порядок — *big-endian*, в нём байты хранятся от старших к младшим: `0x0A`, `0x0B`, `0x0C`, `0x0D`. Само понятие порядка байт называется **endianness**.



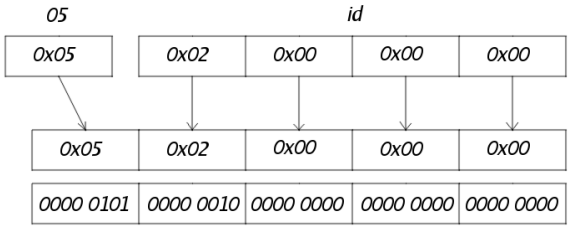
Исправленная команда выглядит так: `0xB8 0x01 0x00 0x00 0x00`.



Теперь ассемблируем вторую команду: `add eax, 2`. Мы решили использовать вариант:

05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.

На сей раз схема кодирования совсем простая: `05 id`. Понятное дело, `05` — однобайтный шестнадцатеричный опкод, а `id` — 4-байтный непосредственный операнд в форме little-endian. Итого получаем команду `0x05 0x02 0x00 0x00 0x00`:



Объединяя обе команды, получаем итоговую программу: `0xB8 0x01 0x00 0x00 0x00 0x05 0x02 0x00 0x00 0x00`.

На самом деле можно было получить её гораздо проще с помощью какого-нибудь готового ассемблера. (Здесь я понимаю ассемблер как программу, а не как язык.) Синтаксис разных ассемблеров немного отличается. Для ассемблера `nasm` исходная программа выглядит так (файл example.asm):

```
1  BITS 64
2
3  mov eax, 1
4  add eax, 2
```

Пришлось добавить директиву `BITS 64` — она указывает ассемблеру, для какого режима процессора ассемблировать программу. Ассемблируем:

```
$ nasm -O0 example.asm
```

Флаг -O0 (нулевые оптимизации) заставляет `nasm` сохранять исходные команды (по умолчанию `nasm` подыскивает более короткие эквивалентные команды). Содежимое бинарного файла example можно посмотреть с помощью `mc`, открыв файл по `F3` и затем нажав `F4`:

```
0xB8 0x01 0x00 0x00 0x00 0x05 0x02 0x00 0x00 0x00
```

Это полностью совпадает с тем, что мы наассемблировали руками. Теперь можно из любопытства опустить флаг -O0:

```
$ nasm example.asm
```

Первая команда (`mov eax, 1`) осталась неизменной, а вот вторую команду (`add eax, 2`) `nasm` оптимизировал: он заменил её более короткой командой `add r/m32, imm8` (воспользовавшись тем, что константа 2 влазит в один байт):

```
0xB8 0x01 0x00 0x00 0x00 0x83 0xC0 0x02
```

(Команда `add r/m32, imm8` узнаётся по опкоду `0x83` из таблицы для команды `ADD`). Вот так мы узнали, что оказывается можно было составить более эффективную программу.

Дальше я не буду останавливаться на кодировании команд. Некоторые команды кодируются намного хитрее, чем те, что мы видели, особенно команды, работающие с памятью: у них есть сложные режимы адресации. В нашем компиляторе будут использоваться в основном простые команды. Это не всегда будут самые эффективные команды, но я и не замахивалась на оптимизирующий компилятор — понятно, что исходный язык настолько прост, что позволяет любую программу вычислить на этапе компиляции (программы не зависят ни от каких внешних данных).

Ну что ж, побаловались с маленькой программой, пора переходить к компиляции принципиально *любой* программы. Чтобы не погибнуть в мутных примерах и частных случаях, понадобятся радикальные меры. :D

Как говорится, "воспользуемся методом математической индукции". Программа представлена в виде AST, и индукцию мы будем проводить по высоте этого AST. Листьями AST являются числа, узлами — арифметические операции. Сначала мы рассмотрим AST нулевой высоты: лист, и покажем, как его скомпилировать — это будет база индукции. Затем мы предположим, что любой AST высоты не более k скомпилирован, и покажем, как на его основе скомпилировать любой AST высоты $k + 1$ — шаг индукции. Поехали!

Итак, база индукции: нужно скомпилировать лист AST, то есть константу C. Нет ничего проще:

```
mov eax, C
```

Переходим к шагу индукции: нужно скомпилировать программу $p_1 \circ p_2$, где p_1 и p_2 — уже скомпилированные программы, а $\circ$ — арифметическая операция. Сразу возникает вопрос: где хранятся результаты выполнения программ p_1 и p_2? Если p_1 — лист, то результат p_1 хранится в регистре EAX. Однако если p_2 — тоже лист, то результат p_2 хранится тоже в регистре EAX! Очевидно, если p_2 выполняется после p_1, то p_2 затрёт результат p_1:

```
<p1>          ; result in EAX
<p2>          ; result in EAX
<op> eax, eax ; bad :(
```

Что с этим делать? Можно попытаться заставить разные подпрограммы использовать разные регистры: например, пусть p_1 сохраняет результат в EAX, а p_2 — в EBX:

```
<p1>          ; result in EAX
<p2>          ; result in EBX
<op> eax, ebx ; result in EAX
```

Всё хорошо, пока не задумываешься, что из себя представляют p_1 и p_2. Это ведь *любые* программы, они могут быть очень большими и ветвиться на сколько угодно подпрограмм. Все эти подпрограммы как-то делят между собой регистры. Что если подпрограмм больше, чем регистров? Например, если $p_1 = C_1 + (C_2 + (... + (C_{N-1} + C_N)...))$, то по нашей логике программа для p_1 должна скомпилироваться так:

```
mov r1, C1          ; result in r1
mov r2, C2          ; result in r2
...
mov r(N-1), C(N-1) ; result in r(N-1)
mov r(N), C(N)      ; result in r(N)
add r(N-1), rN      ; result in r(N-1)
...
add r1, r2          ; result in r1
```

Ясно, что для больших N регистров не хватит. Есть и другая неприятность: запрет использования отдельных регистров вносит асимметрию. Две одинаковых подпрограммы могут скомпилироваться по-разному только потому, что им будут доступны разные регистры. Сама компиляция тоже усложняется: нужно постоянно следить за тем, какие регистры свободны. Короче, ничё хорошего. Явно регистрами тут не обойтись.

Очевидный выход — сохранять результат программы p_1 в память. (Результат программы p_2 сохранять не надо, он сразу же используется.) На вопрос "а если памяти не хватит?" я могу ответить только "будет плохо!". На самом деле, если нам попадётся *настолько* большая программа, проблемы начнутся гораздо раньше и решать их придётся другими способами (разбивать программу на куски). Нас эти проблемы не особо интересуют, поскольку они общие для большинства программ, а не специфичные для компилятора. Будем считать, что памяти много.

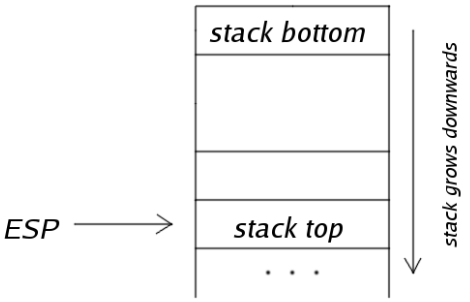
Итак, сохраняем результат программы p_1 в память (допустим, у нас есть большой кусок памяти):

```
<p1>
mov [addr], eax ; result in [addr]
<p2>            ; result in eax
mov ebx, [addr]
<op> eax, ebx   ; result in eax
```

Эта схема вполне работоспособна и можно было бы на ней остановиться. Но она, честно говоря, малось корявая. Компилятору придётся много возиться с адресами: программа p_2 может рекурсивно ветвиться на другие подпрограммы, которые тоже будут сохранять результат в память. Компилятору придётся запоминать, где чей результат. Хуже того: когда результат подпрограммы читается из памяти, он больше не нужен, и хорошо бы эту память использовать повторно. Значит, компилятору придётся следить за тем, какие ячейки памяти свободны (ну или использовать память нерационально).

Этого всего можно избежать, если заметить, что в нашем случае порядок вычислений позволяет хранить значения в стеке.

Стек — это кусок памяти, доступ к которому осуществляется с помощью команд `PUSH` и `POP`: `PUSH` "запихивает" значение на вершину стека, а `POP` "выпихивает" последнее записанное значение. Получается, что значения читаются в порядке, обратном порядку записи. Указатель на вершину стека храниться в регистре ESP (Stack Pointer), и команды `PUSH` и `POP` сами обновляют его: `PUSH` уменьшает ESP на размер операнда и записывает операнд в память по адресу [ESP], а `POP` читает значение из памяти по адресу [ESP] в операнд и увеличивает ESP на размер операнда.



Казалось бы, довольно специфичная структура — но стек оказывается полезен во многих задачах. Он позволяет работать со вложенными (древовидными) структурами: узел-потомок начинает обрабатывться позже, чем узел-предок, а заканчивает раньше (что соответствует принципу LIFO). Если бы AST представляло из себя не дерево, а произвольный граф, то стеком мы бы не отделались.

С учётом стека произвольный узел AST вида $p_1 \circ p_2$ компилируется так:

```
<p1>
push eax
<p2>
pop ebx
<op> eax, ebx ; result in eax
```

Это уже очень хороший и почти окончательный вариант. :) Осталось подправить две мелочи, связанные с кодированием команд.

Первая мелочь касается неразберихи с битностью, командами и режимами процессора. Мы нигде явно не договаривались, для какого режима процессора мы компилируем программы. Помнится, я сказала, что мы будем использовать 32-битную, а не 64-битную арифметику, потому что она проще кодируется. Значит ли это, что мы автоматически выбрали 32-битный режим? Нет. Команды 32-битной арифметики доступны в обоих режимах (это написано в их таблицах в колонках "64 Bit Mode" и "Compat/Leg Mode"). Давай определимся: пусть наши программы будут 64-битные — как-никак мы компилируем для 64-битного процессора. Это решение никак не повлияет на арифметические команды, но немного повлияет на команды `PUSH` и `POP`. Как видно из таблицы для команды `PUSH`, в 64-битном режиме нет команды `PUSH r/m32` — есть только команда `PUSH r/m64`. Эти две команды кодируются абсолютно одинаково — строго говоря, это одна команда, интерпретация которой зависит от режима. Аналогично с командой `POP`. Исправленная программа выглядит так:

```
<p1>
push rax
<p2>
pop rbx
<op> eax, ebx ; result in eax
```

Может показаться, что нельзя так перемешивать 32-битные и 64-битные регистры, но заметь, что в вычислениях участвуют только 32-битные регистры, а их старшие половины просто занимают лишнее место в стеке.

Вторая мелочь связана с кодированием арифметических команд. У нас первый операнд находится в регистре EBX, а второй — в регистре EAX. Результат мы хотим получить в регистре EAX. Со сложением всё хорошо: есть команда `add eax, ebx`, котоая прибавляет EBX к EAX. С умножением тоже всё хорошо: команда `imul ebx` умножает EAX на EBX. Из-за коммутативности сложения и умножения нам безразличен порядок операндов. С вычитанием всё хуже: команда `sub eax, ebx` вычитает EBX из EAX (перепутаны уменьшаемое и вычитаемое), а команда `sub ebx, eax` делает что надо, но записывает результат в EBX — значит, придётся менять операнды местами или перемещать результат. С делением совсем плохо: делимое обязательно должно быть в регистре EAX (других вариантов просто нет). Очевидное решение — поменять операнды местами (для этого есть даже специальная команда `XCHG`). Но есть более элегантное решение: можно поменять местами подпрограммы p_1 и p_2:

```
<p2>
push rax
<p1>
pop rbx
<op> eax, ebx ; result in eax
```

Наконец, есть ещё одна мелочь: произведение двух 32-битных чисел может быть 64-битным числом, поэтому команда `IMUL` записывает результат не просто в EAX, а в пару регситров EDX:EAX. Мы работаем с 32-битными числами и игнорируем старшую часть результата: что не влезло, то не влезло. Но с командой `IDIV` просто проигнорировать не получится: она берёт старшую часть делимого из регистра EDX, поэтому если там будет мусор, результат деления будет неправильным. Перед делением мы должны заполнить регистр EDX старшим (знаковым) битом делимого (sign-extend EDX of EAX): если делимое положительное, EDX = 0, если отрицательное — EDX = 0xFFFFffff. В этом нам поможет команда `CDQ` (Convert Double to Quad). (Если непонятно, почитай про двоичное представление отрицательных чисел в виде дополнения до двух: two's complement.)

Окончательный алгоритм компиляции AST в код для x86-64 выглядит так:

```
compile(p):
    if p = C:
        mov eax, C
    else if p = p1 + p2:
        compile(p2)
        push rax
        compile(p1)
        pop rbx
        add eax, ebx
    else if p = p1 - p2:
        compile(p2)
        push rax
```

```
        compile(p1)
        pop rbx
        sub eax, ebx
    else if p = p1 * p2:
        compile(p2)
        push rax
        compile(p1)
        pop rbx
        imul ebx
    else if p = p1 / p2:
        compile(p2)
        push rax
        compile(p1)
        pop rbx
        cdq
        idiv ebx
```

Всё! С генерацией кода разобрались. :) Вот кусок компилятора, который генерирует код по AST:

```
1  #include "compiler.h"
2
3  void compile (AST * ast, std::vector<unsigned char> & code)
4  {
5      switch (ast->type)
6      {
7          case AST::NUMBER:
8          {
9              // mov eax, <imm32> ; b8 <byte1> <byte2> <byte3> <byte4> (little-
                   endian)
10             unsigned char byte1 = (ast->value.number >> (8 * 0)) & 0xFF;
11             unsigned char byte2 = (ast->value.number >> (8 * 1)) & 0xFF;
12             unsigned char byte3 = (ast->value.number >> (8 * 2)) & 0xFF;
13             unsigned char byte4 = (ast->value.number >> (8 * 3)) & 0xFF;
14             code.push_back (0xb8);
15             code.push_back (byte1);
16             code.push_back (byte2);
17             code.push_back (byte3);
18             code.push_back (byte4);
19             return;
20         }
21         case AST::ADD:
22         {
23             compile_operands (ast, code);
24             // add eax, ebx ; 01 d8
25             code.push_back (0x01);
26             code.push_back (0xd8);
27             return;
28         }
29         case AST::SUB:
30         {
31             compile_operands (ast, code);
32             // sub eax, ebx ; 29 d8
```

```
            code.push_back (0x29);
            code.push_back (0xd8);
            return;
        }
        case AST::MUL:
        {
            compile_operands (ast, code);
            // imul ebx ; f7 eb
            code.push_back (0xf7);
            code.push_back (0xeb);
            return;
        }
        case AST::DIV:
        {
            compile_operands (ast, code);
            // cdq ; 99, sign-extend edx:eax of eax
            code.push_back (0x99);
            // idiv ebx ; f7 fb
            code.push_back (0xf7);
            code.push_back (0xfb);
            return;
        }
    }
}

void compile_operands (AST * ast, std::vector<unsigned char> & code)
{
    compile (ast->value.operands.right, code);
    // push rax ; 50
    code.push_back (0x50);
    compile (ast->value.operands.left, code);
    // pop rbx ; 5b
    code.push_back (0x5b);
}
```

Компилятор, как и интерпретатор, рекурсивно обходит AST, только вместо немедленных вычислений он генерирует машинные инструкции и сохраняют их в массив байтов. Компиляцию операндов я вынесла в отдельную функцию, потому что она везде повторяется.

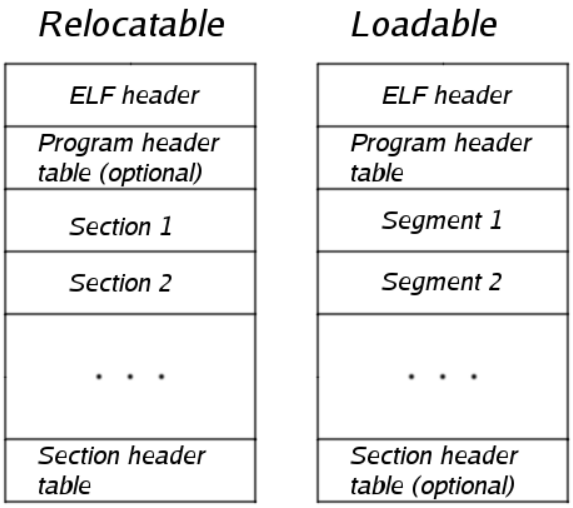
Вообще говоря, на этом задача компилятора выполнена: мы разобрались с архитектурой x86 и научлились компилировать AST в машинный код. Но для души этого мало: надо ещё запустить скомпилированную непосильным тудом программу на настоящем процессоре. (Как минимум, проверить, что она работает. :D)

Обычно компилятор сохраняет скомпилированную программу в виде объектного файла. Объектный файл содержит машинный код и разную информацию: статические данные, таблицу симовлов (экспортируемые и импортируемые функции, глобальные переменные и т.д.), релокации (адреса в программе надо настраивать в соответствии с адресом, по которому программа будет загружена в память), информацию для дебага и прочее. Когда компилятор скомпилировал все единицы трансляции в объектные файлы, наступает черёд линкера. Линкер связывает все объектные файлы в один исполняемый файл: настраивает релокации, разрешает зависимости и т.д. Линковка — отдельная тема, сложная и интересная, и я не буду её трогать. Нам линкер не нужен — наши программы всегда состоят из одной самодостаточной единицы транс-

ляции и не возятся с адресами в памяти. Мы можем сразу же сохранять программу в виде исполняемого файла.

Формат исполняемых файлов специфичен для операционной системы. В линуксе принят формат ELF (Executable and Linkable Format), и мы рассмотрим его 64-битную версию: ELF-64. Описание этого формата ты можешь найти в `man elf` (или в интернедах).

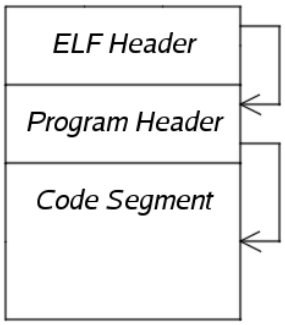
ELF-файл предназначен для хранения двух типов файлов: объектных (relocatable) и исполняемых (loadable). С объектными файлами работает линкер, с исполняемыми — загрузчик.



Любой ELF-файл начинается с ELF-заголовка, который описывает ключевые характеристики файла и взаимное расположение остальных частей файла: таблицы программных заголовков, таблицы разделов и самих сегментов или разделов. Таблица программных заголовков необязательна для объектных файлов, а таблица разделов необязательна для исполняемых файлов.

Наши ELF-файлы будут состоять из трёх частей:

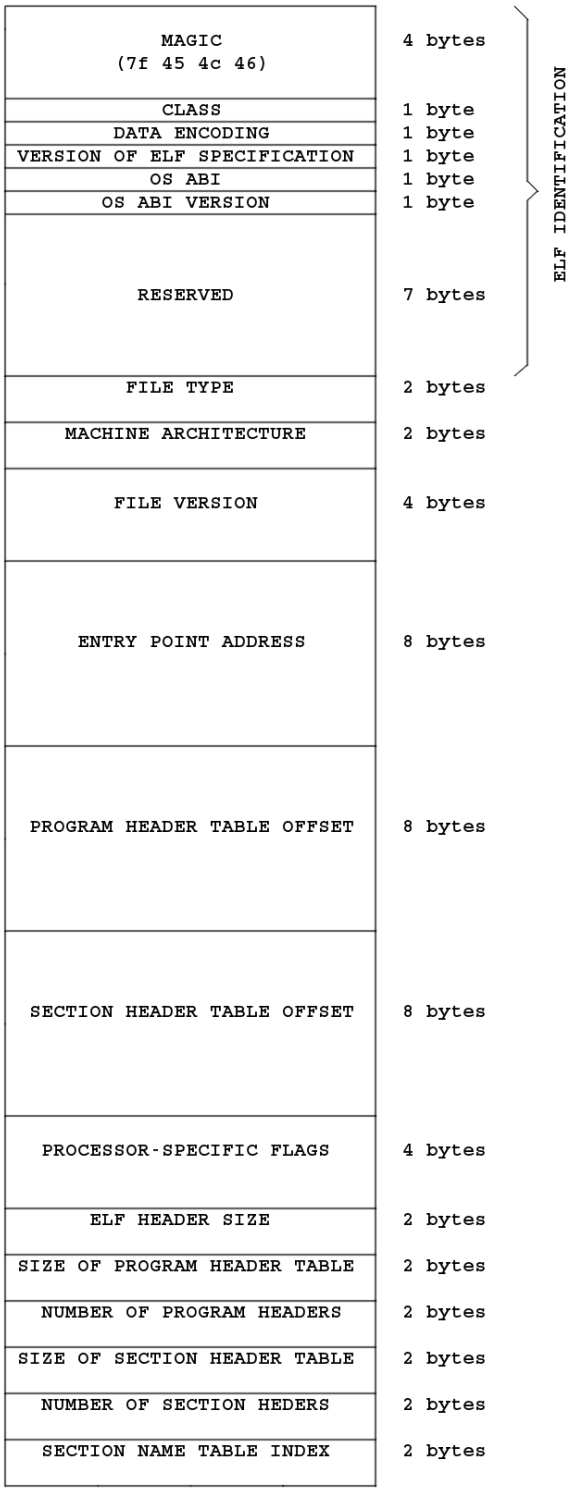
1. ELF-заголовок.
2. Таблица программных заголовков с единственным заголовком, описывающим сегмент кода.
3. Сегмент кода, содержащий код программы.



Как нам проще всего сгенерировать заголовки? Можно руками записать нужные байты в файл, пользуясь спецификацией и внимательно следя за размером и содержимым каждого поля. Но в линуксе есть С-шный заголовочный файл “elf.h”, в котором определены структуры заголовков и много удобных макросов и констант с красивыми названиями. Использование этого хедера сделает наш компилятор непортабельным: компилятор невозможно будет собрать в другой операционной системе, где нет такого хедера. Но честно говоря, наш компилятор и так не отличается особой портабельностью: генерирует код для специфичной архитектуры x86_64 и сохраняет его в специфичном формате ELF-64.

ELF-заголовок состоит из следующих полей:

- Группа полей “ELF Identification”:
  - Четырёхбайтовая сигнатура `0x7f 0x45 0x4c 0x46` (“magic”). Если ты откроешь в `mc` по Shift+F3 какой-нибудь ELF-файл, ты увидишь эту сигнатуру: первый символ (`0x7f`) непечатный, а остальные три (`0x45 0x4c 0x46`) образуют слово “ELF”. Наличие уникальной сигнатуры характерно для бинарных форматов.
  - Класс (разрядность): у нас ELFCLASS64.
  - Кодирование данных: у нас ELFDATA2LSB.
  - Номер версии формата ELF:у нас EV_CURRENT.
  - OS ABI (Operation System Application Binary Interface, бинарный интерфейс операционной системы): у нас, видимо, ELFOSABI_LINUX.
  - Номер версии OS ABI: в `man elf` советуют 0.
  - Семь зарезервированных байтов (нулевых).
- Тип: у нас ET_EXEC (исполняемый).
- Архитектура: у нас EM_X86_64.
- Номер версии файла: у нас, видимо, EV_CURRENT.
- Точка входа (адрес в памяти, на который загрузчик передаст управление): пока непонятно, откуда его взять.
- Смещение таблицы программных заголовков: у нас она идёт сразу после ELF-заголовка, поэтому смещение равно размеру ELF-заголовка.

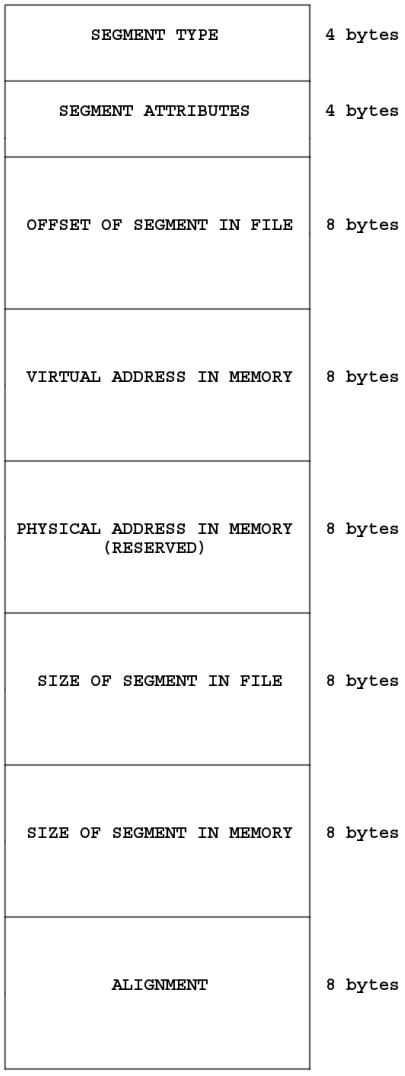


- Смещение таблицы разделов: в `man elf` говорят, что если этой таблицы нету, то смещение 0.
- Процессоро-специфичные флаги: в `man elf` советуют 0.

- Размер ELF-заголовка: фиксирован, можно посчитать (но в C есть удобный оператор `sizeof`, позволяющий получить размер структуры).
- Размер таблицы программных заголовков: равен размеру программного заголовка (у нас всего один заголовок в таблице).
- Число программных заголовков: у нас 1.
- Размер таблицы разделов: у нас её нет, поэтому 0.
- Число разделов: 0.
- Номер раздела, содержащего таблицу имён разделов: у нас такого раздела нет, поэтому SHN_UNDEF.

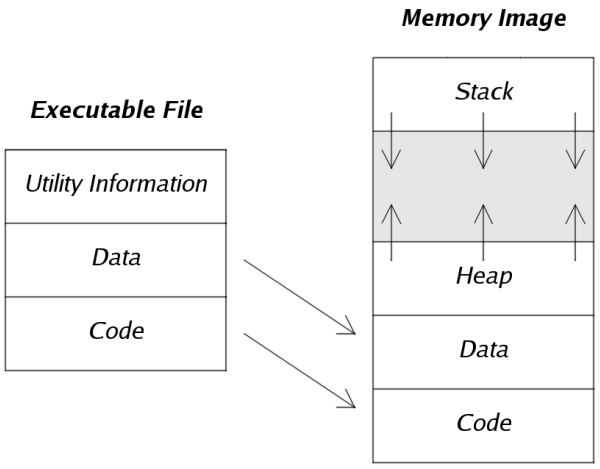
Программный заголовок описывает сегмент (в нашем случае сегмент кода) и состоит из следующих полей:

- Тип сегмента: у нас PT_LOAD (загрузочный).
- Атрибуты сегмента: у нас PF_X и PF_R (доступен на исполнение и чтение).
- Смещение сегмента от начала файла (равно размеру ELF-заголовка).
- Адрес загрузки сегмента в виртуальную память (пока непонятно, откуда взять этот адрес).
- Адрес загрузки сегмента в физическую память (поле зарезервировано для систем, в которых напрямую адресуется физическая память, у нас должно быть нулевым).
- Размер сегмента в файле (равен размеру кода).
- Размер сегмента в памяти (равен размеру кода).
- Выравнивание. Обычно программа ничего не знает о том, в какой области памяти располагается её код и данные. Но иногда при работе с памятью хочется иметь какие-то гарантии: например, что адрес выделенного блока памяти достаточно “круглый”, то есть несколько его младших битов — нули. Это позволяет использовать младшие биты адреса для своих нужд (поскольку адрес однозначно определяется старшими, ненулевыми битами). Наша программа не делает никаких подобных предположений, поэтому выравнивание нам не критично — сойдёт 0.



Итак, осталось два непонятных поля: точка входа (из ELF-заголовка) и адрес загрузки сегмена кода в память (из программного заголовка). В нашем случае оба поля совпадают: это адрес в памяти, на который загрузчик передаст управление, когда загрузит программу в память. Но откуда взять это адрес: может ли он быть любым, или нужен какой-то стандартный? Если у двух программ этот адрес совпадёт, они что, загрузятся в одну и ту же область памяти? Чтобы ответить на эти вопросы, надо представлять, как программы загружаются в память.

На файловой системе программа хранится в виде исполняемого файла (например, в формате ELF). Это очень компактное представление: собственно код программы, инициализированные статические данные (у нас их нет) и вспомогательная информация.



Чтобы исполнить программу, загрузчик разворачивает её из компактного представления в *образ* в памяти: к коду и статическим данным добавляются стек и куча — области памяти, в которых программа будет сохранять данные по мере исполнения.

Размер стека и кучи не фиксирован (во время исполнения программа может выделять и освобождать память), поэтому стек располагается в старших адресах памяти и растёт вниз, а куча располагается в младших адресах (сразу после кода и даных) и растёт вверх. При таком раскладе возникают два вопроса:

1. Что если одни части программы налезут на другие (например, стек на кучу)?
2. Что если надо одновременно загрузить в память несколько программ, как поделить память между ними?

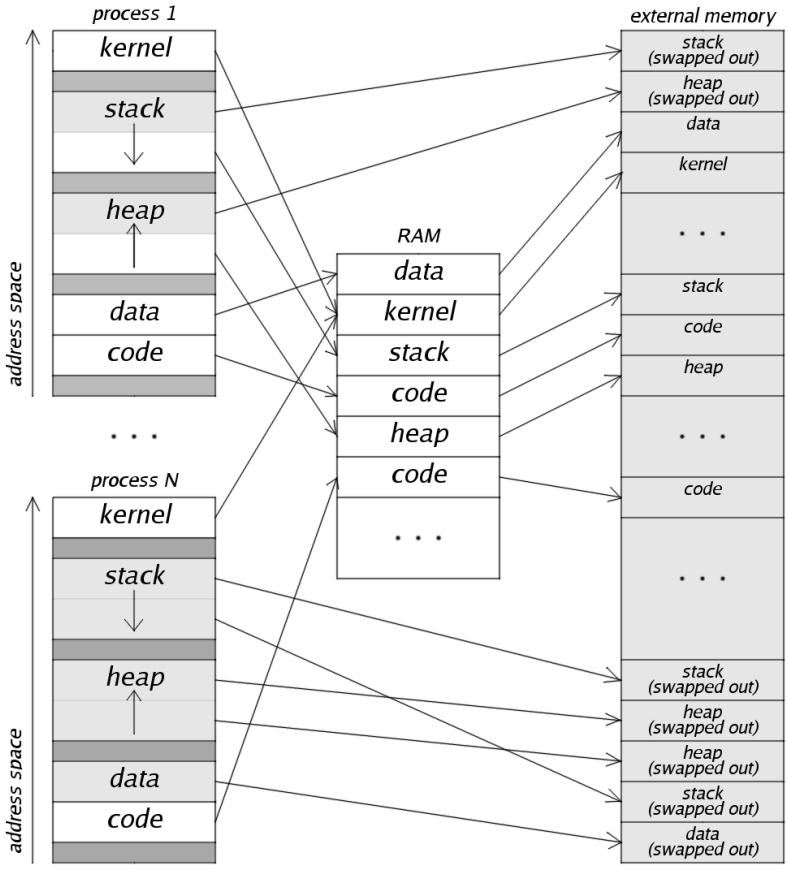
На оба эти вопроса ответ один — виртуальная память.

Я уже немного рассказывала про виртуальную память: это логическое представление оперативной памяти (обычно поверх другого логического представления — сегментной модели памяти). Каждой программе память представлется в виде виртуального адресного пространства. В этом виртуальном адресном пространстве нет других программ (кроме ядра операционной системы и, возможно, динамических библиотек): программе надо неслабо постараться, чтоб влезть в чужое адресное пространство. Размер виртуального адресного пространства зависит от разрядности адреса: 2^N для архитектуры с N-битными адресами, то есть 4 гигабайта для 32-битных адресов и 2 эксабайта для 64-битных адресов. В архитектуре x86_64 размер виртуального адресного пространства — 256 терабайт (не все 64 бита адреса используются). Это значительно превышает объём физической оперативной памяти (обычно несколько гигабайт).

Виртуальная память реализована на основе страничного механизма процессора: адресное про-

странство разбито на страницы, которые по мере надобности загружаются в физическую память и выгружаются из неё. Операционная система настраивает процессор и следит за отображением виртуальных страниц на физическую память. Часть страниц виртуальной памяти привязана к реальной (оперативной или внешней) памяти (mapped), другие не привязаны (unmapped). У каждой страницы есть права доступа на запись/чтение/исполнение (unmapped страницы недоступны ни на что). Всё управление виртуальной памятью (выделение/освобождение/изменение прав доступа) программа вынуждена делать через операционую систему. Если программа попытается нарушить права доступа к странице, операционная система поймает это нарушение и прервёт исполнение программы.

Виртуальная память решает сразу несколько проблем. Во-первых, разные части программы не налезут друг на друга: в виртуальном адресном пространстве между ними находятся "буферные" страницы (unmapped), доступ к которым запрещён. Во-вторых, загрузить несколько программ в память тоже не проблема: операционная система просто должна распределять физическую память сразу между несколькими виртуальными пространствами. При этом разные программы изолированы друг от друга. В-третьих, для самих программ управление памятью значительно упрощается. Из недостатков — доступ к памяти становится ещё медленнее (особенно с учётом page fault'ов).



Я уже говорила (и нарисовала на картинке), что в адресном пространстве любой программы присутствует ядро операционной системы. Зачем? Во-первых, полезная программа не может без операционной системы: выделение памяти, работа с файлами, межпроцессные взаимодействия — всё требует системных вызовов. Во-вторых, даже если сама программа не взаимодействует с операционной системой, операционной системе может понадобиться провзаимодействовать с программой (напрмер, прервать её выполнение). В-третьих, ядро операционной системы всё равно должно быть загружено в физическую память (операционная система исполняется дольше всех программ, которые запущены из неё), поэтому ничего не стоит замапить его в виртуальную память любой программы.

Кроме ядра операционной системы, программы часто используют библиотеки. Следует разли-

чать *статические* и *динамические* библиотеки: статические зашиваются прямо в программу (линкером), а динамические отдельно грузятся в адресное пространство программы (загрузчиком или самой программой во время исполнения). Если программа написана на языке высокого уровня, она почти наверняка будет слинкована со стандартной библиотекой этого языка.

Теперь можно вернуться к нашим недовычисленным полям: точке входа (`e_entry`) и адресу загрузки сегмента кода (`p_vaddr`). Оба поля означают виртуальный адрес загрузки программы в память. На этот адрес есть некоторые ограничения. Первое ограничение — по нулевому адресу грузить программу нельзя: там располагается одна из "буферных" областей виртуальной памяти. Размер этой области в линуксе записан в файле /proc/sys/vm/mmap_min_addr. Второе ограничение — если внимательно прочитать `man elf`, то там написано, что `p_vaddr` и `p_offset` должны быть сравнимы по модулю `p_align` (давать одинаковый остаток от деления на `p_align`). С учётом обоих ограничений положим `e_entry = p_vaddr = mmap_min_addr + p_offset`.

Ну что, всё? Почти. :) Есть одна мелочь: наша программа не останавливается. Скомпилировать-то мы её скомпилируем, потом загрузим в память и передадим на неё управление. Программа радостно выполнится, и ...? Никакой магии не произойдёт, начнут выполняться следующие за сегментом кода байты в памяти. В лучшем случае эти байты окажутся в области памяти, недоступной на выполнение, и операционная система прибьёт программу сегфолтом. В худшем случае может быть что угодно: начнёт исполняться неизвестный код (который, может быть, только и ждал, пока какой-нибудь лапоть передаст на него управление).

Чтобы избежать этого позора, нам надо как-то завершить программу. Вообще, это сделать довольно просто: достаточно вызвать какое-нибудь гадкое исключение. Можно поделить на ноль, можно обратиться по нулевому адресу, можно попытаться исполнить невалидную команду процессора. Во всех этих случаях операционная система гарантированно прибьёт программу (гарантированно — это важно!). Но есть способ завершить программу по-хорошему: сделать системный вызов `exit`. Как видишь, даже наша простецкая программа не обошлась без системных вызовов.

В линуксе, чтобы сделать системный вызов, нужно положить номер вызова в EAX, аргументы в другие регистры, и выполнить 80-е прерывание. Номер `exit` — 1, единственный аргумент — код возврата в EBX. Наша программа сохраняет результат выполнения в EAX, поэтому получаем такой код:

```
mov ebx, eax
mov eax, 1
int 80
```

Этот код должен идти сразу после кода программы. Результат будет в коде возврата. Посмотреть его можно командой `echo $?`, выполнив её сразу после выполнения программы. Правда, если результат не влазит в 1 байт, мы его не увидим: старшие байты операционная система использует для своих нужд (выставляет там разные ошибки и статусы). Кроме того, шелл может обрубить ещё половину кодов (сужая диапазон до $[0, 127]$). Наконец, код возврата интерпретируется как беззнаковое число, поэтому отрицательные результаты мы тоже не увидим. По-хорошему, нельзя возвращать 32-битный знаковый результат программы в коде возврата,

но в противном случае пришлось бы выводить результат на экран (конвертировать число в строку и делать системный вызов), а я решила, что с тебя уже хватит. ;)

Вот теперь действительно всё. Код генерации ELF-файла вот (файл gen_elf64.h):

```
#include <elf.h>
#include <stdio.h>
#include <string.h>

#include "compiler.h"

void gen_elf64 (AST * ast, FILE * out)
{
    std::vector<unsigned char> code;
    compile (ast, code);
    compile_exit (code);

    const uint64_t code_offs  = sizeof (Elf64_Ehdr) + sizeof (Elf64_Phdr);
    const uint64_t load_addr  = get_mmap_min_addr () + code_offs;
    const uint64_t total_size = code_offs + code.size ();

    // elf64 elf header
    Elf64_Ehdr ehdr;
    ehdr.e_ident[EI_MAG0]       = ELFMAG0;         // magic (7f 45 4c 46)
    ehdr.e_ident[EI_MAG1]       = ELFMAG1;
    ehdr.e_ident[EI_MAG2]       = ELFMAG2;
    ehdr.e_ident[EI_MAG3]       = ELFMAG3;
    ehdr.e_ident[EI_CLASS]      = ELFCLASS64;      // class (64-bit)
    ehdr.e_ident[EI_DATA]       = ELFDATA2LSB;     // 2's complement little-endian
    ehdr.e_ident[EI_VERSION]    = EV_CURRENT;      // version of ELF specification
    ehdr.e_ident[EI_OSABI]      = ELFOSABI_LINUX; // OS ABI (Linux)
    ehdr.e_ident[EI_ABIVERSION] = 0;               // ABI version
    memset (&ehdr.e_ident[EI_PAD], 0, 7);          // reserved
    ehdr.e_type = ET_EXEC;                    // executable file
    ehdr.e_machine = EM_X86_64;               // machine architecture
    ehdr.e_version = EV_CURRENT;              // file version
    ehdr.e_entry = load_addr;                 // entrypoint
    ehdr.e_phoff = sizeof (Elf64_Ehdr);       // program header offset
    ehdr.e_shoff = 0;                         // section header offset
    ehdr.e_flags = 0;                         // flags
    ehdr.e_ehsize = sizeof (Elf64_Ehdr);      // elf64 header size
    ehdr.e_phentsize = sizeof (Elf64_Phdr); // program header entry size
    ehdr.e_phnum = 1;                         // program header count
    ehdr.e_shentsize = 0;                     // section header entry size
    ehdr.e_shnum = 0;                         // section header count
    ehdr.e_shstrndx = SHN_UNDEF;              // section header strtab section index

    // elf64 program header
    Elf64_Phdr phdr;
    phdr.p_type = PT_LOAD;       // segment type: loadable
    phdr.p_flags = PF_X | PF_R; // segment permissions: executable and readable
    phdr.p_offset = code_offs;  // offset of segment in file
    phdr.p_vaddr = load_addr;   // virtual address of segment in memory
    phdr.p_paddr = 0;           // reserved for systems with physical addressing
    phdr.p_filesz = total_size; // size of file image of segment
    phdr.p_memsz = total_size;  // size of memory image of segment
```

```
    phdr.p_align = 0;           // alignment

    fwrite (&ehdr, 1, sizeof (Elf64_Ehdr), out);
    fwrite (&phdr, 1, sizeof (Elf64_Phdr), out);
    for (unsigned int i = 0; i < code.size (); ++i)
        fwrite (&code[i], 1, 1, out);
}

void compile_exit (std::vector<unsigned char> & code)
{
    // mov ebx, eax ; 89 c3
    code.push_back (0x89);
    code.push_back (0xc3);
    // mov eax, 0x01 ; b8 01 00 00 00 (little-endian)
    code.push_back (0xb8);
    code.push_back (0x01);
    code.push_back (0);
    code.push_back (0);
    code.push_back (0);
    // int 80 ; cd 80
    code.push_back (0xcd);
    code.push_back (0x80);
}

uint64_t get_mmap_min_addr ()
{
    uint64_t mmap_min_addr;
    FILE * f = fopen ("/proc/sys/vm/mmap_min_addr", "r");
    fscanf (f, "%lu", &mmap_min_addr);
    fclose (f);
    return mmap_min_addr;
}
```

А вот хедер компилятора (файл compiler.h):

```
#include <stdint.h>
#include <stdio.h>
#include <vector>

#include "parser.h"

void compile (AST * ast, std::vector<unsigned char> & code);
void compile_operands (AST * ast, std::vector<unsigned char> & code);
void compile_exit (std::vector<unsigned char> & code);
void gen_elf64 (AST * ast, FILE * out);
uint64_t get_mmap_min_addr ();
```

Фух. Тяжёлая выдалась глава про компилятор. Тут тебе и архитектура x86_64 с её регистрами и набором команд, и алгоритм компиляции AST в машинные инструкции, и формат ELF-64, и виртуальная память. И главное, результат — непортабельный компилятор для одной-единственной архитектуры. В сравнении с полстраницей кода интерпретатора это как-то слишком. Возникает вопрос: это всегда так просто писать интерпретаторы и так тяжело — компиляторы? Ответ — в общем, да. В компиляторе добавляется серьёзный этап — генерация кода (причём хороший компилятор умеет компилировать сразу под много архитектур). Зато как это прекрасно — машинные инструкции, и всё такое. :D

Кроме того, помни, что в нашем примере полностью отсутствует оптимизация промежуточного представления. С учётом оптимизаций разница в сложности компилятора и интерпретатора была бы не такой разительной.

4.5 Виртуальная машина: байткод

Байткод виртуальной машины будет представлять из себя запись оригинальной программы в обратной польской нотации (Reverse Polish Notation, RPN). Это постфиксная запись: сначала идут операнды, потом оператор. Например, программа “11 + (42 - 318) * 7” в RPN выглядит так: “11 42 318 - 7 * +”.

Виртуальная машина у нас будет *стековая*: у неё будет память в виде стека (для хранения промежуточных результатов), и она будет понимать команды:

- ADD — достать из стека два верхних числа, сложить и запихнуть в стек
- SUB — достать из стека два верхних числа, вычесть и запихнуть в стек
- MUL — достать из стека два верхних числа, умножить и запихнуть в стек
- DIV — достать из стека два верхних числа, поделить и запихнуть в стек
- NUMBER n — запихнуть в стек число n.

Это, в общем-то, всё: очень простая виртуальная машина. Вот компилятор программы в байткод (файл vm_bytecode.cpp):

```
1  #include "vm_bytecode.h"
2
3  void vm_bytecode (AST * ast, std::vector<Insn> & bytecode)
4  {
5      switch (ast->type)
6      {
7          case AST::NUMBER:
8              bytecode.push_back (Insn (ast->value.number));
9              return;
10         case AST::ADD:
11         {
12             vm_bytecode (ast->value.operands.left, bytecode);
13             vm_bytecode (ast->value.operands.right, bytecode);
14             bytecode.push_back (Insn (Insn::ADD));
15             return;
16         }
17         case AST::SUB:
18         {
19             vm_bytecode (ast->value.operands.left, bytecode);
20             vm_bytecode (ast->value.operands.right, bytecode);
21             bytecode.push_back (Insn (Insn::SUB));
22             return;
23         }
24         case AST::MUL:
25         {
26             vm_bytecode (ast->value.operands.left, bytecode);
```

```
            vm_bytecode (ast->value.operands.right, bytecode);
            bytecode.push_back (Insn (Insn::MUL));
            return;
        }
        case AST::DIV:
        {
            vm_bytecode (ast->value.operands.left, bytecode);
            vm_bytecode (ast->value.operands.right, bytecode);
            bytecode.push_back (Insn (Insn::DIV));
            return;
        }
    }
}
```

А вот его прототип (файл vm_bytecode.h):

```
#ifndef __VM_BYTECODE__
#define __VM_BYTECODE__

#include <vector>

#include "parser.h"

struct Insn
{
    enum Opcode {ADD, SUB, MUL, DIV, NUMBER} opcode;
    int number;

    Insn (Opcode op)
        : opcode (op)
        , number (0)
    { }

    Insn (int n)
        : opcode (NUMBER)
        , number (n)
    { }
};

void vm_bytecode (AST * ast, std::vector<Insn> & bytecode);

#endif // __VM_BYTECODE__
```

4.6 Виртуальная машина: интерпретатор байткода

Первый вариант реализации нашей виртуальной машины — это интерпретатор байткода. Для реализации стека в интерпретаторе используется `std::stack<int>`. Больше никаких хитростей нет, интерпретатор просто в цикле читает и исполняет инструкции. Вот код интерпретатора байткода (файл vm_run.cpp):

```
#include <stack>

#include "vm_run.h"

int vm_run (std::vector<Insn> & bytecode)
```

```
{
    std::stack<int> stack;
    for (unsigned int i = 0; i < bytecode.size (); ++i)
    {
        switch (bytecode[i].opcode)
        {
            case Insn::ADD:
            {
                int l = stack.top (); stack.pop ();
                int r = stack.top (); stack.pop ();
                stack.push (l + r);
                break;
            }
            case Insn::SUB:
            {
                int l = stack.top (); stack.pop ();
                int r = stack.top (); stack.pop ();
                stack.push (r - l);
                break;
            }
            case Insn::MUL:
            {
                int l = stack.top (); stack.pop ();
                int r = stack.top (); stack.pop ();
                stack.push (l * r);
                break;
            }
            case Insn::DIV:
            {
                int l = stack.top (); stack.pop ();
                int r = stack.top (); stack.pop ();
                stack.push (r / l);
                break;
            }
            case Insn::NUMBER:
                stack.push (bytecode[i].number);
                break;
        }
    }
    return stack.top ();
}
```

А вот его прототип (файл vm_run.h):

```
#include <vector>

#include "vm_bytecode.h"

int vm_run (std::vector<Insn> & bytecode);
```

4.7 Виртуальная машина: JIT-компилятор байткода

Второй вариант реализации виртуальной машины чуть посложнее — это JIT-компилятор байткода в нативный код x86_64. Но нам очень повезло: архитектура виртуальной машины — это подмножество архитектуры x86_64. У x86_64 есть готовый стек и инструкции работы с ним

PUSH и POP, есть все необходимые арифметические инструкции. Так что мы можем напрямую транслировать инструкции нашей виртуальной машины в инструкции x86_64.

Ну хорошо, скомпилировать код проблем нет. А вот как нам вызвать его, получить результат и вернуть управление в нашу программу? В ассемблере для этого есть команды передачи управления, принимающие адрес в памяти как аргумент. В C++ нет таких команд, и нам придётся немного схитрить: представить скомпилированный код как обычную C++ функцию без аргументов, возвращающую int:

```
int f ();
```

Затем вызвать эту функцию и получить результат программы:

```
int result = f ();
```

Для воплощения этого плана нам понядобятся три вещи:

- кусок памяти, доступной на исполнение
- указатель на функцию
- конвенция вызова (calling convention)

mmap

Мы должны хранить скомпилированный код в непрерывном массиве памяти, и эта память должна быть доступна на исполнение. Обычный malloc выделяет память, доступную на чтение и запись, но не на исполнение. Если мы попытаемся исполнить память, выделенную malloc-ом, операционная система прибьет нашу программу сегфолтом. Нам придётся выделять память по-другому: функцией mmap. Эта функция отображает часть виртуального адресного пространства процесса на реальную память, и принимает набор флагов, указывающих права доступа к выделенному куску памяти.

Указатель на функцию

В C++ есть такая штука как *указатель на функцию*. Например, если прототип функции выглядит вот так:

```
int f ();
```

То указатель на эту функцию можно создать так:

```
int (* pf) () = f;
```

То есть создаётся переменная pf типа int (*) (), которая инициализируется значением f. Это немного необычная запись: имя переменной засунуто в середину типа. Вызывать функцию по указателю можно так же, как и по обычному имени. Эти два вызова эквивалентны:

```
f ();
pf ();
```

В данном примере указатель на функцию проинициализирован другой функцией с такой же сигнатурой. Но ведь это не обязательно. Если у нас есть адрес `addr` скомпилированного кода, мы можем сделать, чтобы указатель на функцию показывал на `addr`. Для этого нам придётся привести тип `addr` к типу `int (*) ()`:

```
int (* pf) () = (int (*) ()) addr;
```

Вообще, преобразование типов — гадкая и нехорошая вещь. Его следует использовать с большой осторожностью, только в крайних случаях. Оно позволяет делать страшные вещи (например, передавать управление на какой-то странный кусок памяти, чем мы и заняты). Недаром в C++ есть целых четыре разных преобразования типов (`static_cast`, `const_cast`, `dynamic_cast`, `reinterpret_cast`) в дополнение к старому C-шному `()`.

Теперь мы можем спокойно передавать управление на наш код:

```
f ();
```

Конвенция вызова

Мало просто просто состряпать указатель на функцию, показывающий на код в исполняемой памяти, и передать на него управление. Надо ещё вернуться из этого кода и каким-то образом вернуть результат. Чтобы это сделать, надо прежде всего задуматься: во что транслируется вызов функции в C++?

В этом всём нам поможет знание *конвенции вызова*. Конвенция вызова, или соглашение о вызове — это договорённость о следующих особенностях реализации вызова подпрограммы:

- какую память использовать для передачи аргументов (регистры, стек, кучу, etc.)
- в каком порядке передавать аргументы (в прямом, в обратном, etc.)
- какую память использовать для возвращаемого значения (регистры, стек, кучу, etc.)
- кто должен чистить память после иполнения подпрограммы (вызывающий, вызываемый, оба)
- какие машинные команды использовать для передачи управления на подрограмму и возврата из неё

Конвенция вызова — это часть бинарного интерфейса программы (Application Binary Interface, ABI), то есть интерфейса взаимодействия программ на уровне машинного кода. (Есть ещё API, Application Programming Interface — это интерфейс взаимодействия программ на уровне исходного кода.)

В стандарте C++ конвенция вызова не определена — она зависит от компилятора. Обычно компилятор поддерживает несколько конвенций. Мы будем использовать конвенцию cdecl ("C-declaration"): аргументы передаются через стек, справа налево; очистку стека производит вызывающая программа; результат функции возвращается через регистр RAX. Это конвенция-по-умолчанию в большинстве компиляторов C++, поэтому её не обязательно указывать.

Нашей функции надо сделать две вещи: положить в RAX возвращаемое значение вернуть управление. Это делается двумя командами (учитывая, что результат выполнения программы у нас лежит на вершине стека):

```
pop rax
ret
```

Итак, мы будем хранить скомпилированный код в структуре следующего вида (файл code.h):

```
#ifndef __CODE__
#define __CODE__

class Code
{
    unsigned int size_max;
    unsigned int size;
    unsigned char * buffer;

  public:
    Code ();
    ~Code ();
    void save_byte (unsigned char byte);
    int exec ();
};

#endif // __CODE__
```

У структуры есть конструктор, деструктор и два метода: `void save_byte (unsigned char)` и `int exec ()`. Вот их реализация (файл code.cpp):

```
#include <string.h>
#include <sys/mman.h>

#include "code.h"

Code::Code ()
    : size_max (1024)
    , size (0)
    , buffer ((unsigned char *) mmap (0, size_max, PROT_READ | PROT_WRITE |
        PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0))
{ }

Code::~Code ()
{
    munmap (buffer, size_max);
}

void Code::save_byte (unsigned char byte)
{
    if (size >= size_max)
    {
        unsigned char * buffer_old = buffer;
        unsigned int size_max_old = size_max;
        size_max *= 2;
```

```
24          buffer = (unsigned char *) mmap (0, size_max, PROT_READ | PROT_WRITE |
                PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
25          memcpy (buffer, buffer_old, size_max_old);
26          munmap (buffer_old, size_max_old);
27      }
28      buffer[size++] = byte;
29  }
30
31  int Code::exec ()
32  {
33      int (* f) () = (int (*) ()) buffer;
34      return f ();
35  }
```

Заметь, что вся память выделяется через `mmap` (освобождается через `munmap` соответственно), а передача управления на скомпилированный код осуществляется при помощи указателя на функцию, который грубой силой настроили на начало кода.

Сам JIT-компилятор выглядит проще некуда (файл vm_jit.cpp):

```
1   #include "code.h"
2   #include "vm_jit.h"
3
4   int vm_jit (std::vector<Insn> & bytecode)
5   {
6       Code code;
7       for (unsigned int i = 0; i < bytecode.size (); ++i)
8       {
9           switch (bytecode[i].opcode)
10          {
11              case Insn::ADD:
12              {
13                  // pop rbx ; 5b
14                  code.save_byte (0x5b);
15                  // pop rax ; 58
16                  code.save_byte (0x58);
17                  // add eax, ebx ; 01 d8
18                  code.save_byte (0x01);
19                  code.save_byte (0xd8);
20                  // push rax ; 50
21                  code.save_byte (0x50);
22                  break;
23              }
24              case Insn::SUB:
25              {
26                  // pop rbx ; 5b
27                  code.save_byte (0x5b);
28                  // pop rax ; 58
29                  code.save_byte (0x58);
30                  // sub eax, ebx ; 29 d8
31                  code.save_byte (0x29);
32                  code.save_byte (0xd8);
33                  // push rax ; 50
34                  code.save_byte (0x50);
35                  break;
36              }
```

```
37              case Insn::MUL:
38              {
39                  // pop rbx ; 5b
40                  code.save_byte (0x5b);
41                  // pop rax ; 58
42                  code.save_byte (0x58);
43                  // imul ebx ; f7 eb
44                  code.save_byte (0xf7);
45                  code.save_byte (0xeb);
46                  // push rax ; 50
47                  code.save_byte (0x50);
48                  break;
49              }
50              case Insn::DIV:
51              {
52                  // pop rbx ; 5b
53                  code.save_byte (0x5b);
54                  // pop rax ; 58
55                  code.save_byte (0x58);
56                  // cdq ; 99, sign-extend edx:eax of eax
57                  code.save_byte (0x99);
58                  // idiv ebx ; f7 fb
59                  code.save_byte (0xf7);
60                  code.save_byte (0xfb);
61                  // push rax ; 50
62                  code.save_byte (0x50);
63                  break;
64              }
65              case Insn::NUMBER:
66              {
67                  // push <imm32> ; 68 <byte1> <byte2> <byte3> <byte4> (little-
                        endian)
68                  unsigned char byte1 = (bytecode[i].number >> (8 * 0)) & 0xFF;
69                  unsigned char byte2 = (bytecode[i].number >> (8 * 1)) & 0xFF;
70                  unsigned char byte3 = (bytecode[i].number >> (8 * 2)) & 0xFF;
71                  unsigned char byte4 = (bytecode[i].number >> (8 * 3)) & 0xFF;
72                  code.save_byte (0x68);
73                  code.save_byte (byte1);
74                  code.save_byte (byte2);
75                  code.save_byte (byte3);
76                  code.save_byte (byte4);
77                  break;
78              }
79          }
80      }
81      // pop rax ; 58
82      code.save_byte (0x58);
83      // ret ; c3
84      code.save_byte (0xc3);
85
86      return code.exec ();
87  }
```

Вот его прототип (файл vm_jit.h):

```
1  #include "vm_bytecode.h"
2
```

```
3  int vm_jit (std::vector<Insn> & bytecode);
```

4.8 Итоги

Что мы видели в этом примере?

Язык арифметических выражений. Формальную грамматику для этого языка и алгоритм избавления от левой рекурсии. Парсер, написанный по грамматике методом рекурсивного спуска. AST. Крайне простой интерпретатор, рекурсивно обходящий AST и вычисляющий результат. Чуть более сложный компилятор для архитектуры x86_64, генерирующий исполняемые файлы в формате ELF-64. Стековую виртуальную машину, понимающую арифметические выражения в RPN, и две её реализации: в виде простого интерпретатора и в виде JIT-компилятора для архитектуры x86_64.

Чего мы не видели в этом примере?

Формально определённой семантики. Типичной для языков программирования двухуровневой грамматики (лексической и синтаксической). Оптимизаций. Сложной взаимосвязи между разными фазами языкового процессора. Наконец, из примера не чувствуется разительных достоинств и недостатков разных подходов: захватывающей скорости скомпилированного машинного кода, удивительной гибкости изменяющихся на лету интерпретируемых программ, хитро адаптирующихся к жизни JIT-компиляторов и платформенно-независимого байткода виртуальных машин. Что действительно бросилось в глаза — так это трудность портирования компилятора на x86_64.

Ну что — на этом всё. :) Спасибо за (возможно, напрасно) потраченное внимание и время. :)