ПРОГРАММИРОВАНИЕ
введение в профессию

А.В.СТОЛЯРОВ

ЗАДАЧИ И ЭТЮДЫ

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова «Программирование: введение в профессию. Задачи и этюды», опубликованное в издательстве МАКС Пресс в 2022 году, называемое далее «Произведением», защищено действующим российским и международным авторско-правовым законодательством. Все права на Произведение, предусмотренные законом, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является противозаконным и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1) воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2) копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями оригинального файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искажённых экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через веб-сайты, оказывающие платные услуги, через сайты коммерческих компаний и индивидуальных предпринимателей (включая файлообменные и любые другие сервисы, организованные в Сети Интернет коммерческими компаниями, в том числе бесплатные), а также **через сайты, содержащие рекламу любого рода**; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, бесплатная запись данного файла на носители, принадлежащие другим пользователям, распространение данного файла через бесплатные децентрализованные файлообменные P2P-сети и т. п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

А. В. СТОЛЯРОВ

ПРОГРАММИРОВАНИЕ
ВВЕДЕНИЕ В ПРОФЕССИЮ

задачи и этюды

Москва — 2022

УДК 519.683+004.2+004.45
ББК 32.97
С81

Столяров, Андрей Викторович.

С81 Программирование: введение в профессию. Задачи и этюды / А. В. Столяров. – Москва : МАКС Пресс, 2022. – 156 с.
ISBN 978-5-317-06732-8

Сборник содержит задачи, упражнения и практические задания в поддержку учебника «Программирование: введение в профессию».

Для школьников, студентов, преподавателей и всех, кто интересуется программированием.

УДК 519.683+004.2+004.45
ББК 32.97

ISBN 978-5-317-06732-8

Оглавление

От автора

Вы держите в руках задачник в поддержку курса изучения программирования, изложенного в моей книге «Программирование: введение в профессию»; этот факт довольно примечателен сам по себе, поскольку изначально я совершенно не планировал никаких задачников и вообще задач. Сам я, изучая программирование в начале 1990-х, никогда не нуждался в том, чтобы кто-то ставил мне задачи или подсказывал этюды; напротив, идей, что́ бы такого ещё запрограммировать, у меня всегда было (и до сих пор есть) намного больше, чем имеется на это времени. Признаюсь, я грешным делом был уверен, что это у всех так, а у кого не так, те мою книжку читать не станут; но я ошибался.

Интересоваться, как там насчёт задач «для закрепления материала» или «для самопроверки», публика начала сразу после выхода первого тома первого издания «Введения в профессию». Конечно, поначалу я крепился, отбивался, держал марку и всё такое прочее, но в какой-то момент мою оборону всё-таки прорвали. В конце концов, по большей части материала книги я так или иначе вёл или семинары, или частные уроки, и у меня, естественно, успел накопиться запас наработанных задач, заданий и прочих заморочек; почему бы всем этим добром не поделиться с публикой, раз уж так хотят.

Есть только одна очень важная вещь, которую я хотел бы озвучить, прежде чем вы приметесь за собранные в этой книжке задачи. **Если у вас возникла идея, которую хочется реализовать в виде программы — немедленно бросайте задачник, хоть этот, хоть любой другой, и воплощайте свою идею.** Толку от этого будет заведомо больше. Вообще учиться программировать намного эффективнее, когда вы заставляете компьютер делать то, чего от него хотите сами, а не то, что вам предписано каким-нибудь очередным добрым дядей. Не беспокойтесь, работа на идиотов-заказчиков от вас никуда не денется, всё впереди; так что, пока есть возможность, занимайтесь программированием как искусством, а программирование как ремесло пусть пока подождёт.

Сказанное, впрочем, не относится к задачам в поддержку первой (вводной) части «Введения в профессию», где в основном математика; задачи, не предполагающие написания программ, встречаются и в других частях, но такого тут, прямо скажем, немного.

Стоит сказать несколько слов о том, как пользоваться этой книжкой. В тексте задачника встречаются ссылки на «главы» и «параграфы»; во всех таких случаях имеются в виду главы и параграфы трёхтомника «Программирование: введение в профессию», в той их нумерации, которая сложилась во втором издании. К сожалению,

нумерация глав и параграфов первого издания (того, в котором было четыре тома) несколько отличается. Впрочем, задачник можно использовать и с первым изданием тоже, найти нужный параграф обычно не так сложно, а если совсем не получается — воспользуйтесь электронной версией второго издания. Как можно заметить из оглавления, задачник снабжён рубриками «указания» и «ответы». Скажу сразу, ответами снабжены все задачи, **кроме** тех, которые подразумевают написание программы или её фрагмента; решение таких задач по понятным причинам всё равно невозможно «сверить с ответом», а проверить, правильно вы всё сделали или нет, проще будет обычным тестированием написанной программы. Для многих (но, опять же, не всех) задач в разделе «Указания» можно найти дополнительные подсказки. Иногда подсказки содержатся прямо в тексте задачи — это означает, что мне как автору задачника показалось бессмысленным тратить время читателя на самостоятельный поиск нужной информации или нужного подхода; но если подсказка вынесена в раздел указаний, то сначала стоит попробовать обойтись без неё, это может оказаться полезно. Подсказку, впрочем, стоит прочитать, даже если вы уже решили задачу — может оказаться, что решить её можно проще или в каком-нибудь смысле «правильнее».

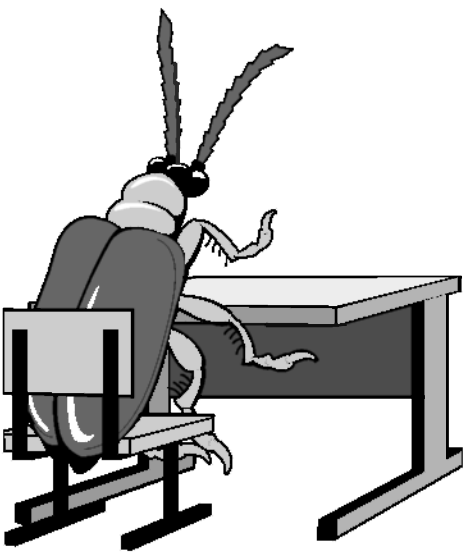
Ну и ещё один момент, который, возможно, разочарует моих коллег-преподавателей. Большинство сборников задач, какому бы предмету их ни посвящали, включают огромные массивы всевозможных примеров, часто однотипных, что позволяет организовать работу в классе или аудитории, предлагая подопечным одну задачу за другой до полного «освоения» (в действительности тут было бы уместно совсем другое слово) очередной темы, а потом ещё и провести контрольную работу, используя тот же задачник. Увы, на такое меня не хватило, да простят меня те, кто чего-то подобного ожидал. Сборник упражнений, который вы держите в руках, рассчитан на человека, осваивающего предмет самостоятельно в гордом одиночестве, «запасных» и похожих друг на друга задач тут, за небольшими исключениями, нет.

Издание этого задачника благополучно завершает проект, растянувшийся на семь лет. Пользуясь случаем, ещё раз благодарю всех тех, кто поддержал проект своими пожертвованиями; ниже приведён актуальный на 11 января 2022 г. список всех, кто присылал пожертвования, кроме тех, кто предпочёл сохранить инкогнито.

Gremlin, Grigoriy Kraynov, Шер Арсений Владимирович, Таранов Василий, Сергей Сетченков, Валерия Шакирзянова, Катерина Галкина, Илья Лобанов, Сюзана Тевдорадзе, Иванова Оксана, Куликова Юлия, Соколов Кирилл Владимирович, jeckep, Кулёва Анна Сергеевна, Ермакова Марина Александровна, Переведенцев Максим Олегович, Костарев Иван Сергеевич, Донцов Евгений, Олег Французов, Степан Холопкин, Попов Артём Сергеевич, Александр Быков, Белобородов И. Б., Ким Максим, artyrian, Игорь Эльман, Илюшкин Никита, Кальсин Сергей Александрович, Евгений Земцов, Шрамов Георгий, Владимир Лазарев, eupharina,

Николай Королев, Горошевский Алексей Валерьевич, Леменков Д. Д., Forester, say42, Аня «canja» Ф., Сергей, big_fellow, Волканов Дмитрий Юрьевич, Танечка, Татьяна 'Vikora' Алпатова, Беляев Андрей, Лошкины (Александр и Дарья), Кирилл Алексеев, kopish32, Екатерина Глазкова, Олег «burunduk3» Давыдов, Дмитрий Кронберг, yobibyte, Михаил Аграновский, Александр Шепелёв, G.Nerc=Y.uR, Василий Артемьев, Смирнов Денис, Pavel Korzhenko, Руслан Степаненко, Терешко Григорий Юрьевич 15e65d3d, Lothlorien, vasiliandets, Максим Филиппов, Глеб Семёнов, Павел, unDEFER, kilolife, Арбичев, Рябинин Сергей Анатольевич, Nikolay Ksenev, Кучин Вадим, Мария Трофимова, igneus, Александр Чернов, Roman Kurynin, Власов Андрей, Дергачёв Борис Николаевич, Алексей Алексеевич, Георгий Мошкин, Владимир Руцкий, Федулов Роман Сергеевич, Шадрин Денис, Панфёров Антон Александрович, os80, Зубков Иван, Архипенко Константин Владимирович, Асирян Александр, Дмитрий С. Гуськов, Тойгильдин Владислав, Masutacu, D.A.X., Каганов Владислав, Анастасия Назарова, Гена Иван Евгеньевич, Линара Адылова, Александр, izin, Николай Подонин, Юлия Корухова, Кузьменкова Евгения Анатольевна, Сергей «GDM» Иванов, Андрей Шестимеров, vap, Грацианова Татьяна Юрьевна, Меньшов Юрий Николаевич, nvasil, В. Красных, Огрызков Станислав Анатольевич, Бузов Денис Николаевич, capgelka, Волкович Максим Сергеевич, Владимир Ермоленко, Горячая Илона Владимировна, Полякова Ирина Николаевна, Антон Хван, Иван К., Сальников Алексей, Щеславский Алексей Владимирович, Золотарев Роман Евгеньевич, Константин Глазков, Сергей Черевков, Андрей Литвинов, Шубин М. В., Сыщенко Алексей, Николай Курто, Ковригин Дмитрий Анатольевич, Андрей Кабанец, Юрий Скурский, Дмитрий Беляев, Баранов Виталий, Новиков Сергей Михайлович, maxon86, mishamm, Спиридонов Сергей Вячеславович, Сергей Черевков, Кирилл Филатов, Чаплыгин Андрей, Виктор Николаевич Остроухов, Николай Богданов, Баев Ален, Плосков Александр, Сергей Матвеев a.k.a. stargrave, Илья, aykar, Олег Бартунов, micky_madfree, Алексей Курочкин aka kaa37, Николай Смолин, I, JDZab, Кравчик Роман, Дмитрий Мачнев, bergentroll, Иван А. Фролов, Александр Чащин, Муслимов Я. В., Sedar, Максим Садовников, Яковлев С. Д., Рустам Кадыров, Набиев Марат, Покровский Дмитрий Евгеньевич, Заворин Александр, Павлочев Сергей Юрьевич, Рустам Юсупов, Noko Anna, Андрей Воронов, Лисица Владимир, Алексей Ковура, Чайка Леонид Николаевич, Коробань Дмитрий, Алексей Вересов, suhorez, Ольга Сергеевна Цаун, Боборыкин Сергей Владимирович, Олохтонов Владимир, Александр Смирницкий, Максим Клочков, Анисимов Сергей, Вадим Вадимович Чемодуров, rumiantcev, babera, Артём Коротченко, Евгений Шевкунов, Александр Смирницкий, Артём Шутов, Засеев Заурбек, Konstantin Slobodnyuk, Yan Zaripov, Виталий Бодренков, Александр Сергиенко, Денис Кузаков, Пушистый Шмель, Сергей Спивак, suuuumcaa, Гагарин, Валерий Гайнуллин, Александр Махаев (mankms), VD, А. Б. Лихачёв, Col_Kurtz, Espanka, Дмитрий Сергеевич Х., Анатолий Камчатнов, Табакаев Евгений Владимирович, Александр Трошенков, Малюга Андрей, Сорокин Андрей, Буркин Иван, Александр Логунов, moya_bbf3, Vilnur_Sh, Кипнис Александр, Oleg G. Geier, Владимир Исаев (fBSD), Филимонов Сергей Владимирович, vsudakou, AniMath, Данилов Евгений, Воробьёв В. С., mochalov, Камчатская LUG, Логинов Сергей Владимирович, Чистяков Артём Андреевич, A&A Сулимовы, Денисов Денис Юрьевич, Сутупов Андрей Михайлович, kuddai, Озерицкий Алексей Владимирович, alexz, Владимир Цой, Владимир Бердовщиков, Сергей Дмитриченко, Иванцов Данил Игоревич, Замыслов Д. А., Владимир Халямин, Максим Карасев (begs), ErraticLunatic, А. Е. Артемьев, FriendlyElk, Алексей Спасюк, Андриевский Константин Андреевич, Сукачёв Владислав Вячеславович, Артём Абрамов, maxon86, Соколов Павел Андреевич, Алексей Н, Никита Гуляев, Бодюль Евгений, rebus_x, Рачинский Максим Юрьевич, Мезенцев Андрей Владимирович, Ижмяков Павел Андреевич, Денис К., Умнова Наталья Евгеньевна, Артём Жилеев, Легоцкий Николай Евгеньевич, Виталий Саган, Александр Д., Иванов Павел Николаевич, Роман Викторович Щербинин, Константин Хомутов, barsik, Чистяков Сергей, Дмитрий Нурисламов, Евгений lipklim Климов, Азаров Владимир, Иванов Виталий Сергеевич, Бичекуев Аслан Расулович, Владимир Бахтин, Иванцов Данил Игоревич, Смирнов Иван, lexizz19, Содномов Алексей Эрдэни-Баирович, yshawn, Авдеенков Василий, Андрей, Игорь, Денис Николаевич Ж., Aglaro, aoratos, Шарапов Кирилл Владимирович, Александр, Нестеренко Владислав Валерьевич, Рябчиков Дмитрий Александрович, Андрей Александрович Герман, pashak, Морозов Михаил Павлович, Тихонов Руслан Владимирович, Копкин Максим, ЛАГ-5891, Евгений Феликсович Набоков, Плотников Сергей, Худяев Глеб, Максим Шаталин, Пётр Захаров, Михаил, Тимофей Дубский, Андрей Анатольевич Бородин, gunpi, Зеленин Андрей, Плаксицкая Наталия, enoelia, Юрий Сидоров, Dizzy, Дмитрий Серегин, Михаил К., Павел Егоров, Даниил Шелепанов, Дьяконов Олег, Данилов Роман Викторович, le34, Константин Плахетка, Илья Денисов, Вадим Марченко, Ратников Сергей Валерьевич, Илья Алексеевич Белоусов, Кузьмин Ринат, savtog, Валеев Эдуард Накипович, paq1311, e_matveev, Ефремов Артур, братишка, Иван Михайлович Пузырёв, Jamezzz, Мелкумян Завен, Игорь Романович Шульга, Сергей Спивак, Василий Дзюбенко, sadsnake, Игорь Середов, Pavel Heiden

З А Д А Ч И



К части 1 «предварительные сведения»

Командная строка Unix

1.01. Мы находимся в директории `/opt/light/foo`, т. е. она является для нас текущей; известно, что на одном уровне с директорией `foo` находится директория `bar`, а в ней есть файл `f1.txt`. Как будет выглядеть абсолютное имя этого файла? Каково кратчайшее *относительное* (т. е. отсчитываемое от текущей директории) имя для обращения к этому файлу?

1.02. Пользователь дал следующие команды:

```
mkdir tetris
cp tetris.pas tetris
cd tetris
ls
```

Все команды отработали успешно. Что напечатала последняя из них?

1.03. Какая директория стала текущей после команд

```
cd /usr/local/bin/hunter
cd ../../lib
cd hunter/run
```

если известно, что все команды отработали успешно?

1.04. Какая директория станет текущей, если дать команды

```
cd ..
cd ../..
cd ../john
cd work/tetris/src
cd ..
```

и все они отработают успешно, а текущей изначально была директория `/home/lizzie/work/pas/tetris`?

1.05. Почему в условиях задачи 1.03 не указано, какая директория была текущей изначально?

1.06. Что означают права доступа к обычному файлу, заданные следующими числами?
a) `0700` b) `0440` c) `0511` d) `0660` e) `0644` f) `0555`

1.07*. Что означают права доступа к обычному файлу, заданные следующими числами?
a) `04711` b) `06751` c) `02550`

1.08. Что означают права доступа к директории, заданные следующими числами? Имеет ли смысл устанавливать такие права, а если нет, то какие права доступа следовало бы поставить вместо них?
a) `0700` b) `0711` c) `0751` d) `0733` e) `0744` f) `0660`

1.09*. Что означают права доступа к директории, заданные следующими числами? В каких ситуациях такие права доступа могут потребоваться?
a) `01770` b) `03777`

Следующие задачи посвящены скриптовому программированию на языке командного интерпретатора (Bourne Shell); для их решения потребуется владение материалом, который в книге объявлен как «необязательный» (см. §1.2.15[1]) — набран уменьшенным шрифтом и предваряется комментарием о причинах. Повторим эту мысль здесь: начинать изучение программирования с языка Bourne Shell — идея крайне неудачная, поэтому, если вы раньше ни на каких языках не программировали, правильнее будет пропустить и сами эти задачи, и нужный для них фрагмент книги, и вернуться к ним позднее, когда у вас уже будет опыт программирования, путь даже небольшой.

1.10. Напишите скрипт на Bourne Shell, принимающий ровно два аргумента командной строки, которые должны быть целыми числами (назовём их N и M). Скрипт должен выдать на печать M натуральных чисел подряд, начиная с числа N; числа должны быть напечатаны через пробел, а в конце следует выдать перевод строки. Например, если заданы числа `15` и `5`, ваш скрипт должен напечатать:

```
15 16 17 18 19
```

1.11. Напишите скрипт на Bourne Shell, который выдаёт (в одну строку) столько символов «`@`», сколько имеется файлов в текущем каталоге.

[1] Напомним, что ссылки здесь и далее — на главы и параграфы трёхтомника «Программирование: введение в профессию».

1.12. В языке Bourne Shell сочетание символов «`$*`» означает «все параметры командной строки, кроме "нулевого"» (т. е. кроме имени самого скрипта). Добавим к этому, что команда «`date +%s`» выдаёт текущее время в виде числа секунд, прошедших с 1 января 1970 г. Зная это, напишите скрипт, который запускает на выполнение программу, имя и аргументы которой заданы параметрами командной строки, и после её завершения печатает число секунд реального времени, которое прошло с момента её запуска (погрешность в одну секунду в любую сторону считаем допустимой).

1.13. Усовершенствуйте скрипт из предыдущей задачи так, чтобы он завершался с тем же кодом, с которым завершилась запущенная им программа.

1.14. Напишите скрипт, получающий ровно один параметр — имя директории, которую нужно создать; создав её, скрипт должен разместить в этой директории символические ссылки на каждый из файлов, имеющихся в текущей директории (за исключением свежесозданной директории, если она создана внутри текущей), причём имена ссылок должны быть образованы из слова `file` и порядкового номера (`file1`, `file2`, ..., `file124`, ...). Если по каким-то причинам директорию не удаётся создать (например, она уже существует), скрипт должен выдать сообщение об ошибке и завершиться, не создавая никаких ссылок.

Комбинаторика

1.15. У Коли есть четыре воздушных шарика разных цветов, и ему хочется взять некоторые из них (или даже все) на прогулку. Сколькими способами он может выбрать шарики, если идти совсем без шариков ему не хочется?

1.16*. У Димы есть белые, синие, красные и зелёные воздушные шарики, причём шариков каждого цвета есть столько, до скольки Дима считать ещё не научился (можете предполагать, что шариков каждого цвета есть сколько угодно). Дима хочет взять с собой на прогулку какие-нибудь шарики, совсем без шариков ему гулять скучно, а больше трёх шариков он не удержит. Сколькими способами он может выбрать шарики?

1.17. В забеге участвуют десять бегунов, примерно равных по подготовке, так что никто не может предсказать заранее, кто кого победит. Сколькими разными способами могут распределиться призовые места? Под «призовыми» подразумеваются первые три места; считайте, что показать совершенно одинаковые результаты два бегуна не могут.

1.18. На подоконнике стоят кувшин, горшок с кактусом, статуэтка и две совершенно одинаковые фарфоровые вазы. Сколькими отличимыми друг от друга способами можно эти предметы выстроить в один ряд?

1.19. У командира есть две сигнальные ракетницы, а также зелёные, красные и белые ракеты. Сколько он может подать разных сигналов, если сигнал подаётся либо одной ракетой, либо ракетами, запускаемыми одновременно? Сколько сигналов можно будет подать, если ракетниц будет три?

1.20. На столе лежат восемь разных игрушек. К столу подошли Вася, Петя и Маша; каждый из них может взять себе какие-то игрушки (в том числе не взять ни одной); если какая-то игрушка никому не понравится, она останется на столе. Сколькими способами могут распределиться игрушки?

1.21. В шкатулке лежат пять разных кулонов. Отправляясь в отпуск, Катя решила взять с собой не меньше двух из своих кулонов, но при этом хотя бы один оставить в шкатулке. Сколькими способами она может это сделать?

1.22. Сколькими способами на пустой шахматной доске можно разместить две одинаковые ладьи так, чтобы они защищали друг друга (то есть любая из них могла пойти в позицию, в которой находится другая)?

1.23. Обруч разделён на три одинаковых сектора, каждый из которых нужно покрасить в один из пяти цветов. Сколькими способами можно это сделать так, чтобы все три использованных цвета были разными?

1.24. В деятельности клуба туристов-водников принимают участие 9 мужчин и 5 женщин. Сколькими способами клуб может выставить на соревнования три смешанных (т. е. состоящих из мужчины и женщины) экипажа двухместных байдарок? Экипажи равноправны.

1.25. В рабочей группе из десяти человек, пять из которых имеют высшее образование, нужно назначить руководителя и трёх его заместителей. Сколькими способами это можно сделать, если руководителем может быть только человек с высшим образованием, заместителем можно назначить кого угодно, и все три заместителя равноправны?

1.26. У мастера есть бусины пяти разных видов, причём бусин каждого вида имеется сколько угодно. Сколько можно сделать разных бус, составленных из семи бусин, если бусы должны быть симметричны?

1.27*. У мастера есть семь разных видов бусин; сколько он может сделать различных украшений, если на каждых бусах должно быть

ровно 17 бусин, бусы должны быть симметричны, в каждых бусах нужно применять не более трёх разных видов бусин, а одинаковые бусины не должны располагаться рядом?

1.28*. Чему равно C_{777}^2?

Системы счисления

1.29. Запишите следующие десятичные числа в двоичной системе счисления:
a) 91 b) 149 c) 344 d) 660 e) 881 f) 1100

1.30. Запишите следующие числа в двоичной системе счисления, воспользовавшись их близостью к степеням двойки:
a) 1 b) 2 c) 15 d) 33 e) 67 f) 126 g) 252

1.31. Переведите в десятичную систему счисления следующие числа, записанные в двоичной системе:
a) 1011001 b) 1101010 c) 1001110111 d) 1100010010
e) 111001100110 f) 101000010001

1.32. Запишите число $35 \cdot 2^{10} + 3$ в двоичной системе, по возможности не производя никаких арифметических действий (во всяком случае, умножение и деление для этого совершенно ни к чему).

1.33. Число 30! (факториал тридцати) записали в двоичной системе. Оказалось, что эта запись содержит довольно много разрядов, причём младшие n из них — нули; иначе говоря, двоичная запись числа 30! заканчивается некоторым количеством нулей. Каким конкретно?

1.34. Следующие числа, записанные в шестнадцатеричной системе, запишите в двоичной системе, **не переводя в десятичную**:
a) `2F` b) `8A3` c) `77777` d) `1234ABCD` e) `5A5A`

1.35. Следующие числа, записанные в восмеричной системе, запишите в двоичной системе, **не переводя в десятичную**:
a) `27` b) `453` c) `33333` d) `123567` e) `6363`

1.36. Следующие числа, записанные в двоичной системе, запишите в восьмеричной и шестнадцатеричной системах, **не переводя в десятичную**:
a) `11101` b) `1111111111` c) `111110100000` d) `1000000000001`

1.37. Имеется запись некоторого числа в девятичной системе: 286501348675_9. Запишите это число в троичной системе. Можно ли это сделать, не используя деление и умножение?

1.38. Представьте в форме обыкновенных дробей следующие периодические дроби (в этой задаче используется только десятичная

система счисления)[2]:

a) 0,(63) b) 3,5(27) c) 7,2(325)

1.39. Переведите следующие периодические двоичные дроби в десятичную систему и запишите их в виде несократимых обыкновенных дробей:

a) 0,(1001) b) 0,(110) c) 11,0(011) d) 0,1(0110) e) 0,(000100111011)

1.40. Следующие дроби записаны в десятичной системе счисления. Представьте их в виде двоичных дробей (возможно, периодических):

a) $\frac{2}{5}$ b) $\frac{1}{6}$ c) $\frac{5}{7}$ d) $\frac{7}{9}$ e) $\frac{9}{11}$

1.41*. Фродо и Сэм попали в кладовую с эльфийскими верёвками. Эльф, охраняющий кладовую, сказал, что готов подарить им 333 метра верёвки, не больше, но и не меньше, поскольку так велела великая Галадриэль. Кроме того, эльф сказал Фродо и Сэму, что эльфийская верёвка почти живая и поэтому очень не любит, когда её измеряют, но, пожалуй, 13 измерений как-нибудь потерпит, а дальше начнёт сопротивляться и откажется служить тем, кто с ней так нехорошо поступил. Фродо вспомнил, что его размах рук в точности равен одному метру, а Сэм сообразил, что, используя уже отмеренную верёвку и прикладывая к ней верёвку, которая ещё не отмерена, можно увеличить её количество вдвое за одно измерение. Могут ли друзья отмерить себе ровно 333 метра верёвки за отведённые им 13 измерений?

1.42*. Для условий предыдущей задачи опишите, как отмерить произвольное количество верёвки, кратное одному метру, за минимальное число измерений.

Двоичная логика

Здесь и далее используются те же обозначения логических операций, как и в §1.3.3: конъюнкция обозначена символом $\&$, дизъюнкция — символом $\vee$, отрицание — $\overline{x}$, и т. д. Символ конъюнкции в некоторых случаях опущен, так что xy — это то же самое, что $x \ \& \ y$.

1.43. Упростите выражения:

a) $x \vee \overline{x}y$ b) $xy \vee \overline{x}y$ c) $xy \vee \overline{x}y \vee x\overline{y}$

1.44. Перепишите следующие выражения, используя только конъюнкцию, дизъюнкцию и отрицание:

[2]На всякий случай напомним, что запись с круглыми скобками обозначает бесконечную периодическую дробь, а цифры в скобках — это её период, т. е. как раз те цифры, которые в записи дроби повторяются до бесконечности; например, 0,1(25) означает дробь 0,125252525...

a) $x \vee (x \oplus y)$ b) $x \& (x \oplus y)$ c) $x \vee (x \equiv y)$ d) $x \& (x \equiv y)$
e) $x \vee (x \to y)$ f) $x \& (x \to y)$ g) $y \vee (x \to y)$ h) $y \& (x \to y)$
i) $(x \to y) \vee (y \to x)$ j) $(x \to y) \& (y \to x)$

1.45. Упростите выражения:

a) $x \vee \overline{(x \& y)}$ b) $x \& \overline{(x \& y)}$ c) $x \& \overline{(x \vee y)}$ d) $x \vee \overline{(x \vee y)}$

1.46. Как известно, среди булевых функций двух переменных есть две функции, через каждую из которых, используя только её одну (и имена переменных, но не константы — константы тоже являются функциями), можно выразить любые булевы функции (не только двух переменных, вообще любые). Эти две «хитрые» функции называются стрелкой Пирса и штрихом Шеффера. Стрелка Пирса обозначается символом «$\downarrow$»; она истинна тогда и только тогда, когда ложны оба её аргумента — иначе говоря, стрелка Пирса представляет собой отрицание дизъюнкции: $x \downarrow y = \overline{x \vee y}$.

Используя только стрелку Пирса, обозначения переменных x и y и круглые скобки, запишите каждую из остальных 15 булевых функций двух переменных.

1.47. Штрих Шеффера обозначается символом «$|$»; выражение $x|y$ истинно всегда, кроме случая, когда одновременно истинны оба его аргумента, т. е. штрих Шеффера представляет собой отрицание конъюнкции: $x \,|\, y = \overline{x \& y}$.

Запишите каждую из остальных 15 булевых функций двух переменных, используя только штрих Шеффера, обозначения переменных x и y и круглые скобки.

1.48. Для следующих выражений постройте таблицы истинности и определите, какие получились функции:

a) $x \oplus \overline{y}$ b) $x \& (x \to y)$ c) $\overline{y} \& (x \oplus y)$ d) $(x \,|\, y) \& (x \equiv y)$ e) $(x \downarrow \overline{y})$

1.49. Под ***совершенной дизъюнктивной нормальной формой*** (СДНФ) понимается представление булевой функции n переменных в виде дизъюнкции (логической суммы) членов, каждый из которых представляет собой конъюнкцию, в которую входит для *каждой* из n переменных либо она сама, либо её отрицание. Например, для $f(x, y) = x \to y$ СДНФ будет такой: $\overline{x}\,\overline{y} \vee \overline{x}y \vee xy$; для $g(x, y) = x > y$ СДНФ состоит всего из одного «слагаемого»: $x\,\overline{y}$. Обратите внимание, что в каждый член входят (с отрицанием или без) обе переменные x и y; для функции от трёх переменных x, y и z в каждый член СДНФ входили бы все эти три переменные.

В общем случае (для произвольной функции) СДНФ состоит из стольки членов, сколько единиц в правой части её таблице истинности; каждый такой член формируется, исходя из левой части соответствующей строки таблицы истинности: переменные, которые в этой строке равны единице, входят в конъюнкт сами по себе, а те, что равны нулю, берутся с отрицанием. Например, функция

$h(x, y, z) = z(x \oplus y)$ будет истинна (равна единице) на двух наборах: $(0, 1, 1)$ и $(1, 0, 1)$; следовательно, её СДНФ будет состоять из двух членов: $\overline{x}yz \vee x\overline{y}z$.

Постройте таблицы истинности для следующих выражений и представьте их в совершенной дизъюнктивной нормальной форме:

a) $(x \to y) \oplus (x \to z)$ b) $(x \vee y)(x \to (yz))$

1.50. Имеется 500 логических (булевых) переменных, обозначенных $x_1, x_2, x_3, \ldots, x_{500}$. Сколько решений имеет каждая из следующих систем уравнений?

$$\begin{cases} x_1 \oplus x_2 = 1 \\ x_2 \oplus x_3 = 1 \\ x_3 \oplus x_4 = 1 \\ \ldots \\ x_{499} \oplus x_{500} = 1 \end{cases} \qquad \begin{cases} x_1 \oplus x_2 = 0 \\ x_2 \oplus x_3 = 0 \\ x_3 \oplus x_4 = 0 \\ \ldots \\ x_{499} \oplus x_{500} = 0 \end{cases}$$

Какие это решения?

1.51*. Логическая функция называется *существенно зависящей от переменной* x, если существует хотя бы одна такая пара наборов значений всех переменных, что функция на этих наборах принимает противоположные значения, а сами наборы различаются только значением переменной x, все остальные переменные в этих наборах имеют одинаковые значения. Так, конъюнкция и дизъюнкция существенно зависят от обеих своих переменных, функция $f(x, y) = \overline{x}$ существенно зависит только от x, а константы не являются существенно зависимыми ни от одной переменной. Сколько существует логических функций от двух переменных, существенно зависящих от *каждой* своей переменной? А сколько таких функций от трёх переменных?

Количество информации

1.52. В некоторой игре, чтобы сделать ход, игрок должен бросить три игральные кости (обычные кубики, грани которых помечены числами от 1 до 6), причём правилами игры установлено, что учитываются только выпавшие единицы и двойки, а кости, упавшие вверх гранями от 3 до 6, дают ноль очков. Какова информационная ценность сообщений, что игрок в результате своего броска:

a) получил ноль очков;
b) получил ненулевое количество очков;
c) получил шесть очков;
d) получил одно очко?

1.53. Игральная кость для некоторой настольной игры выполнена в виде додекаэдра — правильного 12-гранника, каждая грань которого представляет собой правильный пятиугольник; считается, что падения этой кости любой из её граней вверх равновероятны. В связи с особенностями правил игры три из 12 граней пусты (считается, что их выпадение соответствует нулю очков), остальные грани помечены числами от 1 до 9. Какова информационная ценность сообщения «результат броска кости — ноль очков»?

1.54. Бросив игральную кость, описанную в предыдущей задаче, игрок получил больше трёх очков. Какова информационная ценность сообщения об этом?

1.55. Устроители благотворительной лотереи выпустили миллион билетов и сообщили, что будет разыграно 15625 призов (каждый билет может выиграть один приз или не выиграть ни одного). Зная это, Сергей купил два билета. Через некоторое время ему сообщили, что оба его билета выиграли. Сколько бит информации он при этом получил?

1.56. На участие в забеге на 10000 метров заявились 27 спортсменов, среди них был Виктор, за которого болели Маша и Оля. Оля не смогла прийти на соревнования, поэтому Маша сообщала ей о происходящем, посылая SMS-сообщения. Сначала Маша сообщила, что 13 спортсменов из числа заявившихся не вышли на старт, но Виктор благополучно стартовал. Когда забег был в самом разгаре, от Маши пришло сообщение, что на дистанции один из спортсменов сделал другому подножку, так что тот второй упал и вынужден был сойти с дистанции, а виновника происшествия судьи дисквалифицировали. Ещё через какое-то время Маша написала, что ещё шестеро спортсменов сошли с дистанции по разным причинам, а Виктор благополучно добежал до финиша, но табло на стадионе сломалось и зрители не знают, кто с каким результатом финишировал, так что придётся ждать награждения.

Какова информационная ценность заключительной SMS-ки от Маши, в которой сообщалось, что Виктора, увы, нет на пьедестале, если никакой информации об исходном уровне подготовки спортсменов Оля не получала?

1.57. В турнире участвовали восемь шахматистов, имеющих примерно одинаковый рейтинг. Один из зрителей после подведения итогов записал на бумажке фамилии участников турнира, занявших три призовых места. Сколько информации получит его знакомый, для которого предназначается бумажка, если список всех участников, сформированный до начала турнира, у него уже есть?

К части 2 «язык Паскаль и начала программирования»

Закрепление базовых понятий

2.01. Для каждого из следующих выражений определите его значение и тип:

a) `2+3` b) `2+3*10` c) `5-(6-7)` d) `18/6` e) `15/6`
f) `18 div 6` g) `18 mod 6` h) `17 div 3` i) `17 mod 3` j) `4 mod 7`
k) `2/1 + 3/1` l) `3 < 4` m) `2 = 1.0 + 1.0` n) `3 <> 4-1`

2.02. Какие значения будут в переменных `a` и `b`, имеющих тип `integer`, после выполнения следующих присваиваний?

```
a := 3;
b := 5;
a := b;
b := a;
```

2.03. Что напечатает следующая программа?

```
program demo1;
var
    x, y, z: integer;
begin
    x := 5;
    y := 10;
    z := x + y;
    x := z * 3;
    writeln(x)
end.
```

2.04. Переменная `x`, имеющая тип `integer`, содержит число 7; какое число она будет содержать после выполнения следующих присваиваний?

```
x := x + 3;
x := 100 - x;
x := 10 * x + x;
```

2.05. Что будет напечатано в результате выполнения следующих фрагментов программы?

a)
```
i := 0;
while i < 10 do
begin
    writeln('Hello');
    i := i + 1
end
```

b)
```
i := 1;
while i < 20 do
begin
    writeln('Good Bye');
    i := i + 1
end
```

c)
```
i := 15;
while i < 27 do
begin
    writeln('abrakadabra');
    i := i + 1
end
```

d)
```
i := 40;
while i > 25 do
begin
    writeln('foobar');
    i := i - 1
end
```

e)
```
i := 5;
while i < 104 do
begin
    writeln('Johnny be good!');
    i := i + 10
end
```

f)
```
i := 12;
while i > 22 do
begin
    writeln('abcdefgh');
    i := i + 1
end
```

2.06. Что будет напечатано в результате выполнения следующих фрагментов программы?

a)
```
i := 0;
repeat
    writeln('Hello');
    i := i + 1
until i > 10;
```

b)
```
i := 1;
repeat
    writeln('Good Bye');
    i := i + 1
until i > 20;
```

c)
```
i := 12;
repeat
    writeln('abrakadabra');
    i := i - 1
until i > 0;
```

d)
```
i := 12;
repeat
    writeln('abcdefgh');
    i := i + 1
until i < 100;
```

Сравните ваши результаты с результатами предыдущей задачи.

2.07*. Переменную, которая меняет своё значение на каждой итерации цикла и на основании значения которой принимается решение, продолжать цикл или прекратить, часто называют ***счётчиком***

цикла. Как вы считаете, в каких примерах из задач 2.05 и 2.06 переменную `i` можно с полным на то основанием называть «счётчиком» и что конкретно она считает?

2.08. Вернитесь к задаче 2.05 и те примеры, которые это позволяют, перепишите с использованием цикла `for` так, чтобы переменная цикла пробегала те же самые значения. Какие из примеров не допускают такого переписывания и почему?

Побитовые операции

2.09. Каковы значения следующих выражений?

a) `99 and 78` b) `99 or 78` c) `99 xor 78` d) `not 0`
e) `not -13` f) `not 177` f) `1 shl 7` g) `7 shl 2`
h) `255 shr 4` i) `240 shr 3` j) `42 shr 3` l) `-2 shl 2`

2.10. Переменные `x` и `y` имеют тип `longword` (32-битное беззнаковое целое). Какие в них будут значения после выполнения следующих операторов?

```
x := $abcdef57;
y := $12346891;
x := ((x shr 8) and $ffff0000) or ((y shl 8) and $ffff);
y := ((y and $ff0000) shr 16) or ((y and $ff) shl 8);
```

Напомним, знак «$» в Паскале означает шестнадцатеричную систему счисления; ответ дайте также в виде шестнадцатеричных чисел.

2.11. Напишите функцию, которая принимает параметром число типа `longint` и возвращает целое число — количество единиц в двоичном представлении параметра.

Задачи на циклы и процедуры

Последующие задачи стоит пробовать решать не раньше, чем вы справитесь с содержанием главы 2.3, рассказывающей о подпрограммах, или хотя бы с первым параграфом из неё. **Обязательно** применяйте процедуры при решении задач, исключая дублирование кода и упрощая структуру программы, иначе эти задачи окажутся практически бесполезны.

2.12. Задействовав процедуру `PrintChars`, разобранную в §2.3.1 в качестве одного из примеров, напишите две программы, по смыслу аналогичные программе `DiamondProc` (эта программа разобрана в том же параграфе чуть раньше) и точно так же, как она, использующие построчный вывод, но немного отличающиеся от неё: пусть одна

из них печатает «ромбик», он же «алмаз», *заполненный* звёздочками, а вторая — «алмаз» из пробелов на фоне, заполненном звёздочками. Так, если пользователь введёт число 5, ваши программы должны напечатать следующее:

```
    *                       *******
   ***                      *** ***
  *****                     **   **
   ***                      *     *
    *                       **   **
                            *** ***
                            *******
```

(слева показан вывод первой программы, справа — второй). Обязательно убедитесь, что в вашей программе нет одинаковых фрагментов кода, если они есть — вынесите их в процедуры.

2.13. Напишите программу, запрашивающую у пользователя два числа: высоту «алмаза» и желаемое количество «алмазов», и выдающая (построчно!) фигуру из звёздочек, содержащую заданное число алмазов заданной высоты, расположенных горизонтально и отстоящих друг от друга на один пробел. Так, если пользователь ввёл числа 7 и 6, получиться должно следующее:

```
   *       *       *       *       *       *
  * *     * *     * *     * *     * *     * *
 *   *   *   *   *   *   *   *   *   *   *   *
*     * *     * *     * *     * *     * *     *
 *   *   *   *   *   *   *   *   *   *   *   *
  * *     * *     * *     * *     * *     * *
   *       *       *       *       *       *
```

2.14. Напишите программу, выдающую состоящую из «звёздочек» букву Z с горизонтальной поперечиной, имеющую заданную (введённую пользователем) высоту (и такую же ширину). Например, для введённого числа 7 результат должен быть таким:

```
*******
     *
    *
*******
  *
 *
*******
```

Если пользователь пытается ввести чётное число или число меньше пяти, выдавайте сообщение об ошибке и просите пользователя повторить ввод.

2.15*. Усовершенствуйте программу из задачи 2.14, чтобы она запрашивала два числа (высоту буквы и количество таких букв) и печатала нужное количество букв Z заданной высоты, причём каждая следующая буква Z должна отстоять от предыдущей на один

пробел по горизонтали и на половину высоты буквы по вертикали. Например, для чисел 7 и 3 картина должна получиться такая:

```
*******
     *
    *
******* *******
  *           *
 *           *
******* ******* *******
          *           *
         *           *
        ******* *******
                  *
                 *
                *******
```

Символы

2.16. Напишите программу, которая читает из стандартного потока ввода строку (используйте переменную типа `string` и оператор `readln`) и печатает те *непробельные* символы из этой строки, которые в ней встречаются больше одного раза. Каждый такой символ должен быть напечатан ровно один раз; печатать их следует в том порядке, в котором они *в первый раз* встречаются во введённой строке. Не забудьте про перевод строки в конце вашего вывода. Например, работа программы может выглядеть так:

```
lizzie@host:~$ ./prog
Humpty Dumpty sat on a wall
umptyal
lizzie@host:~$
```

2.17. В §2.5.1 рассматривается программа `PrintAscii`, печатающая часть ASCII-таблицы в виде семи строк по 16 символов в каждой, причём коды символов нарастают вдоль горизонтальных строк (на единицу при переходе вправо на один столбец), а при переходе к следующей строке увеличиваются на 16. Переделайте эту программу (или напишите свою с чистого листа) так, чтобы она печатала таблицу из 8 строк по 14 символов в каждой, причём коды символов нарастали на единичку при движении по вертикали (вниз на одну строку), а при переходе к соседнему столбцу увеличивались на восемь — иначе говоря, чтобы таблицу нужно было читать не слева направо строка за строкой, а сверху вниз столбец за столбцом.

Посимвольный ввод, программы-фильтры

2.18. В §2.5.2 рассматривается процедура `ReadLongint`, позволяющая прочитать с клавиатуры десятичную запись числа и превратить её в числовое значение. Усовершенствуйте эту процедуру так, чтобы она могла читать число в любой системе счисления по основанию от 2 до 36, используя латинские буквы вместо цифр, как это делается в шестнадцатеричной системе (так, буква E потребуется во всех системах по основанию от 15 до 36 и всегда будет обозначать 14, тогда как буква Z будет задействована только в системе по основанию 36, а обозначать будет «цифру» 35). Основание системы счисления передавайте дополнительным параметром. Заглавные и строчные латинские буквы не различайте, т. е. сделайте так. чтобы ваша процедура могла работать и с теми, и с другими.

> При решении следующих задач, а также задач, которые встретятся вам позже и будут оперировать понятием «слова», примите во внимание, что под словом обычно (если явно не указано иного) понимается произвольная последовательность непробельных символов; пробельными же символами традиционно считаются собственно пробел (`' '`), табуляция (#9) и перевод строки (#10), плюс в некоторых случаях ещё возврат каретки (#13). Возврат каретки вам при работе в Unix-системах на вводе с клавиатуры не встретится, а перевод строки во многих задачах обрабатывается отдельно, так что остаются только пробел и табуляция.

2.19. Напишите программу, которая читает **посимвольно** из стандартного потока ввода текст и в ответ **на каждую введённую строку** печатает следующую информацию:

a) целое число, равное количеству слов в только что введённой строке (под «словом» здесь и далее подразумевается последовательность любых непробельных символов);

b) два целых числа — количество встреченных в только что введённой строке слов, состоящих из чётного количества символов, и количество встреченных слов, состоящих из нечётного количества символов;

c) два целых числа — количество встреченных во введённой строке слов, состоящих из более чем семи букв, и количество встреченных слов, состоящих из не более чем двух букв;

d) целое число — количество встреченных во введённой строке слов, начинающихся с буквы `A` и заканчивающихся буквой `z`;

e) три целых числа — общее количество слов во введённой строке, длину самого длинного слова и длину самого короткого слова;

f) два целых числа — длину самого длинного слова в строке и длину самой длинной встреченной последовательности пробелов;

g) слово `YES`, если во введённой строке соблюдается баланс круглых скобок, и `NO` в противном случае; баланс скобок считается соблюдённым, если количество открывающих скобок совпадает с количеством закрывающих, при этом ни в каком месте строки количество уже встреченных закрывающих не превышало количество встреченных открывающих;

h) число, соответствующее количеству встреченных в этой строке открывающих круглых скобок, за которыми немедленно (следующим символом) следует закрывающая скобка — иначе говоря, количество встреченных в строке двухсимвольных комбинаций «`()`».

При возникновении ситуации «конец файла» завершайте работу. Вводить ограничения на длины строк, слов и др. запрещается.

2.20. Напишите программу, которая читает текст из стандартного потока ввода и печатает слова из этого текста, заключая каждое слово в круглые скобки. При получении символа перевода строки — переводите строку на печати. При возникновении ситуации «конец файла» завершите работу. Вводить ограничения на длины строк, слов и др. запрещается.

2.21. Напишите программу, которая читает текст из стандартного потока ввода и печатает только те слова из этого текста, которые состоят ровно из двух символов. При получении символа перевода строки — переводить строку на печати. При возникновении ситуации «конец файла» закончить работу. Вводить ограничения на длины строк, слов и др. запрещается.

Строковые переменные; аргументы командной строки

Следующие задачи используют параметры командной строки в качестве источника строк для обработки, просто чтобы не вводить эти строки с клавиатуры. В действительности целью этих задач является создание и закрепление навыков работы с переменными типа `string`. Это, впрочем, никоим образом не оправдывает неумения работать с параметрами командной строки.

2.22. Напишите программу, которая получает произвольное количество аргументов командной строки и печатает:

a) самый длинный из аргументов, а также его длину; если наибольшую длину имеют несколько аргументов, печатать первый из них;

b) те из аргументов, которые не содержат повторяющихся букв;

c) те из аргументов, в которых содержится ровно один символ «©» и не меньше одной точки;

d) аргументы, состоящие только из цифр;

e) аргументы, составленные из одного и того же символа, в том числе аргументы длиной в один символ (например, `'A'`, `'ccc'`, `'7777'` и т. п.);

f) аргументы, в которых содержится хотя бы одна латинская буква;

g) те из аргументов, начиная со второго, которые содержат хотя бы один общий символ с первым аргументом.

Если аргументов не задано, не печатать ничего; каждый аргумент печатать не более одного раза.

2.23. Напишите программу, принимающую ровно один аргумент командной строки, и печатающую количество слов в нём (см. замечание на стр. 22). На всякий случай напомним, что передать аргумент командной строки, содержащий пробельные символы, можно, заключив его в апострофы или двойные кавычки (см. §1.2.6). Если аргументов не задано или задано больше одного, выдайте сообщение об ошибке и завершите программу; помните, что сообщение должно быть англоязычным и при этом осмысленным.

2.24. Напишите программу, принимающую через параметры командной строки два числа типа `real` (назовём их x и y) и целое число n, и выдающую на печать произведение $x \cdot y$ в виде десятичной дроби, имеющей не более чем n знаков после запятой, причём **без незначащих нулей**; если в дробной части вообще не оказалось значащих цифр, её следует отбросить вместе с десятичной точкой.

2.25. Вернитесь к задачам 2.12–2.15 и перепишите программы так, чтобы они принимали нужные числа не из стандартного ввода, а через параметры командной строки. Если пользователь указал неправильное количество параметров или недопустимые значения чисел, выдавайте сообщение об ошибке и завершайте программу, не пытаясь «рисовать» фигуры.

Полноэкранные программы

2.26. Напишите программу, которая очищает экран, после чего отображает в средней строке экрана звёздочку, попеременно пробегающую весь экран (кроме крайней правой позиции) слева направо и справа налево; предусмотрите задержку в $\frac{1}{10}$ секунды, чтобы перемещения звёздочки можно было наблюдать. После трёх проходов слева направо и обратно завершите программу, очистив экран.

2.27. Усовершенствуйте программу, написанную по условиям предыдущей задачи, так, чтобы она немедленно завершалась при нажатии пользователем любой клавиши, но пока ни одной клавиши не нажато — продолжала работу.

2.28. Напишите программу, которая очищает экран, в центре экрана заполняет звёздочками прямоугольную область 10x10 знакомест и по границе заполненной области перемещает символ «#» по одному знакоместу за раз в направлении по часовой стрелке с задержкой в $\frac{1}{10}$ секунды, пока пользователь не нажмёт любую клавишу; когда клавиша окажется нажата, очистите экран и завершите программу. Если при старте программы экран по длине и/или ширине окажется меньше, чем 12 знакомест, выдайте сообщение об ошибке и завершите программу, не очищая экран и ничего на нём не изображая.

2.29. Модифицируйте программу из задачи 2.28 так, чтобы при нажатии клавиши «пробел» менялось направление, в котором перемещается символ «#», а при нажатии «стрелки влево» и «стрелки вправо» величина задержки между отдельными перемещениями увеличивалась или уменьшалась, так что визуально «движение» символа «#» будет замедляться и ускоряться. Завершение программы производите при нажатии клавиши `Escape`, все остальные клавиши игнорируйте.

2.30. Напишите программу, которая очищает экран, в центре экрана заполняет звёздочками прямоугольную область 3x3 знакоместа, после чего при нажатии на клавиши «стрелка влево» и «стрелка вправо» уменьшает и увеличивает заполненную область по горизонтали, а при нажатии «стрелки вверх» и «стрелки вниз» — по вертикали; считайте наименьшим допустимым размером по каждому из измерений единицу, а наибольшим — соответствующий размер экрана, уменьшенный на два. При нажатии клавиш «пробел» и `Escape` завершайте работу, остальные клавиши игнорируйте.

Во всех задачах, предполагающих управление цветом выводимых символов, при завершении обязательно очищайте

экран и восстанавливайте исходные настройки цвета. Как это сделать, написано в §2.8.4.

2.31. Модифицируйте программу из задачи 2.29 так, чтобы символ «#» по мере своего перемещения менял цвет, последовательно проходя все 16 цветов, доступных для текста (см. §2.8.4). Цвет фона не меняйте ни для этого символа, ни для других.

2.32. Напишите программу, которая очищает экран, в центре экрана заполняет звёздочками прямоугольную область 5x5 знакомест, после чего при нажатии на клавиши «стрелка влево» и «стрелка вправо» меняет цвет фона этой области (сразу всей), а при нажатии «стрелки вверх» и «стрелки вниз» — цвет текста (самих звёздочек). Цвета можете выстроить в любом порядке, какой вам нравятся; противоположные «стрелки» должны менять цвета в прямом и обратном порядке «по кругу» — следом за последним цветом должен снова идти первый, а перед первым — последний. При нажатии клавиш «пробел» и `Escape` завершайте работу, остальные клавиши игнорируйте.

Задачи 2.33–2.37 позволяют, начав с совсем простой программы, шаг за шагом прийти к созданию абстрактной, но довольно «затягивающей» игры; каждая следующая задача предполагает модификацию программы, написанной в качестве решения предыдущей задачи. Обратите внимание, что такого рода постепенное совершенствование одной программы — это один из ключевых подходов к разработке программного обеспечения.

2.33. Напишите программу, которая, очистив экран, изображает на нём движущуюся в одном из четырёх направлений (по одному знакоместу за раз) звёздочку. Предусмотрите задержку в $\frac{1}{10}$ секунды. На каждом шаге с вероятностью $\frac{1}{10}$ меняйте направление движения на случайно выбранное другое, так что в среднем звёздочка будет проходить в одном направлении 10 позиций. Если ваша звёздочка упёрлась в край экрана, пусть она после этого появится из-за противоположного края (ушла вправо — появилась слева, ушла вверх — появилась снизу и т. д.), сохранив направление движения. При нажатии клавиш «пробел» и `Escape` завершайте работу, остальные клавиши игнорируйте.

2.34. Модифицируйте программу из задачи 2.33 так, чтобы направление движения менялось только под прямым углом (влево или вправо), то есть чтобы звёздочка не могла сразу начать двигаться в противоположном направлении.

2.35. Модифицируйте программу из задачи 2.34 так, чтобы при нажатии на любую из четырёх «стрелок» звёздочка немедленно меняла направление на соответствующее нажатой клавише (вниз — значит вниз, влево — значит влево, и т. д., даже если она двигалась в противоположном направлении), но если пользователь ничего не нажимает — продолжала менять направление случайным образом, как это указано в предыдущей задаче.

2.36. Модифицируйте программу из задачи 2.35 так, чтобы заставить звёздочку принудительно изменить направление можно было не чаще одного раза в десять перемещений. Если пользователь нажал одну из «стрелок» раньше, следует это запомнить, но направление изменить только когда истекут десять перемещений; если в течение времени, когда менять направление запрещено, пользователь нажимал разные стрелки, использовать направление, соответствующее последней нажатой стрелке. Случайным образом направление должно меняться по-прежнему в соответствии с условиями задач 2.33 и 2.34.

2.37. Модифицируйте программу из задачи 2.36, превратив её в полноценную игру. Для этого звёздочку изначально всегда располагайте в левом верхнем углу, начальным направлением устанавливайте движение направо; в центре экрана расположите область 3x3 знакоместа, заполненную символами «`@`» (или любыми другими, какие вам больше нравятся). Цель игры — «загнать» звёздочку в заполненную область в центре; если эта цель достигнута, очистите экран и в его центр выдайте надпись «YOU WIN», после чего дождитесь нажатия любой клавиши.

Чтобы играть было интереснее, введите ограничения на время игры (например, не больше 1000 перемещений звёздочки) и на количество воздействий (например, не больше 50). Играть станет ещё интереснее, если текущие остатки по обоим лимитам будут отображаться на экране, например, в правом верхнем углу (тогда верхнюю строку экрана следует исключить из игры). Когда один из лимитов исчерпан, выдайте в центр экрана надпись «GAME OVER». Клавишу «пробел» задействуйте, чтобы ставить игру на паузу; для немедленного её завершения используйте `Escape`.

Описанная игра допускает бесчисленное множество дополнительных улучшений — появление на поле всевозможных «бонусов», перемещение «мишени», применение цвета, изменение уровня сложности (для этого можно варьировать скорость и значения лимитов) и многое другое; примените фантазию.

Указатели

2.38. В программе описаны целочисленные переменные `x`, `y` и `z`, а также тип массива, массив этого типа и указатель на такой массив:

```
type
    iptr3 = array [1..3] of ^integer;
var
    a: iptr3;
    p: ^iptr3;
```

Программа выполнила следующие присваивания:

```
    a[1] := @x;
    a[2] := @y;
    a[3] := @z;
    p := @a;
```

Как (с помощью каких выражений) теперь можно обратиться к переменным `x`, `y` и `z`, используя только имя `p` и не упоминая никаких других имён?

2.39. В программе описаны два типа-записи:

```
pair = record
    a: integer;
    b: real;
end;
```

```
twoptrs = record
    g, h: ^pair;
    t: ^integer;
end;
```

Дальше в той же программе описаны следующие переменные:

```
tp2: twoptrs;
m, n: pair;
x: integer;
```

Программа выполнила следующие присваивания:

```
tp2.g := @m;
tp2.h := @n;
tp2.t := @x;
```

В процедуру `proc1`, принимающую один параметр типа `^twoptrs`, имеющий имя `p`, передали значение выражения `@tp2`. С помощью каких выражений в этой процедуре можно обратиться к переменным `m.a`, `m.b`, `n.a`, `n.b` и `x`, если имена `m`, `n` и `x` в тексте процедуры недоступны?

2.40. Напишите программу, которая читает из потока стандартного ввода числа типа `longint`, пока не возникнет ситуация «конец

файла»; после этого печатает те из них, которые встретились в пользовательском вводе ровно три раза. Числа должны быть напечатаны в том порядке, в котором встретились в пользовательском вводе в первый раз.

2.41. Напишите программу, которая читает из потока стандартного ввода числа типа `longint`, пока не возникнет ситуация «конец файла»; после этого печатает то (или те) из них, которые встретились в пользовательском вводе чаще, чем все остальные. Если таких чисел больше одного (т. е. несколько чисел встречались одинаковое число раз, а все остальные — меньше), то такие числа должны быть напечатаны в том порядке, в котором встретились в пользовательском вводе в первый раз.

2.42. Напишите программу, которая читает из стандартного потока ввода строки, состоящие из слов (слова могут разделяться произвольными группами пробельных символов), и в ответ на каждую прочитанную строку печатает слова из этой строки *в обратном порядке*; например, в ответ на фразу «`Hympty Dumpty sat on a wall`» должно быть напечатано «`wall a on sat Dumpty Humpty`». Вводить ограничения на длины строк, слов и др. нельзя; в частности, недопустимо использовать тип `string` для хранения читаемых строк и/или отдельных слов. Вся выделенная динамическая память должна быть корректно освобождена сразу после обработки очередной строки.

2.43. Напишите программу, которая читает из стандартного потока ввода строки, состоящие из слов (слова могут разделяться произвольными группами пробельных символов), и в ответ на каждую прочитанную строку печатает слова из этой строки «вертикально», то есть сначала печатает первые буквы всех слов, переводит строку, печатает вторые буквы слов и т.д., пока буквы во всех словах не кончатся. Для слов, которые короче других, вместо недостающих букв нужно выводить пробелы. Например, в ответ на фразу «`Happy New Year to everyone`» должно быть напечатано:

```
HNYte
aeeov
pwa e
p r r
y   y
    o
    n
    e
```

Вводить ограничения на длины строк, слов и др. нельзя; в частности, недопустимо использовать тип `string` для хранения читаемых строк и/или отдельных слов. Вся выделенная динамическая память

должна быть корректно освобождена сразу после обработки очередной строки.

2.44. Напишите программу, которая читает из стандартного потока ввода текст, содержащий, в числе прочего, последовательности цифр (возможно, не отделённые пробелами от остального текста), и в ответ на каждую прочитанную строку печатает самую длинную из последовательностей цифр, встреченных в этой строке. Если таких (самых длинных) последовательностей несколько, следует напечатать их все, разделив пробелами. При возникновении ситуации «конец файла» закончить работу. Вводить ограничения на длины строк, слов и др. нельзя; в частности, недопустимо использовать тип `string` для хранения читаемых строк и/или отдельных слов. Вся выделенная динамическая память должна быть корректно освобождена сразу после обработки очередной строки.

2.45*. Найдите в Интернете или в книгах описание ***красно-чёрных деревьев***. Реализуйте в виде набора процедур и функций красно-чёрное дерево, в котором ключом выступает строка типа `string`, при этом каждый узел содержит ещё нетипизированный указатель (типа `pointer`), чтобы со строкой можно было связать произвольные данные. Напишите демонстрационную программу, с помощью которой можно будет проверить корректность вашей реализации.

Файлы

2.46. Найдите в Интернете английский оригинал стихотворения про Шалтая-Болтая (*Humpty Dumpty*). Напишите программу, которая получает имя файла через аргумент командной строки, после чего записывает в этот файл найденный вами текст. Обязательно проверьте, что всё работает как нужно, используя команду `cat`.

2.47. Напишите программу, которая получает через аргумент командной строки имя текстового файла и распечатывает его содержимое подобно тому, как это делает команда `cat`.

2.48. Напишите программу, которая получает через аргумент командной строки имя текстового файла, прочитывает его и выдаёт (печатает в поток стандартного вывода) целое число — количество строк текста, содержащихся в файле.

2.49. Напишите программу, которая получает через аргумент командной строки имя текстового файла, после чего читает из стандартного потока ввода («с клавиатуры») текст до наступления ситуации «конец файла» и записывает прочитанное в заданный файл.

2.50. Напишите программу, которая получает через командную строку имя файла, а также значения начального угла, конечного угла и шага (числа типа `real`) в градусах, т. е. всего четыре параметра, и формирует в заданном файле таблицу синусов, косинусов, тангенсов и котангенсов для заданных углов с заданным шагом в виде отформатированного текста, которым можно пользоваться при расчётах; желательно, чтобы в вашей таблице использовались символы–разделители, такие как «|» в качестве вертикальной черты и обычный минус — в качестве горизонтальной; от вертикальной черты следует всегда отступать хотя бы один пробел, чтобы таблица лучше читалась. Не забудьте про перевод из градусов в радианы; число π в Паскале так и обозначется: `pi`.

2.51*. Усовершенствуйте предыдущую программу так, чтобы она принимала пятый параметр командной строки — количество знаков после запятой для значений тригонометрических функций, их аргумент (в градусах) всегда печатала без незначащих нулей, а для целых значений отбрасывала всю дробную часть; так, если пользователь задал целые пределы и целый же шаг, программа вообще не должна использовать дроби при печати аргументов.

2.52*. Напишите программу, которая получает через аргументы командной строки имена произвольного количества текстовых файлов; предполагается, что все эти файлы уже существует. Нужно определить самую длинную строку в каждом из файлов и самую длинную строку среди всех заданных файлов. По каждому файлу выдать в поток стандартного ввода отдельной строкой (по одной на каждый файл) имя файла, символ двоеточия и найденную в этом файле самую длинную строку; если эта строка вдобавок является самой длинной среди всех файлов, напечатать символ звёздочки перед именем файла. Например, работа программы может выглядеть так:

```
vasya@host:~$ ./prog yesterday.txt girl.txt tree.pas
yesterday.txt:Why she had to go, I don't know, she wouldn't say.
girl.txt:Was she told when she was young that pain would lead to pleasure
*tree.pas:function SearchTree(var p: TreeNodePt; val: longint): TreeNodePt;
vasya@host:~$
```

Если наибольшую длину в отдельно взятом файле имеет несколько строк, печатать первую из них; если среди выделенных наиболее длинных строк наибольшую длину имеют несколько, вывести звёздочки перед каждым из соответствующих имён файлов. Нельзя вводить ограничения на количество файлов, количество строк в них, длины строк; в частности, недопустимо использовать тип `string`. Также не следует использовать вспомогательные файлы.

2.53. Напишите программу, которая получает через аргументы командной строки имена трёх файлов, первый из которых уже существует, остальные два программа должна сформировать в ходе своей работы. Рассматривая первый файл как текстовый, запишите во второй файл (тоже текстовый) только те строки из первого, которые начинаются с пробела. В третий файл (типизированный) запишите в виде чисел типа `integer` длины всех строк первого файла. Применив программу `hexdump`, убедитесь в корректности сформированного типизированного файла.

2.54. Напишите программу, которая получает через аргументы командной строки имена нескольких (произвольного количества, не менее одного) существующих типизированных файлов, состоящих из **четырёхбайтных** целых чисел, и одного текстового файла, который программа должна создать во время работы (его имя задаётся последним). Проанализировав каждый из заданных типизированных файлов, запишите в текстовый файл с заданным именем строки (по одной для каждого проанализированного файла), содержащие имя файла, общее количество чисел в нём, наименьшее и наибольшее из чисел, содержащихся в файле.

Для отладки вашего решения напишите вспомогательную программу, которая формирует типизированный файл нужного вам формата, состоящий из чисел, введённых из стандартного потока ввода (до наступления ситуации «конец файла»). Имя файла проще всего задать аргументом командной строки.

2.55. Для сбора статистики потребовалось поддерживать (в виде файла на диске) таблицу, состоящую из двух полей: строки, идентифицирующей некие предметы, и целочисленного счётчика, показывающего, сколько раз данный предмет встретился в ходе сбора статистики. Было принято решение, что идентификатор предмета не может в длину превышать 59 символов, так что файл, в котором будут храниться результаты сбора, может состоять из записей по 64 байта: 60 байт на хранение строки типа `string` (максимальная длина строки при этом составит как раз 59 символов) и 4 байта на целочисленный счётчик, имеющий тип `longint`.

Напишите программу, которая принимает три аргумента командной строки: имя файла, «команду» и идентификатор. Если второй аргумент («команда») представляет собой строку `add`, нужно в заданном файле найти запись, содержащую заданный идентификатор, и счётчик в этой записи увеличить на единицу; если такой записи нет, создать её (в конце файла) и установить счётчик равным единице. Если второй аргумент представляет собой строку `query`, нужно найти запись с заданным идентификатором и выдать в стандартный поток вывода значение счётчика из этой записи; если такой записи

нет, выдать ноль (это соответствует ситуации, что интересующий нас предмет пока ни разу не встречался). Если второй аргумент представляет собой строку `list`, выдать в стандартный поток вывода все пары «идентификатор+счётчик», имеющиеся в настоящий момент в файле; третьего аргумента командной строки в этом случае может не быть, но если он есть, его следует проигнорировать.

Об эффективности поиска можете пока не думать и просто просматривать весь файл от начала, пока либо не найдётся нужная запись, либо файл не кончится.

Пример к задаче 2.56

Файл 1		Файл 2		Результат	
apple	5	carrot	3	apple	5
pear	3	onion	4	pear	3
tomato	7	tomato	2	tomato	9
carrot	2	cucumber	5	carrot	5
				onion	4
				cucumber	5

2.56. Исходя из условий задачи 2.55, напишите программу, которая объединяет результаты сбора статистики, содержащиеся в двух файлах. Программа должна принимать три аргумента командной строки: первые два — имена существующих файлов описанного формата, которые программа будет читать, третий — имя файла, который программа должна создать. Множество идентификаторов, содержащихся в записях результирующего файла, должно совпадать с объединением множеств идентификаторов из первых двух файлов; если некоторый идентификатор содержался только в одном из двух исходных файлов (любом), соответствующее значение счётчика должно быть просто скопировано в файл результата, если же идентификатор содержался в обоих файлах, значения счётчиков из исходных файлов для данного идентификатора следует сложить и в результирующий файл занести полученную сумму. Пример такого объединения показан в таблице.

2.57*. Попробуйте предложить для условий задачи 2.55 такой формат файла, который обеспечит быстрый поиск нужной записи — например, за время, пропорциональное $\ln N$, где N — количество записей в файле, или ещё лучше — за константное время, вообще не зависящее от количества записей. Реализуйте программы, описанные в задачах 2.55 и 2.56, с использованием вашего формата файла. Исследуйте быстродействие программ при использовании вашего формата для ускоренного поиска в сравнении с временем линейного поиска; для этого вам потребуется файл, содержащий несколько миллионов

записей — как создать такой файл, чтобы в нём не повторялись идентификаторы, попробуйте сообразить самостоятельно.

Этюды посерьёзнее

2.58*. Напишите свою версию знаменитого «Тетриса». Фигуры можно сформировать, например, из символов «#» или даже просто выводить в соответствующих местах экрана пробелы, установив другой цвет фона. Обычный размер «стакана» в Тетрисе составляет десять колонок на 14-15 «строк»; поскольку обычный размер терминального окна больше, каждую «клетку» фигуры Тетриса можно представить прямоугольником шириной в три и высотой в два знакоместа, так они будут выглядеть «более квадратными»:

Если размер окна при старте программы недостаточен для рисования «стакана» того размера, который вы задумали, можно попросить пользователя увеличить окошко терминала и перезапустить программу.

2.59*. В эпоху MS-DOS была довольно популярна игра «Tetriller», которую придумал Алексей Миносянц. Согласно её сюжету некий фанат Тетриса, заснув за клавиатурой, во сне обнаружил себя на дне «стакана» Тетриса, а сверху на него посыпались фигуры, состоящие из строительного мусора — бетонные балки с торчащей арматурой, блоки из кирпичей, обрывки каких-то цепей и всё такое прочее. Сама игра отличается от обычного Тетриса тем, что внизу — по дну «стакана» и по лежащим фигурам — бегает персонаж, занимающий две клетки по вертикали, но если его прижать сверху, он может присесть, скрючиться и занимать только одну клетку; если же ему не оставить даже одной клетки, очередная упавшая фигура его попросту раздавит, что приводит к тому же эффекту, как и заполнение «стакана» до верха — тратится очередная «жизнь». Персонаж умеет подниматься и спускаться на одну клетку, что и делает, если уровень фигур перед ним отличается от того уровня, на котором он стоит, ровно на одну клетку вверх или вниз; если же персонаж упирается «в стену» или «в обрыв» — т. е. когда перед ним стенка самого «стакана» или когда следующий шаг потребовал бы вертикального перемещения больше чем на одну клетку — персонаж разворачивается и идёт в другую сторону. Управлять персонажем напрямую нельзя, можно только выстраивать перед ним «лестницы» из фигур или, наоборот, возводить непроходимые уступы. На уровне девяти клеток от дна по

бокам «стакана» расположены «двери», через которые персонаж может сбежать из «стакана», при этом всё, что успело нападать, разом исчезает; собственно говоря, подвести персонажа к этим дверям — и есть настоящая задача игры.

Если, реализуя Тетрис, вы последовали нашему совету и строили фигуры из прямоугольников 3x2 знакоместа, то изобразить персонажа, стоящего в полный рост, можно, например, так:  , а изобразить его же сидящим скрючившись — так:  ; разумеется, можно придумать и другие способы. Всё это позволяет реализовать сюжет Тетриллера в текстовом режиме, что вам и предлагается сделать. Кстати, возможно, вы лучше поймёте, что делать, если найдёте в Интернете оригинал игры и потратите минут сорок, чтобы с этой игрой освоиться.

2.60*. Игра «змейка» известна в сотнях вариантов, объединённых общим сюжетом: по игровому полю, состоящему из клеток, ползает «змейка», занимающая несколько клеток, причём соседние сегменты змейки должны располагаться на поле в соседних клетках («соседних» по вертикали или горизонтали, но не по диагонали). Направление движения змейки изменяется нажатием стрелок; одновременно перемещается на соседнюю клетку голова змейки и «поджимается» её хвост, так что её длина при движении остаётся постоянной. На поле в разных местах возникают предметы, которые надлежит «съесть», при этом змейка увеличивается в длину. Змейка при движении не должна пересекать сама себя, это означает конец игры. Пока змейка короткая, гонять её по полю, не допуская самопересечений, довольно просто, но по мере её удлинения это становится всё труднее и труднее, пока не превысит возможности игрока.

Попробуйте реализовать свою версию «змейки». Например, голову змейки можно обозначить символом «`@`», остальные её сегменты — звёздочками или буквами «`o`». Эта игра примечательна тем, что для её реализации вам обязательно потребуется односвязный список — чтобы хранить координаты сегментов вашей змейки.

2.61*. Придумайте и реализуйте (на Паскале с использованием модуля `crt`) свою собственную игровую программу. Предложите поиграть в неё вашим друзьям или родственникам, но после объяснения правил больше ни на чём не настаивайте; если ваш «подопытный» не потеряет интереса к игрушке в течение пяти минут — считайте, что дело сделано. Если же, оставив игрока один на один с вашей игрой, через час вы обнаружите, что игрок всё ещё не сбежал — поздравьте себя: вы стали программистом. Конечно, такое получается редко, но вы не расстраивайтесь, ведь десять из двенадцати частей «Введения в профессию» у вас пока ещё впереди.

К части 3 «возможности процессора и язык ассемблера»

Регистры и флаги

3.01. Какое число будет находиться в регистре EAX после выполнения следующих команд?

a)
```
mov eax, 4f682bh
mov ax, 19905
mov al, 130o
```

b)
```
mov eax, 99f6eeh
mov ax, 63172o
mov al, 99
```

c)
```
mov eax, db9fe3h
mov ax, 2486
mov ah, 5bh
```

d)
```
mov eax, $7f2176
mov ax, 62877
mov al, 134o
```

e)
```
mov eax, 0x55623b
mov ax, 31071
mov ah, c9h
```

f)
```
mov eax, 5e1bdah
mov ax, 45102
mov ah, $40
```

3.02. Какое число окажется в регистре AL и какие из флагов ZF, SF, OF и CF будут установлены в результате выполнения команды `add al, bl`, если перед этим в регистрах AL и BL находились числа:

a) fe и 01 b) 02 и fe c) 1f и 1e d) 1f и ff e) 7a и 07 f) 8b и f0

(все числа шестнадцатеричные)?

3.03. Какое число окажется в регистре AL и какие из флагов ZF, SF, OF и CF будут установлены в результате выполнения команды `sub al, bl`, если перед этим в регистрах AL и BL находились числа:

a) 1a и 1c b) 29 и fe c) ed и ee d) 7e и 6d e) 8a и 1a f) 7c и 88

(все числа шестнадцатеричные)?

3.04*. По результатам операции сложения процессор устанавливает значения флагов ZF, SF, OF и CF (и некоторых других, но мы будем рассматривать только эти четыре). Поскольку каждый флаг

может быть равен нулю или единице, *без учёта их смысла* можно предположить, что всего возможно 16 комбинаций значений этих четырёх флагов — от `0000` до `1111`, хотя в действительности не все эти комбинации могут получиться.

Для каждой из 16 комбинаций предложите такие значения регистров `AL` и `BL`, что выполнение команды `add al, bl` приведёт к возникновению рассматриваемой комбинации флагов, либо укажите, что это невозможно.

3.05*. При решении предыдущей задачи вы выяснили, что некоторые комбинации флагов с помощью команды `add` получить невозможно. Можно ли какие-то из этих комбинаций получить командой `sub al, bl`? Какие для этого нужны исходные значения?

3.06. Какое число окажется в `EAX` после выполнения следующих команд? Ответ дайте в шестнадцатеричной системе счисления.

a)
```
mov eax, $12345678
and eax, $0cccccccc
```

b)
```
mov eax, $12345678
and eax, $33333333
```

c)
```
mov eax, $12345678
or eax, $0cccccccc
```

d)
```
mov eax, $12345678
shl eax, 4
```

e)
```
mov eax, $12345678
shr eax, 4
```

f)
```
mov eax, $0f000
shr eax, 2
```

g)
```
mov eax, $9abcdef1
shr eax, 16
```

h)
```
mov eax, $9abcdef1
sar eax, 16
```

i)
```
mov eax, $45678abc
sar eax, 16
```

j)
```
mov eax, $0abcdef12
mov ebx, $3456789a
shl eax, 12
and eax, $00ff0000
shr ebx, 20
and ebx, $0ff
or eax, ebx
```

k)
```
mov eax, $1234abcd
xor eax, $33333333
```

l)
```
xor eax, eax
neg eax
sal eax, 16
sar eax, 8
```

Ознакомительные этюды

Следующие задачи предполагают использование макросов из файла `stud_io.inc`; описание этих макросов приведено в §3.1.5, а сам файл можно скачать на домашней странице книги «Программирование: введение в профессию» (`http://stolyarov.info/books/programming_intro/e2`).

3.07. Напишите программу, которая считывает из стандартного потока ввода («с клавиатуры») один символ и печатает слово `YES`, если этот символ — заглавная латинская буква `A`, в противном случае печатает `NO`.

3.08. Напишите программу, которая считывает из стандартного потока ввода («с клавиатуры») один символ, и если этот символ — цифра, печатает соответствующее ей количество звёздочек (от одной до девяти), в противном случае ничего не печатает.

3.09. Напишите программу, которая читает данные из потока стандартного ввода и немедленно выдаёт их в поток стандартного вывода, пока не наступит ситуация «конец файла» (т. е. делает то же самое, что делает программа `cat`, если её запустить без параметров).

3.10. Напишите программу, которая читает из потока стандартного ввода символы до конца первой строки (т. е. пока не будет встречен символ с кодом 10 либо не возникнет ситуация «конец файла») и подсчитывает, сколько в этой строке встретилось символов «+» и «-»; все прочие символы игнорирует. Полученные результаты подсчёта назовём N_+ и N_-. После окончания ввода строки программа вычисляет число $N_* = N_+ \times N_-$ и печатает такое количество звёздочек.

3.11. Напишите программу, которая читает из потока стандартного ввода символы до конца первой строки (т. е. пока не будет встречен символ с кодом 10 либо не возникнет ситуация «конец файла») и подсчитывает сумму всех встреченных цифр (напомним, что коды цифр в ASCII-таблице располагаются подряд). По окончании чтения строки программа должна напечатать столько звёздочек, какова получилась сумма.

3.12. Напишите программу, которая читает текст из потока стандартного ввода, пока не наступит ситуация «конец файла», и в ходе работы печатает «`OK`» в ответ на каждую введённую пользователем строку.

3.13. Напишите программу, которая читает текст из потока стандартного ввода, пока не наступит ситуация «конец файла», и в ответ на каждую введённую пользователем строку печатает столько звёздочек, какова была длина введённой строки.

3.14. Напишите программу, которая читает текст из стандартного потока ввода и в ответ на каждую введённую строку печатает столько звёздочек, какова длина:

a) самого длинного слова строки;
b) последнего слова строки.

Программа должна заканчивать работу при возникновении ситуации «конец файла» (но не раньше).

3.15. Напишите программу, которая читает текст из потока стандартного ввода, пока не наступит ситуация «конец файла», и в ответ на каждую введённую пользователем строку печатает слово `YES`, если во введённой строке соблюдается баланс круглых скобок, и `NO` в противном случае; баланс скобок считается соблюдённым, если количество открывающих скобок совпадает с количеством закрывающих, при этом ни в каком месте строки количество уже встреченных закрывающих не превышало количество встреченных открывающих.

3.16. Напишите программу, которая читает текст из потока стандартного ввода, пока не наступит ситуация «конец файла», и в ответ на каждую введённую пользователем строку печатает столько звёздочек, сколько было слов во введённой строке.

3.17. Напишите программу, которая читает текст из стандартного потока ввода, пока не наступит ситуация «конец файла», и печатает слова из этого текста, заключая каждое слово в круглые скобки. При получении символа перевода строки — переводите строку на печати.

> Обратите особое внимание на задачи 3.18 и 3.19! Они несколько сложнее, чем все предшествующие, но решить их нужно всенепременнейшим образом и обязательно без посторонней помощи. Опыт, полученный при решении этих двух задач, потребуется вам в дальнейшем, при этом никакие другие задачи вам этого опыта не предоставят. Если вы с ходу не знаете, как к этим задачам подступиться, воспользуйтесь подсказками в разделе «Указания» (см. стр. 120), но от использования посторонней помощи всё равно воздержитесь. Если не получается — отложите эти задачи до лучших времён, но обязательно рано или поздно вернитесь к ним и решите их сами.

3.18. Напишите программу, которая читает из стандартного потока ввода десятичное представление неотрицательного целого числа; чтение выполняйте по одной цифре, до тех пор, пока не будет прочитан символ, не являющийся цифрой, либо не возникнет ситуация «конец файла»; по окончании чтения числа программа должна вывести столько звёздочек, какое число было введено, и завершить вывод символом перевода строки. Переполнение — ситуацию, когда введённое пользователем число не умещается в 32-битный регистр или переменную — можете никак не обрабатывать.

3.19. Напишите программу, которая читает из стандартного потока ввода произвольный текст и подсчитывает количество символов

в нём, а после наступления ситуации «конец файла» печатает результат подсчёта в десятичном представлении, переводит строку и завершается.

3.20. Воспользовавшись фрагментами текстов программы, написанных при решении задач 3.18 и 3.19, напишите программу, которая читает из стандартного потока ввода два числа, разделённые символом пробела, и выдаёт в стандартный поток вывода их сумму, разность и произведение. Предусмотрите выдачу сообщений об ошибках на случай некорректного пользовательского ввода.

3.21. Напишите программу, которая читает из потока стандартного ввода строки, каждая из которых содержит десятичную запись первого числа, знак арифметического действия (`+`, `-`, `*`, `/`) и десятичную запись второго числа (без пробелов). В ответ на каждую прочитанную строку программа должна напечатать результат выполнения указанного действия над числами (деление считайте целочисленным) и сделать перевод строки; при наступлении ситуации «конец файла» — завершить работу. Если очередная строка не соответствует условиям задачи, следует напечатать слово «`Error`» и перевод строки, работу продолжить.

Стек, подпрограммы, модули

3.22. Напишите программу, которая читает из стандартного потока ввода текст до тех пор, пока либо не будет встречен символ перевода строки, либо не возникнет ситуация «конец файла», а после окончания чтения выдаёт все символы введённой строки в обратном порядке.

3.23*. Напишите программу, которая читает из стандартного потока ввода текст до наступления ситуации «конец файла», а после окончания чтения выдаёт все строки введённого текста, причём символы каждой отдельной строки выдаются в обратном порядке, но сами строки выдаются в том порядке, в котором были введены.

3.24. Вернитесь к задачам 3.18–3.20; используя фрагменты кода, написанного при их решении, создайте следующие подпрограммы:

- для перевода десятичного представления числа (в виде строки символов) в собственно число; подпрограмма должна получать на вход два параметра — адрес начала строки и её длину (например, в регистрах `EAX` и `CL`); возвращать число лучше всего в регистре `EAX`, но потребуется какой-то дополнительный регистр (например, тот же `CL`), чтобы показать вызывающему, не возникло ли ошибки при анализе строки;

- для перевода числа в строковое представление; входных параметров здесь стоит предусмотреть два — само число и адрес области памяти, где следует разместить строку (например, используйте регистры `EAX` и `ECX`); можете предполагать, что в указанной области памяти всегда есть не менее 11 байт, больше вам для представления 32-битного числа не потребуется; подпрограмма может возвращать (через всё тот же EAX) длину сформированной строки, либо можно после кода последней цифры разместить в области памяти ноль (не код символа «ноль», а настоящий ноль, байт с нулевым значением), тогда ваша подпрограмма может вообще ничего не возвращать;
- печать по одному символу строки из заданной области памяти; в зависимости от решения, принятого в предыдущем пункте, подпрограмма может получать адрес начала области памяти и длину строки, либо только адрес начала — в этом случае предполагать, что в очередном байте встретится арифметический ноль, свидетельствующий об окончании строки;
- ввод строкового представления числа из стандартного потока ввода по одному символу с размещением в заданной области памяти, пока не будет прочитан символ, не являющийся цифрой, либо не настанет ситуация «конец файла»; эта подпрограмма должна получить на вход адрес области памяти и её размер, а вернуть количество символов, которые были успешно прочитаны, и информацию о причине прекращения чтения: код символа, отличного от цифры, если таковой был прочитан, либо какие-то специальные значения, указывающие на возникшую ситуацию «конец файла» или на исчерпание доступного размера памяти (символов оказалось больше, чем переданная длина области памяти).

Пользуясь написанными подпрограммами, перепишите программу, соответствующую условиям задачи 3.20.

3.25. Вернитесь к предыдущей задаче и переделайте свои подпрограммы так, чтобы они получали все входные параметры через стек и строили полноценный стековый фрейм (для этого снабдите их стандартным прологом и эпилогом); возврат значений по-прежнему производите через регистр `EAX` и, если нужно, через регистр `ECX` или его части. Соблюдайте конвенцию `CDECL`. Головную программу модифицируйте для работы с новыми версиями подпрограмм; убедитесь, что после завершения переделки программа по-прежнему успешно транслируется и правильно работает.

3.26. Возьмите своё решение задачи 3.24 или 3.25 и разбейте его на модули, по одному на каждую подпрограмму и на головную про-

грамму (всего пять модулей). Убедитесь, что все модули успешно транслируются, программа проходит сборку и по-прежнему правильно работает.

3.27*. Напишите программу, которая читает из стандартного потока ввода текст до тех пор, пока либо не будет встречен символ перевода строки, либо не возникнет ситуация «конец файла», а затем вычисляет арифметическое выражение, представленное в прочитанной строке, и печатает полученный результат. Выражение может состоять из целых чисел, записанных в десятичной системе, а также символов четырёх действий арифметики и круглых скобок.

Макропроцессор

Для проверки правильности работы написанных макросов вам будет полезен флаг командной строки `-E`; получив этот флаг, ассемблер NASM подвергает указанный исходный текст макропроцессированию, а результат выдаёт в поток стандартного вывода (на экран), и это позволяет посмотреть, что в действительности получилось.

3.28. Напишите макрос, получающий три параметра: начальное значение, шаг и количество, и создающий область памяти, заполненную 4-байтными целыми числами (в заданном количестве), первое из которых равно заданному начальному значению, а каждое последующее больше предыдущего на заданный шаг.

3.29. Напишите макрос, получающий в качестве параметров произвольное количество меток и генерирующий код, который в зависимости от значения регистра `EAX` выполняет переход на одну из этих меток: при значении 1 — на первую, при значении 2 — на вторую и т. д., а в случае, если в `EAX` содержится число, не соответствующее никакой из меток — не выполняющий никакой переход.

3.30. Напишите макрос, получающий на вход строковый литерал и создающий область памяти, заполненную *четырёхбайтными* числами, каждое из которых соответствует коду очередного символа из полученной строки.

3.31. С помощью директив условной компиляции модифицируйте макрос из предыдущей задачи так, чтобы в зависимости от значения некоторого управляющего макросимвола, введённого специально для этой цели, область памяти составлялась из двухбайтных, четырёхбайтных либо восьмибайтных значений.

Взаимодействие с ОС

Все последующие задачи до конца части, посвящённой ассемблеру, следует решать без использования файла `stud_io.inc`; к соответствующим системным вызовам нужно уметь обращаться напрямую.

3.32. Напишите программу, которая завершается успешно, если ей при запуске указать ровно три аргумента командной строки, и неуспешно — в любом другом случае. Проверить, успешно ли завершилась программа, можно, запустив её примерно так:

```
./prog abra schwabra kadabra && echo "YES"
```

Обратите внимание, что условия этой задачи не предполагают никаких операций ввода-вывода.

3.33. Напишите программу, которая завершается успешно, если:

a) ей переданы ровно два аргумента командной строки, имеющие одинаковую длину;
b) ей переданы ровно два аргумента командной строки, и у них совпадает последний символ;
c) оба вышеуказанных условия выполнены одновременно.

В противном случае (если аргументов не два или если не выполнено условие) программа должна завершаться неуспешно. Обратите внимание, что программа не должна ничего печатать или тем более читать.

3.34. Вернитесь к решению последнего пункта предыдущей задачи и разбейте его на три отдельные единицы трансляции. Убедитесь, что все три модуля успешно проходят трансляцию, полученный набор объектных файлов проходит сборку, исполняемый файл по-прежнему корректно работает.

3.35. Напишите программу, принимающую произвольное количество аргументов командной строки и печатающую самый длинный из них. Имя программы аргументом не считается и не учитывается; если ни одного аргумента при запуске не указано — ничего не печатать.

3.36. Напишите программу, принимающую произвольное количество аргументов командной строки и печатающую:

a) те из них, в которых есть хотя бы одна заглавная латинская буква;
b) те из них, в которых содержится ровно один символ «@» и не менее одного символа точки «.»;

c) те из них, которые начинаются заглавной латинской буквой, а заканчиваются строчной латинской буквой;
d) те из них, которые состоят из одного и того же символа (например, `zzz`, `11111`, `M` и т. п.).

Имя программы аргументом не считается и не учитывается.

3.37. Напишите программу, принимающую два аргумента командной строки, каждый из которых представляет собой беззнаковое целое число, записанное в текстовом представлении в восьмеричной системе счисления. Программа должна напечатать сумму и произведение этих чисел, представленные в восьмеричной системе; если аргументов не два, завершить выполнение неуспешно, ничего не печатая.

3.38. Напишите программу, которая получает три аргумента командной строки. Первый и второй состоят каждый ровно из одного символа (цифры или заглавной латинской буквы) и задают основания систем счисления (от 2 до 35): «`2`» — двоичная, «`3`» — троичная, ..., «`9`» — девятичная, «`A`» — десятичная, «`G`» — шестнадцатеричная, «`Z`» — система по основанию 35. Третий аргумент представляет собой запись числа в первой из двух систем счисления; программа должна напечатать то же самое число во второй системе счисления. Например, при запуске «`./prog 2 A 110011`» должно быть напечатано `51`. Если всё правильно, программа должна завершаться успешно, если же при формировании аргументов допущена ошибка (аргументов не три, первый или второй имеют длину, отличную от единицы, или содержат недопустимый символ, в третьем аргументе недопустимый символ) — завершиться неуспешно, при этом ничего не напечатав.

3.39. Напишите программу, которая принимает имя файла через аргумент командной строки, открывает этот файл на чтение, читает его блоками по 4096 байт (не забудьте, что прочитано может быть меньше, чем запрошено) и (в предположении, что файл текстовый) подсчитывает в нём количество строк. Результат подсчёта напечатайте в восьмеричной, десятичной или шестнадцатеричной системе — выберите ту, которая вам удобнее.

3.40. Напишите программу, которая свой первый аргумент командной строки рассматривает как имя файла, а остальные — как простые слова, составляющие строку; программа должна сформировать текстовый файл с заданным именем и десять раз записать в него строку, образованную из заданных слов. Если всё в порядке, программа должна завершиться успешно, если же файл открыть не

удалось или произошла ошибка при записи в него, следует завершиться неуспешно.

3.41. Напишите программу, которая принимает через аргументы командной строки имена двух файлов и число N, записанное в восьмеричной, десятичной или шестнадцатеричной системе — выберите ту, которая вам удобнее. Первый файл откройте на чтение, второй — на запись (прикажите системе создать этот файл, если его ещё нет, а если он уже есть — сбросить его старое содержимое). Если любой из файлов не откроется — напечатайте сообщение об ошибке и завершите работу неуспешно. Во второй файл запишите первые N байтов из первого файла. Чтение выполняйте блоками по 4096 байт; не забудьте, что прочитано может быть меньше, чем запрошено. Записывайте каждый раз столько, сколько было прочитано, или меньше, если в противном случае будет превышено число N.

3.42. Напишите программу, которая читает из потока стандартного ввода строки, пока не наступит ситуация «конец файла», и по окончании работы формирует бинарный (не текстовый!) файл с именем «`result.dat`», в который записывает три четырёхбайтных целых числа: общее количество прочитанных строк, их суммарную длину и длину самой длинной из прочитанных строк.

3.43. Напишите программу, которая через первый аргумент командной строки получает текстовое представление числа в десятичной системе счисления и печатает слово «`OK`», если это число делится без остатка на три, в противном случае ничего не печатает. Число может быть столь большим, что не поместится ни в 32-битное, ни в 64-битное целое.

Плавучая арифметика

При создании подпрограмм, имеющих дело с арифметикой плавающей точки, считайте, что регистры `EAX`, `ECX`, `EDX`, а также весь стек регистров сопроцессора находятся в вашем распоряжении; если этого недостаточно, используйте стековую память. Проще всего будет сбросить сопроцессор в «чистое» состояние командой `finit` (см. §3.8.9), при необходимости предварительно сохранив в локальной переменной (в локальной области стекового фрейма) числа, переданные вам через регистры сопроцессора (если такие параметры предполагаются). Если вы проводите некие расчёты и вам требуется вызвать такую подпрограмму, сохраните состояние сопроцессора командой `fsave`, а когда вызванная

подпрограмма завершит работу, восстановите его командой `frstor`.

Задачи этого параграфа довольно трудоёмки. В принципе вы можете их пропустить, особенно если у вас проблемы со свободным временем: опыт, который могут дать эти задачи, весьма любопытен, но, надо признать, совершенно не критичен для постижения последующего материала.

Не слишком расстраивайтесь, если с задачами этого параграфа ничего не получится. Они реально мозголомны; впрочем, в этом случае мы настоятельно посоветуем вернуться к ним позднее и, как говорят психологи, «закрыть гештальт» — когда вы накопите больше опыта, справиться с этими задачами окажется проще.

3.44*. Напишите подпрограмму, которая получает число с плавающей точкой через вершину стека регистров сопроцессора (т. е. на момент передачи управления этой подпрограмме число находится в `ST0`), через регистр `EAX` получает адрес области памяти, через регистр `ECX` — длину этой области памяти в байтах. Подпрограмма должна построить в указанной области памяти строковое представление переданного числа в виде десятичной дроби; позиция конца строки должна быть обозначена нулевым байтом. Десятичные знаки после точки стройте до тех пор, пока не закончится выделенная вам память (не забывайте, что в конце нужно оставить один байт, чтобы записать в него ограничительный ноль); завершить работу раньше можно лишь в том случае, если совершенно точно все остальные десятичные знаки будут нулями. Если в отведённую память не помещается даже целая часть числа (с учётом ограничивающего нулевого байта), в знак ошибки сделайте строку «пустой», т. е. запишите ноль в самый первый байт переданной области памяти.

Перед возвратом управления занесите в `EAX` общую длину сформированной строки *без учёта ограничивающего нулевого байта* (таким образом, значение, которое вы возвращаете через `EAX`, всегда будет хотя бы на единицу меньше значения, полученного через `ECX`).

Обязательно проверьте работу вашей подпрограммы на различных числах; для этого сформированную подпрограммой строку следует напечатать, как, например, это делалось в §3.7.2.

3.45*. Напишите программу, которая получает в качестве аргументов командной строки текстовое представление чисел в форме десятичных дробей и печатает их сумму. Для этого напишите подпрограмму, которая получает (например, через `EAX`) адрес строки, содержащей десятичную дробь, и возвращает через `ST0` полученное

из этого представления число. Для печати числа воспользуйтесь наработками из решения предыдущей задачи.

3.46*. Напишите программу, которая через аргументы командной строки, которых должно быть ровно три, получает три коэффициента квадратного уравнения и решает его (в действительных числах; если действительных корней нет, следует напечатать «`No roots`»).

3.47*. Вернитесь к условиям задачи 3.27 и напишите аналогичную программу-калькулятор для чисел с плавающей точкой.

3.48*. Вернитесь к задачам 2.50, 2.51 из части, посвящённой Паскалю (см. стр. 31), и попытайтесь написать аналогичные программы на языке ассемблера.

К части 4 «программирование на языке Си»

Ознакомительные задачи

4.01. Чему равны следующие выражения? Ответ запишите в десятичной системе.

a) `'2' + 3`	b) `3 + 'a'`	c) `'Z'-'X'`	d) `' ' * 10`	e) `'\n' * 4`
f) `17 / 5`	g) `17 % 5`	h) `17 / 15`	i) `!25`	j) `!0`
k) `23 >> 3`	l) `15 << 3`	m) `~35`	n) `~(-47)`	o) `21 ^ 15`
p) `21 && 10`	q) `21 & 10`	r) `21 || 7`	s) `21 | 7`	t) `252 & ~15`

4.02. Дано описание:

```
enum en25 { alpha = 25, beta, gamma = beta * 10, delta };
```

Каково численное значение идентификатора `delta`?

4.03. Переменная `i` имеет тип `int` и содержит число 493. Что будет напечатано в результате следующего вызова `printf`? Записывая ответ, обозначьте пробелы символом «␣».

```
printf("%d:%6d:%-6d:%06d:%7.5d:%x:%X\n", i, i, i, i, i, i, i);
```

4.04. Указатель `str` типа `char*` содержит адрес строки `"Hello"` Что напечатает `printf` при следующих обращениях к ней? Пробелы обозначьте символом «␣».

a) `printf("%3s", str);`
b) `printf("%8s", str);`
c) `printf("%-8s", str);`
d) `printf("%3.4s", str);`
e) `printf("%6.4s", str);`

4.05. Напишите функцию `get_and_zero`, которая каким-то способом получает на вход (через параметр) целочисленную переменную, в эту переменную заносит ноль, а то число, которое в переменной находилось раньше, возвращает в качестве своего значения. Как нужно будет вызывать эту функцию?

4.06. Напишите функцию `put_sum`, которая получает через параметры три целочисленные переменные, вычисляет сумму их значений, заносит полученную сумму во все три переменные и возвращает её же в качестве своего значения.

4.07. Напишите функцию, которая получает на вход адрес начала текстовой строки и возвращает в качестве значения количество пробелов в этой строке.

4.08*. Вернитесь к решению предыдущей задачи; если оно использует цикл, напишите альтернативное решение, использующее рекурсию и не применяющее никаких циклов; если же ваше исходное решение было рекурсивным, напишите итеративное решение (т. е. с циклом и без применения рекурсии). Как вы считаете, в чём, при всей его красоте, состоит фундаментальный недостаток рекурсивного решения?

4.09. Вернитесь к задачам 2.19–2.21 (см. стр. 22) и перепишите их решения на Си, при этом из библиотечных функций используйте только `getchar`, `putchar` и `printf`; цикл в каждой из программ должен быть ровно один, и его заголовок следует сделать таким:

```
while((c = getchar()) != EOF) {
```

4.10. Напишите программу, которая печатает свои аргументы командной строки, *кроме* тех, которые начинаются с символа «-».

Анализ строк

В следующих задачах аргументы командной строки используются как источник анализируемых строк, чтобы не вводить их с клавиатуры.

4.11. Вернитесь к задачам 2.22 и 2.23 (см. стр. 24) и напишите такие же программы на Си, **используя библиотечные функции только для вывода** (в частности, не применяя никаких функций из заголовочника `string.h`). Несмотря на небольшой объём каждой из программ, старайтесь максимально применять вспомогательные функции. Учтите, что их применение осмысленно в *каждой* из этих задач, так что, если вы написали хотя бы одну из указаных программ

целиком в функции `main` — вы делаете фундаментальную ошибку в подходе к созданию программного кода.

4.12. Напишите функцию, которая в заданной строке находит первое вхождение заданной подстроки и, если таковое обнаружено, возвращает *адрес* (не индекс в массиве, а именно адрес!) начала подстроки в строке, если же заданная подстрока не обнаружена — возвращает `NULL`.

Используя эту функцию, напишите программу, которая принимает на вход произвольное (не менее одного) количество аргументов командной строки, первый из них рассматривает как подстроку, и печатает:

a) те из оставшихся аргументов, которые содержат в себе эту подстроку (каждый на отдельной строке);
b) для каждого из аргументов, содержащих в себе подстроку — сам этот аргумент и количество вхождений в него указанной подстроки.

Обратите внимание, что написанной вами функции, если она соответствует спецификации, *достаточно* для решения обоих пунктов задачи, в том смысле что никакой дополнительный анализ строк здесь не нужен, можно вообще не обращаться к элементам строк ни напрямую, ни как-то ещё, кроме как через написанную функцию.

4.13. Рассмотрим бесконечную последовательность символов $S_k (k = 1, 2, \ldots)$, которая состоит из **цифр** десятичной записи квадратов натуральных чисел, между собой ничем не разделённых; в частности, первые двадцать символов в этой последовательности такие: `14916253649648110012` (десятичная запись чисел 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 и первые две цифры от 121). Обратите внимание, что номера в последовательности принадлежат отдельным символам (цифрам), а не каждому числу целиком; скажем, S_4 — это символ `1`, а S_5 — символ `6` (вместе образующие десятичное представление числа `16`). Напишите программу, которая принимает через аргументы командной строки два натуральных числа N и M и выдаёт (в виде одной строки) члены последовательности S_k, начиная с S_N, заканчивая S_M. Например, при запуске с параметрами `10 16` должна быть напечатана строка «`4964811`» (это 7^2, 8^2, 9^2 и единичка от 10^2); при параметрах `150 175` должна быть выдана строка «`2401250026012704280929163O`». Считайте, что разрядности числа типа `long long` вам при вычислениях будет достаточно.

4.14*. Напишите функцию, которая получает адрес строки и удаляет из этой строки (прямо на месте, то есть не создавая никаких копий) все пробелы. Задача решается **за один проход**.

Указатели и динамическая память

4.15. Есть следующие описания:

```
double m[] = { 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0 };
double *p = &m[2];
```

Какой тип имеет выражение `*(p+1)` и чему оно равно? А как насчёт выражения `3[p]`?

4.16. Какой тип имеет выражение `*"A"` и чему оно равно?

4.17. Напишите функцию, которая:

a) получает через параметры адрес начала массива элементов типа `int` и его длину, строит список из тех же чисел и возвращает адрес первого элемента построенного списка;
b) получает единственным параметром адрес первого элемента списка целых чисел, затем создаёт в памяти массив на один элемент больше, чем было элементов в списке, в его самый первый элемент (тот, что имеет индекс `0`) заносит длину списка, в остальные элементы — значения из звеньев списка, и возвращает адрес начала массива.

Считайте, что звенья списка имеют тип

```
struct item {
    int x;
    struct item *next;
};
```

Если ваше решение (в каждом из пунктов задачи) использует цикл, попробуйте сделать ещё одно решение, на этот раз без циклов, применяя только рекурсию. И наоборот, если вы сразу написали рекурсивное решение, обязательно сделайте также и итеративное, без рекурсии.

4.18. Вернитесь к задачам 2.40–2.44 (см. стр. 28) и напишите аналогичные программы на Си, используя списки. За неимением типа `longint` используйте тип `long`.

4.19. Напишите программу, которая читает из потока стандартного ввода с помощью функции `scanf` целые числа до тех пор, пока не возникнет ситуация «конец файла»; после этого печатает (в произвольном порядке — выберите тот, что вам больше удобен) те пары среди введённых чисел, которые, встретившись в пользовательском вводе подряд, при этом отличались друг от друга не более чем на пять. Для хранения чисел используйте список.

В следующих задачах **воздержитесь от применения функции `realloc`, даже если знаете о ней!** Для изменения размера массива всегда создавайте массив нового размера, копируйте в него данные из старого, после чего ликвидируйте старый.

4.20. Напишите программы, отвечающие условиям задач 2.42–2.44, используя в этот раз массивы символов, увеличивая по мере надобности их размер.

4.21. Реализуйте абстракцию стека чисел типа `double`, используя в качестве реализации массив, размер которого (при исчерпании свободного места) увеличивается вдвое. Для этого напишите функции `stackdbl_init`, `stackdbl_destroy`, `stackdbl_push`, `stackdbl_pop` и `stackdbl_empty` соответственно для инициализации стека, освобождения памяти от стека, занесения числа в стек, извлечения числа из стека и проверки, не пуст ли стек. Все переменные, нужные для обслуживания стека, объедините в структуру, а в функции передавайте адрес этой структуры.

4.22*. Вернитесь к задаче 2.45 (см. стр. 30). Если вы решили её, перепишите решение на Си; если не решали — возможно, теперь (с высоты опыта, полученного уже после работы с Паскалем) красно-чёрные деревья вам покорятся; попробуйте!

Высокоуровневый ввод-вывод

4.23. Напишите на Си с использованием функций высокоуровневого ввода-вывода программы, соответствующие условиям задач 2.46–2.49 (см. стр. 30).

4.24. Напишите на Си программу, соответствующую условиям задачи 2.52 (см. стр. 31). Для хранения строк используйте массивы, изменяя их размеры по мере надобности.

Полноэкранные программы

4.25. С использованием библиотеки `ncurses` перепишите на Си решения задач 2.26–2.30 (см. стр. 2.26). В тех задачах, где это нужно, для задания задержек применяйте функцию `usleep` (её единственный параметр задаёт величину задержки в миллионных долях секунды и не может превышать по своему значению миллион). Все задачи из указанных переписывать не обязательно, выбирайте те, что вам больше нравятся; если вы почувствуете определённую уверенность в

работе с `ncurses`, можете сэкономить время и оставшиеся задачи не решать.

4.26. Напишите на Си с использованием `ncurses` программу, соответствующую условиям задачи 2.32 (см. стр. 26), при этом для смены цветов используйте изменение содержания цветовой пары, что позволит не выводить каждый раз сами символы.

4.27*. Напишите программу, которая принимает на вход от двух до ста параметров командной строки, рассматривает их как текст пунктов меню, выводит на очищенном для этого экране вертикальное меню из указанных пунктов, причём, если количества строк на экране не хватает, выводит столько пунктов, сколько помещается на экран, но в дальнейшем при необходимости осуществляет скроллинг, позволяя, таким образом, выбрать любой из пунктов. Если пунктов меньше, чем строк на экране — центрируйте меню по вертикали. После старта программы подсвечивается (сменой цвета фона) самый первый из пунктов, далее клавиши «стрелка вверх» и «стрелка вниз» соответствующим образом меняют выбранный пункт (при необходимости — со скроллингом), клавиши `Enter` и `Escape` завершают работу программы. Программа должна завершиться с кодом `0`, если была нажата клавиша `Escape`, а при нажатии `Enter` — с кодом от `1` до `argc`, соответствующим номеру выбранного пользователем пункта меню.

Напомним, что код завершения предыдущей команды можно узнать, дав команду `echo $?`.

Сложные описания

4.28. Массив `arr` описан следующим образом:

a) `int arr[245][12];`
b) `double arr[100][10][2];`
c) `char arr[5][5];`
d) `char *arr[5][5];`
e) `struct item *arr[20][3];` (считайте, что структура `item` описана в тексте программы ранее);
f) `struct item *arr[20];`

Опишите указатель `p` так, чтобы присваивание `p = arr` было корректным и при его выполнении не происходило преобразований типов.

4.29. Функция `f` имеет следующий заголовок:

a) `void f();;`
b) `int f(int t);;`

c) `void *f(int x);;`
d) `double f(int x, const char *str);;`
e) `void f(double (*vecp)[3]);;`
f) `double (*f(int len, double (*vecp)[3]))[3];`.

Опишите указатель на такую функцию.

4.30. Как известно, текстовые строки можно упорядочить разными способами; самый популярный из них — лексикографический, он же словарный, но есть и другие — например, по возрастанию встречающихся в строках чисел, или такой, при котором строки одинаковой длины упорядочиваются как в словаре, но более короткая строка, являющаяся префиксом более длинной, ставится *после* длинной. Есть и другие способы, здесь всё зависит от решаемой задачи.

Предположим, некий массив строк хранится в программе как массив указателей на их первые элементы, так что для работы с этим массивом применяются указатели вида

```
char **strptr;
```

(т. е. работа с этим массивом организована точно так же, как, например, с элементами командной строки). Потребовалась функция для сортировки такого массива; при этом известно, что в разных ситуациях, возникающих в одной и той же программе, подходы к порядку строк будет отличаться. В связи с этим принято решение задавать порядок строк (в строгих математических терминах — ***отношение порядка на строках***; напомним, что в общем случае отношение порядка на множестве задаётся операцией «меньше») с помощью функции, которая будет получать через параметры адреса двух строк, сравнивает их в соответствии с нужным отношением порядка и возвращает истину (`1`), если первая строка *меньше* второй (т. е. должна в итоговом массиве стоять раньше). Функция сортировки будет принимать три параметра: адрес начала массива указателей, длину массива и адрес функции, задающей требуемый порядок.

Как будет выглядеть описание указателя на функцию, задающую порядок? Не забывайте про модификатор `const`.

4.31. Расшифруйте следующие объявления и описания, переведя их на русский язык; опишите то же самое с использованием директивы `typedef`, снизив, насколько это возможно, трудность восприятия.

a) `int *(*f(int))[10];`
b) `void (*fpvec[15])(int, void (*)(int, void*), void*);`
c) `double (*(*repfptr)(double(*)(double)))(double);`
d) `int (*(*fvecpos)[4])(void(*)(double*));`

4.32. Заголовок функции `set_sr_func` введён с помощью директив `typedef` следующим образом:

```
typedef double (*d3vptr)[3];
typedef d3vptr (*search_for_vec_fptr)(d3vptr, int, double);
search_for_vec_fptr set_sr_func(int num, search_for_vec_fptr func);
```

Как будет выглядеть заголовок той же функции (с теми же именами параметров), если его записать без применения директив `typedef`?

Программы без стандартной библиотеки

4.33. Вернитесь к задаче 3.27 и перепишите её решение на Си без использования библиотечных возможностей; для оформления точки входа в программу и обёрток системных вызовов воспользуйтесь ассемблерными модулями, описанными в §4.12. Текст этих модулей не обязательно набирать вручную, они есть в архиве примеров к книге «Программирование: введение в профессию» (см. файлы `start.asm` и `calls.asm` в поддиректории `no_libc`).

4.34. В §3.6.9 разобран пример на языке ассемблера — программа для копирования файла. Напишите аналогичную программу на языке Си без использования стандартной библиотеки, воспользовавшись ассемблерными модулями для оформления точки входа в программу и обёрток системных вызовов; недостающие системные вызовы допишите сами.

Этюды и практикум

4.35. Вернитесь к задачам 2.58–2.60 и напишите описанные там программы на Си с использованием библиотеки `ncurses`. Не обязательно реализовывать все игры, выберите те, что вам больше нравятся, или придумайте свою.

4.36 (`shell-I`). Напомним, что программа, которая ведёт с пользователем диалог, обеспечивая нам возможность управления компьютером с помощью командной строки, называется ***командным интерпретатором***. Чрезвычайно полезно попробовать написать свой собственный (конечно, очень упрощённый) командный интерпретатор — такую программу, которая будет в цикле читать из потока стандартного ввода строки, вводимые пользователем, и, проанализировав очередную строку, выполнять её в качестве команды — в большинстве случаев это означает, что нужно запустить некую внешнюю программу с заданными параметрами командной строки и дождаться её завершения. Бóльшая часть сведений, которые для этого

потребуются, изложена в пятой части «Введения в профессию», так что основную часть командного интерпретатора мы оставим до следующей части нашего задачника; но командный интерпретатор — программа довольно сложная, и её создание правильнее разбить на отдельные этапы, к первому из которых вы уже, можно надеяться, готовы.

Первый этап подразумевает только чтение и анализ строк; иначе говоря, программа должна всё подготовить, чтобы можно было выполнять команды, но сами команды пока не запускать, поскольку мы этого ещё не умеем. Что до анализа строки, то фактически её нужно *разбить на слова*, используя символы пробела и табуляции как разделительные. Единственное «но» тут состоит в том, что в словах тоже бывают пробелы, и настоящий командный интерпретатор, такой как привычный нам `bash`, предусматривает несколько разных способов такие пробелы в слова вставить; из всех этих способов мы задействуем только двойные кавычки: внутри кавычек и пробел, и табуляция должны восприниматься как простые символы, не отделяющие слова друг от друга. Нечётное количество кавычек в строке должно рассматриваться как ошибка; после выдачи пользователю соответствующего сообщения нужно очистить всю занятую динамическую память и *продолжить работу* (!) — вспомните, ведь настоящий shell не завершает сеанс работы из-за каждой ошибки пользователя! Вообще завершать работу ваш командный интерпретатор должен только в одном случае: если в потоке стандартного ввода возникла ситуация «конец файла». Во всех остальных случаях следует продолжать чтение строк.

Учтите, что символ кавычки может встретиться где угодно, в том числе в середине слова; сам он в слова не входит, он просто переключает режим анализа: воспринимать ли пробелы и табуляции как разделители (обычный режим) или как простые символы (режим внутри кавычек).

Прочитав строку из стандартного потока ввода, ваша программа должна **сформировать список строк**, элементы которого будут содержать в ходе анализа слова. Список следует построить из элементов-структур, содержащих два поля: указатель на строку (имеющий тип `char*`) и указатель на следующий элемент списка, например:

```
struct word_item {
   char *word;
   struct word_item *next;
};
```

В таком виде введённая пользователем команда практически готова к выполнению, но, поскольку мы ещё не знаем, как её выполнить,

поступим проще: *напечатаем* полученные слова (каждое на отдельной строке; будет лучше, если каждое слово мы на печати на всякий случай возьмём в квадратные скобки, чтобы было видно, если вдруг у нас в словах окажутся ненужные пробелы), затем освободим всю захваченную динамическую память и продолжим работу, то есть приступим к чтению следующей строки.

Перед вводом очередной строки выдайте какое-нибудь приглашение к вводу, например, символ «>» и пробел. Работа с вашей программой должна в итоге выглядеть примерно так:

```
user@host:~$ ./shell1
> abra schwabra kadabra
[abra]
[schwabra]
[kadabra]
>     abra          schwabra          kadabra
[abra]
[schwabra]
[kadabra]
> abra "schwabra kadabra" "foo    bar"
[abra]
[schwabra kadabra]
[foo    bar]
> abra schw"abra ka"dab"ra" foo"   "bar
[abra]
[schwabra kadabra]
[foo    bar]
> abra schwabra kadabra"  foo bar
Error: unmatched quotes
> abraschwabrakadabra
[abraschwabrakadabra]
> ^D
user@host:~$
```

Обязательно проверьте свою программу на утечки памяти! Для этого откройте два терминальных окна так, чтобы можно было видеть их оба одновременно; в одном из них запустите программу `top`. Переключившись в другое окно, запустите там вашу программу в составе примерно такого конвейера (обратите внимание на апострофы!):

```
yes 'abra schw"abra ka"dab"ra" foo"  "bar' | ./shell1 > /dev/null
```

Сразу после этого ваша программа (в нашем примере её исполняемый файл называется `shell1`) появится в верхних строчках списка, выдаваемого программой `top` в соседнем окне; подождите несколько

секунд, и если за это время числа, показывающие количество занимаемой вашей программой памяти (в особенности число в колонке `VIRT`), не начнут плавно расти, то всё, скорее всего, в порядке; если же они действительно стали расти, немедленно прибейте ваш конвейер нажатием `Ctrl-C` (иначе ваша программа запросто может сожрать всю доступную память и «повесить» систему) и приступайте к поиску — где-то в вашей программе память выделяется, но не освобождается.

4.37. Доработайте решение задачи 4.36 так, чтобы в слово можно было включить символ двойной кавычки. Для этого предусмотрите ещё один «специальный» символ — «\», который будет «экранировать» символ, стоящий следом за ним; таких «экранируемых» символов следует предусмотреть два: символ двойной кавычки и сам символ «\», ведь его тоже может потребоваться ввести в слово. Как реагировать, если следом за «\» в строке располагается какой-то ещё символ — придумайте сами, только не завершайте при этом работу программы (увы, приходится об этом напоминать).

4.38. Снабдите решение задачи 4.36 возможностью сформировать пустое слово. Обозначением пустого слова считайте два символа двойных кавычек, встреченные подряд *вне слов*, то есть когда накопленное слово пусто, а сразу после закрывающей кавычки идёт либо пробельный символ, либо конец строки.

К части 5 «объекты и услуги операционной системы»

Файлы и потоки (уровень системных вызовов)

5.01. Напишите программу, которая получает аргументом командной строки имя файла и, открыв его на чтение, с помощью системного вызова `lseek64` определяет (и печатает) его длину в байтах. Обязательно проверьте корректность работы вашей программы на разных файлах.

Конечно, в реальной задаче следует для определения размера файла воспользоваться вызовом `stat`, тогда файл не придётся открывать, а узнать размер можно будет даже для такого файла, к которому нет доступа на чтение. Всему своё время!

5.02. Используя для чтения и записи только системные вызовы (`read` и `write`), напишите программу, которая копирует свой поток стандартного ввода в поток стандартного вывода, т. е. ведёт себя как хорошо известная команда `cat` без параметров. Чтение и запись производите блоками (например, чтение запрашивайте по 1024 или 4096 байт; записывайте всегда столько, сколько прочитано). Проверьте корректность работы вашей программы как с терминалом, так и с различными комбинациями перенаправлений; особое внимание обратите на сохранение размера файла, когда перенаправлены и ввод, и вывод.

5.03. Напишите на Си программы, соответствующие условиям задач 2.53 и 2.54 (см. стр. 32), адаптированным к терминологии Си (термин «типизированный файл» замените термином «бинарный файл»). Для работы с бинарными файлами используйте системные вызовы, для работы с текстовыми — высокоуровневый ввод-вывод.

5.04. Используя для чтения файла только системные вызовы, напишите программу, которая получает через параметр командной

строки имя файла и (в предположении, что файл текстовый) подсчитывает в нём количество строк. Чтение всегда запрашивайте блоками (например, по 4096 байт). Полученное количество строк напечатайте (используйте обычный `printf`).

5.05. Напишите программу, которая получает четыре параметра командной строки: имя файла, начальную позицию, длину и значение байта; указанный файл открывает на запись и, начиная с заданной позиции, заполняет в этом файле отрезок нужной длины указанным байтом. Запись организуйте так, чтобы, если указанная длина отрезка превышает 4096 байт, вывод производился блоками по 4096 байт, и лишь один (последний) вызов `write` записывал меньше (сколько ещё осталось). Проверить корректность работы вашей программы можно с помощью утилиты `hexdump`.

5.06. Известен очень простой (и совершенно не стойкий к взлому, но сейчас это не так важно) способ зашифровать данные, используя секретный ключ: просто применить к данным и указанному ключу операцию «исключающее или» (`xor`). Напомним на всякий случай, что в языке Си эта операция обозначается символом «`^`».

Напишите программу, которая получает два параметра командной строки: имя файла и ключ (ключ рассматривается как 32-битное беззнаковое целое число), и *прямо на месте* (не создавая никаких дополнительных файлов) «зашифровывает» заданный файл заданным ключом. Насколько это возможно, работу производите блоками по 4096 байт. Учтите, что длина файла может оказаться не кратна четырём байтам. Длина файла, естественно, не должна измениться в результате зашифровки; если всё сделать правильно, это никак не усложнит вашу программу.

Поскольку $x \oplus a \oplus a \equiv x$ для любых x и a, расшифровка производится точно так же, как и зашифровка, то есть нужно второй раз прогнать вашу программу с тем же самым ключом. Это позволяет проверить корректность программы: возьмите какой-нибудь файл более-менее серьёзного размера (например, фотографию или музыкальный трек), создайте его копию, «зашифруйте» её, убедитесь, что содержимое файла теперь не имеет ничего общего с оригиналом, повторно запустите вашу программу и теперь (например, с помощью `diff` или `md5sum`) убедитесь, что результат расшифровки тождественно (байт в байт) совпадает с оригинальным файлом.

5.07. Вернитесь к условиям задач 2.55–2.57 (см. стр. 32) и напишите аналогичные программы на Си с использованием файловых системных вызовов. Текст условия задачи 2.55 адаптируйте к реалиям Си — за неимением аналога паскалевского типа `string` используйте

массив из 60 элементов типа `char`, предполагающий нулевой байт на конце.

5.08. Напишите программу, которая получает через аргумент командной строки имя файла и, используя библиотечные функции `opendir`, `readdir` и `closedir`, находит в текущей директории и во всех её поддиректориях файлы с заданным именем. Для каждого найденного файла напечатайте его *относительное* имя, отсчитываемое от текущей директории.

5.09. Напишите программу, которая получает через аргумент командной строки имя файла и, используя вызовы `stat` и `lstat`, выдаёт (в виде текста на английском языке) все свойства этого файла, какие только возможно, начиная с типа файла; для символических ссылок печатайте информацию как по самой ссылке, так и по файлу, на который она ссылается (если, конечно, таковой есть; если его нет — напечатайте слово `dangling`).

Управление процессами

5.10. Что может напечатать следующая программа, если вызов `fork` и все операции вывода пройдут успешно? Перечислите все варианты. Считайте, что вывод идёт на терминал, так что выдача символа перевода строки приводит в вытеснению буфера вывода.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int pid;
    printf("start\n");
    pid = fork();
    if(pid == 0) {
        printf("child\n");
    } else {
        printf("parent\n");
    }
    printf("finish\n");
    return 0;
}
```

5.11. В некоторой программе содержится описание локальной переменной:

```
int *status;
```

а позже в тексте — такой вызов:

```
wait(status);
```

Больше переменная `status` в программе нигде не упоминается. В чём состоит ошибка программиста, писавшего эту программу? На всякий случай подчеркнём, что ошибка здесь совершенно фатальная и выдаёт глубочайшее непонимание происходящего, но при этом программа пройдёт компиляцию и даже будет делать вид, что «работает», хотя, конечно, работать будет неправильно.

Как следует исправить эту программу?

5.12. Напишите программу, которая принимает произвольное количество (не менее одного) аргументов командной строки и рассматривает их как имя и аргументы для запуска внешней программы; например, если ваша программа называется `prog` и пользователь запустил её командой

```
./prog ls -l -R
```

— то ваша программа должна запустить программу `ls` с аргументами `-l -R`. Сделайте так, чтобы в зависимости от того, как запущенная программа завершится, ваша программа печатала либо слово `exited` и код завершения, либо слово `killed` и номер сигнала, которым запущенная программа была убита.

5.13. Напишите программу, которая через аргументы командной строки получает имена и аргументы для запуска внешних программ (произвольное их количество), разделённые параметром, состоящим из двух символов «`;`»; чтобы при запуске командный интерпретатор не считал параметр «`;;`» имеющим особый смысл, заключайте его в апострофы, например:

```
./prog ls -l / ';;' sleep 10 ';;' cat file1.txt file2.txt
```

Ваша программа должна запустить на одновременное (параллельное) исполнение все указанные программы с заданными аргументами и напечатать **имена** тех из них, которые завершились успешно, то есть вызовом `_exit` с параметром `0`. Печатать имена следует по мере завершения запущенных программ. Закончить работу следует сразу после завершения последней из запущенных программ, но не раньше.

5.14. Допустим, в вашей программе запускается много дополнительных процессов, и вы знаете, что некоторые из них уже могли завершиться (и превратиться в зомби), но не все. Нужно убрать всех «готовых» зомби, но при этом не блокироваться в ожидании тех, кто ещё не завершился. Каким фрагментом кода это можно сделать?

5.15 (`shell-II`). Вернитесь к программе, которую вы писали, решая задачи 4.36–4.38, и модифицируйте её так, чтобы введённые пользователем команды не печатались, а выполнялись, то есть чтобы ваша программа, прочитав очередную команду, запускала внешнюю программу, имя которой задано первым словом, а остальные слова представляют собой аргументы командной строки.

Поскольку процесс в ОС Unix не может изменить текущую директорию *другому* процессу, а только себе, команду `cd` (то есть — буквально — ситуацию, когда первое слово введённой команды состоит из двух символов — «`c`» и «`d`») придётся рассматривать как особый случай. Запуск внешней программы здесь бесполезен; вместо этого проверьте. что задан ровно один параметр (т. е. введённая команда состоит ровно из двух слов: собственно «`cd`» и имени директории), и смените текущую директорию с помощью системного вызова `chdir`.

5.16. Доработайте программу, написанную при решении предыдущей задачи, так, чтобы команду `cd` можно было дать без параметров, и она при этом, как и в «настоящем» командном интерпретаторе, устанавливала в качестве текущей домашнюю директорию пользователя.

5.17 (`shell-III`). Когда речь идёт о разборе текста на формальном языке, в большинстве случаев первой стадией такого разбора становится *лексический анализ*, то есть разбиение текста на *лексемы* или *токены*; решение задачи 4.36 можно рассматривать как простейший лексический анализатор, а токенами здесь выступают те самые слова, на которые наша программа делит введённую пользователем строку.

В языках программирования некоторые символы или определённые последовательности символов (как правило, короткие — в два символа, очень редко в три) выделяются в отдельные лексемы, даже если их не отделить от остального текста пробелами. Таковы обычно всевозможные скобки, пунктуационные символы вроде «`;`», «`,`» или «`:`», обозначения арифметических операций — такие как «`+`», «`/`», и т. п. Примерами разделителей, состоящих более чем из одного символа, служат обозначения некоторых операций языка Си: `+=`, `&&`, `->`, а, скажем, `<<=` и `>>=` — это тот самый редкий случай разделителя из трёх символов. Отметим, что обозначения операций не обязаны быть разделителями: так, в Паскале слова `div`, `mod`, `and`, `or` или `not` разделителями не являются, ведь если написать, к примеру, `aandb` — это будет совсем не то же самое, что `a and b` (тогда как в Си `a&&b` и `a && b` — совершенно одно и то же, поскольку здесь конъюнкция обозначена *разделителем* `&&`).

Существуют свои разделители и во входном языке командного интерпретатора. В нашем упрощённом интерпретаторе можно ограничиться небольшим количеством разделителей, но совсем без них обойтись невозможно.

Начнём мы с того, что снабдим наш командный интерпретатор возможностью запуска процессов в фоновом режиме, а для этого нам потребуется символ «`&`» (амперсанд), и его нужно оформить как разделитель. Говоря более конкретно, нужно, чтобы ваша программа, встретив этот символ *вне кавычек*, рассматривала его как «отдельное слово» вне всякой зависимости от того, есть вокруг него пробелы или нет, причём это должно быть некое «хитрое» слово, отличающееся от простого слова из одного амперсанда (такое можно получить, если заключить амперсанд в кавычки, а вокруг кавычек поставить пробелы).

Амперсанд означает, что команду следует выполнить «в фоновом режиме», то есть, попросту говоря, *не ждать её завершения* — запустить и «забыть», продолжая читать команды с клавиатуры и выполнять их. Нужно будет только не забыть вовремя убрать зомби, когда такая фоновая команда всё-таки завершится. Настоящие командные интерпретаторы позволяют ввести сразу несколько команд, разделив их амперсандами, при этом все команды, кроме последней, будут выполняться в фоновом режиме; но эта возможность всё равно применяется очень редко, так что мы поступим проще: будем считать, что амперсанд должен располагаться в конце строки, т. е. если после него есть ещё что-то, кроме пробелов, будем выдавать ошибку.

Если пользователь ввёл всё правильно, то есть амперсанд, встреченный вне кавычек, является в строке последним значащим символом, то ваша программа должна введённую команду исполнить как фоновую (сам амперсанд, конечно, в такую команду не входит, используются только настоящие слова, расположенные до него). Само по себе это даже проще, чем обычное выполнение — просто не делать `wait`, и всё; но сам факт возможности существования фоновых процессов означает, что выполнение команд в *обычном* режиме, не фоновом, становится несколько сложнее, хотя и не сильно. Попробуйте сами сообразить, в чём тут проблема и как с ней справиться, если же сразу не догадаетесь — воспользуйтесь указаниями на стр. 129.

На всякий случай отметим, что в последующих задачах, посвящённых командному интерпретатору, потребуются ещё разделители `>`, `<`, `>>`, `|`, `;`, `(`, `)`, `&&` и `||`. Если вы планируете довести эту программу до конца — возможно, имеет смысл предусмотреть все разделительные лексемы прямо сейчас, вместе с одиночным амперсандом, чтобы больше не переделывать ваш анализатор строки; пока соответствующие возможности не реализованы, при появлении

в пользовательском вводе любого из этих разделителей следует фиксировать ошибку с диагностикой вроде «`Feature not implemented yet`».

Перенаправление ввода-вывода

5.18. Напишите программу, которая принимает не менее двух аргументов командной строки и рассматривает первый аргумент — как имя файла, остальные — как имя и аргументы для запуска внешней программы; например, если ваша программа называется `prog` и пользователь запустил её командой

```
./prog file1.txt ls -l -R
```

— то здесь имеется в виду файл `file1.txt`, внешняя программа `ls` и аргументы `-l -R`. Запустите указанную внешнюю программу с указанными аргументами так, чтобы:

a) её стандартный вывод был перенаправлен в указанный файл (если такого файла нет, он должен быть создан, если он уже есть — перезаписан; если открыть файл не удалось, должна быть выдана диагностика, соответствующая возникшей ситуации);
b) её стандартный ввод шёл из указанного файла (если такого файла нет, должна быть зафиксирована ошибка и выдана соответствующая диагностика);
c) стандартный вывод был перенаправлен в указанный файл *на добавление в конец* (если файла нет, создайте его, но если он есть, новое содержимое должно быть добавлено к старому);
d) стандартный вывод был перенаправлен в указанный файл на добавление в конец, причём если такого файла нет, должна быть зафиксирована ошибка.

5.19 (`shell-IV`). Возьмите за основу программу, написанную в соответствии с условиями задачи 5.17; если вы ещё не решали её, можно начать с версии программы, написанной в качестве решения задачи 5.15, а к фоновым процессам вернуться позднее. Условия задачи 5.17 стоит хотя бы прочитать и понять, о чём там идёт речь, поскольку повторять рассуждения о лексическом анализе и разделительных лексемах мы здесь не будем.

Модифицируйте ваш анализатор строки так, чтобы он воспринимал в качестве разделителей, наряду с уже имеющимся токеном «`&`», также токены «`>`», «`<`» и «`>>`». Обратите внимание, что любые из этих символов должны иметь специальный смысл *только вне кавычек*, тогда как внутри кавычек любой символ, кроме собственно кавычки, считается просто символом. Таким образом, например, `">>"`

и просто >> — это совершенно не одно и то же. Кроме того, >> — это не то же самое, что два отдельно стоящих символа >.

Модифицируйте остальную часть программы так, чтобы любой из этих токенов, если после него поставить обычное слово (предполагается, что это слово — имя файла), производил соответствующее перенаправление: знак < означает перенаправление стандартного потока ввода на чтение из заданного файла, знак > — перенаправление стандартного вывода на запись в заданный файл (если файла нет, он должен быть создан, если он есть — перезаписан с нуля), знак >> — перенаправление стандартного вывода на добавление в конец заданного файла (если файла нет, он должен быть создан).

Фиксируйте ошибки в случаях, если:

- пользователь пытается перенаправить один и тот же поток два и более раза — дважды встречается один и тот же символ перенаправления или в одной команде сочетаются > и >>;
- следующий токен после любого из токенов <, > или >> не является простым словом (т. е. является разделителем);
- после специальных токенов наблюдаются какие-то ещё простые слова, кроме ожидаемых (имён файлов сразу после <, > или >>).

Сочетания перенаправлений с символом фонового режима (который по-прежнему должен быть последним во введённой строке), а также и между собой, если они осмысленны (одно перенаправление на ввод, другое на вывод) должны нормально работать.

Сигналы

5.20. Напишите программу, которая после запуска выдаёт сообщение «`Press Ctrl-C to quit`» и после этого ничего не делает, но и не завершается; при нажатии `Ctrl-C` выдаёт сообщение «`Good bye`» и завершается. Обязательно убедитесь с помощью программы `top`, что ваша программа во время своего «ничегонеделания» не потребляет процессорное время.

5.21. Напишите программу, которая после запуска ничего видимого не делает, но и не завершается, в том числе и при нажатии `Ctrl-C`; при получении сигнала `SIGUSR1` выдаёт в поток стандартного вывода число, сколько раз к текущему моменту пользователь успел нажать `Ctrl-C` (на самом деле — сколько раз приходил сигнал `SIGINT`).

Для завершения работы программы используйте `Ctrl-\` или убейте её из другого терминального окна.

5.22. Напишите программу, которая один раз в секунду печатает (без перевода строки) некоторый символ, причём сразу после запуска это символ «+», после нажатия `Ctrl-C` (то есть по сигналу `SIGINT`) печатаемый символ меняется на «-», после нажатия `Ctrl-\` (то есть по `SIGQUIT`) — меняется обратно на «+», причём в обоих случаях соответствующий символ печатается немедленно после нажатия комбинации клавиш, и очередная секунда отсчитывается с этого момента. При нажатии `Ctrl-C` дважды с интервалом менее одной секунды программа завершает работу, двойное нажатие с бóльшим интервалом никакого эффекта не оказывает. Для отслеживания секундного интервала используйте системный вызов `alarm` и сигнал `SIGALRM`.

5.23. Напишите программу, которая ожидает пользовательского ввода, причём если пользователь ничего не вводит в течение пяти секунд, спрашивает пользователя, не заснул ли он (конкретный текст вопроса можете придумать сами). При нажатии `Ctrl-C` (сигнал `SIGINT`) выдаёт информацию о том, сколько строк и символов успел к настоящему моменту ввести пользователь (если пользователь дальше ничего не вводит, пять секунд до очередного вопроса отсчитывается от момента нажатия `Ctrl-C`). При повторном нажатии `Ctrl-C` менее чем в течение пяти секунд программа завершает работу. Для отслеживания пятисекундного интервала используйте системный вызов `alarm` и сигнал `SIGALRM`.

5.24. Вернитесь к задаче 5.19 (если её ещё не решали — то к задаче 5.17) и перенесите цикл «очистки от зомби» в обработчик сигнала `SIGCHLD`, как это, собственно говоря, и положено делать. Учтите, что этот цикл помешает работе цикла обычных `wait`'ов, который выполняется при ожидании завершения нефоновой команды, так что на время этого цикла следует возвращать для сигнала `SIGCHLD` диспозицию по умолчанию.

Каналы и конвейеры

5.25. Напишите программу, в которой порождается дополнительный процесс, связанный неименованным каналом (pipe) с родительским процессом; родительский процесс читает информацию из канала, пока она не кончится, и всё прочитанное тут же выдаёт в свой стандартный вывод; порождённый процесс записывает в канал несколько текстовых строк (например, какое-нибудь стихотворение) и завершается. Родительский процесс должен, дочитав всё из канала, дождаться завершения порождённого процесса (на самом деле порождённый, скорее всего, уже завершился, но `wait` сделать всё

равно стоит) и закончить работу. Если всё сделать правильно, строки, которые порождённый процесс передавал в канал, окажутся в итоге выведенными на экран (родительским процессом), после чего программа завершит работу.

Эта задача может показаться бессмысленной, но она позволит, как говорится, «опробовать инструмент» (в данном случае — `pipe`). Если что-то пошло не так, обратитесь к указаниям на стр. 132.

Для большей наглядности добавьте в вашу программу сразу после вызова `pipe` простой `printf`, который напечатает численные значения обоих дескрипторов созданного канала. Если вам придёт в голову какой-нибудь ещё эксперимент — обязательно сделайте его, компьютер не взорвётся; осознать, что это за сущность такая — «канал», — очень важно для дальнейшего развития.

5.26. Напишите программу, которая через аргументы командной строки получает имена и аргументы для запуска **двух** внешних программ, разделённые параметром, состоящим из двух символов «`;`», как в задаче 5.13, и запускает эти программы на одновременное (параллельное) выполнение, связав их конвейером, т. е. стандартный вывод первой программы должен идти на стандартный ввод второй программы.

5.27. Напишите программу, которая получает через аргументы командной строки имя и аргументы для запуска внешней программы (так же, как это было сделано в задаче 5.12). Запустите эту внешнюю программу, подав ей на стандартный ввод **через неименованный канал** текстовое представление чисел 1, 2, 3 и т. д. до 1000000, по одному числу в строке. После исчерпания чисел закройте свой конец канала, чтобы показать партнёру, что здесь больше ничего не ожидается. Ваша программа должна работать (т. е. не должна завершаться) до завершения внешней программы; после такового напечатать информацию об обстоятельствах её завершения (код завершения при самостоятельном завершении, номер сигнала при завершении по сигналу). Ситуация с досрочным завершением внешней программы (когда цикл записи чисел в канал ещё крутится) должна быть обработана корректно.

5.28. Напишите программу, которая получает через аргументы командной строки имя и аргументы для запуска внешней программы (как в предыдущей задаче). Запустите эту внешнюю программу, перехватив её поток стандартного вывода и напечатав:

а) только первые 10 строк выведенной ею информации (при этом внешней программе следует дать возможность доработать до конца, продолжая чтение из канала, но не выдавая прочитанное);

b) только те из выданных ею строк, которые начинаются с пробела или табуляции;
c) первые 20 символов каждой выданной ею строки (если строка короче, печатать её целиком; символ перевода строки печатать в любом случае);
d) первое слово каждой строки.

5.29. Модифицируйте решение задачи 5.27 так, чтобы стандартный вывод запускаемой внешней программы тоже перехватывался, а после завершения работы на экран было выведено количество символов и количество строк, выданных внешней программой за время работы.

5.30. Напишите программу, которая через аргументы командной строки получает имена и аргументы для запуска **двух** внешних программ, разделённые параметром, состоящим из двух символов «`;`», как в задаче 5.26, и запускает эти программы на одновременное (параллельное) выполнение, причём на стандартный ввод второй программе подаётся:

a) каждая вторая строка из выданных первой программой;
b) текст, составленный из тех строк, выданных первой программой, которые начинаются с пробела или табуляции;
c) первые 10 символов каждой строки, выданной первой программой (если очередная строка короче 10 символов, её следует подать целиком; части разных строк следует разделять переводом строки).

5.31 (`shell-V-simple`). Возьмите за основу решение задачи 5.19, или более позднюю её модификацию (например, полученную при решении задачи 5.24). Доработайте анализатор строки, добавив в этот раз в качестве разделителя символ «`|`». Остальную часть программы модифицируйте так, чтобы этот символ рассматривался как обозначение конвейера. Пока можете считать, что больше одного такого символа во введённой команде быть не может, т. е. ваш конвейер не может состоять более чем из двух элементов; если такое ограничение кажется вам ненужным, можете сразу же решать задачу 5.32, которая отличается только отсутствием ограничения на количество элементов конвейера.

Если во введённой команде присутствует символ конвейера, следует сформировать две отдельные командные строки для первой команды и для второй; символ фонового режима по-прежнему допускается только последним во всей строке, что касается символов перенаправления ввода-вывода, то вы можете сами для себя решить, удобнее вам допускать их непосредственно перед знаком конвейера

или же тоже только в конце всей строки. В обоих случаях перенаправление ввода должно относиться к первому элементу конвейера, перенаправление вывода — к последнему.

Программы, запускаемые в составе конвейера, должны работать одновременно, то есть нельзя сначала дожидаться завершения первой из них, и только потом запускать вторую — объём буфера канала ограничен, и если первая программа будет выдавать достаточно большой объём данных в свой стандартный вывод, а читать эти данные будет некому, ваша система процессов просто повиснет. Моментом завершения конвейера считается момент завершения последнего из составляющих его процессов, причём «последнего» в *хронологическом* смысле; так, если второй элемент конвейера уже завершился, а первый всё ещё работает, конвейер следует считать продолжающим работу.

5.32* (`shell-V`). Переделайте решение задачи 5.31, убрав ограничение на количество элементов конвейера.

Терминал

5.33. Напишите программу, которая получает ровно один аргумент командной строки — имя текстового файла, первая строка которого содержит некий пароль, т. е. паролем считаются символы от начала файла до перевода строки, не включая его; программа должна запросить у пользователя ввод пароля, перепрограммировать терминал так, чтобы вводимые символы не отображались на экране, считать вводимый пользователем пароль и проверить, совпадает ли он с находящимся в файле. Если пароль совпал — завершить работу «успешно», ничего не печатая, если не совпал — выдать в диагностический поток сообщение `incorrect password` и завершиться с кодом неуспеха. Не забудьте сразу после ввода пароля вернуть настройки терминала в исходное состояние!

В случае, если ввод идёт не с терминала (перенаправлен), откажитесь работать.

5.34*. Напишите программу, которая позволяет вводить текст (строчка за строчкой) и записывать его в заданный файл, при этом в каком-то другом файле имеется словарь слов (простое их перечисление по одному в строке); когда пользователь нажимает клавишу `Tab`, программа проверяет вводимое слово и, если введённые буквы, относящиеся к текущему слову, представляют собой префикс какого-то одного слова из словаря — дописывает это слово во вводимой строке, как если бы пользователь ввёл его целиком; если на момент нажатия `Tab` введённые символы окажутся префиксом *нескольких* слов —

следует поступить так же, как поступает, например, командный интерпретатор: перевести строку, показать все подходящие слова из словаря, затем снова напечатать ту часть строки, которую пользователь уже ввёл, и позволить пользователю продолжить ввод.

Программа должна получать два аргумента командной строки: файл словаря и тот файл, куда будет записываться результат пользовательского ввода.

5.35*. Дополните решение предыдущей задачи возможностью редактировать вводимую строку с помощью стрелок влево и вправо, `Backspace`, `Del` и (желательно) комбинации `Ctrl-W` для удаления последнего слова (до ближайшего пробельного символа).

5.36* (`shell-edit`). Возьмите какую-то из версий вашего командного интерпретатора (лучше всего, конечно, самую последнюю) и оснастите её возможностями редактирования вводимой команды (как в задаче 5.35) и дописывания имён команд и файлов (как в задаче 5.34, только без словаря). Во всех словах, кроме первого в строке, пытайтесь «дописать» имя файла, найдя на диске соответствующий файл или все подходящие файлы (напомним, имена файлов бывают абсолютные и относительные, здесь это важно), а в первом слове, если в нём пока нет ни одного слэша, следует дописывать имена исполняемых файлов из директорий, содержащихся в переменной `PATH`.

Эта задача не входит в основной набор этапов создания командного интерпретатора, поскольку всё остальное там можно сделать и без этого, а трудоёмкость этой задачи довольно высока; но опыт, который вам даст решение этой задачи, определённо стоит потраченного времени.

5.37 (`daemon`). Напишите программу-демона (*daemon*, см. §5.4.5). Пусть ваш демон «просыпается» один раз в пять минут и дописывает в некий файл (лучше всего — расположенный в директории `/tmp`, запись туда разрешена всем) строку с указанием того, кто он такой, каков его `pid`, сколько времени, по его сведениям, он уже работает и сколько за время работы получил сигналов `SIGUSR1`; при получении такого сигнала стоит аналогичную строку в тот же файл записать немедленно, не дожидаясь очередных пяти минут.

5.38 (`daemon-logging`). Дополните решение предыдущей задачи выдачей диагностических сообщений через системную журнализацию. Такое сообщение стоит выдать при старте, а также дублировать журнальными сообщениями информацию, выводимую в файл.

5.39 (`shell-VI`). Дополните ваш командный интерпретатор (любой степени готовности, начиная от второго этапа, где просто выполнялись команды) правильно организованной работой с группами

процессов. Процессы, запускаемые для исполнения очередной введённой команды, должны образовывать отдельную группу, новую для каждой команды; если для выполнения одной команды нужно породить больше одного процесса (например, при выполнении команды, содержащей конвейер), все эти процессы должны быть в *одной* группе, отличной от группы, в которой находится сам командный интерпретатор, и от групп, в которых выполняются запущенные ранее фоновые команды (у каждой из них, заметим, тоже должна быть своя группа). При выполнении *нефоновой* команды интерпретатор должен на время её выполнения объявить её группу процессов текущей в сеансе, а после окончания её выполнения снова сделать текущей группой свою собственную.

Если всё сделать правильно, то, в частности, при нажатии `Ctrl-C` во время работы программы, запущенной в ответ на очередную команду, будет убиваться только эта программа, а сам ваш командный интерпретатор сигнала `SIGINT` не получит и продолжит работу. Собственно говоря, так и должно быть.

Дополнение

5.40* (`shell-VII`). Возьмите командный интерпретатор в той версии, которая описана в задаче 5.32 (возможно, с дополнениями из задач 5.39 и/или 5.36, они никоим образом не помешают). Дополните анализатор вводимой строки, чтобы он обрабатывал ещё пять токенов-разделителей: «`(`», «`)`», «`;`», «`||`» и «`&&`». Реализуйте *связки* команд: команды, связанные точкой с запятой, выполняются последовательно, сначала первая, потом вторая, вне зависимости от результатов; в связке `&&` выполняется сначала первая команда, а вторая выполняется лишь в случае, если первая завершилась успешно; связка `||` работает аналогично, только вторая команда выполняется при *неуспехе* первой. Связки и скобки рассмотрены в §1.2.15; возможно, есть смысл перечитать тот параграф, а ещё — поэкспериментировать со связками в настоящем командном интерпретаторе, чтобы понять, как выглядит их работа.

Круглые скобки реализуются порождением отдельного процесса, который и выполнит (проинтерпретирует) все те команды и связки, которые указаны в скобках; мы ведь помним, что процесс порождается как копия текущего, в данном случае — копия командного интерпретатора? Этот дополнительный процесс (вместе со всеми теми, кого он будет порождать при выполнении содержимого скобок), очевидно, имеет свои стандартные потоки, которые могут быть перенаправлены; как следствие, группа команд, заключённых в скобки, может (как целое) выступать элементом конвейера, для неё может

быть выполнено перенаправление стандартных потоков (в файлы и из файлов), она может быть как правым, так и левым элементом любой из связок.

Ещё работа над этой задачей — прекрасный повод превратить, наконец, амперсанд «&» из флажка, которым мы его сделали, в то, чем он должен быть на самом деле — в связку, в которой левый элемент выполняется как фоновый процесс, а правый — как обычный процесс (нефоновый, завершения выполнения которого дожидаются).

К части 6 «сети и протоколы»

IP-адреса

6.01. Что из этого **не является** валидной записью ip-адреса и почему?

```
195.42.170.128    192.168.210.5    1.1.1.1    129.253.254.255
198.260.101.15    231.0.0.0     10.10.10.10   100.200.300.400
127.0.0.1       127.128.129.130     0.1.2.3         254.1.1.1
```

6.02. Определите, **не подглядывая в таблицы и прочие материалы**, как будет выглядеть маска подсети при указанной длине префикса:
a) `/24` b) `/16` c) `/8` d) `/32` e) `/30` f) `/25` g) `/23` h) `/20`

6.03. Какая длина префикса соответствует маске подсети `255.240.0.0`?

6.04. Перечислите (с указанием адреса сети и длины префикса) все подсети, в которые входит ip-адрес `195.42.170.129`.

6.05. Напишите (на Си или даже на Паскале, здесь это неважно) программу, которая решает предыдущую задачу для произвольного ip-адреса: принимает десятичную запись ip-адреса через аргумент командной строки и печатает (через запятую, через пробел или просто на отдельных строках) все ip-подсети, в которые он входит.

Простейшее взаимодействие по сети

Задачи этого и последующих параграфов предполагают написание программ на языке Си. Решение задач, предполагающих связь по компьютерной сети, можно проверить на локальной машине, указывая в качестве адреса сервера `127.0.0.1` или просто `0`, но если у вас есть хоть какая-то возможность разнести взаимодействующие процессы по разным компьютерам — обязательно сделайте это.

Сложность программ при этом никак не изменится (им, собственно, всё равно, как работать), а вот наглядность происходящего (лично для вас!) сильно возрастёт, и скорее всего — вместе с вашей самооценкой.

6.06. Напишите две программы — для передачи дейтаграмм и для их получения. Первая программа получает три аргумента командной строки: ip-адрес, номер порта и произвольную строку; указанную строку (**без** ограничивающего нуля!) она должна отправить в виде дейтаграммы на заданные адрес и порт. Вторая программа получает через командную строку один аргумент — номер порта; на этом порту (и на всех адресах, имеющихся на локальной машине) она должна ожидать прихода дейтаграмм. Получив очередную дейтаграмму, программа должна напечатать адрес и порт отправителя, а содержание дейтаграммы воспроизвести посимвольно, заменяя вопросительными знаками те символы, которые не могут быть напечатаны (не входят в диапазон 32–126 и не являются ни переводом строки, ни табуляцией).

6.07. Напишите простой UDP-сервер — программу, которая подсчитывает количество полученных дейтаграмм и их общий объём, и в ответ на каждую полученную дейтаграмму отправляет (также в виде дейтаграммы) текстовое представление обоих счётчиков. Для проверки работоспособности вашей программы придётся написать также клиент — программу, которая отправляет на заданный адрес/порт дейтаграмму заданной длины, дожидается ответа и распечатывает его, если же ответ не поступает в течение 15 секунд (или какого-то другого интервала времени) — сообщает об этом и завершается.

Обе программы должны всю необходимую информацию получать через аргументы командной строки; программа-сервер должна после запуска «демонизироваться» (см. §5.4.5).

6.08. Напишите простой TCP-сервер, который сразу после получения очередного запроса на соединение принимает его, отправляет клиенту (в текстовом виде) текущие дату и время, а также ip-адрес и порт *клиента*. Это, заметим, имеет самый прямой смысл: если клиент находится за NAT'ом, он может не знать, с какого — реального, а не «внутреннего» — ip-адреса он работает. Сразу после отправки данных сервер разрывает соединение.

Клиентскую программу писать не обязательно, можно воспользоваться программой `telnet`; если вдруг выяснится, что в вашей системе её нет — установите её, она ещё не раз потребуется.

В этой задаче можно не беспокоиться относительно проблемы очерёдности действий: вся передаваемая вашим сервером информация

«уляжется» в один ip-пакет, сам он ждать данных от клиента не будет, так что никаких блокировок не произойдёт и отвалиться по тайм-ауту никто не успеет. Учтите, что этот случай — исключение, а не правило.

Системный вызов `select`

Задачи этого параграфа не имеют прямого отношения к компьютерным сетям; они призваны помочь понять, как работает и для чего нужен вызов `select`, без которого при создании сетевых программ придётся туго.

6.09. Напишите программу, которая задаёт пользователю вопрос «`What is your name, please?`» и, дождавшись ответа, выдаёт вежливое «`Nice to meet you, dear` *(имя)*`!`»; если пользователь не вводит имя в течение 15 секунд, программа заявляет «`Sorry, I'm terribly busy.`» и завершается. Для отслеживания времени используйте вызов `select`.

6.10. Вернитесь к задаче 5.23 и напишите программу, удовлетворяющую её условиям, без применения сигнала `SIGALRM`. Для отслеживания интервала используйте `select`.

6.11. Напишите программу, которая получает через аргументы командной строки имена двух файлов (предполагается, что это именованные каналы, т. е. файлы типа `FIFO`), первый из них использует для приёма информации, и всё, что принято, тут же выводит в свой стандартный вывод, а во второй файл передаёт то, что будет введено через стандартный ввод.

Если всё сделать правильно, можно будет с помощью `mkfifo` создать два файла нужного типа, например, `f1` и `f2`, а затем в разных терминалах запустить два экземпляра вашей программы, указав им созданные каналы «крест-накрест» — например, `./prog f1 f2` и `./prog f2 f1`; получится «чат»: всё, что набирается на клавиатуре в одном окошке терминала, после нажатия `Enter` появляется во втором, и наоборот. Не забудьте обработать конец файла на обоих потоках ввода: если «кончился» стандартный ввод, можно просто выйти, а если канал — желательно сначала сообщить пользователю, что наш партнёр по чату нас покинул.

В этой задаче есть одна хитрость; если с ходу ничего не работает, прочитайте указания к задаче на стр. 139, но лучше сначала попробуйте сами догадаться, что происходит; если воспользоваться дебаггером и выяснить, *где* всё повисло, а ещё (до или после) перечитать параграф про каналы (§5.3.15), то понять, что к чему, будет не так уж сложно.

TCP-сервера

Задачи этого параграфа предполагают использование вызова `select`. Условия некоторых задач позволяют применить решение с обслуживающими процессами, но так делать следует разве что с ознакомительными целями; после обязательно перепишите ваши программы через мультиплексирование ввода-ввода.

В роли клиентской программы можно использовать утилиту `telnet`, если условиями задачи не подразумевается что-то другое.

6.12. Напишите TCP-сервер, который принимает от каждого клиента произвольный текст и отвечает на каждую присланную строку отправкой сообщения «`Ok`». Единственное ограничение на количество обслуживаемых клиентов может быть обусловлено предельным числом одновременно открытых в процессе дескрипторов, сама ваша программа никаких ограничений вводить не должна.

6.13. Усовершенствуйте программу, написанную при решении предыдущей задачи: снабдите ваш сервер одной на всех клиентов целочисленной переменной, изначально равной нулю, а пришедшие от клиента строки анализируйте. Если полученная от клиента строка содержит слово «`up`» (возможно, с пробельными символами спереди и сзади), значение переменной увеличьте на единицу, если слово «`down`» — наоборот, уменьшите на единицу; клиенту в обоих случаях отправьте сообщение «`Ok`». Если полученная строка содержит слово «`show`» — отправьте клиенту строку текста, содержащую десятичное представление текущего значения переменной. Если прочитана строка, в которой есть непробельные символы, но она после отбрасывания пробелов совпадает ни с одной из трёх команд `up`, `down` и `show`, отправьте клиенту сообщение об ошибке, что-нибудь вроде «textttunknown command». Обязательно проверьте, что сервер ведёт себя корректно при массовых подключениях и отключениях клиентов.

6.14. Придумайте подходящий протокол и напишите серверную и клиентскую программы, реализующие функциональность *BBS* (*bulletin board system*). При старте программа должна через аргументы командной строки получить имя рабочей директории; в ней должны содержаться файл заставки (текстовый), файл учётных записей (имён и паролей пользователей), а также файлы, доступные пользователям для скачивания, и к каждому из них — служебный

файл, содержащий описание и права доступа (кому из пользователей этот файл можно скачивать; обязательно нужно предусмотреть вариант «всем», в том числе незарегистрированным).

Учётные записи пользователей должно быть возможно пометить особым образом; такая отметка должна позволять пользователю не только скачивать файлы, но и размещать на BBS новые файлы, указав к ним описание и права доступа. Кроме того, стоит предусмотреть «привилегированных» пользователей, которые могут удалять файлы, изменять их описания и права, а также добавлять новых пользователей.

Клиентская программа после установления связи должна продемонстрировать пользователю содержимое заставки, позволить ввести логин и пароль (либо при желании продолжить работу как незарегистрированный пользователь), просмотреть описания к доступным для скачивания файлам и, конечно, скачать любой из них, а также (при наличии полномочий) закачать новые файлы, изменить описания и права к имеющимся файлам, удалить файлы. Очень желательно предусмотреть возможность оставить сообщение для владельца системы.

6.15*. Напишите серверную программу, принимающую электронную почту по протоколу SMTP и складывающую все принятые письма в виде файлов в указанную (например, через параметр командной строки) директорию. Для проверки вашей программы возьмите какой-нибудь существующий клиент электронной почты, такой как Sylpheed, Thunderbird или любой другой, и настройте его на отправку почты через smtp-сервер, расположенный на локальной машине (ip-адрес `127.0.0.1`) и порт, соответствующий порту, используемому вашей программой. Отправьте несколько писем и убедитесь, что ваша программа их получила и сложила куда следует.

Обязательно убедитесь, что ваша программа способна без ограничений одновременно обслуживать несколько сеансов работы с клиентами. Для этого, запустив программу, подключитесь к ней из нескольких терминальных окон с помощью утилиты `telnet`, в каждом из подключений начните сеанс взаимодействия, бросьте эти сеансы на разных стадиях (не завершая их), после чего попробуйте отправить почту из вашего клиента; если всё в порядке, вернитесь к брошенным сеансам и доведите каждый из них до отправки письма.

6.16. Напишите серверную программу, отдающую клиентам заданный файл в формате HTML по протоколу `http 1.1`; для проверки работы программы воспользуйтесь браузером.

6.17*. Напишите сервер, позволяющий нескольким людям (пользователям) сыграть друг с другом в хорошо известного *подкидного*

дурачка, используя текстовый режим. Чтобы играть могло больше людей, используйте «полную» колоду (включающую карты каждой масти от двойки до туза, всего 52 карты; джокеры в игре в дурачка задействовать проблематично); тогда в игре могут участвовать до восьми человек. Сервер должен начинать игру, когда к нему присоединилось не меньше двух игроков и один из них нажал `Enter` (прислал пустую строку), либо к игре присоединилось восемь человек — в этом случае нажатия `Enter` ждать уже бессмысленно.

Сервер должен «перетасовать» колоду, т. е. расположить её карты в случайно выбранном порядке, «сдать» каждому игроку по шесть карт, одну из оставшихся карт «открыть», показав, какая масть в этой игре будет козырной. Для обозначения старшинства карт можно использовать цифры от `2` до `9`, десятку так и обозначать «`10`», вальта, даму, короля и туза — латинскими буквами `J`, `Q`, `K` и `A` (*jack, queen, king, ace*); для обозначения мастей можно воспользоваться, например, символами `#`, `%` `^` `v` или какими-то ещё (здесь можно проявить фантазию).

Игрок должен видеть «состояние стола» и «свою руку». Пока не начался очередной ход, «состояние стола» представляет собой количество карт у каждого из остальных игроков, карту, показывающую козырную масть, и количество оставшихся карт в колоде; «свою руку» — список карт, имеющихся в настоящий момент у игрока — лучше показать, снабдив каждую карту меткой, причём, чтобы не путались между собой «номера» карт и их достоинства, правильнее будет использовать в роли меток строчные латинские буквы. Вывести всё это можно в две строки, например

```
< 6 >   < 6 >   < 6 >                  J^  [ 27 ]
a: 10^   b: J#   c: 2%   d: Qv   e: 7^   f: 9#
```

будет означать, что у игрока есть три противника, у каждого из них в руках по шесть карт, козыря показывает валет пик, в колоде (прикупе) ещё 27 карт, а на руке у игрока десятка пик, валет бубей, двойка крестей, дама червей, семёрка пик и девятка бубей. Ситуация, когда латинских букв не хватит для нумерации карт на руке, маловероятна, но теоретически возможна; в этом случае следует, например, привлечь заглавные латинские буквы (это вроде бы не очень хорошо, поскольку некоторые из них игрок может перепутать со старшинством карт, но ситуация эта настолько редка, что на мелкие неудобства при её возникновении можно не обращать внимания). Тому игроку, чей сейчас ход, следует вывести приглашение и (желательно) список возможных ходов; например, к вышеприведённым строчкам, если сейчас ход этого игрока, следует добавить строку

```
abcdef >
```

После начала хода к «состоянию стола» добавляются сыгранные на этом ходу карты, выглядеть всё вместе может, например, так:

```
< 4 >   < 6 >   < 6 >                  J^  [ 27 ]

   2% \ J%
   J# \ Q#

a: 10^   b: Qv   c: 7^   d: 9#
b >
```

Здесь игрок зашёл с двойки крестей, партнёр покрыл её вальтом, атакующий подбросил вальта бубей, валет был покрыт дамой, теперь атакующему выдано приглашение, показывающее, что он может подбросить ещё даму пик. Игрок может набрать букву `b` и нажать `Enter`, либо просто нажать `Enter`, отказавшись от дальнейшего подбрасывания.

Приглашение для «отбивающегося» игрока должно выглядеть как-то иначе, например, можно перечислить метки тех карт его руки, которыми он может покрыть последнюю сыгранную карту, и к этому добавить какие-то ещё символы; получится, например, «`be =>`». Все игроки, которые на текущем состоянии хода могут «подбросить» отбивающемуся ещё карты, должны своевременно получить соответствующее приглашение. Отбивающийся игрок может отказаться от дальнейшей защиты, даже если у него есть подходящие карты — для этого он отправляет пустую строку (нажимает `Enter`). В этом случае все карты, сыгранные на данном ходу, передаются ему на руку. Если у отбивающегося нет подходящих карт для обороны, такое завершение хода должно происходить автоматически. Если у остальных игроков имеются карты, подходящие для подбрасывания, и пока ещё не достигнут лимит на количество карт при атаке (этот лимит равен числу карт на руке у отбивающегося в на момент начала хода), им должно быть выдано приглашение с указанием подходящих карт («добавки»); ход считается завершённым, когда ни у кого нет больше подходящих карт или все игроки отказались от «добавления», нажав `Enter`.

При любом изменении в игре все игроки должны получить обновлённую «картинку», проще всего это сделать, передав 25 символов перевода строки. что при традиционной высоте терминала приведёт к очистке экрана, после чего передать строки состояния стола, руки и приглашения.

В наше время при игре в подкидного дурака обычно устанавливается очерёдность «подкидывания», но классические правила этой игры такой очерёдности не подразумевали: действовало правило «кто успел, тот молодец». Именно такая реализация, когда игроки, имеющие подходящие карты, могут посоревноваться в скорости реакции, *интереснее* в качестве этюда. Проблема разве что в том, что наличие на столе одновременно нескольких непокрытых карт приводит к некоторой неопределённости: какую из них отбивающийся пытается покрыть очередным своим ходом? Проще всего решить эту проблему, не допуская на стол больше одной непокрытой карты; сервер может принять и запомнить, какие карты игроки пытаются подбросить, но показывать их на столе по одной, либо показывать все, но игроку предлагать крыть в каждый момент только одну из них.

6.18*. Усовершенствуйте свой сервер для игры в дурачка так, чтобы он позволял одновременно проводить несколько игр, а пользователей и их результаты запоминал. При входе на сервер у пользователя следует спросить его имя; если такого пользователя пока нет, спросить, желает ли пользователь завести новую учётную запись и если да, то запросить пароль; если пользователь с таким именем уже известен — спросить и проверить пароль. После успешного входа на сервер пользователь оказывается в общем чате, где можно договориться об игре с другими игроками, создать игру или присоединиться к созданной, но ещё не начавшейся игре. В режиме игры теперь должны показываться имена игроков; следует предусмотреть возможность чата между игроками в ходе игры.

6.19*. Придумайте свою текстовую многопользовательскую игру и реализуйте сервер для неё.

6.20*. Реализуйте сервер для многопользовательских игр (см. условия задачи 6.18), позволяющий одновременно проводить игры различных видов.

К части 7 «параллельные программы и разделяемые данные»

7.01. В неправильно спроектированной системе имеется сегмент разделяемой памяти, в котором содержится массив целых чисел (как положительных, так и отрицательных), общая сумма которых равна нулю. Некая программа из числа входящих в систему начала подсчёт суммы элементов массива. В это же самое время другая программа каким-то неизвестным способом выбрала три элемента массива, первый из них уменьшила на 100, второй уменьшила на 200, третий увеличила на 300, так что общая сумма элементов осталась нулевой. Какими могут оказаться результаты подсчёта, выполняемого первой программой, если известно, что каждая отдельно взятая операция извлечения элемента из массива и записи элемента в массив выполняется атомарно, но никакого взаимоисключения в обеих работающих программах не предусмотрено? Перечислите все возможные варианты. Номера элементов, выбранных второй программой, а также конкретная последовательность, в которой первая программа суммирует элементы, неизвестны (могут быть любыми).

7.02. В разделяемой памяти расположен массив из трёх элементов типа `int`, причём исходно первый элемент равен 100, второй — 200, третий — 300. Есть функция, принимающая параметрами адреса двух переменных, вычисляющая их среднее арифметическое и записывающая результат в обе переменные:

```
void set_average(int *x, int *y)
{
    int z = (*x + *y) / 2;
    *x = z;
    *y = z;
```

```
}
```

Эту функцию вызвали в параллельных процессах, причём в одном — для первого и второго элементов массива, а в другом — для второго и третьего. Каковы могут быть значения элементов после этого, если никаких средств синхронизации и взаимоисключения не предусмотрено, но при этом извлечение значения из отдельно взятого элемента массива, а также занесение в него нового значения работают атомарно? Перечислите все возможные варианты.

7.03. В §7.1.3 приведён ***алгоритм Петерсона***, позволяющий выполнить корректное взаимоисключение для двух процессов с использованием переменной `who_waits` и массива флагов `interested` из двух элементов:

```
void enter_section() {
    interested[0] = TRUE;
    who_waits = 0;
    while(who_waits==0 &&
          interested[1]) {}
}
void leave_section()
{ interested[0] = FALSE; }
```

```
void enter_section() {
    interested[1] = TRUE;
    who_waits = 1;
    while(who_waits==1 &&
          interested[0]) {}
}
void leave_section()
{ interested[1] = FALSE; }
```

Останется ли алгоритм корректным, если в функциях `enter_section` поменять местами первые две строки, вот так:

```
void enter_section() {
    who_waits = 0;
    interested[0] = TRUE;
    while(who_waits==0 &&
          interested[1]) {}
}
```

```
void enter_section() {
    who_waits = 1;
    interested[1] = TRUE;
    while(who_waits==1 &&
          interested[0]) {}
}
```

— или он «сломается»? Ответ обоснуйте.

7.04. Напишите программу, соответствующую условиям задачи 6.13, без применения вызова `select` — вместо этого в главном треде выполняйте вызов `accept`, а для обслуживания каждого нового подключившегося клиента порождайте дополнительный тред. Не забывайте про взаимное исключение при обращении к любым данным, доступным в разных тредах! Сравните объём и прочие характеристики этого решения с решением в событийно-ориентированном стиле (на вызове `select`).

К части 8 «ядро системы: взгляд за кулисы»

8.01. Дана программа:

```
int main()
{
    double d;
    scanf("%lf", &d);
    d = cos(d);
    printf("%f", d);
    return 0;
}
```

Предположим, мы используем такую версию компилятора и библиотеки, что полученный исполняемый файл использует абсолютный минимум системных вызовов. Сколько системных вызовов выполнит данная программа, и какие это будут системные вызовы?

8.02. Программа `true` выглядит так:

```
int main() { return 0; }
```

Предположим, мы используем такую версию компилятора и библиотеки, что полученный исполняемый файл использует абсолютный минимум системных вызовов. Будет ли данная программа выполнять системные вызовы, и если да, то какие?

8.03. Среди библиотечных функций `printf`, `scanf`, `fgetc`, `fopen`, `fclose`, `read`, `malloc`, `free`, `kill`, `signal`, `exit`, `getpid` есть ровно одна, которая не является системным вызовом и никогда не обращается к системным вызовам. Какая это функция?

8.04. Среди библиотечных функций `sin`, `atoi`, `sprintf`, `sscanf`, `strcpy`, `strcmp`, `malloc`, `free`, `rand` ни одна не является системным

вызовом, но есть ровно одна, которая может обратиться к системному вызову. Какая это функция?

8.05. На некотором компьютере реализована страничная модель виртуальной памяти с двумя уровнями страничных таблиц. Ячейки памяти восьмибитные, разрядность адреса составляет 32 бита, таблица первого уровня содержит 512 записей, таблицы второго уровня — по 4096 записей каждая. Каков размер страницы?

8.06. На некотором компьютере реализована страничная модель виртуальной памяти с двумя уровнями страничных таблиц. Ячейки памяти восьмибитные, разрядность указателей составляет 64 бита, но из них используется только 40, остальные зарезервированы; таблица первого уровня содержит 8192 записи, таблицы второго уровня — по 4096 записей каждая. Каков размер страницы?

8.07. На некотором компьютере реализована страничная модель виртуальной памяти с тремя уровнями страничных таблиц. Ячейки памяти восьмибитные, разрядность указателей составляет 32 бита; таблицы каждого из трёх уровней содержат по 128 записей. Каков размер страницы?

К части 9 «парадигмы в мышлении программиста»

Рекурсия

9.01. Дан целочисленный массив (адрес его начала и длина); нужно определить, чему равен его максимальный элемент. Сформулируйте алгоритм решения этой задачи в терминах рекурсивного мышления (рекурсии как парадигмы), т. е. без явного указания последовательности действий. Напишите на Си функцию, реализующую этот алгоритм и имеющую следующий профиль:

```
int intvecmax(const int *arr, int len);
```

Обратите внимание, что при правильном решении функция не должна использовать циклы и модифицирующие («разрушающие») операции (присваивания и инкременты/декременты).

Обязательно сформулируйте ответы на следующие вопросы:

- какой случай используется в роли базиса рекурсии?
- чем (конкретно) случай, решаемый с помощью рекурсивного вызова, проще, нежели исходный случай?

9.02. Является ли остаточной рекурсия в вашем решении предыдущей задачи? Почему? Перепишите решение так, чтобы рекурсия стала остаточной. Можно ли теперь сформулировать принцип её работы в рамках рекурсии как парадигмы?

9.03. В целочисленном массиве длиной не менее двух элементов нужно найти два наибольших значения его элементов; решение требуется в виде функции с профилем

```
void intvec2max(const int *arr, int len, int res[2]);
```

Предполагается, что функция запишет найденные значения в элементы массива `res`: наибольшее — в элемент `res[0]`, второе по величине — в элемент `res[1]`. Напишите такую функцию (возможно, с использованием вспомогательных функций) **без применения циклов**; присваивания используйте только для занесения значений в `res`.

Является ли получившаяся у вас рекурсия остаточной? Если нет, исправьте ваш код так, чтобы рекурсия стала остаточной, в этой задаче это несложно.

9.04. Дан массив беззнаковых целых; требуется определить максимальное значение его элементов и все элементы, имеющие это значение (в общем случае их больше одного) обнулить. Предложите решение этой задачи, не использующее циклы.

9.05. Можно ли в решении предыдущей задачи обойтись также и без присваиваний, за исключением присваивания нулей элементам, в которых находилось максимальное значение?

9.06. В стандартной библиотеке языка Си есть функция `strstr`, позволяющая найти подстроку в строке (см. §4.10.2). Напишите на Си функцию, делающую то же самое, не применяя циклов и разрушающих операций.

В качестве дополнительного (факультативного) упражнения попробуйте обойтись без вспомогательных функций.

9.07*. В §§2.11.3, 3.3.9, 4.3.22 и 9.2.4 разобраны решения задачи о сопоставлении строки с образцом — на Паскале, на языке ассемблера и на Си. Эти решения друг от друга несколько отличаются по структуре (в том числе слегка различаются между собой два решения на Си), но их объединяет главное: все они используют рекурсию. Попробуйте решить эту задачу на языке Си без применения рекурсии.

9.08. Требуется программа, принимающая на вход (параметром командной строки) целое число N и печатающее N первых чисел Фибоначчи. Напишите такую программу без применения разрушающих операций.

Работа в явных состояниях

В задачах этого параграфа под «реализацией автомата» понимается создание трёх функций: первая из них выделяет область памяти для хранения состояния автомата (скорее всего, в виде какой-то структуры, но это не обязательно), приводит условный автомат в исходное состояние (инициализирует выделенную область памяти) и возвращает адрес

созданного объекта; вторая функция получает адрес объекта автомата через один из параметров и выполняет один шаг автомата, выдав при этом нужную информацию (если информация скалярная, её можно вернуть как значение функции, в противном случае придётся использовать выходной параметр — адрес, куда нужно записать полученную из автомата информацию); третья функция получает адрес объекта автомата и уничтожает его, освобождая память (в простейшем случае — вызывает `free`).

9.09. Реализуйте автомат, последовательно выдающий числа Фибоначчи.

9.10. Реализуйте автомат, который на каждом шаге выдаёт очередной элемент треугольника Паскаля, а также его «индексы» — номер строки и номер элемента в этой строке, начиная с нулей. Иначе говоря, автомат на каждом шаге должен генерировать очередные n, k и C_n^k: $(0,0,1)$, $(1,0,1)$, $(1,1,1)$, $(2,0,1)$, $(2,1,2)$, $(2,2,1)$, $(3,0,1)$, $(3,1,3)$, $(3,2,3)$, $(3,3,1)\ldots$ На всякий случай напомним, что при вычислении C_n^k не следует использовать никаких умножений, треугольник Паскаля как раз и предназначен для вычисления членов этой последовательности с использованием только операции сложения.

9.11. ***Перестановкой*** из N элементов называется некий упорядоченный набор, составленный из чисел $1, 2, \ldots, N$, в котором каждое число встречается ровно один раз. Так, перестановок из двух элементов существует две ($\{1,2\}$ и $\{2,1\}$), перестановок из трёх элементов — шесть ($\{1,2,3\}$, $\{1,3,2\}$, $\{2,1,3\}$, $\{2,3,1\}$, $\{3,1,2\}$, $\{3,2,1\}$), а в общем случае перестановок из N элементов существует $N!$ (эн-факториал). Перестановки подробно обсуждаются в §1.3.1, хотя там они формально не вводятся в качестве кортежей из чисел, вместо этого используются занумерованные бильярдные шарики.

Реализуйте автомат, который при его создании получает число N (то есть функция создания автомата имеет целочисленный параметр), а на каждом шаге выдаёт очередную перестановку из N элементов, для чего функции, которая реализует шаг автомата, передаётся адрес начала массива из N целых. После исчерпания перестановок, т. е. после того, как функция шага автомата была вызвана $N!$ раз, при последующих вызовах она должна заполнять переданный массив нулями, чтобы показать, что больше перестановок нет.

К части 10 «язык Си++, ООП и АТД»

Абстрактные типы данных

Следующие задачи построены в предположении, что вы уже прочитали главу 10.3.

10.01. Дана функция main:

```
int main()
{
    A first = 1;
    A second(10);
    printf("first: %d %d\n", first[100], first[200]);
    printf("second: %d %d\n", second[100], second[200]);
    return 0;
}
```

Опишите класс `A`, объекты которого ведут себя как неизменяемые массивы целых чисел, причём значение каждого элемента больше значения его индекса на число N, которое передаётся как параметр конструктора. Так, данная функция `main` должна напечатать:

```
first: 101 201
second: 110 210
```

10.02. Дана функция main:

```
int main()
{
    B first(1), second = 2;
    first += 10; second += 1000;
    printf("%d %d\n", first.Get(), second.Get());
    return 0;
}
```

Опишите класс `B` таким образом, чтобы программа с такой функцией main компилировалась (без предупреждений) и работала, печатая строку «`11 1002`».

10.03. Дана функция main:

```
int main() {
    D x;
    D y(x);
    D z = y;
    printf("%d %d %d\n", x.Get(), y.Get(), z.Get());
    return 0;
}
```

Опишите класс `D` так, чтобы эта функция компилировалась без предупреждений и работала, печатая строку «`0 1 2`».

10.04. Опишите класс `M`, объекты которого представляют целочисленную матрицу 3×3 с индексацией от `1` до `3`, причём для задания позиции используются объекты другого класса, называемого `I`. Реализуйте операцию сложения матриц, а операцию индексирования перегрузите так, чтобы можно было выполнять как присваивания, так и извлечения элементов матрицы. При создании матрицы делайте её единичной (единицы на главной диагонали, остальные нули). Ваши классы должны работать со следующим текстом функции `main()`:

```
int main()
{
    M m1;
    printf("%d %d %d\n", m1[I(1,1)], m1[I(2,2)], m1[I(2,3)]);
    M m2;
    m1[I(2,3)] = 7;
    m2[I(2,3)] = 350;
    M m3(m1 + m2);
    printf("%d %d %d\n", m3[I(1,1)], m3[I(2,2)], m3[I(2,3)]);
    return 0;
}
```

— причём должно печататься

```
1 1 0
2 2 357
```

Использовать глобальные переменные и статические поля в этой задаче не следует.

10.05*. Усовершенствуйте класс `M`, описанный в предыдущей задаче, устранив потребность в объектах класса `I`; двойной индекс должен записываться обычным для Си и Си++ способом, через двойное применение операции индексирования. Текст функции `main` после этого должен стать таким:

```
int main() {
    M m1;
    printf("%d %d %d\n", m1[1][1], m1[2][2], m1[2][3]);
    M m2;
    m1[2][3] = 7;
    m2[2][3] = 350;
    M m3(m1 + m2);
    printf("%d %d %d\n", m3[1][1], m3[2][2], m3[2][3]);
    return 0;
}
```

10.06. Опишите класс `E`, объект которого представляет собой «бесконечную» целочисленнцю матрицу, элементы которой на главной диагонали равны нулю, а остальные элементы равны разности индексов (т. е., например, элемент `e[3][200]` равен `-197`, а элемент `e[12905][105]` равен `12800`). Так, следующая функция `main()`:

```
int main() {
    E e;
    printf("%d %d %d\n", e[0][0], e[100][100], e[-10][-10]);
    printf("%d %d %d\n", e[1500][7], e[7][55], e[-8][-16]);
    return 0;
}
```

должна компилироваться, работать и печатать

```
0 0 0
1493 -48 8
```

Естественно, в этой задаче не следует использовать глобальные переменные и статические поля; кроме того, здесь не нужно описывать какие-либо настоящие массивы (операции `[]` должны применяться только к объектам ваших классов).

10.07. Что напечатает следующая программа?

```
#include <stdio.h>
class I {
    int i;
public:
    I() : i(6) { printf("sun\n"); }
    I(int a) : i(a) { printf("venus %d\n", i); }
    I(const I& other) : i(other.i)
         { printf("earth %d\n", i); }
    ~I() { printf("moon\n"); }
    int Get() { return i; }
    void operator+=(const I& op) { i+=op.i; }
};
```

```
void f(I& x, I y)
{
    y += 1000;
    x += y;
}
int main()
{
    I i1;
    I i2(50);
    i2 += 800;
    f(i1, i2);
    printf("%d %d\n", i1.Get(), i2.Get());
    return 0;
}
```

10.08. Напомним, что *рациональное число* — это несократимая дробь вида $\frac{n}{m}$, где n — целое, m — натуральное. Опишите класс `Rational`, реализующий понятие рационального числа, воспользовавшись числами типа `long long` для представления числителя и знаменателя дроби; требование о несократимости дроби пока проигнорируйте. Снабдите класс четырьмя основными действиями арифметики, соответствующими им операциями присваивания (`+=`, `-=`, `*=`, `/=`), функциями преобразования к числу типа `double` и к целому (здесь желательны две функции: через отбрасывание дробной части и через округление к ближайшему). Будет полезно предусмотреть метод (или два) для извлечения из объекта текущих значений числителя и знаменателя.

Напишите набор тестов, демонстрирующий корректность работы вашего класса (всех его методов).

10.09. Возможности класса, созданного при решении предыдущей задачи, несколько ограничены: все арифметические операции имеют тенденцию наращивать значение знаменателя — как напрямую при выполнении умножения и деления, так и через приведение к общему знаменателю, которое необходимо при сложениях и вычитаниях; рано или поздно (и скорее рано, чем поздно) значение знаменателя превышает возможности разрядности даже для `long long`.

Напишите тесты, демонстрирующие ограничения нехватки разрядности; желательно при этом найти точные границы возможностей подобно тому, как это сделано в §2.2.5.

Воспользовавшись алгоритмом Евклида, напишите функцию, отыскивающую наибольший общий делитель для двух чисел типа `long long`; обратите внимание, что наибольший общий делитель всегда по определению положителен и не меняется при любой смене знаков исходных чисел. Поскольку эта функция потребуется вам в

классе `Rational`, она должна быть его приватным методом; но так как объект типа `Rational` ей для работы не нужен, следует сделать её статической.

Воспользовавшись этой функцией, усовершенствуйте ваши арифметические операции так, чтобы, во-первых, результат их работы всегда представлял собой несократимую дробь; во-вторых, чтобы «лишние» множители по возможности изымались из чисел *до* выполнения других операций: так, при перемножении двух дробей $\frac{n}{m} \cdot \frac{p}{q}$ следует не только предварительно сократить обе дроби, если этого ещё не сделано, но и проверить, нет ли нетривиального (т. е. отличного от единицы) общего делителя в парах (n, q) и (m, p).

С помощью ранее написанных тестов убедитесь, что теперь возможности вашего класса стали шире; напишите тесты, демонстрирующие границы возможностей новой реализации, и сравните эти границы со старыми.

Наследование

10.10. Что напечатает следующая программа?

```
#include <stdio.h>
class A {
    int i;
public:
    A(int x) { i = x; printf("first\n"); }
    virtual ~A() { printf("second\n"); }
    int f() const { return i + g() + h(); }
    virtual int g() const { return i; }
    int h() const { return 6; }
};
class B : public A {
public:
    B() : A(0) { printf("third\n"); }
    ~B() { printf("fourth\n"); }
    int f() const { return g() - 5; }
    virtual int g() const { return 8; }
    int h() const { return 1; }
};
int main() {
    B b;
    A* p = &b;
    printf("result = (%d ; %d)\n", p->f(), b.f());
    return 0;
}
```

10.11. Опишите абстрактный класс `Body`, представляющий некое абстрактное физическое тело, для которого не задана форма, но задана плотность составляющего его вещества. Плотность задаётся полем типа `double`. Предусмотрите в классе **чисто виртуальную** функцию `Volume()`, задающую объём тела, и простую функцию `Mass()`, вычисляющую массу как произведение объёма и плотности.

Унаследуйте от класса `Body` класс `Cube`, представляющий кубик, сделанный из однородного вещества и задаваемый длиной ребра (первый параметр конструктора) и плотностью вещества (второй параметр конструктора), а также класс `Tetrahedron`, представляющий правильный тетраэдр, сделанный из однородного вещества и задаваемый длиной ребра и плотностью. Напомним, что объём правильного тетраэдра составляет $\frac{\sqrt{2}}{12}a^3$. Для задания значения $\sqrt{2}$ можно воспользоваться константой `M_SQRT2`, которая определена в заголовочном файле `math.h`.

В результате следующая функция `main()`:

```
int main() {
    const Body *p, *q, *r;
    Cube a(2, 10), b(5, 0.1);
    Tetrahedron t(6, 2.5);
    p = &a; q = &b; r = &t;
    printf("Volumes: %3.3lf %3.3lf %3.3lf\n",
           p->Volume(), q->Volume(), r->Volume());
    printf("Weights: %3.3lf %3.3lf %3.3lf\n",
           p->Mass(), q->Mass(), r->Mass());
    return 0;
}
```

должна откомпилироваться без ошибок и предупреждений, отработать и выдать

```
Volumes: 8.000 125.000 25.456
Weights: 80.000 12.500 63.640
```

Проверьте, что ваша реализация соответствует следующим условиям:

- все поля находятся в закрытой (`private`) части класса, открытыми и защищёнными могут быть только функции-члены;
- директива `friend` не используется;
- для инициализации объектов используются конструкторы;
- никакие методы не изменяют внутренние поля объектов, могут быть только функции, возвращающие значения полей (но не меняющие ничего); как следствие, все методы, кроме конструкторов, помечены как константные;

- данные базового класса нигде не дублируются полями порождённого класса.

10.12. Опишите абстрактный класс `Prism`, представляющий прямоугольную призму с известной высотой, но неизвестной формой основания. Предполагается, что все величины имеют тип `double`. Предусмотрите **чисто виртуальную** функцию `Square()`, возвращающую площадь основания призмы, и простую функцию `Volume()`, вычисляющую объём как произведение высоты на площадь основания.

Унаследуйте от класса `Prism` класс `Box`, представляющий прямоугольный параллелепипед с квадратом в основании, задаваемый длиной бокового ребра (первый параметр конструктора) и длиной стороны основания (второй параметр). Унаследуйте от класса `Box` класс `Cube`, представляющий собой куб, задаваемый длиной ребра (единственный параметр конструктора). В результате следующая функция `main()`:

```
int main()
{
    const Prism *p, *q, *r;
    Box a(0.5, 2), b(5, 0.2);
    Cube c(0.5);
    p = &a; q = &b; r = &c;
    printf("Squares: %3.3lf %3.3lf %3.3lf\n",
           p->Square(), q->Square(), r->Square());
    printf("Volumes: %3.3lf %3.3lf %3.3lf\n",
           p->Volume(), q->Volume(), r->Volume());
    return 0;
}
```

должна откомпилироваться без ошибок и предупреждений, отработать и выдать

```
Squares: 4.000 0.040 0.250
Volumes: 2.000 0.200 0.125
```

Проверьте выполнение условий, перечисленных в конце предыдущей задачи.

Обработка исключений

10.13. Что напечатает следующая программа?

```
#include <stdio.h>
class Ex {
    int code;
public:
    Ex(int i) : code(i) {}
    Ex(const Ex& ex) : code(ex.code) {}
    int Get() const { return code; }
};
class Ex60 : public Ex {
public:
    Ex60() : Ex(60) {}
};
void f(int i)
{
    if(i>0)
        throw Ex(i);
    printf("owl\n");
}
void g()
{
    try {
        f(60);
    }
    catch(Ex60 &x) {
        printf("dog\n");
    }
    printf("cat\n");
}
void h()
{
    try {
        g();
        printf("sheep\n");
    }
    catch(Ex &x) {
        printf("horse\n");
        throw Ex60();
    }
    catch(Ex60 &x) {
        printf("cow\n");
    }
    printf("elephant\n");
}
void t()
{
    try { h(); }
    catch(Ex &x) { printf("wolf %d\n", x.Get()); }
```

```
    printf("monkey\n");
}
int main() {
    try { t(); }
    catch(...) { printf("fox\n"); }
    return 0;
}
```

10.14. Вернитесь к решению задачи 10.09 и снабдите конструкторы класса `Rational` проверкой, что знаменатель отличен от нуля; если проверка не прошла, выбрасывайте исключение, для которого опишите специальный класс.

Шаблоны

10.15. Опишите идентификатор `swap3` так, чтобы вычисление выражения вида `swap3(x,y,z)` (где `x`, `y`, `z` — переменные какого-то неизвестного типа) приводило к тому, что в переменной `x` оказывается значение, которое находилось до этого в переменной `y`, в ней, в свою очередь, значение, находившееся в переменной `z`, а в переменной `z` — то, что раньше было в переменной `x`. Про тип переменных известно только, что он допускает присваивание и описание переменных без указания инициализирующих параметров.

10.16. Опишите идентификатор `get_and_zero` так, чтобы вычисление выражения вида `a = get_and_zero(b)`, где `a` и `b` — переменные некоторого (неизвестного) типа, приводило к тому, что в переменной `a` оказывается старое значение переменной `b`, а в самой переменной `b` — ноль. Про тип переменных известно только, что он допускает копирование, присваивание переменных друг другу и присваивание нуля (т.е. выражение `a = 0`, где `a` — переменная такого типа, является допустимым).

Этюды для закрепления навыков

10.17. Вернитесь к задачам 9.09–9.11 и оформите описанные в них автоматы в виде объектов классов, имеющих конструктор для инициализации и задания начальных параметров, если таковые нужны, деструктор для высвобождения захваченных ресурсов, если такие есть, и один обыкновенный публичный метод, выполняющий шаг автомата. Приватные методы можно вводить без ограничений.

10.18. Вернитесь к условиям задач 6.12–6.17 и перепишите эти программы на Си++ с использованием объектно-ориентированного подхода.

Библиотека FLTK

10.19. С использованием библиотеки FLTK напишите программу, которая получает через параметр командной строки некое сообщение, после запуска формирует на экране диалоговое окно, содержащее это сообщение и две кнопки с надписями `Yes` и `No`; если пользователь нажимает кнопку `Yes`, программа должна завершиться успешно (с нулевым кодом завершения), если же пользователь предпочтёт кнопку `No` — завершиться следует с кодом `1` (неуспех).

10.20. Если не предпринять специальных мер, нажатие клавиши `Escape` в окошке программы, написанной на FLTK, приводит к *нормальному* завершению программы; по смыслу условия предыдущей задачи это совершенно не то, что нужно. Усовершенствуйте программу, написанную для решения предыдущей задачи, так, чтобы при нажатии `Escape` она завершалась с кодом `1`.

10.21. Напишите программу, которая принимает произвольное (но не менее одного) количество аргументов командной строки, каждый из аргументов использует как текст кнопки, формирует диалоговое окно, содержащее соответствующие кнопки, и в случае, если пользователь нажимает на одну из кнопок — завершается успешно, выдав в поток стандартного вывода десятичное представление номера кнопки, начиная с единицы (этот номер будет совпадать с индексом аргумента в массиве `argv`); если же пользователь нажал `Escape`, программа должна завершиться с кодом `1`, ничего не напечатав.

10.22. Напишите с использованием FLTK программу — клавиатурный тренажёр. После старта программа должна прочитать текст из некоторого текстового файла, в котором каждая строка — это «упражнение», т. е. фраза, которую пользователь должен набрать. Далее программа формирует диалоговое окно, в котором есть виджет для отображения очередной фразы и поле текстового ввода; в этом окне программа последовательно воспроизводит фразы, прочитанные из файла. Пользователю предлагается набрать такую же фразу в поле ввода и нажать `Enter`. Если набранная фраза совпадает с ожидаемой, программа должна перейти к работе со следующей фразой из файла, когда же они закончатся — выдать пользователю информационный диалог, отображающий общее количество потраченного на упражнение времени, с кнопкой «`Ok`»; после нажатия на кнопку программа завершается. Если очередную фразу пользователь набрал с ошибками, при нажатии клавиши `Enter` набранный текст должен исчезнуть, а фраза, которую нужно набрать — остаться той же самой, чтобы её пришлось набрать повторно, и так до тех пор, пока она не будет набрана правильно.

10.23. С конца XIX века хорошо известна «игра в 15», она же «такен». В классическом варианте эта игра представляет собой квадратную коробку, в которой располагаются 15 квадратных фишек («плиток») размером вчетверо меньше коробки; при этом один квадратик в коробке остаётся пустым. Плитки исходно расположены в случайной последовательности; задача игры — перемещая за один ход одну плитку на пустое место, расставить все плитки в порядке возрастания номеров. Хорошо известен также тот факт, что ровно половина исходных позиций неразрешима: их можно привести к позиции, в которой все плитки стоят по порядку, но последние две (14 и 15) поменены местами.

Рискнём предложить читателю модификацию «игры в 15», которую можно назвать «13». От классического варианта она отличается только нумерацией плиток: они занумерованы от `0` до `13`, причём плиток с нулём в коробке две, и они не отличаются одна от другой. Целевая позиция в этом случае — сначала два «нуля», потом остальные плитки от первой до тринадцатой. Поскольку две «нулевые» плитки неразличимы, их можно поставить в любом порядке, что делает все возможные начальные позиции заведомо разрешимыми.

Реализуйте программу, позволяющую пользователю играть в «13». Программа должна формировать диалоговое окно с 15 квадратными кнопками, имеющими соответствующий текст (размером 70–80% от размеров кнопки) и расположенными на поле размером 4×4 кнопки; ход производится нажатием на одну из кнопок рядом со свободным местом, после чего соответствующая кнопка перемещается в свободную позицию (меняются координаты её виджета). После достижения целевой позиции следует выдать диалог с поздравлениями.

Желательно, чтобы ваша программа помнила начальную позицию и все сделанные пользователем ходы, позволяла возвращаться к началу, откатывать любое количество ходов, записывать текущее состояние (позицию и ходы) в файл и считывать его из файла, а также генерировать новую начальную позицию, используя псевдослучайные числа. Кроме того, желательно предусмотреть для пользователя возможность самостоятельно задать начальную позицию; впрочем, если файл для сохранения состояния будет простым текстовым, эту возможность можно отдельно не реализовывать, предложив пользователю вручную сформировать соответствующий файл.

10.24*. Вернитесь к задаче 6.18 (см. стр. 81) и снабдите ваш сервер графической клиентской программой; изображения игральных карт, составляющих различные колоды, можно легко отыскать в Интернете.

К части 11 «неразрушающие парадигмы»

Лисп и Scheme

Для решения задач из этого параграфа рекомендуется воспользоваться любым из интерпретаторов Common Lisp, рассматриваемых во «Введении в профессию» — SBCL, GCL или ECL, а также интерпретатором Chicken Scheme (хотя на самом деле можно использовать едва ли не любой интерпретатор). В задачах, предполагающих написание *программы* (в отличие от отдельной функции), такую программу следует обязательно оформить как исполняемый скрипт (см. §§11.1.2, 11.2.1), либо для случая Chicken Scheme — откомпилировать. Помните, что запуск программ из интерактивного сеанса работы с интерпретатором — это атавизм давно ушедшей эпохи, когда не существовало конечных пользователей; конечному же пользователю ваша программа вообще не должна показывать, на чём она написана.

Если вы хотите освоить и Лисп, и Scheme (что в целом полезно), правильно будет поначалу решение каждой задачи писать на обоих языках; на более поздних этапах можно чередовать языки (одну задачу на одном, другую на другом), можно для каждой задачи выбирать один из двух языков, исходя из любых соображений вплоть до сегодняшнего настроения, тут главное — не зацикливаться на одном из двух языков, иначе второй вы так толком и не попробуете.

11.01. Напишите функцию, которая принимает параметром натуальное число N и возвращает список натуральных чисел от `1` до N, например, `(1 2 3 4)`.

11.02. Напишите функцию, которая принимает два параметра — произвольное выражение E и целое число N — и возвращает построенный список, состоящий из N элементов E.

11.03. Напишите функцию, которая принимает один параметр — список произвольной длины, элементами которого тоже могут быть списки на произвольную глубину вложенности — и подсчитывает сумму входящих в этот список (на любой глубине вложенности) числовых атомов, а все остальные атомы (как и структуру самих списков) игнорирует.

11.04. Напишите функцию, которая получает параметром список произвольной длины и вложенности, и возвращает список, построенный из атомов, входящих в исходный список на любом уровне вложенности. Например, из аргумента `(1 2 (3 (4 5) 6 ((7 8)) 9))` должен получиться список `(1 2 3 4 5 6 7 8 9)`.

11.05. Напишите функцию `properlist`, которая принимает один аргумент и возвращает значение логической истины, если аргумент является «правильным» списком, и ложь в любом другом случае. Напомним, что правильным (*proper*) считается либо пустой список, либо такой, в котором правый элемент (`cdr`) последней точечной пары является пустым списком. Отметим, что встроенная функция `listp` (`list?` в Scheme) возвращает истину в том числе и для точечных списков, таких как `(1 . 2)` или `(a b c . d)`.

11.06. Не используя арифметические действия, напишите функцию, которая возвращает истину, если её (единственный) аргумент — список из чётного числа элементов (верхнего уровня), а во всех остальных случаях возвращает ложь.

11.07. Напишите функцию, которая получает два аргумента — список L и целое число N, и возвращает список, состоящий из N первых элементов списка L; если список L состоит всего из N элементов или окажется ещё короче, следует вернуть его весь (допускается вернуть его копию).

11.08. Напишите функцию `list-iter`, которая принимает один параметр — произвольный список, а возвращает *объект новой функции* (замыкание); порождённая функция должна, не принимая аргументов, при каждом вызове возвращать очередной элемент исходного списка, пока этот список не кончится, после этого в случае дальнейших вызовов возвращать `NIL` (или `#F` для Scheme). Например, если результат вызова `(list-iter '(1 2 3))` присвоить переменной `f`, то обращения к `(funcall f)` (для Scheme — просто `(f)`) должны последовательно вернуть `1`, `2`, `3`, а дальше соответственно `NIL` или `#f`.

11.09. Используя свойства замыканий, напишите функцию, которая по заданному списку строит «перевёрнутый» список, используя для просмотра списка функционал `mapcar` (`map` для языка Scheme).

11.10. В Common Lisp узнать длину строки можно с помощью встроенной функции `length`, извлечь из строки символ с заданным номером — с помощью функции `char`; в Scheme аналогичные функции называются `string-length` и `string-ref`. Напишите на этих языках программы, соответствующие условиям задач 2.22, 2.23 и 4.12.

11.11. Напишите на Лиспе и/или Scheme программу, соответствующую какому-нибудь из пунктов задачи 2.19 (см. стр. 2.19), *не применяя циклы*. Внимательно проанализируйте получившуюся рекурсию; если она не является остаточной (хвостовой), обязательно скорректируйте своё решение. Подав своей программе на стандартный ввод бесконечный поток строк (например, с помощью команды `yes`), убедитесь, что количество занимаемой ею памяти не увеличивается со временем.

11.12. Напишите программу, принимающую ровно один аргумент командной строки, в котором должно быть записано арифметическое выражение, вычисляет это выражение и выдаёт результат вычисления на стандартный вывод. Выражение может включать целые числа, круглые скобки и знаки пяти основных действий целочисленной арифметики (`+`, `-`, `*`, `/` и `%`); кроме того, между любыми элементами выражения (но, конечно, не внутри чисел) допускаются (но не являются обязательными) произвольные группы пробельных символов. Для упрощения картины можете не обрабатывать унарные минусы и плюсы, хотя их наличие если и усложняет разбор выражения, то не так чтобы очень сильно.

11.13. Создайте программу, главная функция которой написана на Си, при этом некоторые из остальных функций написаны на Chicken Scheme; например, можно сделать программу, печатающую те из аргументов командной строки, которые являются палиндромами (без учёта пробелов), а проверку строки на палиндромичность реализовать на Scheme.

Создайте программу, основная часть которой написана на Scheme, но при этом из неё вызываются части, написанные на Си; например, это может быть какой-то демонстрационный TCP-сервер, часть которого, отвечающая за сокеты, написана на Си, а прикладная функциональность — на Scheme.

Всю необходимую информацию о том, как интегрировать между собой модули на Chicken Scheme и Си, найдите в Интернете.

Пролог

В задачах на обработку списков не следует применять встроенные и библиотечные предикаты, иначе вы не получите навыков, на формирование которых даны эти задачи.

11.14. Без использования арифметики напишите одноместный предикат, истинный на списках, которые:

a) состоят из чётного числа элементов;
b) состоят из нечётного числа элементов;
c) содержат в произвольном месте два соседних одинаковых элемента;
d) в каждой чётной позиции (вторым, четвёртым, шестым элементом и т. д.) содержат пустой список (`[]`),

и ложный на любых других аргументах.

11.15. Напишите без использования арифметики предикат `atleast2(E,L)`, верный в случае, если элемент `E` встречается в списке `L` не менее двух раз. Правильно написанный предикат должен работать в прототипах (+,+) и (−,+) то есть когда первый аргумент не задан, предикат должен найти все такие элементы, которые входят в список больше одного раза.

11.16. Напишите предикат `similar(L1, L2)`, истинный тогда и только тогда, когда `L1` и `L2` — списки одинаковой длины, причём `L2` либо является копией `L1`, либо получен из `L1` заменой одного или нескольких (возможно, всех) элементов, являющихся списками, на пустой список `[]`. Например, такому свойству удовлетворяют списки `L1=[f(c),[2,3],[a],[c,b],3]` и `L2=[f(c),[],[a],[],3]`. Проверьте работу в прототипах (+,+) и (+,−).

11.17. Напишите предикат `ins_one(List, Elem, Res)`, устанавливающий отношение «`Res` есть список, полученный из списка `List` добавлением элемента `Elem` в любое его место — перед первым элементом, между любыми двумя или после последнего». При правильном решении ваш предикат должен работать в прототипах (+,+,+), (+,+,−), (+,−,+), (−,+,+) и (−,−,+); например, в ответ на запрос `ins_one([1,2], 0, R)` вы получите три ответа: `[0,1,2]`, `[1,0,2]` и `[1,2,0]`, запрос `ins_one(L, 2, [1,2,3])` даёт ровно одно решение (`[1,3]`), запрос `ins_one(L, X, [1,2,3])` даст три решения (догадайтесь, какие).

11.18. Напишите предикат `shrink(L,R)`, истинный, когда список `R` получен из списка `L` вычёркиванием:

a) ровно двух произвольных элементов;

b) не более чем двух элементов;
c) ровно двух элементов, стоящих в списке подряд.

11.19. Напишите предикат `strike(L,M)`, который истинен тогда и только тогда, когда список `M` получен путём вычёркивания из списка `L` некоторого (возможно, нулевого) количества элементов, стоящих подряд, и вставления вместо них одного элемента `[]`. Предикат должен работать как в прототипе (+,+), так и в прототипе (+,−); в частности, запрос `strike([1,2,3],M)` должен выдать (в произвольном порядке) следующие решения: `[[]]`, `[1,[]]`, `[[],3]`, `[[],2,3]`, `[1,[],3]`, `[1,2,[]]`, `[[],1,2,3]`, `[1,[],2,3]`, `[1,2,[],3]`, `[1,2,3,[]]`.

11.20. Напишите предикат `permute(L, M)`, задающий отношение «список `M` является перестановкой списка `L`», т. е. списки состоят из одинаковых элементов, возможно, стоящих в разном порядке. Проверьте работу в прототипах (+,+) и (+,−).

Как ни странно, этот предикат обычно не желает работать в инвертированном прототипе, т. е. если вы заставили его работать в прототипе (+,−), то, скорее всего, в прототипе (−,+) он не заработает (в лучшем случае уйдёт в бесконечную рекурсию после выдачи всех результатов). Пусть вас это не беспокоит, не вы первые, не вы последние, кто обнаруживает такое свойство у этой задачи. В библиотеке SWI-Пролога есть предикат `permutation`, решающий ровно эту задачу и корректно работающий во всех прототипах, но его реализация использует примитив `var`.

11.21. Напишите двухместный предикат, оба аргумента которого — списки, причём второй аргумент получен из первого вычёркиванием повторений элементов (т. е. если какие-то элементы списка одинаковы, из них должен остаться только первый).

11.22. Напишите на Прологе программы, соответствующие условиям задач 2.22, 2.23 и 4.12.

11.23 (`queens`). В классической формулировке ***задача о восьми ферзях*** выглядит следующим образом: «как на шахматной доске расположить восемь ферзей так, чтобы никакие из них не могли друг друга побить». Вполне очевидное обобщение даёт «задачу об N ферзях»: как на доске размером $N \times N$ клеток расположить N ферзей так, чтобы они не могли друг друга побить; известно, что при $N \in \{2, 3\}$ задача решений не имеет, минимальная доска, на которой такое расположение возможно — 4×4.

Напишите программу на Прологе, принимающую ровно один аргумент командной строки — число N, и отыскивающую все возможные решения задачи об N ферзях. Поскольку, очевидно, каждый ферзь должен находиться на своей собственной вертикали, найденное решение можно представить в виде последовательности из N чисел,

каждое от 1 до N, задающих номера горизонталей, где размещены ферзи на каждой из вертикалей; ваша программа должна каждое из решений выдать на отдельной строке в виде последовательности чисел в десятичном представлении, разделённых пробелами. Так, при $N = 4$ программа должна напечатать:

```
2 4 1 3
3 1 4 2
```

(или наоборот). Для самопроверки можете использовать тот факт, что при $N = 8$ должно получиться 92 решения — естественно, отличных друг от друга.

11.24* (`knight's tour`). Задача «ход конём», впервые (во всяком случае, в сохранившейся до наших дней опубликованной работе) исследованная Леонардом Эйлером в XVIII веке, состоит в том, как, используя ход шахматного коня, обойти все клетки доски, побывав в каждой клетке ровно один раз. Маршрут коня называется *замкнутым*, если после посещения всех клеток конь может вернуться за один ход в ту клетку, с которой начинал обход. Известно, что замкнутые маршруты существуют на квадратных досках $N \times N, N = 6, 8, 10, \ldots$ (т. е. чётного размера, начиная с 6), что касается маршрутов незамкнутых, то их можно найти на досках 3×4, $3 \times M, M \geqslant 7$, 4×5 и на всех более крупных.

Напишите на Прологе программу, принимающую два аргумента командной строки — числа N и M, задающие размер доски — и отыскивающую какой-нибудь один замкнутый маршрут, если для заданной доски такой возможен, если же невозможен — то какой-нибудь один незамкнутый маршрут. Найденный маршрут следует выдать на стандартный вывод в виде последовательности клеток, заданных, например, их координатами через запятую, разделёнными пробелами (что-то вроде «`1,1 2,3 3,1`...») или как-то иначе.

Искать *все* маршруты на доске заданного размера мы не предлагаем, их может оказаться неожиданно много. Так, для доски 5×5 существует 1728 маршрутов (все незамкнутые, поскольку доска хоть и квадратная, но нечётного размера), а на доске 8×8 будет больше 26 *триллионов* одних только замкнутых маршрутов, всего же их (и замкнутых, и незамкнутых) оказывается 19.5 квадриллионов.

11.25*. Напишите на Прологе программу, которая принимает ровно 16 аргументов командной строки, задающих начальное расположение фишек (плиток) игры «15» или «13» (правила игры «13» см. в условиях задачи 10.23, см. стр. 99) — первые четыре аргумента задают верхний ряд плиток, следующие четыре — второй и т. д. Пустая клетка задаётся символом «`@`». Если аргументов не 16, следует выдать сообщение об ошибке и завершиться. Какая игра имеется в

виду — «15» или «13» — программа должна заключить по набору значений аргументов: если набор состоит из чисел от `1` до `15`, то это «15», если из чисел от `1` до `13` и двух «нулей» (обозначаемых «`0`» и «`00`») — то «13», если же ни то, ни другое — это ошибка, о ней следует сообщить и завершить работу.

Программа должна найти решение головоломки; для игры «15» задачу следует считать решённой, когда либо все плитки расположены по порядку, либо по порядку расположены все плитки, кроме двух последних (`15` и `14`), а эти две поменены местами; для игры «13» задача решена, если плитки расположены по порядку, начиная с двух «нулей», взаимное расположение которых неважно. В обоих случаях пустой должна быть нижняя правая клетка. Решение представляет собой последовательность ходов, при которых плитка, соседняя с пустой клеткой, передвигается на пустое место, а её исходная клетка становится пустой; *правильное* решение не содержит повторяющихся позиций (отметим, что если это не учесть с самого начала — скорее всего, вы не сможете создать работающий решатель, поскольку ваша программа будет до бесконечности рассматривать циклические последовательности ходов и никогда не найдёт путь к искомой финальной позиции). Ход обозначается номером плитки, которую на этом ходу передвигают.

Результат работы — последовательность ходов — следует выдать на стандартный вывод.

Хоуп

11.26. Какой тип имеет в языке Хоуп выражение `("abc", 'a', true, 1)`?

11.27. С интерпретатором `hopeless` можно провести следующий диалог (напомним, «`>:`» — это приглашение интерпретатора, т. е. строки, перед которыми стоит эта комбинация, набраны пользователем; свои ответы интерпретатор зачем-то предваряет символами «`>>`»):

```
>: 1,2,'a',true;
>> (1, 2, 'a', true) : num # num # char # bool
>: 1,2,('a',true);
>> (1, 2, 'a', true) : num # num # char # bool
>: 1,(2,'a'),true;
>> (1, (2, 'a'), true) : num # (num # char) # bool
```

Какой полезный вывод можно сделать из этого диалога?

11.28. Пусть нам потребовалась функция, принимающая три числовых параметра и возвращающая список чисел. Каков будет тип такой функции?

11.29. Напишите функцию `MakeSeq`, которая принимает три числовых параметра (назовём их x, y и d) и строит список из членов арифметической прогрессии с разностью d, первым членом x, не превосходящих y. Например, `MakeSeq(1, 4.1, 0.6)` должна вернуть список `[1, 1.6, 2.2, 2.8, 3.4, 4]`.

11.30. Напишите функцию `Twist`, которая принимает два списка, состоящих из элементов какого-то одного (произвольного) типа и возвращает список, состоящий из первых элементов обоих списков, вторых элементов обоих списков и т. д.; например, выражение `Join1([1,2,3], [7,8,9]))` должно вернуть список `[1,7,2,8,3,9]`. Если списки оказались разной длины, «лишние» элементы более длинного списка проигнорируйте.

11.31. Чему равно выражение

```
letrec lst == 1 :: (map ((+3),lst)) in FirstN(lst, 10);
```

— если `FirstN` строит список из первых N элементов своего первого аргумента, получив число N вторым аргументом? (эта функция встречается в примерах, разобранных в книге). А что представляет собой список `lst` в правой части, если к нему не применять `FirstN`?

11.32. В §11.5.6 рассмотрена функция `DropElems` для списка чисел, в §11.5.8 она переписана для произвольных списков. Напишите аналогичную функцию в каррированной форме.

11.33. В §11.5.12 разобраны примеры `Sieve` (решето Эратосфена) и `PascalsTriangle` (построение треугольника Паскаля). Перепишите эти примеры, не применяя кортежей, т. е. без использования символа `#`. Для этого все функции придётся записать в каррированной форме.

11.34. Напишите на Хоупе две-три программы, соответствующие каким-нибудь пунктам задачи 2.19 (см. стр. 2.19). Следите за тем, чтобы ответ на введённую пользователем строку ваша программа давала немедленно после ввода строки — для этого придётся всерьёз задействовать возможности ленивой семантики. Подав своей программе на стандартный ввод бесконечный поток строк (например, с помощью команды `yes`), убедитесь, что количество занимаемой ею памяти не увеличивается со временем.

Задачи для решения на всех языках

Каждую из задач этого раздела можно решить на любом из рассмотренных четырёх «экзотических» языков, и любое такое решение можно сделать достаточно эффектным, раскрыв с новой стороны возможности той или иной парадигмы. Добиться от приведённых задач большей, если можно так выразиться, учебной эффективности можно, если каждую из них решать на трёх языках: на каком-то из лиспов (Common Lisp или Scheme), на Прологе и на Хоупе.

Все программы должны быть оформлены так, чтобы ни при их запуске, ни при работе с ними, в том числе некорректной, пользователь не видел никаких особенностей, обусловленных используемой реализацией языка; программа должна запускаться как обычный исполняемый файл (возможно, представляющий собой скрипт **на используемом языке**, т. е. именно интерпретатор используемого языка должен быть указан в его первой строке), и не должна выдавать приглашений к вводу, сообщений об ошибках и других сообщений, формируемых интерпретаторами (а не вашей программой).

11.35. Напишите программу, которая принимает через аргументы командной строки произвольное количество слов (не менее одного), затем читает из стандартного потока ввода произвольный текст и выдаёт на печать те строки из него, в которых содержится хотя бы одно из слов, заданных в командной строке. Строки следует выдавать (если строка должна быть выдана) немедленно после прочтения; в памяти нельзя хранить весь текст целиком, только текущую строку.

11.36. Напишите программу, которая читает из стандартного потока ввода описание ориентированного графа в виде последовательности предложений, каждое из которых описывает дуги, исходящие из одной вершины. Предложение начинается с имени исходной вершины, после которого ставится двоеточие и перечисляются через запятую вершины, в которые проведены дуги. Предложения разделяются символом точки с запятой, в конце последнего предложения стоит точка. Символы пробела, табуляции и перевода строки могут встречаться где угодно (кроме имени вершины) в любых количествах и должны игнорироваться. Имена вершин состоят из цифр и латинских букв (заглавных и строчных). Например:

```
Living: Animals, Plants, Mushrooms, Protists, Bacteria;
```

```
Animals: Chordata, Arthropoda, Mollusca;
Chordata: Fish, Birds, Mammals, Reptiles, Amphibians;
Arthropoda: Insects, Spiders, Crustaceans.
```

Программа должна прочитать представление графа и установить, является ли заданный граф ***ориентированным деревом***, т. е. орграфом, имеющим ровно один корень, в котором все вершины, кроме корня, имеют ровно одну входящую дугу, а корень не имеет ни одной входящей дуги. В случае, если граф является деревом, напечатать сообщение «`THIS IS A TREE`», если же не является — указать причины (отсутствует корень; имеется более одного корня — перечислить их; имеются вершины с более чем одной входящей дугой — перечислить их).

В случае обнаружения ошибок во вводимых данных следует обязательно выдать осмысленное сообщение об ошибке с указанием номера строки вводимого текста.

11.37. Напишите программу, которая принимает в качестве первого аргумента командной строки текстовую запись арифметического выражения, представленного в польской инверсной записи. Выражение может содержать целочисленные константы (одна или несколько десятичных цифр подряд), переменные (*одна* строчная латинская буква: `a`, `b`, `c` и т. д.) и символы целочисленных операций: `+`, `-`, `/`, `*` и `%` (имеющих тот же смысл, что в языке Си). Напомним на всякий случай, что скобки в ПОЛИЗе не нужны. Переменные и константы должны друг от друга отделяться пробелами, знаки операций пользователь отделять пробелами не обязан (хотя и имеет право). Остальные аргументы командной строки представляют собой значения переменных: второй — значение `a`, третий — значение `b` и т. д.. Ваша программа должна вычислить выражение и напечатать результат (целое число).

11.38. В условиях предыдущей задачи замените польскую инверсную нотацию на прямую (сначала операция, затем операнды; примерно как в Лиспе, только без скобок) и напишите соответствующую программу, по возможности используя уже написанные подпрограммы.

11.39. Используя код программ, решающих задачи 11.37 и 11.38, напишите программу, которая может работать как с прямой, так и с инверсной польской нотацией; если выражение (первый аргумент командной строки) начинается со знака операции, считается, что это прямая нотация, если с операнда (константы или переменной) — обратная нотация.

11.40. Модифицируйте решение предыдущей задачи так, чтобы выражение (в прямой или инверсной польской записи) программа

по-прежнему извлекала из первого аргумента командной строки, а значения переменных читала из потока стандартного ввода: в каждой введённой строке первое число — значение переменной `a`, второе — значение переменной `b` и так далее. После прочтения и анализа каждой строки вычисляйте и печатайте результат (либо сообщение об ошибке, например, если количество значений не совпадает с количеством переменных, либо если введено не число), затем читайте следующую строку и т. д., работайте до наступления ситуации «конец файла».

11.41. Усовершенствуйте решение предыдущей задачи так, чтобы в роли имён переменных в выражении могли использоваться произвольные последовательности латинских букв и цифр, начинающиеся с буквы. Значения переменных задаются в строках, вводимых из стандартного потока: первое слово — имя переменной, второе — её значение, третье — имя другой переменной, четвёртое — её значение и т. д. Вводимая строка может задавать любой набор переменных, не обязательно все, какие есть в выражении; значения переменных сохраняются при вводе последующих строк, заменяются только те, которые указаны в только что введённой строке; первое вычисление значения произведите, когда после анализа очередной введённой строки все переменные, упоминаемые в выражении, окажутся заданы, затем вычисляйте новый результат после ввода каждой строки (это может быть полезно, например, чтобы вычислить некую функцию в большом количестве разных точек).

11.42. Напишите программу, которая **читает из потока стандартного ввода** последовательность выражений, записанных **в польской инверсной записи** (сначала операнды, потом операция), и вычисляет (выполняет) операции по мере их чтения. Вводимый текст может содержать целочисленные константы (одна или несколько десятичных цифр подряд), переменные (**одна или больше** строчных латинских букв подряд: `a`, `xyz`, `kadabra` и т. д.), символы целочисленных операций: `+`, `-`, `/`, `*` и `%` (имеющих тот же смысл, что в языке Си, все операции бинарные), символы служебных операций: «`=`» — присваивание, причём переменная в записи присваивания ставится *после* выражения (например, «`25 x =`» означает «присвоить `25` переменной `x`»; «`?`» — напечатать значение, находящееся на вершине стека, не удаляя из стека; «`!`» — извлечь из стека значение и напечатать его; «`#`» — напечатать на отдельной строке текущее содержимое стека (в угловых скобках через пробел) и значения всех переменных (парами имя-значение, через пробел). При выполнении команд «`?`» и «`!`» выдавайте пробел после числа. Переменные и константы должны друг от друга при вводе отделяться пробелами, знаки операций

пользователь отделять пробелами не обязан (хотя и имеет право). Вычисления должны производиться, когда получен символ перевода строки (либо раньше); если за время обработки введённой строки хоть что-то было напечатано, выдать символ перевода строки. Программа должна продолжать работу до наступления ситуации «конец файла». Например, в ответ на введённую строку

```
15 15*? x= x 5/? 20 foo = 12 13! foo#
```

должно быть напечатано

```
225 45 13
<45 12 20> x 225 foo 20
```

(допускается стек напечатать в виде «`<45 12 foo>`»). **Предусмотрите обработку ошибок!** (некорректное выражение, несуществующая переменная, деление на ноль и т. п.)

11.43. Усовершенствуйте программу, написанную для решения предыдущей задачи, так, чтобы она допускала ровно один аргумент, задающий (в виде числа от `2` до `36`) основание системы счисления, в которой потом будут проводиться вычисления. На вводе «цифры», превышающие `9`, обозначаются латинскими буквами `A` (10), `B` (11) и т. д., вплоть до `Z`, которая в системе по основанию 36 означает цифру 35. Буквы верхнего и нижнего регистра не различайте (ни в именах, ни в литералах). Операнды, запись которых начинается с десятичной цифры, ваш калькулятор должен рассматривать как числа (константы), а начинающиеся с буквы — как идентификаторы переменных; если первая цифра числа обозначена буквой, к нему следует добавить незначащий ноль, чтобы программа не воспринимала его как идентификатор (например, `A25` — это идентификатор переменной, а `0A25` — запись некоторого числа, при условии, что система задана как минимум 11-ричная, в противном случае — ошибка).

11.44. Модифицируйте программу, написанную по условиям задачи 11.42, чтобы она производила вычисления не в целых числах, а в числах с плавающей точкой. Добавьте одноместные операции для округления к ближайшему целому и отбрасывания дробной части. Постарайтесь сделать так, чтобы числа, имеющие нулевую дробную часть, печатались как целые.

11.45*. Напишите программу, которая принимает два аргумента командной строки; первый из них задаёт выражение в обычной инфиксной (не польской!) нотации, включающее константы, записанные в виде целых чисел и десятичных дробей, символы четырёх действий арифметики, символ `^` для обозначения возведения в степень, имена переменных, состоящие из *одной* строчной латинской буквы, отличной от «`e`»; буква `e` считается обозначением числа e (основания натуральных логарифмов), идентификатор `pi` используется для

обозначения числа π, также выражение может включать обозначения тригонометрических функций и обратных к ним (`sin`, `cos`, `tg`, `arcsin` и т. п.), натурального логарифма `ln`, десятичного логарифма `lg`; обычный логарифм в систему лучше не включать из-за сложностей с обозначением его основания. Второй аргумент командной строки должен состоять ровно из одной латинской буквы, обозначающей одну из используемых в заданном выражении переменных.

Ваша программа должна, рассматривая выражение как функцию от (возможно) нескольких переменных, взять её производную по заданной переменной. Обязательно выполните тривиальные оптимизации, такие как замена выражения $0+x$ выражением x, замена $0 \cdot x$ на 0, $1 \cdot x$ на x; позаботьтесь, чтобы итоговое выражение не содержало лишних круглых скобок.

К части 12 «компиляция, интерпретация, скриптинг»

Язык Tcl

12.01. Выберите два-три примера из условий задач 2.19–2.21 (см. стр. 22) и напишите скрипты на языке Tcl, решающие те же задачи. Для посимвольного ввода используйте команду `read stdin 1`, для обнаружения ситуации «конец файла» — команду `eof stdin`; помните, что `eof` проверяет, не настал ли конец файла в ходе последней операции ввода, так что применять её следует *после* `read`.

12.02. Перепишите скрипты, созданные при решении предыдущей задачи, используя в этот раз команду `gets` и последующий анализ прочитанной строки. Для анализа строк достаточно знать, что команда `string length` *<строка>* выдаёт длину указанной строки, а `string index` *<строка>* *<номер>* возвращает символ заданной строки (нумерация идёт с нуля). Сравните получившиеся решения с теми, что использовали посимвольное чтение.

12.03. Выберите два-три примера из условий задач 2.22, 2.23 (см. стр. 24) и напишите соответствующие скрипты на Tcl. Для анализа строк используйте `string length` и `string index` (см. условия предыдущей задачи).

12.04. Напишите на Tcl скрипт, решающий задачу 2.52 (стр. 31).

Библиотека Tcl/Tk

12.05. Для ознакомления с возможностями Tcl/Tk напишите (в форме скриптов для интерпретатора `wish`) программы, соответствующие условиям задач 10.19 и 10.21 (см. стр. 98).

12.06. Используя геометрический менеджер `grid`, напишите на Tck/Tk программу для игры в «15» и «13» (см. задачу 10.23).

12.07*. Вернитесь к задаче 11.24 (стр. 105) и модифицируйте своё решение так, чтобы в ходе поиска программа выдавала координаты клеток, из которых состоят рассматриваемые недостроенные маршруты. Напишите на Tcl/Tk программу, которая принимает те же параметры, что и ваша программа на Прологе, рисует шахматную доску, запускает пролог-программу как внешнюю и в ходе её работы визуализирует происходящий поиск с возвратами.

С задачей 12.07 у автора задачника связаны интересные воспоминания. Задача «ход конём» для решения на Прологе предлагалась слушателям спецкурса «Парадигмы программирования» (да, когда-то очень давно это был именно спецкурс). Заставить программу, написанную на Прологе, работать в связке с Tcl/Tk для визуализации поиска придумали сами студенты, слушавшие курс, автор их никак к такому решению не подталкивал; по правде говоря, в той версии связка была устроена проще: прологовская программа выдавала команды для `wish`, а запускать её (и `wish`) нужно было конвейером.

Когда студент, получивший задание, подходит к нему *настолько* творчески — это, пожалуй, самое вдохновляющее, что может произойти в практике преподавателя. Автору остаётся лишь сожалеть, что за двадцать с лишним лет работы все встретившиеся ему случаи такого рода можно пересчитать по пальцам (и, кажется, даже одной руки), а в последние лет семь такого вообще не встречалось.

У К А З А Н И Я

1.06–1.09. Права доступа к файлам подробно рассмотрены в т. 1, §1.2.13.

1.16. Здесь придётся комбинации из одного, двух и трёх шариков рассмотреть отдельно. С одним шариком всё понятно. С двумя чуть сложнее: как первый, так и второй шарик можно выбрать четырьмя способами, только при этом мы все комбинации, *кроме тех, которые состоят из одинаковых шариков*, посчитаем дважды, ведь условия задачи не предполагают *упорядоченности* выбранных шариков (красный плюс синий — это то же самое, что и синий плюс красный). Ещё хуже обстоят дела с тремя шариками: каждый из них по-прежнему можно выбрать четырьмя способами, но комбинации, состоящие из трёх разных шариков, мы посчитаем по много раз каждую (кстати, по сколько раз? если вы не знаете, что ответить на этот вопрос, перечитайте §1.3.1); с комбинациями, в которых два шарика одинковы, а третий имеет другой цвет, дела обстоят проще — два одинаковых шарика выбираются четырьмя способами, а третий к ним — тремя, так мы изначально игнорируем порядок, что и требуется. Наконец, комбинации, целиком состоящие из шариков одного цвета, нам встретились ровно по одному разу каждая. Все эти случаи придётся рассмотреть отдельно и просуммировать.

1.20. Каждая игрушка может оказаться у Маши, у Пети, у Васи или остаться на столе; исходите из этого.

1.21. Проще всего рассмотреть общее количество возможных подмножеств из всех имеющихся кулонов, включая пустое множество (все кулоны остались в шкатулке) и множество из всех кулонов (в шкатулке не осталось ничего); напомним, что каждый элемент исходного множества может в подмножество входить или не входить, так что количество всевозможных подмножеств вычисляется очень просто. Из этого количества следует выкинуть все «запрещённые» варианты — когда не взято ни одного кулона, когда взят только один или когда взяты вообще все. Задачу можно решить и другим способом: с собой нужно взять два, три или четыре кулона из пяти, что соответствует C_5^2, C_5^3 и C_5^4, остаётся их вычислить с помощью треугольника Паскаля и сложить.

1.22. Первую из двух ладей можно поставить на любую клетку доски, а их, как известно, 64. Для второй ладьи, где бы ни стояла первая, можно выбрать одну из семи оставшихся клеток горизонтального ряда, в котором стоит первая ладья, либо одну из семи клеток соответствующего вертикального ряда, т. е. всего клеток для её размещения оказывается 14. Кроме того, нужно учесть, что ладьи по условию одинаковые, так что каждые две комбинации, получающиеся друг из друга, если ладьи поменять местами, неразличимы; следовательно, нужно ещё разделить полученное число пополам.

1.23. Начните с обруча, зафиксированного в пространстве, сектора которого закрашиваются в определённом порядке — например, по часовой стрелке. Первый сектор можно закрасить одним из пяти цветов, второй — одним из оставшихся четырёх, для третьего цветов останется три. Заметим теперь, что, «сдвинув» выбранные цвета на один или два сектора, мы получим точно такую же комбинацию, поскольку сектора одинаковые и никто не мешает повернуть обруч. Следовательно, каждые три из рассмотренных нами отдельных сочетаний цветов в действительности неразличимы. Кроме того, обруч можно перевернуть, и это сделает неразличимыми каждые два из оставшихся сочетаний.

1.27. Обратите внимание, что сначала мастеру придётся выбрать центральную бусину, потом два дополнительных вида бусин, и после этого, повесив на нить центральную бусину, мастер каждую следующую пару симметричных бусин сможет выбрать всего двумя способами. Все бусы, состоящие из бусин ровно трёх разных видов, в этом числе будут учтены по одному разу; есть ещё комбинации, где третий тип бусины вообще не применялся — такие окажутся учтены по пять раз каждая, поскольку их считали отдельно для каждого из пяти возможных вариантов третьего типа бусины, но эти пять вариантов в итоге неразличимы, так что по четыре из них нужно вычесть.

1.28. Попробуйте выписать три левых колонки треугольника Паскаля — начните с первых трёх строк, для последующих строк выписывайте только три первых числа. Очень легко заметить, что $C_n^1 \equiv n$, что и понятно, если вспомнить смысл чисел C_n^k; заметить закономерность для C_n^2 (для произвольных натуральных n) лишь немногим сложнее.

1.32. Не забывайте, что умножение на основание системы счисления (каково бы оно ни было) технически выглядит как «дописывание нолика»; также примите во внимание, что $3_{10} = 11_2$, а $35_{10} = 32_{10} + 3_{10} = 100000_2 + 11_2 = 100011_2$.

1.33. Искомое количество нулей равно максимальной степени двойки, на которую делится число 30!; эта степень двойки, в свою очередь, соответствует общему количеству двоек, которые получатся при разложении чисел от 1 до 30 на простые множители. Чтобы не пришлось действительно раскладывать на простые множители все тридцать чисел, задайте сами себе вопрос, сколько из этих чисел делятся на два, сколько — на четыре, сколько на восемь и на шестнадцать.

1.38. Как это делается, подробно рассказано в §1.3.2.

1.40. Принцип перевода тот же, что и для десятичных дробей: умножаем на два, если получилось больше единицы — единицу уби-

раем; после каждого умножения выписываем очередную двоичную цифру — единицу, если пришлось её убирать из получившегося числа, ноль в противном случае.

1.41, **1.42**. Воспользуйтесь двоичным представлением длины верёвки в метрах.

1.46, **1.47**. В обеих задачах начните с отрицания $\overline{x}$; имея его в своём распоряжении, через стрелку Пирса можно очевидным образом выразить дизъюнкцию, а через штрих Шеффера — конъюнкцию; после этого попробуйте выразить константы 0 и 1. С остальными функциями будет проще. Список всех 16 существующих функций двух переменных см. в §1.3.3.

1.51. Проще всего сначала перечислить функции двух переменных, которые *не* обладают нужным свойством, то есть такие, которые от одного из своих аргументов не зависят. Таких функций всего шесть: константы 0 и 1, а также функции, равные одному из своих аргументов или его отрицанию: x, $\overline{x}$, y и $\overline{y}$ (см. §1.3.3, а в нём — таблицу 1.8 и пояснения к ней). Из этого соображения немедленно вытекает ответ для функций двух переменных. С трёхместными функциями несколько сложнее, но принцип тот же: не имеющих существенной зависимости ни от одного аргумента — две (константы 0 и 1), существенно зависящих от одной переменной — шесть (x, $\overline{x}$, y, $\overline{y}$, z, $\overline{z}$), существенно зависящие от двух переменных проще посчитать отдельно для каждой из трёх переменных, не входящих в число существенных зависимостей, и для этого воспользоваться уже известным количеством функций двух переменных, существенно зависящих от обеих.

2.07. Попробуйте подойти к вопросу с другой стороны: *что конкретно* считает переменная `i` в каждом из примеров, или, иначе говоря, *чему*, какой величине она равна в каждый момент?

2.15. Воспользуйтесь следующими соображениями. Обозначим размер одной фигуры (первое число, введённое пользователем) буквой H, а количество фигур (второе число) — буквой N. Весь ваш вывод состоит из N отдельных колонок, имеющих ширину $H+1$ символ, причём в каждой колонке (с номером k) первые s_k строк — «пустые», т. е. состоящие из $H+1$ пробелов, затем H строк составляют искомую фигуру, и последующие строки (с номерами, превышающими $s_k + H$) — опять «пустые». Пусть $h = (H-1)/2$ (та самая «половина высоты» фигуры); тогда (если колонки нумеруются с единицы) $s_1 = 0$, $s_2 = h$ и т. д.; общая высота всего изображения при $N = 1$ составляет H, а при увеличении N на единицу эта общая высота увеличивается на h; общие формулы для s_k и высоты изображения получите сами. После этого останется сообразить, что внутри «буквы Z» (если её строки занумеровать с единицы) строки с номерами

1, $h+1$ и H состоят из H звёздочек и пробела (отделяющего следующую фигуру от предыдущей), все остальные — из одной звёздочки, перед и после которой нужно напечатать определённое число пробелов (как оно вычисляется — попробуйте сообразить самостоятельно).

Процедура, печатающая n-ю строку k-й колонки изображения — состоящую из одних пробелов, из одних звёздочек или из одной звёздочки с пробелами спереди и сзади, — должна знать только число H и, конечно, числа n и k, то есть у неё будет три целочисленных параметра; всё остальное она просто вычислит. Обязательно напишите эту процедуру! Это ключ к решению всей задачи. Главная программа будет состоять из простых вложенных циклов `for`: внешний — для печати строк от первой до только что вычисленной общей высоты фигуры, внутренний — для печати нужной строки от каждой из N колонок (путём вызова вышеупомянутой процедуры). Не забудьте про перевод строки после печати всех колонок, относящихся к одной строке!

Решение можно сделать ещё интереснее, если вместо процедуры написать функцию, возвращающую строку (т. е. имеющую тип `string`).

2.16. Проще всего решить задачу следующим образом: просматривать введённую строку слева направо и, зафиксировав текущий символ и убедившись, что это не пробел и не табуляция, вложенным циклом *просматривать остаток строки* (от символа, следующего за текущим) в поисках символов, совпадающих с текущим; при обнаружении таких символов взводить какой-нибудь флаг, означающий, что символ нужно будет напечатать, а сам символ в строке заменять пробелом, чтобы, дойдя до него, программа не напечатала его снова. Символ печатать (или не печатать) после окончания работы вложенного цикла.

Многим программистам может не понравиться предложение модифицировать исходную строку; в самом деле, такое решение нарушает один из базовых принципов программирования: если по смыслу выполняемого с данными действия их не требуется менять, то менять их не следует. Решить эту проблему очень просто: достаточно создать копию строки и работать с ней, а не с основной строкой. Возможны и другие решения.

2.19. Эти задачи можно (теоретически) решить разными способами, но наиболее правильный — использовать ровно один цикл `while`, одну проверку на конец файла и один оператор чтения, как это делается в примерах §2.5.3 (обратите внимание на программы `FilterLength` и `SkipIndented`). Другие подходы чреваты ошибками с отслеживанием конца файла. «Поймать» ситуации начала слова и конца слова очень легко, если в специально для этого созданной

переменной запоминать символ, прочитанный на предыдущем шаге: тогда переход от пробельного символа к непробельному будет означать начало слова, обратный переход — конец слова; постарайтесь только не запутаться с происходящим в начале и конце строки.

2.20. Проще всего на каждом шаге печатать прочитанный символ, но если на этом шаге наблюдается начало слова (см. комментарий к задаче 2.19), то *перед* выводом текущего символа печатать ещё открывающую скобку, если же имеет место конец слова, то (опять-таки перед выводом текущего символа) печатать закрывающую скобку.

2.24. Для этого придётся десятичное представление числа (по крайней мере его дробной части) сгенерировать вручную, без применения псевдофункции `str` или форматирующих возможностей оператора `write`.

2.40, **2.41**. В этих задачах нужен список, в каждом элементе которого хранится число и то, сколько раз число было введено (счётчик вхождений). Не забудьте, что при чтении чисел отслеживать ситуацию «конец файла» следует с помощью функции `SeekEof` (см. §2.5.4).

2.42–**2.44**. Здесь нужно использовать списки символов; чтение, естественно, выполнять посимвольно.

3.01. Для решения примеров из этой задачи нужно помнить структуру регистров общего назначения, о которой рассказывается в §3.2.1; кроме того, стоит освежить в памяти правила записи целочисленных констант в разных системах счисления (см. §3.2.3). Наиболее удобно будет перевести все числа в шестнадцатеричную систему счисления, после чего вспомнить, что каждый байт — это ровно две шестнадцатеричные цифры.

3.12. Организуйте цикл, в котором вызывайте `GETCHAR`, если в `EAX` оказалась `-1`, завершайте работу, если там же число `10` — печатайте `OK` (не забудьте про перевод строки!), если же ни то, ни другое — просто продолжайте выполнение цикла.

3.18. Макросы, представленные в файле `stud_io.inc`, сохраняют значения регистров, за исключением `EAX`, который используется для возврата результата; вы можете использовать остальные регистры процессора так, как вам удобно. Один из регистров следует выделить для хранения постепенно накапливаемого числа. Учтите, что регистры `EAX` и `EDX` требуются для выполнения команды `mul`, так что при каждом умножении они будут портиться; для накапливаемого числа правильнее будет использовать, например, `EBX` или `ESI`. Исходно этот регистр следует обнулить.

После прочтения очередного символа (обращения к макросу `GETCHAR`) следует прежде всего убедиться, что символ, во-первых,

прочитан (то есть ситуация «конец файла» пока не возникла) и, во-вторых, что он является цифрой; если наступил конец файла или прочитана не цифра, выходим из цикла чтения и идём печатать звёздочки. Если же прочитана цифра, следует получить её численное значение, вычтя из прочитанного кода код символа `'0'` (вы, возможно, помните, что это число `48`, но пользоваться сим тайным знанием не надо: наш ассемблер позволяет написать просто `'0'`, это намного понятнее); далее следует число, накопленное к настоящему моменту в регистре, умножить на десять и к результату прибавить численное значение прочитанной цифры. Например, если пользователь ввёл число `1437`, то накопленное число будет меняться следующим образом: сначала это будет ноль, после прочтения единицы — число $0 \cdot 10 + 1 = 1$, после прочтения четвёрки — $1 \cdot 10 + 4 = 14$, после прочтения тройки — $14 \cdot 10 + 3 = 143$, после прочтения семёрки — $143 \cdot 10 + 7 = 1437$, что нам и требуется.

Запустив программу, введите какое-нибудь небольшое число и убедитесь, что программа действительно печатает звёздочки и переводит строку; подсчёт звёздочек доверьте программе `wc`:

```
user@host~$ echo 325 | ./prog | wc -c
326
```

Обратите внимание, что количество символов, которые насчитает `wc -c`, при корректной работе вашей программы будет на единицу больше введённого числа — за счёт добавленного в конце символа перевода строки.

3.19. Цикл чтения и подсчёта символов делается сравнительно просто, мы не будем на этом останавливаться. Искомые цифры, составляющие число, можно получить как последовательность остатков от деления числа на 10; основная сложность в решении этой задачи состоит в том, что печатать эти цифры по мере их вычисления никак нельзя — они должны быть напечатаны в обратном порядке. Одно из простейших решений — создать в памяти массив для хранения этих символов; поскольку числа у нас по условию 32-битные, вам заведомо хватит десяти байт. Заполнять массив можно с начала к концу, а печатать от конца к началу, но лучше будет поступить наоборот; почему это лучше — вы поймёте при решении одной из более поздних задач.

3.22. Используйте стек. Как известно, в стек данные следует помещать порциями по четыре байта; проще будет использовать из каждой такой порции только один байт для хранения одного символа — например, младший; для этого можно занести прочитанный байт в регистр `BL`, а затем занести в стек весь `EBX` целиком.

3.23. Для решения этой задачи следует прочитанные символы занести в стек, а затем для выдачи использовать заполненное символами пространство стека как обычный массив в памяти, не извлекая из него символы. Конкретику придумайте сами.

3.27. Признаем сразу же, что этот этюд довольно сложен. В §3.8.4 вы найдёте описание так называемой польской инверсной записи выражений (ПОЛИЗ), а также алгоритма Дейкстры, с помощью которого можно перевести в ПОЛИЗ произвольное выражение, записанное в традиционной инфиксной нотации. Сам ПОЛИЗ интерпретируется (вычисляется) за один проход слева направо, что позволяет не хранить его, а вычислять по мере построения; для этого нужно объединить реализацию алгоритма Дейкстры с реализацией вычислителя ПОЛИЗа. Отметим, что алгоритм Дейкстры подразумевает использование стека для временного хранения символов операций, тогда как вычисление ПОЛИЗа требует своего собственного стека, но уже для хранения операндов и промежуточных результатов. Аппаратный стек вам понадобится для работы с подпрограммами, так что и стек операций, и стек результатов придётся реализовать как обычные массивы в памяти; здесь можно использовать для сложности вычисляемого выражения какую-то разумную оценку сверху, например, считать, что в оба стека никогда не будет требоваться занести больше 1024 элементов (чтобы превысить этот лимит, придётся построить выражение, содержащее по меньшей мере 512 вложенных друг в друга пар круглых скобок).

Возможно, есть смысл сначала написать решение этой задачи на Паскале, чтобы понять, как это всё вообще делается, и лишь после этого пытаться реализовать то же самое на языке ассемблера.

3.28. Используйте макроповторение и макропеременную для хранения текущего числа. При проверке не забудьте, что параметры макроса перечисляются через запятую.

3.29. Эту задачу можно решить двумя способами. Первый предполагает генерацию последовательности связок из команд `cmp`/`je`, для чего можно применить макроповторение. Второй способ изящнее: он предполагает построение таблицы меток — обыкновенного массива, элементами которому служат как раз переданные макросу метки. В этом случае следует убедиться, что число в `EAX` допустимое (не меньше единицы и не больше количества меток), после чего сделать косвенный безусловный переход, используя в качестве операнда элемент массива. Макроповторение, впрочем, всё равно придётся применить, только на этот раз — для построения массива. Сам массив следует строить прямо в секции кода, только не забудьте, что через него потребуется «перепрыгнуть», и здесь обязательно надо использовать метку, локальную для макроса (см. §3.5.6).

3.34. В ходе решения задачи 3.33 у вас должны были появиться подпрограммы для подсчёта длины строки и выделения последнего символа из строки. Если это так, достаточно вынести каждую из них в отдельный модуль; третьей единицей трансляции будет головная программа. **Если вы решили задачу 3.33, но процедур при этом не написали, вам необходимо срочно переосмыслить своё отношение к происходящему!** Задача интересна именно тем, что выделение подзадач в ней *очевидно*; на практике остро необходимо «на автопилоте» выделять в подпрограммы любые подзадачи, которые вообще возможно выделить, в том числе в намного менее очевидных случаях.

3.43. Воспользуйтесь признаком делимости на три, это упростит задачу. Всё число целиком вам при этом ни в какой момент не потребуется.

4.05. Обратите внимание на рассуждения в §4.3.12 и на то, как там описана функция `swap`, либо хотя бы припомните, что «передача переменной» в языке Си производится через передачу её адреса.

4.07. Это должна быть функция, возвращающая `int`, с одним параметром, имеющим тип `const char *`. Не забывайте в таких случаях про слово `const`!

4.09. Помните, что результат функции `getchar` нельзя напрямую присваивать переменной типа `char`, поскольку при этом вы не сможете отличить друг от друга ситуацию «конец файла» и прочитанный байт со значением `255`. Как следствие, переменная `c` должна иметь тип `int`.

4.12. Решение первого пункта очевидно, что касается второго, то подсчёт количества вхождений провести очень просто: обнаружив очередное вхождение, учесть его (например, увеличить переменную-счётчик), после чего, имея адрес, возвращённый вашей функцией, применить эту функцию к строке, начиная со *следующей* позиции. Например, если значение, возвращённое функцией, вы разместили в указателе `s`, то нужно будет применить функцию к той же подстроке, что и раньше, и выражению `s+1` в качестве строки. После этого функция либо найдёт, либо не найдёт очередное вхождение подстроки.

4.14. Нужно завести два указателя; один будет указывать на то место, *куда* копируются символы, второй — на то место, *откуда* копируются символы. Исходно оба указателя устанавливаются на начало строки. На каждом шаге, если очередной символ, на который указывает указатель «откуда», является пробелом, то просто этот указатель сдвигается на следующий символ; в противном случае выполняется копирование символа, если был скопирован ноль — рабо-

та завершается, иначе сдвигаются уже *оба* указателя и работа цикла продолжается.

4.17. Для рекурсивного решения в пункте b) потребуется вспомогательная функция, имеющая дополнительный («накапливающий») параметр — текущий номер элемента просматриваемого списка, или, если угодно, количество элементов, просмотренных до данного вызова функции.

4.19. Правильнее будет хранить не все числа, а только те, которые удовлетворяют условию (то есть которые потом нужно будет напечатать).

4.36. Чтение, как обычно в таких случаях, следует выполнять посимвольно *одним* циклом. Если допустить вложенные циклы при реализации чтения, вы, скорее всего, где-нибудь запутаетесь; чаще всего результатом такой путаницы становится некорректная реакция программы на *некоторые* случаи возникновения ситуации «конец файла».

Поскольку наперёд неизвестны ни количество слов, ни длина каждого из них, при работе над этой задачей приходится тем или иным способом решать вопрос с динамическим хранилищем для прочитанного. Основных подходов к этому можно выделить два: построение временного списка символов и использование для хранения символов временного промежуточного массива, который по мере необходимости можно увеличивать в размерах. Профессиональные программисты обычно избегают списков символов, поскольку их применение приводит к непроизводительному расходу оперативной памяти (менеджер кучи обычно работает со «скважностью» в 64, а то и 128 байт, так что именно столько памяти в действительности выделяется на каждый элемент списка, то есть для хранения *одного* байта полезной информации); тем не менее, если такое решение кажется вам проще, воспользуйтесь им.

Если всё же вы решите использовать промежуточный массив, то для его организации потребуются три переменные: указатель на массив, имеющий тип `char*`, и две целочисленные переменные, хранящие текущий размер самого массива и то, сколько в нём в настоящий момент занято элементов. Правильнее будет объединить все три переменные в одну структуру — с тем, чтобы ваши функции, обеспечивающие работу с массивом, получали одним из своих параметров (обычно первым) указатель на эту структуру. Занесение в массив очередного символа следует обязательно реализовать отдельной функцией, которая будет при необходимости изменять размер массива, например, увеличивая его вдвое; для этого нужно создать новый массив, переписать в него информацию из старого, старый удалить.

Функция `realloc` из стандартной библиотеки позволяет несколько сократить объём кода при изменении размера массива; не применяйте эту функцию в учебных задачах, иначе так и не научитесь уверенно работать с массивами, предполагающими переменную длину!

Второе принципиальное решение, которое вам надлежит принять — это следует ли прочитанные слова заносить в итоговый список по мере их прочтения, или же лучше будет сначала прочитать и разместить во временном хранилище (будь то список символов или промежуточный массив) всю строку целиком, и уже потом разбить её на отдельные слова.

В обоих случаях на момент формирования очередного слова уже известна его длина, поэтому следует выделить ровно столько памяти, сколько под эту строку нужно (на один байт больше, чем букв в слове), скопировать в эту память буквы из временного хранилища, не забывая при этом про ограничительный ноль в конце строки, а саму сформированную строку включить в итоговый список слов.

Если вы используете для временного хранения список символов, его элементы следует удалить, как только вся информация из них переписана в итоговые строки. Если же вы предпочли промежуточный массив, то удалять его каждый раз не стоит: достаточно занести ноль в переменную, содержащую текущее количество хранимых символов, и продолжить использование массива (ныне «пустого») при анализе последующих строк.

Большинство новичков при попытке решения обсуждаемой задачи делает одну и ту же тактическую ошибку: не видя толком, что тут можно вынести в отдельные подзадачи, они пытаются весь цикл чтения реализовать в одной функции (обычно `main`, хотя не всегда), и эта функция в момент «отбивается от рук» — достигает в длину ста и более строк. Как правило, когда это уже произошло, декомпозицию проводить поздно, и всё приходится переписывать с нуля. Если это произошло с вами, помните: **функции длиннее 50 строк — это очень плохо, а функции длиннее 70 строк — это уже криминал**, и такие функции можно считать ненаписанными вне всякой зависимости от обстоятельств. Задание, о котором идёт речь, в этом плане очень интересно: оно позволяет понять, готовы ли вы к серьёзным программам в плане декомпозиции задач на подзадачи. Итак, **если у вас получилась громоздкая функция, превосходящая 50 строк, обязательно сотрите её и перепишите ваше решение с нуля** — так, чтобы монстров в этот раз не возникало. Для этого заранее продумайте, какие подзадачи стоит реализовать до того, как вы приступите к написанию главной функции.

Если ничто иное не помогает, здесь можно посоветовать один достаточно универсальный подход. Вы можете объединить в рамках

одной структуры всю информацию, связанную с чтением и анализом строки. Например, в такую структуру могут войти текущее состояние анализатора (находитесь вы внутри кавычек или нет), указатели на начало и конец сформированной части списка слов, а также всё, что нужно для временного хранения того слова, которое находится в процессе считывания (будь то промежуточный массив или список символов). Если так сделать, задача довольно естественным образом распадётся на подзадачи. Цикл чтения символов сводится к тому, чтобы после проверки, не настал ли конец файла, «отдать» очередной символ в обработку, вызвав соответствующую функцию, после чего проверить, не окончено ли чтение строки и не пора ли «выполнять команду» (на первом этапе вместо выполнения команды у нас будет печать слов). Если время для этого пришло, то нужно вызвать ещё две функции: первая из них «выполнит команду» (напечатает слова), вторая — очистит структуры данных, освободив всю занятую динамическую память (кроме, возможно, промежуточного массива, если вы его используете) и подготовит всё для чтения и анализа следующей строки.

Большая часть оставшейся «сложности» при таком подходе перекочует в функцию обработки отдельного символа, которая будет вынуждена проверять, пробельный это символ или нет, если пробельный — то имеется ли какое-то непустое накопленное слово, если имеется — создавать следующий элемент в списке слов, а накопленное во временном хранилище сбрасывать; если символ не просто пробельный, а ещё и перевод строки, нужно взвести какой-то флажок, чтобы другая функция могла отрапортовать о готовности строки к «выполнению команды»; если символ не пробельный, его нужно добавить во временное хранилище, и т. д. Но, во-первых, сложностей на долю этой функции остаётся изрядно меньше, а во-вторых, она тоже прекрасно бьётся на подзадачи.

Забегая вперёд на целый том, можно отметить, что такой подход к борьбе со сложностью соответствует небезызвестному ***объектно-ориентированному программированию***; в роли объекта тут выступает структура, объединяющая всё, что нужно в ходе чтения, ну а функции, которые написаны для работы с этой структурой, можно рассматривать как ***методы***. Если вы не знаете, о чём тут идёт речь, не обращайте внимания — всему своё время; объектно-ориентированный подход к программированию подробно разбирается в третьем томе «Введения в профессию».

4.31, **4.32**. Язык Си позволяет проверить эквивалентность двух способов введения одного и того же имени переменной или функции; для этого достаточно в одной единице трансляции либо *объявить* этот идентификатор дважды (одним способом и другим), либо сначала *объявить* его одним способом, а потом *описать* другим. Для переменных это означает, что нужно предъявить компилятору снача-

ла объявление со словом `extern`, сделанное одним способом, а затем ввести тот же идентификатор вторым способом (extern в этом случае можно ставить, а можно и не ставить, тогда это будет описание). Для функций объявлением считается заголовок без тела, так что первый способ введения имени функции должен иметь вид заголовка, а второй может быть как заголовком, так и описанием функции (т. е. можно снабдить второй вариант заголовка телом).

Например, следующий фрагмент

```
typedef void (*hdlfun)(int);
hdlfun set_handler(int num, hdlfun hdl);


void (*set_handler(int num, void (*hdl)(int)))(int)
{
    return num ? hdl : (void*)0;
}
```

не вызовет возражений у компилятора; это означает, что имя функции `set_handler` в обоих случаях имеет строго один и тот же тип.

5.10. Обратите внимание, что программа сначала напечатает строку `start`, после чего породит второй процесс вызовом `fork` — «раздвоится». Процессы будут выполняться без синхронизации, так что неизвестно, появится ли первой строка `child` или `parent`; формально говоря, неизвестно и то, успеет ли процесс, напечатавший эту строку, напечатать также и свой экземпляр строки `finish` (как легко заметить, эту строку они напечатают оба).

5.14. Следует воспользоваться одним из системных вызовов `waitpid`, `wait3` или `wait4`, указав опцию `WNOHANG`; обращения к системному вызову повторять в цикле, пока возвращаемое значение строго положительно; когда будет возвращён `0` (есть порождённые процессы, но среди них нет ни одного зомби) или `-1` (процессов больше нет) — цикл завершить.

5.12. Вам потребуются вызов `fork`, функция `execvp`, вызов `wait` и макросы для анализа его результатов (см. §§5.3.4–5.3.7). В качестве аргументов `execvp` условия задачи позволяют использовать части вашей собственной командной строки — выражения `argv[1]` (имя запускаемой программы) и `argv+1` (командная строка для неё), новых массивов можно не создавать.

5.15. Согласно условиям задачи 4.36 у вас должен был получиться список строк, содержащих слова введённой пользователем командной строки, то есть фактически список указателей типа `char*`. Для выполнения команды, введённой пользователем (т.;е. для запуска внешней программы), вам потребуется обратиться к функции

`execvp`, которая ожидает своим вторым параметром те же самые указатели, только собранные не в список, а в массив. Сформировать нужный массив очень просто: достаточно описать указатель нужного типа (`char**`, ведь он должен указывать на элементы типа `char*`) посчитать количество элементов вашего списка, прибавить единицу, выделить память под массив нужного размера, состоящий из элементов типа `char*` (например, если подсчитанное количество слов содержится в переменной `wcnt`, то потребуется что-то вроде `malloc((wcnt+1)*sizeof(char*))`), а затем циклом перенести адреса из указателей, содержащихся в списке, в элементы полученного массива; нужно только не забыть занести в последний элемент массива значение `NULL`. Если ваш указатель на начало массива назывался `words`, то первым аргументом для `execvp` послужит `words[0]`, а вторым — сам указатель `words`. Элементы списка можно после этого удалить, но только сами элементы (структуры), а не строки, адреса которых там хранились — эти строки теперь следует считать принадлежащими сформированной командной строке.

Когда массив готов, следует проверить, не является ли первое слово (имя команды) строкой «`cd`», если да — обработать этот особый случай, если же нет — исполнить обычную связку `fork`+`execvp`; не забудьте про `perror`/`exit` (или `perror`/`fflush`/`_exit`) после `exec` — им предстоит сработать, когда пользователь вместо команды введёт белиберду (или просто ошибётся в имени команды).

Для случая команды `cd` следует **в обязательном порядке** обработать значение, возвращённое вызовом `cd`; если он вернул пресловутую минус-единицу — скорее всего, пользователь ошибся при вводе имени директории. Худшее, что можно здесь сделать — это «тихо проигнорировать» ошибку, ведь пользователь тогда будет уверен, что директория благополучно изменилась. Самое правильное (и, заметим, самое простое) решение — вызвать функцию `perror`, подав ей параметром **имя директории**, с которым возникли проблемы (а не что-то другое).

Учтите, что в этой программе очень легко пропустить где-нибудь утечку памяти, то есть забыть где-то что-то очистить. Будьте внимательны на этот счёт! Можно чуть-чуть облегчить себе жизнь, если массив для `execvp` формировать уже в порождённом процессе, непосредственно перед обращением к `execvp`; в этом случае всё, что касается этого массива, можно будет не удалять, ведь впереди либо замена выполняющейся программы, либо (в случае ошибки) завершение процесса, и в обоих случаях ваш сегмент данных перестанет существовать весь целиком. Впрочем, это никак не отменяет удаления списка строк (и в этот раз — вместе с самими строками) в

главном процессе, а ещё при этом будет несколько сложнее (хотя и не намного) проверять особые случаи, связанные с `cd`.

5.16. Узнать домашнюю директорию пользователя можно из переменной окружения `HOME`; соответствующую строку вам вернёт `getenv("HOME")`, только не забудьте, что переменной `HOME` в окружении может вообще-то и не быть, и это совершенно не повод для аварийного завершения. Если пользователь дал команду `cd` без параметров, а `getenv("HOME")` вернул вам `NULL`, можно напечатать диагностику вроде `I don't know where's your home :-(`

5.17. Если «в фоне» уже выполняются какие-то программы, а пользователь при этом требует очередную команду выполнить в обычном режиме, то ваш вызов `wait` может «дождаться не того». В самом деле, команда, которую пользователь ввёл последней, может выполняться достаточно долго, чтобы какой-то из процессов, запущенных ранее в фоновом режиме, за это время завершился, опередив процесс, выполняющий последнюю команду. Проверить это достаточно просто: сначала запустите в фоне (с амперсандом) команду `sleep 15`, а потом (уже без амперсанда) команду `sleep 1000`; если в вашей программе будет только один `wait`, она очередное приглашение командной строки выдаст после завершения `sleep 15`, и вы сможете легко убедиться, что `sleep 1000` всё ещё «висит», но ваш интерпретатор её завершения уже не ждёт, хотя вроде бы должен.

Конечно, решается эта проблема очень просто, её, пожалуй, труднее осознать, чем решить. Достаточно вспомнить, что вызов `wait` возвращает `pid` того процесса, которого он дождался, а функция `fork` в родительском процессе возвращает `pid` процесса, который она запустила. Осталось только при запуске команды запомнить `pid` получившегося процесса, а после, если её завершения нужно дожидаться (то есть если она не фоновая) — сделать не одно обращение к вызову `wait`, а цикл таких обращений, работающий до тех пор, пока не дождётся, кого надо. Например, если идентификатор нужного процесса находится в переменной `pid`, цикл может выглядеть примерно так:

```
do {
    p = wait(NULL);
} while(p != pid);
```

Кроме этого, наличие фоновых задач означает, что время от времени стоит чистить систему от получающихся зомби; на эту тему см. указания и ответ к задаче 5.14. На данном этапе можно ограничиться проведением такой очистки, скажем, непосредственно перед

вводом очередной команды, а также перед запуском фоновой команды (перед запуском нефоновых команд это делать бессмысленно, поскольку с убиранием из системы зомби-процессов прекрасно справится цикл обычных `wait`'ов, который вы начнёте выполнять сразу после порождения процесса).

Для тестирования написанного решения этой задачи удобно применять какой-нибудь из эмуляторов терминала, например, простой `xterm`. Его экземпляры можно запускать в качестве тестовых фоновых задач, в том числе с параметрами — например, команда `xterm -e top` позволяет получить окошко `xterm`'а, в котором сразу же запущена программа `top` (напомним, что выход из неё производится буквой `q`).

При тестировании обратите внимание ещё на один момент, который многие при решении этой задачи упускают — внутри кавычек амперсанд должен быть обычным символом без какой-либо специальной роли. Например, команда «`echo "&"`» должна просто печатать символ «&» и переводить строку, здесь амперсанд не имеет никакого отношения к фоновому режиму, поскольку встречен внутри кавычек. В условии задачи специально подчёркнуто, что в роли разделительной лексемы, обозначающей фоновый режим, выступает амперсанд *вне кавычек*, но на это почему-то часто не обращают внимания.

5.20. Прежде всего напомним, что при нажатии `Ctrl-C` активная группа процессов (в данном случае — ваша программа) получает сигнал `SIGINT`. В этой задаче вам нужно поймать всего один сигнал, так что на возможные вариации семантики вызова `signal` можно не обращать внимания.

Потребуется одна глобальная переменная, показывающая, приходил сигнал или нет; можно использовать простой `int`, но желательно не забывать про слово `volatile`. Глобальные переменные инициализируются нулями по умолчанию, но это тот случай, когда правильнее будет написать инициализатор явно, чтобы избавить читателя программы от необходимости про это вспоминать. В обработчике сигнала следует присвоить этой переменной, например, единицу; это всё, что должен сделать обработчик в этой задаче.

В функции `main` сначала вызываем `signal` для установления новой диспозиции для `SIGINT`, затем выдаём первое сообщение, запускаем цикл вида

```
while(!sig_caught)
    sleep(1);
```

(где `sig_caught` — имя вашей глобальной переменной). После этого цикла выдаём второе сообщение. Собственно говоря, всё. Вместо

`sleep(1)` можно вызвать `pause()`. В принципе с `pause` в данном случае можно даже обойтись без цикла и без переменной (а тело обработчика оставить пустым), поскольку никаких других сигналов программа не обрабатывает, но лучше так не писать — программа сильно проиграет в ясности и к тому же этот вариант перестанет работать, как только потребуется какой-нибудь ещё сигнал.

5.21. Не забывайте, что `printf` использовать внутри обработчика сигнала нельзя, так что выводить число придётся из главной программы. В функции `main` после установки диспозиций для `SIGINT` и `SIGUSR1` следует сделать *бесконечный* цикл, в теле которого сначала вызывается `sleep` или `pause`, потом проверяется флажок, отвечающий за получение `SIGUSR1` (в этой роли вполне сойдёт обычная глобальная переменная типа `int`), и если он поднят, то печатается текущее значение счётчика, а сам флажок сбрасывается обратно.

Обратите внимание, что каждый из сигналов здесь по условию «ловится» больше одного раза, так что нужно позаботиться о разнице семантики функции `signal` — в обоих обработчиках первым действием переустанавливать диспозицию сигнала на случай, если семантика попалась «одноразовая».

5.22. В этой задаче вам потребуется три функции-обработчика (для сигналов `SIGINT`, `SIGQUIT` и `SIGALRM`) и две глобальные переменные: одна для простоты картины может быть типа `char` и хранить печатаемый символ, другая — любого целого типа (хотя бы и просто `int`), она будет использоваться как флажок, *разрешающий выход по `Ctrl-C`*. Обе переменные следует снабдить инициализаторами.

В функции `main`, установив диспозицию всех трёх сигналов и «взведя будильник» (т. е. вызвав `alarm(1)`), организуйте бесконечный цикл, в теле которого вызывается `sleep` или `pause`. Больше `main` ничего делать не будет, всю функциональность возьмут на себя обработчики.

Поскольку печать символа будет происходить изнутри обработчиков, её придётся выполнять прямым обращением к вызову `write`, функции высокого уровня (тот же `putchar`) тут не подходят. Впрочем, от них всё равно в данном случае не было бы никакой пользы — буферизацией мы в этой задаче воспользоваться не можем, ведь символы должны появляться на экране сразу, а не после перевода строки. Печать символа, находящегося в соответствующей глобальной переменной, придётся производить из всех трёх обработчиков, так что правильнее будет соответствующий вызов `write` вынести в отдельную функцию, назвав её, например, `write_the_char` (без параметров).

В обработчике сигналов `SIGINT` и `SIGQUIT` следует присвоить соответствующее значение переменной, хранящей символ, напечатать

этот символ и перевзвести будильник (повторно вызвать `alarm(1)`); в обработчике `SIGINT`, кроме того, нужно проверить флажок, разрешающий выход по `Ctrl-C`, если он взведён — завершить работу с помощью `exit`, а если не взведён — то взвести его. В обработчике `SIGALRM` — перевзвести будильник, сбросить флажок и вывести символ.

Не забывайте про различия в семантике `signal`!

5.23. Если эта задача вызывает сложности, попробуйте сначала решить задачу 5.22, пользуясь приведёнными выше указаниями.

5.25. В этой задаче, как и при любой работе с каналами, следует обратить особое внимание на закрытие всех ненужных дескрипторов, связанных с каналом: в порождённом процессе нужен только дескриптор на запись, в родительском — только дескриптор на чтение, остальные нужно закрыть сразу после порождения процесса. Так, если массив дескрипторов в вашей программе называется `fd`, то порождённый процесс должен начать свою работу с закрытия `fd[0]`, ведь он будет только записывать данные в канал, используя `fd[1]`; родительский процесс, напротив, должен сразу же закрыть `fd[1]`, поскольку он будет только читать данные из `fd[0]`.

Для чтения из канала и записи в канал используйте вызовы `read` и `write`; строки, предназначенные к передаче, лучше всего оформить как константные массивы типа `char`; например, строку, введённую как

```
const char message1[] = "Take a sad song, and make it better";
```

можно отправить в канал примерно так:

```
write(fd[1], message1, sizeof(message1)-1);
```

(обратите внимание на вычитание единицы — отправлять нулевой байт, обозначающий конец строки, нам не надо). Если же у вас нет имени массива, а есть простой указатель на строку, имеющий тип `char*`, отправка такой строки будет выглядеть так (в этот раз ничего вычитать не надо):

```
write(fd[1], str, strlen(str));
```

Приниципиальная разница тут в том, что `sizeof` вычисляется (точнее, просто подставляется) компилятором во время компиляции, и в машинном коде фигурирует обыкновенное число, тогда как `strlen` вычисляется во время работы программы путём просмотра строки в поисках нулевого байта, и на это тратится время.

Цикл чтения и вывода в родительском процессе следует организовать блоками, например, так:

```
int rc;
char buf[1024];
/* ... */
while((rc = read(fd[0], buf, sizeof(buf))) > 0)
    write(1, buf, rc);
```

Обратите внимание, что это как раз тот редкий случай, когда побочный эффект в заголовке цикла вполне оправдан (см. §4.8.3).

5.26. Как легко догадаться, здесь нужен неименованный канал (`pipe`); собственно говоря, для этого они и придуманы. Перенаправление вывода первой программы и ввода второй следует выполнить с помощью `dup2`; если ваш массив дескрипторов называется `fd`, то первый порождённый процесс должен сделать `dup2(fd[1], 1)`, второй — `dup2(fd[0], 0)`. Интересно, что после этого *во всех трёх процессах* — и в родительском, и в обоих порождённых — оба дескриптора `fd[0]` и `fd[1]` оказываются лишними и должны быть закрыты. Не забудьте в родительском процессе дождаться завершения обоих порождённых.

Для тестирования подобных программ удобны команды `cat`, `grep`, `wc`, `head`, `tail` и т. п., а «материалом» для их работы могут послужить всевозможные текстовые файлы — например, ваши исходные тексты разнообразных программ вполне подойдут.

5.27. Поскольку здесь нужен вывод текстовых данных, будет удобно воспользоваться высокоуровневыми средствами вывода (в данном случае — функцией `fprintf`), которые заодно обеспечат правильно организованную буферизацию вывода. Организовать поток вывода, имеющий тип `FILE*`, на основе имеющегося файлового дескриптора позволяет библиотечная функция `fdopen`. Студенты в таких случаях часто предпочитают перенаправить в канал поток стандартного вывода и использовать обычный `printf`, но делать так не надо — в «боевых» случаях (в отличие от демонстрационной задачи из задачника) поток стандартного вывода вам наверняка ещё пригодится.

Отдельное происшествие здесь — своевременно обнаружить, что внешняя программа завершилась досрочно. Если использовать обычный `write`, проще всего будет установить сигналу `SIGPIPE` диспозицию `SIG_IGN`, в этом случае ваша программа не будет убита, вместо этого `write` вернёт `-1`, а `errno` примет значение `EPIPE`. Можно так поступить и с выводом высокого уровня, нужно только не забывать анализировать значение, возвращаемое функцией `fprintf`. Кроме того, сигнал `SIGPIPE` можно перехватить и в обработчике взводить некий флажок, а в цикле печати его проверять. Можно перехватить

и `SIGCHLD`, но с `SIGPIPE` всё равно придётся что-то сделать, ведь иначе он прибьёт ваш главный процесс, скорее всего, ещё до того, как `SIGCHLD` будет отправлен.

5.28. Стандартный вывод внешней программы перенаправляем в канал с помощью `dup2` (что-то вроде `dup2(fd[1], 1)`), после этого в порождённом процессе закрываем оба исходных дескриптора канала и делаем `execvp`. В родительском процессе чтение лучше всего организовать посимвольно, но, конечно, ни в коем случае не следует читать по одному символу с помощью `read`, это будет слишком неэффективно; чтобы воспользоваться буферизацией, реализованной в стандартной библиотеке, создайте высокоуровневый поток ввода (`FILE*`), связанный с нужным концом канала, с помощью функции `fdopen`, а второй конец канала закройте. Для чтения после этого можно будет использовать функцию `fgetc`.

5.29. Здесь потребуется ещё один процесс, который будет анализировать вывод внешней программы и печатать результаты подсчёта. Обойтись одним порождаемым процессом в этой задаче теоретически возможно, но для этого нужно владеть механизмами мультиплексирования ввода-вывода, до которых вы, скорее всего, ещё не дошли, они рассматриваются в части, посвящённой компьютерным сетям; если же про них не знать, скорее всего, ваша программа, пытаясь найти баланс между отправкой информации по одному каналу и получением её по другому, загонит себя в самоблокировку — точнее говоря, как бы вы ни написали вашу программу, можно будет создать такую внешнюю программу, которая вашу «повесит».

Каналов, естественно, тоже потребуется два, и здесь любой забытый дескриптор может привести к «вечной блокировке»; это означает, что после создания каналов и порожения обоих дополнительных процессов ваш родительский процесс должен закрыть три дескриптора концов каналов (из четырёх имеющихся), процесс, в котором будет исполняться внешняя программа, должен закрыть два дескриптора, процесс, предназначенный для анализа вывода — три дескриптора.

Не забудьте в родительском процессе «дождаться» обоих потомков. Кроме того, нужно предусмотреть корректную обработку для случая досрочного завершения внешней программы (см. указания к предыдущей задаче).

5.30. В этой задаче нужен третий порождаемый процесс, который займёт в конвейере место «посерёдке» между первой и второй программой. Проще всего будет в этом процессе перенаправить стандартные потоки и выбирать из потока ввода нужную часть информации, используя обычный цикл посимвольного чтения с посимвольной же записью, примерно как в задаче 4.09.

5.31. Основная проблема при выполнении команды, содержащей знак конвейера — сформировать *две* командные строки, т. е. два массива указателей на строки, и не потерять при этом информацию о перенаправлениях и фоновом режиме. Универсальной рекомендации, как конкретно это делать, дать невозможно, ваша программа уже достаточно сложна — ориентируйтесь по обстоятельствам.

Конвейер считается завершённым, когда завершились все его элементы (в данном случае — *оба*). Как уже говорилось (см. указания к задаче 5.17 на стр. 63), ожидать окончания нефоновой задачи нужно циклом обычных `wait`'ов с проверкой, «того ли» мы дождались; здесь дело осложняется тем, что процессов, которых нужно дождаться, стало два, т. е. нужно хранить информацию о том, не завершился ли уже каждый из них, и прекращать ожидание, когда будет установлено, что завершились оба процесса.

Фоновый конвейер удобно тестировать на связке команд `yes` и `wc`, например, такой командой:

```
yes "abrakadabra" | wc -l > count.txt &
```

Обязательно попробуйте, что произойдёт, если с помощью команды `kill` «пристрелить» любой из элементов конвейера. Если вы сначала убьёте команду `yes`, второй элемент должен завершиться, причём успешно — у него на стандартном вводе возникнет ситуация «конец файла»; если этого не произошло — вы где-то забыли закрыть «лишний» дескриптор канала, предназначенный для записи. Если же пристрелить команду `wc`, то первый элемент конвейера (`yes`) тоже должен завершиться, но уже по сигналу `SIGPIPE`. Если, опять-таки, этого не произошло — значит, где-то остался незакрытым дескриптор канала для чтения.

Ещё рекомендуется попробовать следующие две команды:

```
sleep 15 | sleep 1
sleep 1 | sleep 15
```

Выполнение каждой из двух команд должно длиться пятнадцать секунд, лишь после этого ваш интерпретатор должен выдать приглашение к вводу следующей команды. Пока ждёте, посмотрите в соседнем терминале, не висит ли завершившийся `sleep` в виде «зомби»; если висит — вы как-то не так организовали цикл ожидания.

5.32. Самое сложное тут — не запутаться с дескрипторами каналов в цикле, в котором будут запускаться элементы вашего конвейера. Если с первого раза не получается — не беспокойтесь, это у всех так.

Когда программа заработает, для начала протестируйте её на чём-нибудь вроде

```
cat myprogram.c | grep int | grep -v main
```

Вы должны увидеть все строчки вашей программы `myprogram.c`, в которых есть слово `int`, кроме строки, содержащей заголовок функции `main` (опция `-v` для команды `grep` означает инверсию результатов). Увеличить количество элементов любого конвейера можно, добавляя в него команды `cat`, например:

```
cat myprogram.c | cat | cat | grep int | cat | grep -v main | cat
```

Проверьте, что корректно работает перенаправление ввода, например:

```
cat | grep int | grep -v main < myprogram.c
```

Если ваш интерпретатор не ругается на эту команду, как ошибочную, то перенаправлен при этом должен быть поток ввода *первой* программы; если вы решили такого варианта не допускать, то вот такой уж точно должен работать:

```
cat < myprogram.c | grep int | grep -v main
```

(отметим, что в настоящем интерпретаторе такой вариант работает, а первый вариант там перенаправляет поток ввода для последнего элемента конвейера, делая весь конвейер неработоспособным; так делать не надо).

Наконец, проверьте, всё ли у вас в порядке с дескрипторами. Для этого подойдёт, например, такое:

```
yes | cat | cat | cat | cat | cat | cat > /dev/null &
```

После запуска этого конвейера посмотрите список процессов (желательно команду `ps ax` дать прямо из вашего же интерпретатора, в конце концов, он командный интерпретатор или что?) и отстрелите командой `kill` процесс `cat` в середине конвейера. Если ваша программа написана правильно, после этого все остальные элементы конвейера должны немедленно завершиться: те, что слева — по сигналу `SIGPIPE`, те, что справа — по концу файла. Если этого не происходит — ищите недозакрытые концы каналов.

Кстати, есть ещё один момент, достойный тестирования. Во всех наших примерах мы расставили для красоты «декоративные» пробелы, так что разделители фактически не выполняют свою функцию. Чтобы убедиться, что программа в этом аспекте соответствует сформулированному заданию, попробуйте вот так:

```
yes|cat|cat|cat|cat|cat|cat>/dev/null&
```

5.33. Если непонятно, как переключить терминал в «слепой» режим, перечитайте §5.4.3 и обратите внимание на разобранный там пример.

5.34. Для решения этой задачи придётся вывести терминал из канонического режима, отключить автоматическое отображение вводимых символов (`ECHO`) и всю работу, которую в каноническом режиме делает терминал, взять на себя. Флаг `ISIG` лучше оставьте установленным, иначе вашу программу будет довольно сложно прервать: «спасительные» комбинации `Ctrl-D`, `Ctrl-C` и `Ctrl-\` потеряют свой магический эффект и будут генерировать обычные символы (байты со значениями `4`, `3` и `28` соответственно). Впрочем, вы можете сами предусмотреть возможность корректного выхода.

Читать из потока ввода придётся вызовом `read`, и лучше это делать блоками, но каждый введённый символ обрабатывать отдельно (в цикле после `read`). Если этот символ не имеет специального смысла с точки зрения вашей программы, его следует выдать «на экран» (в стандартный поток вывода), поскольку терминал теперь за вас этого не сделает, вы ведь отключили `ECHO`; только при этом не забудьте про буферизацию: либо производите вывод непосредственно вызовом `write`, либо можно воспользоваться функцией `putchar`, при этом вызывать `fflush` для вытеснения буфера. Преимущество такого варианта в том, что `fflush` можно вызывать не после каждого символа, а после, например, завершения обработки группы символов, введённых одним вызовом `read`.

Кроме кодов перевода строки и табуляции, обязательно обработайте коды `8` (`Backspace`, `'\b'`) и `127` (`Del`), можно обрабатывать их одинаково — удалением последнего введённого символа. Для возврата курсора на позицию назад можно вывести код `Backspace` (просто вывести символ `'b'` обычным путём), но этого одного будет недостаточно, поскольку выведенный символ не исчезнет; чтобы всё получилось как надо, нужно вывести вместо него пробел и снова вернуть курсор обратно, это делается выдачей строки `"\b \b"`.

При проверке работоспособности программы обязательно просмотрите получившийся файл с помощью `hexdump -C`, не затесались ли в итоговые данные какие-нибудь служебные символы. Учтите, что, например, строка `"HelCRAP\b\b\b\blo"`, будучи напечатанной на терминале, выглядит просто как слово `Hello`; заметить разницу можно только с помощью `hexdump` или других аналогичных средств.

5.35. Всё, что нужно здесь знать — это что некоторые клавиши, включая «стрелки», при работе в терминале ОС Unix традиционно генерируют escape-последовательности, причём после собственно кода `Escape` (`27`) в *этой* группе последовательностей всегда идёт символ `[` (код `91`, `0x5B`). Если такую последовательность получает про-

грамма, не умеющая её обрабатывать, а сама последовательность при этом оказывается напечатана — выглядит это довольно забавно; читатель наверняка уже не раз наблюдал на экране что-то вроде `^[[A` — именно это мы увидим, например, если запустим какую-нибудь из наших программ, читающих с клавиатуры (*не меняющих режим терминала*, т. е. это не относится к программам на Паскале, использующим модуль `crt`, и программам на Си с применением библиотеки `ncurses`, а равно и к тем, которые мы пишем сейчас с использованием `termios.h`), а потом случайно нажмём стрелку вверх. Здесь `^[` — это изображение кода `27` (тот самый `Escape`), а остальные два символа — `[` и `A` — изображают, собственно говоря, сами себя, то есть это буквально символ открывающей квадратной скобки и латинская буква `A`.

Мы могли бы здесь привести таблицу последовательностей для всех «хитрых» клавиш, но не будем лишать читателя удовольствия выяснить их все самостоятельно экспериментальным путём. Довольно неожиданным оказывается тот факт, что последовательности, генерируемые разными клавишами, имеют различную длину — так, «стрелки» выдают три символа, а, скажем, клавиша `Ins` — четыре. Впрочем, там есть определённая логика, попробуйте «поймать» её сами, анализируя последовательности от разных клавиш[1]. Если вы хотите, чтобы ваша программа более-менее сносно работала, за этим придётся следить, потому что если вы начнёте путать символы из последовательностей с реально вводимыми пользователем, то пользователь на вас обидится.

В редактировании строки один из хитрых моментов наступает, когда пользователь добавляет и удаляет символы не в конце, а где-нибудь в середине; тогда остаток строки приходится «перерисовывать», а при удалении символов ещё и следить, чтобы после строки было выдано нужное количество пробелов, чтобы затереть символы, «уехавшие» влево. Просто будьте к этому готовы, всё это не так сложно.

5.37. Процедура «демонизации» подробно описана в §5.4.5, но на всякий случай напомним, что полноценный демон обязан закрыть свои стандартные потоки, открыв вместо них `/dev/null`, уйти в отдельный сеанс, отказавшись от управляющего терминала, затем перестать быть лидером этого сеанса, и обязательно сменить текущий каталог на корневой.

6.02. Формально говоря, для этого нужно написать столько единиц, какова длина префикса, затем дописать столько нулей, чтобы всего разрядов стало 32, поделить их на байты по 8 бит и каждый

[1] Если никак не можете понять, о чём идёт речь, обратите внимание на то, каков третий символ последовательности — буква это или цифра.

байт перевести в десятичное представление. В действительности сисадмины так никогда не делают, они просто помнят, во-первых, что префиксы `/8`, `/16` и `/24` заканчиваются на границе байта (так что запись маски состоит из чисел `255` и `0`), а во-вторых, как правило, помнят числа, получающиеся при вычитании степени двойки из `256`: `254`, `252`, `248`, `240`, `224`, `192` и `128`. Это позволяет переводить длину префикса в форму маски подсети и обратно в уме, не прибегая к разложению на биты.

6.04. Если не получается иначе, можно начать с «подсети» из одного хоста (префикс `/32`), адрес сети будет совпадать с исходным; затем записать все биты адреса в двоичной форме и по мере уменьшения длины префикса принудительно сбрасывать в ноль по одному биту справа, а результат переводить обратно в десятичную систему. Обратите внимание, что на каждом шаге меняется только один байт, так что переводить в десятичку нужно только его, остальные байты можно каждый раз брать из предыдущего адреса.

6.09. Достаточно вспомнить, что стандартный ввод имеет дескриптор `1`; далее вызываем `select` и смотрим, что произойдёт раньше — пользовательский ввод или истечение отведённого времени. Прочитать имя можно обычным `fgets`. Формально это не очень хорошо, ведь пользователь при желании может ввести только часть имени, нажав `Ctrl-D` посреди строки, и дальше уже думать над остатком имени сколько ему в голову взбредёт — наш `fgets` будет его прилежно ждать; но в реальной жизни пользователи так не делают, если же кто и сделает — будет сам виноват; защититься от действий пользователя здесь всё равно нельзя, он ведь может вообще приостановить программу, нажав `Ctrl-Z`.

6.10. Для начала стоит отказаться от ввода средствами библиотеки и читать стандартный поток вызовом `read`; после этого уже ничто не помешает рассматривать поступление пользовательского ввода, сигналы и истечение временно́го интервала как три вида событий — именно так, как это предполагается при построении событийно-ориентированных программ с помощью `select`.

Отметим, что на самом деле ввод можно производить и высокоуровневый, но тут уже появляются определённые тонкости, бороться с которыми — дело неблагодарное, особенно в *этой* задаче.

6.11. Напомним, что при попытке открыть канал `FIFO` с любого конца, если на втором конце ещё никого нет, вызов `open` блокируется в ожидании появления партнёра. Несложно сообразить, что — если, конечно, не принять мер против этого — два экземпляра нашей программы благополучно заблокируются на попытках открыть первый из двух каналов. Решить проблему очень просто: первый из

двух вызовов `open` (но не оба!) делать с флагом `O_NONBLOCK`. Лучше, если это будет тот файл, который открывается на запись. После открытия обоих файлов следует организовать бесконечный цикл с `select`'ом, исследуя готовность к чтению тех двух потоков, из которых в программе читаются данные, то есть стандартного потока ввода (дескриптор `1`) и того файла, который в командной строке указан первым (но открывался, по-видимому, всё же вторым).

6.12. Организация работы с TCP-клиентами на основе вызова `select` подробно описана в §6.4.4. При наступлении готовности по чтению на одном из клиентских сокетов следует выполнить чтение блока данных вызовом `read` или `recv`, обязательно сохранив длину прочитанного блока (значение, возвращённое вызовом чтения) в целочисленной переменной; если это значение — `0`, сокет следует закрыть, а сеанс работы с данным клиентом считать завершённым и исключить из списка активных. Если прочитан блок ненулевой длины, его (разумеется, только ту часть буфера, которая прочитана) следует просмотреть в цикле и на каждый байт, равный символу перевода строки `'\n'`, отреагировать отправкой клиенту сообщения `"Ok\n"` длиной три символа. Готовность сокета на запись в этой задаче можно не отслеживать.

6.13. Во всех подобных задачах необходимо помнить, что потоковые сокеты гарантируют сохранение последовательности передаваемых данных, но никоим образом не границ отдельных «посылок». Данные, отправленные в сокет, на другой его конец могут прийти нарезанными в произвольную «лапшу», а могут, напротив, «склеиться»; так, если на одном конце в сокет были записаны строки `"11111111\n"`, `"22222222\n"` и `"33333333\n"`, каждая отдельным вызовом `write` или `send`, то на противоположном конце, возможно, первый вызов `read` вернёт строку `"111"`, второй — что-нибудь вроде `"11111\n22"`, третий — `"2222"`, а четвёртый — `"22\n33333333\n"` Впрочем, с тем же успехом все отправленные данные (в данном случае — все 27 байт) могут прочитаться из сокета за один `read`, если, конечно, хватит размера переданной вызову области памяти. Дело осложняется ещё и тем, что в «тепличных» условиях, в которых обычно тестируются программы, строки, набранные пользователем в клиентской программе (будь то `telnet` или любой другой TCP-клиент), обычно прочитываются каждая за один вызов, изредка «склеиваются», но практически никогда не разрезаются; это совершенно не повод расслабляться, в боевых условиях строка окажется разорванной на две порции читаемых данных в самый неподходящий момент.

Если мы рассматриваем данные, приходящие из сокета, как последовательность неких команд, мы вынуждены предполагать, что

при очередном чтении из сокета мы в любой момент можем получить не всю команду, а только некое её начало, а в более общем случае прочитанный из сокета блок может содержать остатки команды, которую мы начали читать ранее, плюс к этому несколько новых команд и начало следующей за ними. Следовательно, **необходим накопительный буфер**, свой для каждого сеанса работы с клиентом — попросту говоря, сколько есть клиентских сокетов, столько должно быть и накопительных буферов, сокет с буфером можно объединить в одну структуру. Физически накопительный буфер обычно представляет собой массив типа `char` какого-то фиксированного размера, например, 4 КВ — но во всяком случае не меньше, чем предполагаемый размер одного запроса или команды; лучше будет, чтобы в буфер у вас могло поместиться несколько запросов. К массиву должна прилагаться целочисленная переменная, хранящая объём принятых данных, в настоящее время находящихся в буфере — проще говоря, размер *занятой* части вашего массива. Например, если в этой переменной ноль — значит, буфер пуст. Чтение можно выполнять прямо в свободную часть буфера; например, если в вашей программе каждому активному сеансу соответствует структурка вроде следующей:

```
enum { buffer_size = 4096 };
struct client_session {
    int sd;                 /* дескриптор сокета */
    char buf[buffer_size];  /* буфер */
    int bd;                 /* кол-во данных в буфере */
};
```

— и вызов `select` сообщил вам о готовности клиента, на структуру которого указывает указатель `p`, то выполнить чтение из сокета можно примерно так:

```
res = read(p->sd, p->buf + p->bd, buffer_size - p->bd);
if(res == 0) {
    /* конец файла; обработка завершения сеанса */
} else {
    p->bd += res;
    /* обработка команд из буфера */
}
```

После чтения, убедившись, что ситуация «конец файла» пока не возникла, нужно отразить факт успешного чтения данных в поле, хранящем текущее количество данных в буфере, и проанализировать содержимое буфера на предмет наличия в нём законченной команды. В нашем случае достаточно просмотреть первые `bd` элементов буфера в поисках символа перевода строки. Если законченная команда в

буфере есть, её следует обработать и изъять из буфера, для чего придётся оставшиеся (после этой команды) байты буфера, если таковые есть, сдвинуть в его начало (если лень писать это вручную, можно воспользоваться функцией `memmove`), а поле количества данных (в нашем случае `bd`) на длину изъятой команды уменьшить. Учтите, что команд в буфере может быть несколько, так что последовательность анализа, выполнения и изъятия команды следует выполнять в цикле, из которого можно выйти, когда очередной законченной команды в буфере не окажется (в нашем случае — не найдётся ни одного перевода строки); сюда входит и случай, когда буфер пуст.

Для формирования текстовых ответов, отправляемых клиенту, очень удобна функция `sprintf`, см. §4.4.6.

7.03. Очевидно, что, если какое-то различие между версиями и возникнет, то только благодаря тому, что один из процессов будет прерван между этими двумя строками (в противном случае их перестановка не поменяла бы буквально ничего, их вообще можно было бы рассматривать как одно действие). Попытайтесь найти такую последовательность выполнения строк из процедур `enter_section` обоих процессов, что в итоге они окажутся в критических секциях одновременно.

8.03. Здесь стоит припомнить, что память, затребованная под кучу, никогда не возвращается операционной системе (до завершения программы).

10.01. Хранить в объекте нужно только число N; воспользуйтесь перегрузкой операции индексирования.

10.02. Храните в объекте целое число, начальное значение задавайте параметром конструктора, операция `+=` должна иметь естественный смысл.

10.03. Храните в объекте целое число, по умолчанию равное нулю, а при копировании устанавливаемое на единицу больше, чем в оригинальном объекте.

10.04. Здесь достаточно описать в приватной части класса настоящий массив; операция индексирования должна извлекать оба числа из переданного ей объекта класса `I`, уменьшать их оба на единицу и возвращать ссылку на элемент внутреннего массива с полученными индексами.

10.05. Здесь первая операция индексирования должна формировать и возвращать объект некоторого промежуточного класса, хранящего (в своих полях) указатель на объект класса `M` и значение первого индекса; для него должна быть перегружена операция индексирования, возвращающая ссылку на нужный элемент массива

в реализации класса `M` (того объекта, на который указывает сохранённый указатель). Аналогичная техника применяется в примере, разобранном в §10.4.25.

10.06. См. указания к задаче 10.05; здесь в промежуточном объекте достаточно хранить только число, послужившее аргументом для первой операции индексирования.

10.15. Поскольку тип переменных не задан, идентификатор `swap3` должен вводиться как имя шаблона функции; при использовании такого шаблона, как известно, указание конкретного типа не обязательно, поскольку компилятор выведет его из типов фактических параметров. Сами параметры должны, естественно, иметь ссылочный тип.

11.01. Решить эту задачу будет проще, если сразу же написать вспомогательную функцию, получающую *два* параметра — K и N, и строющую список чисел от K до N.

11.08. Решение этой задачи возможно благодаря ***замыканиям***, которые подробно рассмотрены в §11.1.9. Обратите внимание на пример, в котором создаются функции `seq-next` и `seq-reset`; принцип решения этой задачи тот же, только нужно не определять именованную функцию формой `defun`, как в примере, а создавать безымянную функцию — с помощью `#'(lambda` в Common Lisp, а в Scheme — вычислением lambda-формы.

11.09. Создайте локальную переменную с помощью `let`, задав ей в качестве начального значения пустой список. Безымянная функция, которую вы передадите функционалу `mapcar` (или `map`), должна, используя `setq` (соответственно `set!` для Scheme), добавить свой аргумент к списку, связанному с вашей локальной переменной.

11.12. Проще всего будет воспользоваться алгоритмом Дейкстры для перевода выражения в польскую инверсную запись, полученную запись проинтерпретировать. Алгоритм Дейкстры описан в части, посвящённой ассемблеру, см. т. 1, §3.8.4.

11.17. Проще всего выбрать в качестве базиса рекурсии случай, когда `Elem` является первым элементом `Res`. Рекурсивный переход здесь — случай, когда первые элементы `List` и `Elem` совпадают, а «хвост» `Res` получен добавлением `Elem` в «хвост» списка `List`.

11.20. В роли базиса следует выбрать пустой список, его единственная перестановка — это он сам. Для случая, когда список не пуст, один из простейших вариантов — получить перестановку его хвоста и в неё (в произвольную позицию) вставить голову, например, с помощью `ins_one` (см. задачу 11.17).

11.27. Здесь стоит припомнить, что запятая — это *конструктор кортежей*, причём кортеж из одного элемента — это то же самое,

что и просто сам этот элемент, а скобки при формировании кортежей выполняют ту же роль, что и в других выражениях — меняют последовательность применения операций.

11.32. Помните, что ваша функция должна быть функцией *одного* аргумента (в данном случае это будет количество элементов, которые нужно отбросить), а возвращать она должна *функцию*, принимающую список (исходный) и возвращающую список (уже без соответствующего количества первых элементов). Для построения объекта «функция, которая...» следует использовать лямбда-выражения.

12.07. При решении подобных задач можно усмотреть определённое неудобство в том, что Пролог невозможно заставить что-то печатать на *обратном* ходе поиска с возвратами: вывод представляет собой побочный эффект встроенных предикатов (процедур), и этот побочный эффект проявляется, когда процедура вызывается обычным способом, то есть в ходе последовательного вычисления целей в предложении. Это, на первый взгляд, мешает выдавать информацию об отмене последних рассматриваемых позиций и возврате к рассмотрению других вариантов, начинающихся с более короткого недостроенного маршрута. Проблема, однако, решается очень просто: вместе с координатами очередной рассматриваемой клетки следует выдавать *номер* этой клетки в составе текущего маршрута (например, выданная строка «`13 5 4`» может означать, что в текущем маршруте в качестве 13-го элемента рассматривается клетка с координатами 5,4). Уменьшение номера в сравнении с предыдущим выданным как раз и будет означать, что один или несколько последних элементов маршрута пришлось откатить: например, если в какой-то момент программа выдала «`25 2 7`», а сразу после этого — «`22 6 8`», это следует понимать так, что последние четыре элемента текущего маршрута пришлось выбросить, оставив только первые 21 из них, а в качестве нового элемента с номером 22 теперь рассматривается клетка с координатами 6,8.

О Т В Е Т Ы

1.01. `/opt/light/bar/f1.txt`, `../bar/f1.txt`. **1.02**. `tetris.pas`. **1.03**. `/usr/local/lib/hunter/run`. **1.04**. `/home/john/work/tetris`. **1.05**. В задаче 1.03 первая команда `cd` использует абсолютный путь для установки текущей директории, так что эффект от неё и последующих команд `cd` никак не зависит от того, какая директория была текущей до этого. **1.06**. a) владелец может читать, записывать и исполнять, остальные пользователи системы не могут ничего; b) владелец и группа (здесь и далее под «группой» понимаются пользователи, входящие в группу, которая установлена для рассматриваемого файла) могут из файла читать, больше никто ничего делать не может; c) владелец может читать и исполнять, все остальные — только исполнять; d) владелец и группа могут читать и записывать, больше никто ничего не может; e) владелец может читать и записывать, все остальные — только читать; f) все пользователи могут читать и исполнять. **1.07**. a) владелец файла может его читать, записывать в него и запускать файл на исполнение; остальные пользователи могут только запускать на исполнение, при этом запущенная (записанная в файле) программа будет выполняться с правами владельца файла; b) владелец может читать, записывать и запускать, группа — читать и запускать, все остальные — только запускать, при этом исполнение будет происходить с правами владельца и группы, установленной для файла; c) владелец и группа могут читать файл и запускать его, исполнение будет происходить с правами пользователя, запустившего файл, но с правами группы, указанной для файла (а не для пользователя, который его запустил). **1.08**. a) владелец может делать с директорией что угодно, остальные пользователи — ничего; b) владелец может всё, остальные пользователи могут воспользоваться файлом из директории в соответствии с правами на него, но только если знают его имя, поскольку читать содержимое директории (собственно говоря, имена файлов) никому, кроме владельца, не позволено; c) владелец может всё, пользователи группы могут получать листинг директории и пользоваться её содержимым в соответствии с правами на эти файлы, но не могут изменять саму директорию, то есть переименовывать и удалять файлы из неё; остальные пользователи могут пользоваться файлами из директории при условии, что знают их имена; d) владелец может всё, все остальные пользователи могут создавать в директории файлы, использовать их и даже переименовывать/удалять, но только если знают имя файла; такие права доступа обычно применяются, чтобы другие пользователи системы могли «подкладывать» в вашу директорию свои файлы, но не могли ничего сделать с файлами, которые «подложил» кто-то ещё, поскольку не знают имён таких файлов; e) владелец может всё, все остальные пользователи могут

только читать имена файлов; это бессмысленная комбинация, так как сделать с этими файлами другие пользователи ничего не смогут, даже имея права на эти файлы; следует применять либо `0700`, если вы не хотите никому позволять делать что бы то ни было с файлами в этой директории, либо `0755`, если, напротив, хотите позволить всем использовать эти файлы; f) владелец и группа могут узнавать имена файлов, переименовывать их и удалять, но не могут никак этими файлами воспользоваться; эта комбинация бессмысленна, следует применять `0770`, чтобы разрешить владельцу и группе использование файлов, их создание, удаление и переименование. **1.09**. a) владелец и группа могут создавать в директории файлы и использовать имеющиеся файлы в соответствии с правами на них, но удалять и переименовывать существующие файлы могут только их владельцы; простейшая ситуация, в которой такие права требуются — если в системе есть большой диск для хранения личных файлов пользователей, но доступ к нему следует давать не всем; при этом тех пользователей, которым «повезло», следует включить в специальную группу, которую и прописать для корневой директории диска; b) в директории кто угодно может создать файл, но группой для этого файла будет прописана та же группа, которая установлена для самой директории; доступ к файлам разрешён всем в соответствии с правами на сами файлы, но переименовать или удалить файл может только его владелец.

1.15. 15. **1.16**. $4 + ((4 \cdot 3)/2 + 4) + (4 \cdot 3 \cdot 2/6 + 4 \cdot 3 + 4) = 34$. **1.17**. 720. **1.18**. 60. **1.19**. 9; 19. **1.20**. $4^8 = 65536$. **1.21**. $32 - 1 - 1 - 5 = 25$; $C_5^2 + C_5^3 + C_5^4 = 10 + 10 + 5 = 25$. **1.22**. $64 \cdot 14/2 = 448$. **1.24**. 5040. **1.23**. $5 \cdot 4 \cdot 3/3/2 = 10$. **1.25**. 420. **1.26**. 625. **1.27**. 26712. **1.28**. 301476; для любителей нумерологии отметим, что C_{776}^2 выглядит довольно красиво: 300700. В общем случае, внимательно посмотрев на первые три колонки треугольника Паскаля, можно заметить, что $C_n^2 \equiv 1+2+\ldots+(n-1)$, а такая сумма, как известно, составляет $\frac{n(n-1)}{2}$. Последнее соотношение можно получить непосредственно из формулы для C_n^k, но это не так интересно.

1.29. a) 1011011; b) 10010101; c) 101011000; d) 1010010100; e) 1101110001; f) 10001001100. **1.30**. a) 1; b) 10; c) 1111; d) 100001; e) 1000011; f) 1111110; g) 11111100. **1.31**. a) 89; b) 106; c) 631; d) 786; e) 3686; f) 2577. **1.32**. 1000110000000011. **1.33**. $15 + 7 + 3 + 1 = 26$. **1.34**. a) 101111; b) 100010100011; c) 1110111011101110111; d) 10010001101001010101111001101; e) 101101001011010. **1.35**. a) 10111; b) 100101011; c) 11011011011011; d) 1010011101110111; e) 110011110011. **1.36**. a) 35, 1D; b) 1777, 3ff; c) 7640, fa0; d) 10001, 1001. **1.37**. 22220120001101122202112; можно: каждая девятичная цифра соответствует ровно двум

троичным. **1.38**. a) $\frac{7}{11}$; b) $3+\frac{1}{2}+\frac{3}{110}=3\frac{58}{110}=\frac{388}{110}=\frac{183}{55}$; c) $7{,}23+\frac{7}{2700}=\frac{723}{100}+\frac{7}{2700}=\frac{19528}{2700}=\frac{4882}{675}$. **1.39**. a) $\frac{3}{5}$; b) $\frac{6}{7}$; c) $3\frac{3}{14}$; d) $\frac{7}{10}$, e) $\frac{1}{13}$. **1.40**. a) 0,(0110); b) 0,0(01); c) 0,(101); d) 0,(110001); e) 0,(1101000101). **1.41**. Число 333 в двоичной системе записывается как 101001101, так что $333=(((1\cdot2\cdot2+1)\cdot2\cdot2\cdot2+1)\cdot2+1)\cdot2\cdot2+1$; всего здесь как раз 13 операндов, то есть нужно отмерить метр, удвоить, удвоить, добавить метр, удвоить, удвоить, удвоить, добавить метр, удвоить, добавить метр, удвоить, удвоить, добавить метр – получится 333. **1.42**. Следует выразить нужное количество верёвки в виде целого числа (единица измерения — метр) и перевести это число в двоичное представление; далее, просматривая число слева направо, добавлять метр верёвки для каждой увиденной единички и удваивать отмеренную верёвку при каждом переходе к следующей цифре.

1.43. a) $x\vee y$; b) y; c) $x\vee y$. **1.44**. a) $x\vee y$; b) $x\overline{y}$; c) $x\vee\overline{y}$; d) xy; e) 1; f) xy; g) $\overline{x}\vee y$; h) y; i) 1; j) $xy\vee\overline{xy}$. **1.45**. a) 1; b) $x\overline{y}$; c) 0; d) $x\vee\overline{y}$. **1.46**. $\overline{x}=x\downarrow x$; $x\vee y=(x\downarrow y)\downarrow(x\downarrow y)$; $1=\overline{x}\vee x=(x\downarrow x\downarrow x)\downarrow(x\downarrow x\downarrow x)$, но это есть $\overline{x\downarrow x\downarrow x}$, так что $0=\overline{1}=x\downarrow x\downarrow x$; $xy=\overline{\overline{x}\vee\overline{y}}=\ldots$ и т. д. **1.47**. $\overline{x}=x|x$; $xy=\overline{x|y}=(x|y)|(x|y)$; $0=\overline{x}x=(x|x|x)|(x|x|x)$, но поскольку это есть $\overline{x|x|x}$, то $1=\overline{0}=(x|x|x)$; и т. д. **1.48**. a) $x\equiv y$; b) $x\,\&\,y$; c) $x>y$; d) $x\downarrow y$; e) $x<y$. **1.49**. a) $x\overline{y}z\vee xy\overline{z}$; b) $\overline{x}y\overline{z}\vee\overline{x}yz\vee xyz$. **1.50**. Каждая из систем имеет всего два решения; для первой это наборы $(0,1,0,1,\ldots,0,1)$ и $(1,0,1,0,\ldots,1,0)$, для второй — $(0,\ldots,0)$ и $(1,\ldots,1)$. **1.51**. 10; $256-2-6-3\cdot10=218$.

1.52. a) $\log_2\frac{27}{8}=\log_2 27-3=3\log_2 3-3$ бит; b) $4-3\log_2 3$ бит; c) $\log_2 6^3=3\log_2 6=3+\log_2 3$ бит; d) $\log_2\frac{6^3}{3}=\log_2 72=3+2\log_2 3$ бит. **1.53**. 2 бита. **1.54**. 1 бит. **1.55**. 12. **1.56**. 1 бит. **1.57**. $\log_2(8\cdot7\cdot6)=4+\log_2 21$ бит.

2.01. a) `5`, `integer`; b) `32`, `integer`; c) `6`, `integer`; d) `3.0`, `real`; e) `2.5`, `real`; f) `3`, `integer`; g) `0`, `integer`; h) `5`, `integer`; i) `2`, `integer`; j) `4`, `integer`; k) `5.0`, `real`; l) `true`, `boolean`; m) `true`, `boolean`; n) `false`, `boolean`; **2.02**. В обеих переменных будет число `5`. **2.03**. `45`. **2.04**. `990`. **2.05**. a) 10 строк `Hello`; b) 19 строк `Good Bye`; c) 12 строк `abrakadabra`; d) 15 строк `foobar`; e) 10 строк `Johnny be good!`; f) ничего. **2.06**. a) 11 строк `Hello`; b) 20 строк `Good Bye`; c) 12 строк `abrakadabra`; d) 1 строка `abcdefgh`. **2.07**. В примерах 2.05, a) и 2.06, a) переменная `i` на каждой итерации цикла равна тому, *сколько раз напечатана строка*; в примере 2.06, c) значение переменной соответствует тому, сколько раз *осталось* напечатать строку; в этих трёх примерах переменная `i` — действительно *счётчик*. В остальных примерах ничего вразумительного про текущее значение `i` сказать невозможно, так что называть её счётчиком, по-видимому, не

Ответ к задаче 3.04

флаги z s o c	пример al	bl	комментарий
0 0 0 0	01	01	неотрицательные с суммой не больше 127
0 0 0 1	FF	02	беззнаковые, дающие в сумме от 257 до 383
0 0 1 0			*невозможно*
0 0 1 1	8F	F0	отрицательные с суммой от -129 до -255
0 1 0 0	F0	01	отрицательное и мéньшее по модулю неотриц.
0 1 0 1	FF	81	беззнаковые, дающие в сумме 384 и больше
0 1 1 0	5F	6E	положительные с суммой от 128 до 254
0 1 1 1			*невозможно*
1 0 0 0	00	00	сумма двух нулей
1 0 0 1	FE	02	сумма полож. и отриц., равных по модулю
1 0 1 0			*невозможно*
1 0 1 1	80	80	128 + 128
1 1 * *			*невозможно*

стоит. **2.08**. Пример 2.05, e) переписать через `for` нельзя, поскольку переменная цикла изменяется с шагом, отличным от единицы. Во всех остальных примерах в условии применяется строгое неравенство; из-за этого конечное значение, указываемое в цикле `for`, будет отличаться на единичку от значений из исходных примеров. Так, пример 2.05, a) превратится в следующий фрагмент:

```
for i := 0 to 9 do
    writeln('Hello');
```

2.09. a) `66`; b) `111`; c) `45`; d) `-1`; e) `12`; f) `-178`; g) `128`; h) `28`; i) `15`; j) `30`; k) `5`; l) `-8`. **2.10**. `$ab9100`, `$9134`.

2.38. `p^[1]^`, `p^[2]^` и `p^[3]^`. **2.39**. `p^.g^.a`, `p^.g^.b`, `p^.h^.a`, `p^.h^.b`, `p^.x`.

3.01. a) $4f4d58_{16} = 5197144$; b) $996663_{16} = 10053219$; c) $db5bb6_{16} = 14375862$; d) $7ff55c_{16} = 8385884$; e) $55c95f_{16} = 5622111$; f) $5e402e_{16} = 6176814$. **3.02**. a) `ff`; `SF`; b) `00`; `ZF` и `CF`; c) `3d`; никакие; d) `1e`; `CF`; e) `81`; `SF` и `OF`; f) `79`; `OF` и `CF`. **3.03**. a) `fe`; `SF` и `CF`; b) `2b`; `CF`; c) `ff`; `SF` и `CF`; d) `11`; никакие; e) `70`; `OF`; f) `f4`; `SF`, `OF` и `CF`. **3.04**. См. таблицу. **3.05**. Установленный `OF` при остальных сброшенных можно получить, вычитая из отрицательного числа такое положительное, чтобы результат оказался меньше 128, т. е. его нельзя было представить восьмибитным знаковым целым (см. задачу **3.03**, пункт e); установленные одновременно `SF`, `OF` и `CF` получаются при вычитании из достаточно большого положительного такого отрицательного, чтобы

результат превысил 127 (см. там же пункт f); получить одновременно установленные ZF и OF при сброшенном CF операциями сложения и вычитания невозможно, как и установленные одновременно ZF и CF. **3.06**. a) $44448; b) $12301230; c) $0defcdefc; d) $23456780; e) $01234567; f) $3c00; g) $9abc; h) $0ffff9abc; i) $4567; j) $0f10045; k) $210798fe; l) $0ffffff00.

3.28.
```
%macro manynums 3
  %assign num %1
  %rep %3

    dd num

  %assign num num + %2
  %endrep
%endmacro
```

3.29.
```
%macro jmpseq 1-*
  %assign num 1
  %rep %0

    cmp eax, num
    je near %1

  %assign num num + 1
  %rotate 1
  %endrep
%endmacro
```

4.01. a) 53 (код цифры 5); b) 100 (код буквы d); c) 2; d) 320; e) 40; f) 3; g) 2; h) 0; i) 0; j) 1; k) 2; l) 120; m) -36; n) 46; o) 26; p) 1; q) 0; r) 1; s) 23; t) 240. **4.02**. 261. **4.03**. 493:␣␣␣493:493␣␣␣:000493:␣␣00493:1ed:1ED. **4.04**. a) Hello; b) ␣␣␣Hello; c) Hello␣␣␣; d) Hell; e) ␣␣Hell.

4.05.
```
int get_and_zero(int *p)
{
    int t = *p;
    *p = 0;
    return t;
}
```
Вызов функции будет выглядеть так: `a = get_and_zero(&b);`.

4.07.
```
int count_spaces(const char *s)
{
    int t = *p;
    p = 0;
    return t;
}
```

4.08. Рекурсивное решение может выглядеть, например, так:
```
int count_spaces(const char *s)
{
    return *s ? (*s == ' ' ? 1 : 0) + count_spaces(s + 1) : 0;
}
```
Основной его недостаток в том, что для анализа каждого символа строки производится вызов функции и, как следствие, строится стековый фрейм, имеющий отнюдь не нулевую длину. Достаточно длинная (порядка нескольких сотен килобайт) строка обрушит вашу программу по переполнению стека.

4.15. Оба выражения имеют тип `double`, значение первого — `2.5`, второго — `3.5`. **4.16**. Тип `char`, значение 65 (код буквы `A`). **4.28**. a) `int (*p)[12];`; b) `double (*p)[10][2];`; c) `char (*p)[5];`; d) `char *(*p)[5];`; e) `struct item *(*p)[3];`; f) `struct item **p;`. **4.29**. a) `void (*p)();` b) `int (*p)(int);` c) `void *(*p)(int);` d) `double (*p)(int, const char*);` e) `void (*p)(double (*)[3]);` f) `double (*(*p)(int, double (*)[3]))[3]`. **4.30**. `int (*lessfunp)(const char*, const char*);`.

4.31. a) это заголовок (прототип) функции: `f` *есть функция, принимающая параметр типа* `int` *и возвращающая адрес массива из* `10` *указателей на* `int`;

```
typedef int *intptr;
typedef intptr (*int10ptr)[10];
int10ptr f(int);
```

b) это описание массива: `fpvec` *есть массив из* `15` *указателей на функции, принимающие три параметра: типа* `int`, *типа «указатель на функцию, принимающую* `int` *и* `void*` *и не возвращающую значения», типа* `void*`; *и не возвращающие значений*;

```
typedef void (*ivfptr)(int, void*);
typedef void (*f3ptr)(int, ivfptr, void*);
f3ptr fpvec[15];
```

c) это описание переменной — указателя на функцию: `repfptr` *есть указатель на функцию, принимающую один параметр типа «указатель на функцию, принимающую* `double` *и возвращающую* `double`», *и возвращающую адрес функции, принимающей* `double` *и возвращающей* `double`;

```
typedef double (*d2d_fptr)(double);
d2d_fptr (*repfptr)(d2d_fptr);
```

d) это описание переменной — указателя на массив: `fvecpos` *есть указатель на массив из четырёх указателей на функции, которые принимают параметр типа «указатель на функцию, принимающую адрес* `double` *и не возвращающую значения», а возвращают* `int`;

```
typedef void (*fdpv_ptr)(double*);
typedef int (*f2i_fptr)(fdpv_ptr);
f2i_fptr (*fvecpos)[4];
```

4.32.

```
double (*(*set_sr_func(
        int num,
        double (*(*func)(double(*)[3], int, double))[3]
    ))(double(*)[3], int, double))[3];
```

5.10. Возможны следующие четыре варианта, хотя некоторые из них маловероятны:

```
start       start       start       start
parent      child       parent      child
finish      finish      child       parent
child       parent      finish      finish
finish      finish      finish      finish
```

5.11. Указатель `status` не инициализирован, то есть вместо адреса переменной типа `int` он содержит случайный мусор. Единственный параметр вызова `wait` должен представлять собой адрес переменной типа `int`, куда ядру надлежит записать информацию об обстоятельствах завершения порождённого процесса, которого вызов «дождался»; и *куда*, спрашивается, ядру записывать эту информацию, если `status` «указывает», как говорят программисты, «куда-то в Африку»?

Если бы функция `wait` была библиотечной, программа, скорее всего, «упала» бы при таком обращении к ней, но поскольку `wait` — это системный вызов, никакой видимой аварии не произойдёт: ядро проверит валидность переданного ему адреса, убедится, что адрес невалиден, и немедленно вернёт управление вызвавшему процессу; сам вызов при этом вернёт сакраментальную `-1`, а значение `errno` будет равно константе `EFAULT`. Поскольку в программе всё это не проверяется (значение, возвращаемое вызовом `wait`, игнорируется), она об этом не узнает. При этом того, что от него ожидалось, `wait` не сделает — он не станет ни дожидаться завершения какого-нибудь из порождённых процессов, ни убирать из системы получившегося зомби.

Ещё «интереснее» получится, если в указателе `status` случайно окажется валидный адрес некоторой области памяти. В этом случае фрагмент программы, содержащий вызов `wait`, будет работать в точности так, как от него ожидалось, но где-то окажутся испорчены совершенно не относящиеся к делу четыре байта. Как и когда это проявится — предугадать невозможно, как и найти в программе ошибку такого рода.

Исправить «косяк» очень просто: переменную `status` следует описывать не как указатель, а как обычную переменную типа `int` (без всякой звёздочки!), ну а вызов делать так: `wait(&status)`.

5.14. Например, так:

```
do {
    p = wait4(-1, NULL, WNOHANG, NULL);
} while(p > 0);
```

6.01. В «ip-адресах» `198.260.101.15` и `100.200.300.400` присутствуют «байты», превышающие `255`, так что это на самом деле никакие не ip-адреса. С остальными приведёнными адресами всё в порядке; хотя некоторые из них относятся к специальным областям и не могут использоваться обычным способом, они всё же являются ip-адресами. **6.02**. a) `255.255.255.0`; b) `255.255.0.0`; c) `255.0.0.0`; d) `255.255.255.255`; e) `255.255.255.252`; f) `255.255.255.128`; g) `255.255.254.0`; h) `255.255.240.0`. **6.03**. `/12`. **6.04**. `195.42.170.129/32`, `195.42.170.128/31`, `.../25`, `195.42.170.0/24`, `/23`, `195.42.168.0/22`, `/21`, `195.42.160.0/20`, `/19`, `195.42.128.0/18`, `/17`, `195.42.0.0/16`, `/15`, `195.40.0.0/14`, `/13`, `195.32.0.0/12`, `/11`, `195.0.0.0/10`, `../8`, `194.0.0.0/7`, `192.0.0.0/6`, `../2`, `128.0.0.0/1`.

7.01. `0`, `100`, `200`, `300`, `-100`, `-200`, `-300`. **7.02**. `150`, `225`, `225`; `175`, `175`, `250`; `150`, `150`, `250`; `150`, `250`, `250`. **7.03**. Конечно, сломается. В самом деле, пусть первый процесс выполнил первую строку процедуры `enter_section` в её «новом варианте» (`who_waits = 0;`), после чего у него истёк квант времени. Пока первый процесс находится в очереди на выполнение, второй процесс добирается до своей процедуры `enter_section`, проходит её всю (при этом цикл ожидания, как можно легко заметить, выполняться не будет, ведь первый процесс не успел занести единицу в `interested[0]`) и начинает выполнение критической секции, где у него, в свою очередь, истекает квант времени. Между тем первый процесс получает очередной квант, проходит оставшуюся часть процедуры, и здесь цикл ожидания тоже не выполняется, ведь в переменную `who_waits` второй процесс уже занёс единицу. В результате оба процесса оказываются в своих критических секциях одновременно.

8.01. три: `read`, `write` и `_exit`. **8.02**. `_exit` для выхода. **8.03**. `free`. **8.04**. `malloc`. **8.05**. 2048 байт (ячеек). **8.06**. 32768 (2^{15}) байт. **8.07**. 2048 байт.

10.07.
```
sun
venus 50
venus 800
moon
earth 850
venus 1000
moon
moon
1856 850
moon
moon
```

10.10.
```
first
third
result = (14 ; 3)
fourth
second
```

10.13.
```
horse
wolf 60
monkey
```

10.15.
```
template <class T>
void swap3(T &x, T &y, T &z)
{
    T t;
    t = x;
    x = y;
    y = z;
    z = t;
}
```

10.16.
```
template <class T>
T get_and_zero(T& m)
{
    T t(m);
    m = 0;
    return t;
}
```

11.17.
```
ins_one(L, E, [E|L]).
ins_one([H|L], X, [H|R]) :- ins_one(L, X, R).
```

11.26. `list char # char # bool # num`.

11.27. Конструктор кортежа (запятая) — правоассоциативен, т. е. работает справа налево; иначе говоря, $(x,\ y,\ z,\ t)$ есть то же самое, что $(x,\ (y,\ (z,\ t)))$. Впрочем, вложенные кортежи, подобные тем, что возникли в ответ на третье из введённых выражений, никакого практического смысла не имеют и могут привести только к возникновению лишней путаницы; в приведённом примере мы загнали интерпретатор в угол, но лучше так не делать.

11.28. `num # num # num -> list(num)`.

11.29. Например, так:

```
dec MakeSeq : num # num # num -> list(num);
--- MakeSeq(x, y, d) <=
        if x > y then [] else x :: MakeSeq(x+d, y, d);
```

11.30.

```
dec Twist : list(alpha) # list(alpha) -> list(alpha);
--- Twist(_, []) <= [];
--- Twist([], _) <= [];
--- Twist(x :: t1, y :: t2) <= x :: y :: Twist(t1, t2);
```

11.31. `[1, 4, 7, 10, 13, 16, 19, 22, 25, 28]`; список `lst` — это бесконечная (!) арифметическая прогрессия с первым элементом `1` и разностью `3`.

11.32. Здесь возможны варианты, например:

```
dec DropElC : num -> list(alpha) -> list(alpha);
--- DropElC 0 <= \ x => x;
--- DropElC k <= \ [] => [] | (_ :: tail) => DropElC (k-1) tail;
```

(напомним, что `\` в Хоупе означает то же, что и слово `lambda`).

ГЛАВНЫЕ СПОНСОРЫ ПРОЕКТА

список наиболее крупных пожертвований

I: 214999, **Nikolay Ksenev**

II: 65536, **unDEFER**

III: 60000, *анонимно*

IV: 53500, *анонимно*

V: 45763, *анонимно*

VI: 41500, *анонимно*

VII: 32767, **Дмитрий Нурисламов**

VIII: 29855, **Антон Хван**

IX: 29592, *анонимно*

X: 25600, **Ковригин Дмитрий Анатольевич**

XI: 25000, **sadsnake**

XII: 24344, *анонимно*

XIII: 21600, *анонимно*

XIV: 21048, **os80**

XV: 20079, *анонимно*

XVI: 19901, **Заворин Александр**

XVII: 19723, **Максим Филиппов**

XVIII: 18712, **Шер Арсений Владимирович**

XIX: 16384, *анонимно*

XX: 16000, **Спиридонов Сергей Вячеславович**

XXI: 15001, **Аня «canja» Ф.**

XXII: 15000, *анонимно*

http://www.stolyarov.info

Учебное издание
СТОЛЯРОВ Андрей Викторович

ПРОГРАММИРОВАНИЕ: ВВЕДЕНИЕ В ПРОФЕССИЮ
ЗАДАЧИ И ЭТЮДЫ

Рисунок и дизайн обложки *Елены Доменновой*

Напечатано с готового оригинал-макета

Подписано в печать 11.01.2022 г.
Формат 60x90 1/16. Усл.печ.л. 9,75. Тираж 500 (1–300) экз. Изд.№ 001.

Издательство ООО "МАКС Пресс"
Лицензия ИД № 00510 от 01.12.99 г.

119992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891

Отпечатано в полном соотетствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
115201, г. Москва, ул. Котляковская, д. 3, стр. 13

ISBN 978-5-317-06732-8

http://www.stolyarov.info