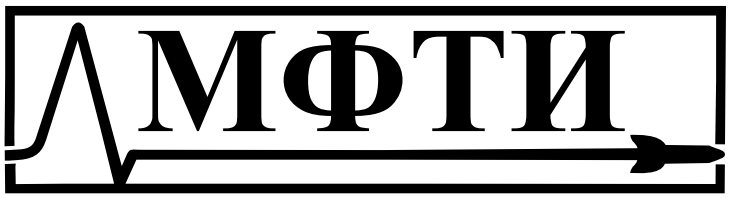
Лекции «Алгоритмы: построение, анализ и реализация на языке программирования Си»

Московский физико-технический институт
http://www.mipt.ru

кафедра информатики
© Ворожцов А.В., Винокуров Н.А.



Москва, 2007

Оглавление

Предисловие

Содержание курса

Данный курс предназначен для студентов младших курсов технических вузов. Он включает как теоретические так и практические аспекты программирования, знакомит с избранными технологиями и инструментами, конкретными задачами, содержит тщательно подобранные учебные материалы по языку Си.

Курс построен как несколько целостных, связанных друг с другом «этюдов», которые посвящены следующим темам:

- вычислимость и невычислимость, абстрактные исполнители;
- сложность вычислений;
- язык Си;
- алгоритмы, методы построения и анализа алгоритмов;
- реализация структур данных и алгоритмов на Си;
- сравнительный анализ парадигм программирования;

Кроме того, затрагиваются следующие практические аспекты:

- принципы разработки надёжных программ;
- визуализация и анализ данных;
- тестирование и верификация программных средств;
- коллективная разработка программных средств.

Первая часть «Алгоритмы, исполнители и формальные языки» является вводной. В ней даётся формальное определение алгоритма, описываются абстрактные исполнители, обсуждается понятие вычислимости. Данная часть позволяет чётко обозначить область задач, решаемых программистами, получить полезный опыт создания формальных описаний логики работы в рамках различных абстрактных исполнителей. В этой части вводится расширенный вариант языка BNF, который активно используется в курсе для описания синтаксиса языка Си и различных грамматик.

Вторая часть «Язык программирования Си» посвящена освоению языка Си, знакомству со средой программирования, приобретению опыта решения простейших программистских задач. Она содержит множество примеров программного кода, после освоения которых, несложно писать код самому. В семинаре 7 сформулировано более 40 задач, которые могут быть использованы для проведения самостоятельных работ и контрольных по технике программирования на Си.

Третья часть «Алгоритмы и структуры данных» содержит материал по классическим алгоритмам и структурам данных: рекурсия и итерации, алгоритмы сортировки, переборные алгоритмы, деревья поиска, хэштаблицы, двоичная куча, «жадные алгоритмы», алгоритмы на графах, формальные грамматики.

Четвёртая часть состоит из разноплановых дополнительных лекций. Она включает две лекции, посвящённые парадигмам и языкам программирования. Другие лекции рассказывают про теорию сложности вычислений, алгебру булевых функций, задачу программирования виртуального исполнителя, методы и инструменты синтаксического разбора, теорему Гёделя.

Особенное внимание в данном курсе уделяется абстрагированию и модульности, понятиям «интерфейс», «реализация», «библиотека», «повторное использование кода», «виртуализация».

В курсе затрагиваются различные важные концепции программирования. Методам тестирования, верификации и отладке программ уделяется внимание в лекциях и семинарах по структурам данных. В дополнительной лекции 26 «Виртуализация исполнителей. Архитектура компьютера» рассказывается про принципы разработки качественного программного обеспечения. Методология повторного использования кода проходит красной нитью по всем лекциям и семинарам.

Педагогика учебника

Учить информатике можно по разному. Этот курс по-своему уникален:

- Главной компонентой курса являются не алгоритмы и язык программирования, а абстрагирование, системной подход к решению задач, методы построения алгоритмов, принципы разработки качественного кода.

- Курс покрывает достаточно большой спектр вопросов, как по сложности, так и по темам. Он, с одной стороны, является стартовой точкой для начинающего программиста (будующего программиста-аналитика), который может познакомиться с различными аспектами теории и практики программирования, наметить себе путь дальнейшего развития. В то же время данные лекции могут стать интересным чтением и для профессионала, который, пропустив простые задачи и темы, уделит внимание сложным моментам.

- Курс в бóльшей степени ориентирован на создание полезных ассоциативных связей и формирование понятийного аппарата, нежели на создание формализованного полного описания чётко очерченной части науки программирования. Авторы считают, что вводный курс в информатику в принципе должен иметь обзорный (метаинформационный) характер, то есть намечать важные вехи (проблемы, общие концепции, развивающиеся технологии и языки программирования), которые заинтересованный читатель сможет изучить глубже самостоятельно. Тем не менее, в курсе многие аспекты рассмотрены достаточно детально.

- Центральным средством программирования выбран язык Си, а не Си++, Java, Python, и др. Язык Си позволяет на первом этапе достигать большего понимания происходящего в компьютере и приучает писать более качественный код. Язык Си++ слишком сложен и слишком много скрывает от программиста, чтобы брать его как первый язык программирования. Но профессиональному программисту крайне важно познакомится с

этим языком и библиотекой шаблонов STL. Программами на языке Си иллюстрируется большое количество программистских и алгоритмических задач. Читатель может по примерам достаточно полно освоить язык Си и научится самостоятельно писать программы. Конечно, полезно при этом иметь полный справочник языка Си и книги [3, 2].

О преподавании информатики

Содержание вводного курса информатики долго было и остаётся спорным вопросом. По объему знаний информатика стала уже превосходить многие классические науки. И это связано не только с большим числом компьютерных технологий, но и с богатой фундаментальной составляющей информатики, — даже она не помещается в рамки двухгодичного институтского курса. Таксономия теоретических знаний по информатике (Computing Curricula 2005 [**?**]) представляет собой 40 разделов, непосредственно относящихся к вычислениям и и разработке программного обеспечения, и еще 22 раздела, связанных с математическими основами, электроникой, проектированием оборудования, системами управлением и анализом. Многие из этих областей сами по себе могут быть представлены в виде плотного двугодичного курса.

Информатика заняла свою нишу и стала наукой, решающей задачу обработки больших объёмов информации и создания многокомпонентных систем. Она стала междисциплинарной и выработала общие принципы, касающиеся проектирования и разработки сложных, надёжных, распределённых и др. систем. Также, в силу своей специфики, информатика первая смогла выработать принципы коллективной работы в больших проектах.

В IT-индустрии важную роль играют точность, абстрагирование, формальные языки, понятийный аппарат, следование стилю и соглашениям. И в результате информатика стала интересным соединением математики, лингвистики, философии, системного анализа и теории управления.

Выпишем несколько наиболее важных и фундаментальных разделов информатики:

Programming Fundamentals	Основания программирования
Algorithms and Complexity	Теория алгоритмов и теория сложности
Theory of Programming Languages	Теория языков программирования
Integrative Programming	Интеграционное программирование. Теория и практика модульного построения программного обеспечения
Computer Architecture and Organization	Компьютерная архитектура, принципы разработки оборудования
Operating Systems Principles & Design	Основы и принципы разработки операционных систем
Intelligent Systems (AI)	Интеллектуальные системы и системы основанные на знаниях
Information Management (Database) Theory	Теория баз данных. Методы обработки данных
Scientific computing (Numerical methods)	Численные методы
Information Systems Development	Методы разработки информационных систем
Software Modeling and Analysis /Design/ Verification and Validation / Evolution (maintenance) /Quality	Разработка программного обеспечения: проектирование, моделирование и анализ, тестирование и верификация, жизненный цикл разработки, методы контроля и управления разработкой
Distributed Systems	Распределённые системы. Методы масштабирования программных средств. Теоретические и технологические аспекты распределённого хранения данных и параллельных вычислений
Security: issues and principles / implementation and management	Защита информации: теория и принципы, реализация и практическое использование
Systems administration	Системное администрирование
Systems integration	Системный анализ и интеграция

Одна книга может претендовать лишь на ознакомление читателя с отдельными темами избранных разделов. В частности, данный курс лекций относится к первым трём разделам и, конечно, не охватывает ни один из них целиком[1].

[1]Опишем примерное содержание трёх первых разделов приведённой классификации.

Раздел «Основания программирования» содержит базовые концепций программирования (исполнители, исполнение, типы данных, функции, массивы, файлы). Он включает в себя методы отладки и тестирования программных средств, принципы объектно-ориентированного программирования (инкапсуляция, наследование, переопределение методов), сравнительный анализ парадигм программирования, а также логики и исчисления, лежащие в основе различных парадигмами программирования.

«Теория алгоритмов и теория сложности» является необъятной темой, включающей большое количество математических основ (см., например, [?], [19]) и требующее много часов практических занятий. Это анализ и построение алгоритмов [1], методы реализации алгоритмов на языках программирования, оценка сложности алгоритмических задач, эвристические алгоритмы решения NP-сложных и плохо формализуемых задач. Отдельный подраздел «Геометрические алгоритмы», представленный в книге [16] «Computational Geometry» издательства Springer с трудом умещается в годовой курс.

Из-за большого объема информации в Computer Science особенно хорошо чувствуется необходимость в принципиально новом подходе к образованию. Основной тезис этого подхода звучит так:

Образование должно перейти на мета-уровень: ориентироваться не на задачи, а на методы решения задач, не на избранные технологии, а на концепции, на которых технологии базируются. Задача учителя – сделать обзор, описать суть и мотивировать, задача ученика – заинтересоваться и самостоятельно изучить избранные темы более глубоко.

В принципе, это простой и хорошо известный принцип фундаментального образования вообще. Не следует возводить его в догму. Необходим разумный баланс между теоретическим и практическим материалом. Хороший учитель хорошо чувствует этот баланс и старается помимо разбора конкретных задач и практики программирования дать ученикам некоторые общие концепции. При этом он старается не увлекаться теорией и дать ученикам самостоятельно «поработать с материей». Это позволяет закрепить материал, увидеть сильные и слабые стороны теории.

Три слова – Простота, Ясность, Общность – написанные на обложке книги «Практика программирования» Брайана Кернигана и Роба Пайка, пожалуй, наиболее ярко отражают то, какими должны быть[2] современные лекционные материалы:

- **Простота.** Учебный курс должен быть легко понимаемым. Если учесть, что лекционный материал, должен восприниматься на слух, то требование простоты становится вполне обоснованным. Процент понимания среднего ученика должен быть более 50%.

- **Ясность и яркость.** Современный курс уже не имеет права быть не ярким. Все примеры и теории должны быть иллюстрированы примерами из жизни, образными аналогиями, схемами, картинками. Яркость материала повышает усвояемость, и способность к использованию полученных знаний (knowledge usability).

- **Общность.** Курс — это не энциклопедический материал, формально и по порядку излагающий основы теории, какие-либо методы программирования или спецификацию языка программирования. Курс должен представлять собой красочную крупномасштабную карту, лишь избранные кусочки которой показаны с большим увеличением. Курс должен затрагивать множество общих вопросов, касающихся логики, философии и фундаментальных концепций. Детальное же изучение избранных тем должно осуществляться в рамках самостоятельной работы учащихся со специализированными учебниками (документациями, web-ресурсами). Общность курса необходима для того, чтобы сделать

Раздел «Теория языков программирования» один из сложнейших разделов науки программирования. Он включает следующие вопросы: какие языки существуют и чем они отличаются, какие концепции в них заложены, формальные языки и грамматики, метаязыки, методы компиляции и оптимизации кода. Задачи связанные с параллельными вычислениями также в значительной степени связаны с языками и специальными методами компиляции под параллельные архитектуры. В российских вузах этот курс частично покрывается курсом «Теория и реализация языков программирования», где изучаются контекстно свободные грамматики, и системы синтаксического разбора подобные lex & bison. Но методы компиляции часто остаются за рамками институтского курса.

[2]Данный курс, скорее всего, не удовлетворяет этим постоянно растущим требованиям к учебным материалам. Но полезно эти требования сформулировать.

образование более индивидуальным. Имея крупномасштабную карту, ученик сможет сориентироваться и углубится в наиболее интересные ему темы и изучать их с удобной для него скоростью.

При движении в сторону простоты, яркости и общности материала нередко приходится жертвовать полнотой и формальной строгостью. Иногда приходится жертвовать целыми темами, исключать из курса большие и важные классы задач и методы решения этих задач, возможно, лишь, коротко аннотируя их и указывая ссылки на источники, в которых заинтересованные ученики могут найти подробные описания.

Современный специалист по информационным технологиям должен быть очень самостоятельным: он должен уметь самостоятельно формулировать и уточнять задачи, самостоятельно придумывать решения и чётко описывать их, критически анализировать результаты, тестировать программное обеспечение, самостоятельно изучать новые технологии, усваивать и фильтровать большие объемы информации.

Если оценить наиболее популярные виды его деятельности, то получиться, что учить нужно двум вещам:

- самостоятельному изучению и фильтрации больших объемов информации (технический английский), с последующим извлечением ключевых моментов и использованием полученной информации для решения конкретной практической задачи;

- абстрагированию и формализации, в частности, трансформированию жизненных задач в строгие формальные описания, созданию математических (логических) моделей, понятийному аппарату, позволяющему проводить анализ задач и разработку решений.

Поэтому, крайне полезными задания вида “Самостоятельно изучить документацию к инструменту X и с его помощью решить практическую задачу Y”.

Приведём список основных положений, которые должен помнить учитель:

- Доступность учебного материала первостепенно, а затем уже следуют полнота и формальная строгость. Необходимо научить учащихся научить внятно отвечать на элементарные, но важные вопросы:

  – В чём состоит задача?
  – Какие существует методы решения данной задачи и каковы их достоинства и недостатки?
  – Какие новые методы можно было бы разработать и какова сложность этой задачи?
  – Для какого класса задач полученное решение приемлемо, а для каких – нет?
  – Какие возможны пути развития (обобщения, доработки) полученного решения?

  Необходимо значительное время выделять анализу как постановки задачи, так и результатов.

- Важно достигнуть баланса между доступностью материала, наличием связи с реальными интересными задачами и фундаментальностью затронутых вопросов.

- В индустрии информационных технологии человек может играть различные роли: разработчик, системный интегратор, тестировщик, проектировщик, генератор идей, ... Важно, чтобы учащиеся успели попробовать себя в разных ролях. Каждой роли соответствует определённый психологический тип, образ мышления, для каждой роли требуются специфические знания и умения. Индивидуальный подход исключительно важен. Плохой разработчик может оказаться гениальным системным интегратором, или тестировщиком.

- У ученика должна быть свобода выбора как темы для углублённого изучения, так и степени её прикладной/теоретической направленности.

Необходимые условия для чтения книги

Благодарности

Игорю Борисовичу Петрову, Виктору Петровичу Иванникову, сотрудникам кафедры информатики МФТИ.

Большое спасибо всем студентам, принявшим активное участие в разработке и проверке данного курса.

Шеню А.Х., Евгению Щетинкину и Юле Мариинченко, Александру Брегеру, Евгению Барскому, Андрею Власову.

TODO

Часть I

Алгоритмы, исполнители и формальные языки

Лекция 1

Общее понятие алгоритма

Краткое описание: Первая лекция посвящена понятиям «алгоритм» и «рекурсия». Указываются базовые свойства алгоритма — детерминированность, конечность и массовость. В качестве примеров приводятся алгоритм Евклида и решение задачи «Ханойские башни». Изучается метод рекурсии. Разбираются два абстрактных исполнителя: исполнитель псевдокода и исполнитель конечных автоматов.
Ключевые слова: алгоритм, исполнитель, детерминированность, конечность, массовость, элементарные объекты и действия, переполнение, алгоритм Евклида, рекурсия, полнота набора элементов (функций), сложность вычислений, асимптотическая сложность, конечные автоматы.

Определение понятия алгоритма

Понятие алгоритма — это одно из основных понятий программирования и математики. : **Алгоритм**[1] — это последовательность команд, предназначенная исполнителю, в результате выполнения которой он должен решить поставленную задачу. Алгоритм должен описываться на формальном языке, исключающем неоднозначность толкования. **Исполнитель** — это человек, компьютер, автоматическое устройство и т.п. Исполнитель должен уметь выполнять все команды, составляющие алгоритм. Множество возможных команд конечно и изначально строго задано. Действия, которые выполняет исполнитель по этим командам называются элементарными.

Запись алгоритма на формальном языке называется **программой**. Иногда само понятие алгоритма отождествляется с его записью, так что слова «алгоритм» и «программа» — почти синонимы. Небольшое различие заключается в том, что при упоминании алгоритма, как правило, имеют в виду основную идею его построения. Программа же всегда связана с записью алгоритма на конкретном формальном языке.

Приведём простой алгоритм, позволяющий человеку уталить жажду:

1. Налить кружку воды.
2. Выпить кружку воды.
3. Если по-прежнему хочется пить, перейти к шагу 1.

[1]Приведённое определение не является строгим. В дальнейшем будет приведено строгое определение понятия *функции вычислимой на определённом исполнителе*, и в частности, будет определено *множество вычислимых функций*. Исторически сложилось так, что термин алгоритм не имеет точного значения и в различных областях математики используется либо в общем смысле (план действий), либо в некотором специальном смысле, которое отдельно оговаривается.

Алгоритмы обладают свойством **детерминированности** — каждый шаг и переход от шага к шагу должны быть точно определены так, чтобы его мог выполнить любой другой человек или механическое устройство.

Кроме детерминированности, алгоритмы должны обладать свойством конечности и массовости:

Конечность: Обычно предполагают, что алгоритм заканчивает исполнение за конечное число шагов. Результат работы алгоритма должен быть получен за конечное время. Но можно расширить понятие алгоритма до понятия процесса, который по различным каналам получает данные, выводит данных и потенциально может не заканчивать свою работу.

Массовость: Алгоритм применим к классу **входных данных**. В качестве данных могут выступать числа, последовательности чисел, последовательности букв и др. Не имеет смысла строить алгоритм нахождения наибольшего общего делителя только для одной пары чисел 10 и 15. Алгоритм должен решать не одну частную задачу, а класс задач.

Поясним эти свойства на простом примере. Рассмотрим следующую формулу вычисления числа π:

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...) = 4\sum_{i=1}^{\infty} \frac{1}{2i-1}.$$

Является ли эта формула алгоритмом вычисления числа π? Ответ на этот вопрос «нет», так как здесь нет ни свойства массовости (нет входных данных), ни свойства конечности (сумма бесконечного количества чисел)[2].

Операция суммирования бесконечного ряда не является элементарной ни для современных компьютеров, ни для человека, а если разложить эту операцию на отдельные шаги сложения, то получим бесконечное число шагов. Алгоритмы же по определению должны выполняться конечное число шагов и через конечное число шагов предоставлять результат вычислений[3]

Элементарные объекты и элементарные действия

Алгоритмы по определению должны сводится к последовательности элементарных действий над элементарными объектами. Какие действия и объекты элементарны, а какие — нет, зависит от исполнителя. *Набор элементарных действий и элементарных объектов для каждого исполнителя конечен и чётко зафиксирован.* Элементарные действия оперируют с небольшим числом элементарных объектов. Все остальные объекты и действия являются совокупностью элементарных. В современных компьютерах числа не являются элементарными

[2]Следует отметить, что в различных системах символьных вычислений число π представляется как число с бесконечной точностью в виде алгоритма, который по мере надобности вычисляет нужное число первых цифр десятичного представления числа π. Такой алгоритм можно построить и на основе приведённой формулы, но этот алгоритм нельзя отождествлять с самой формулой.

[3]Существуют особый класс полезных алгоритмов, для которых не выполнено ни свойство массовости, ни свойство конечности. Например, алгоритм работы светофора. Светофор работает постоянно и у него нет входных данных. Некоторые серверные программы потенциально никогда не завершают свою работу, они занимаются обработкой приходящих на сервер запросов по извлечению и изменению хранимых данных. Такого сорта алгоритмы мы пока рассматривать не будем.

объектами[4]. Для них элементарным объектом является является бит — ячейка памяти с двумя возможными состояниями: 0 и 1. С помощью набора бит можно записывать целые и действительные числа. В частности, существует простой способ представить целые числа от 0 до $2^8 - 1 = 255$ в виде последовательности 8 бит:

0	→ `00000000`	...	→ ...
1	→ `00000001`	250	→ `11111010`
2	→ `00000010`	251	→ `11111011`
3	→ `00000011`	252	→ `11111100`
4	→ `00000100`	253	→ `11111101`
5	→ `00000101`	254	→ `11111110`
...	→ ...	255	→ `11111111`

Указанный способ представления натуральных чисел в виде последовательности нулей и единиц называется **двоичной записью числа**. Каждому биту в этом представлении соответствует степень двойки. Самому правому биту соответствует $1 = 2^0$, второму справа — $2 = 2^1$, третьему справа — $4 = 2^2$, и т.д. Двоичная запись соответствует разложению числа в сумму неповторяющихся степеней двойки. Например:

$$
\begin{array}{lrll}
3 & \to & \texttt{11} & \to 1 \cdot 2^1 + 1 \cdot 2^0 = 2 + 1, \\
5 & \to & \texttt{101} & \to 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 0 + 1, \\
7 & \to & \texttt{111} & \to 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 4 + 2 + 1, \\
31 & \to & \texttt{11111} & \to 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0, \\
32 & \to & \texttt{100000} & \to 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0.
\end{array}
$$

Задача Л1.1. Докажите, что каждое натуральное число единственным образом представляется в виде суммы неповторяющихся степеней двойки. Подсказка: для данного числа n найдите максимальную степень двойки 2^m, которая меньше либо равна n. Рассмотрите число $n' = n - 2^m$. Воспользуйтесь методом математической индукции.

Конечный набор элементарных объектов может принимать лишь конечное число значений. Так, например, упорядоченный набор 8 бит (**байт**) имеет 256 возможных значений. Из этого простого факта следует очень важное утверждение: среди команд исполнителя не может быть команд сложения или умножения *произвольных* натуральных (действительных) чисел.

При изучении языка программирования, вы встретитесь с явлением *переполнения*. **Переполнение** — это ситуация, когда результат элементарной арифметической операции выходит за пределы подмножества чисел, которые можно записать в выбранном машинном представлении.

Итак, для компьютеров лишь некоторые действительные (целые) числа являются элементарными объектами. Множество этих чисел конечно. Какие именно действительные (целые) числа элементарны, зависит о используемого машинного представления.

Несмотря на то, что числа не является элементарными объектами, мы в некоторых алгоритмах позволим себе работать с числами как с элементарными объектами. При этом мы будем

[4]Для человека числа также не являются элементарными — обычно люди представляют большие числа как набор цифр в десятичной записи и для сложения и умножения используют алгоритмы сложения и умножения «столбиком», в которых вычисления сводятся к последовательности элементарных действий с цифрами.

подразумевать, что алгоритм работает лишь для чисел с ограниченным модулем и ограниченной точностью (в случае действительных чисел), или что за арифметическими операциями стоят соответствующие алгоритмы сложения, вычитания и умножения столбиком или алгоритм деления уголком сколь угодно «длинных» чисел.

Многие современные процессоры поддерживают несколько типов машинного представления действительных чисел. Целые числа практически везде представляются одинаковым образом. В процессорах с 32-разрядной архитектурой большая часть команд связана с числами, записанными в 32-х битах. При представлении неотрицательных чисел в 32 бита помещается их двоичная запись. Множество представимых таким образом чисел — это все неотрицательные числа меньше 2^{32}. Этому машинному представлению в языке Си часто соответствует тип данных `unsigned int` (англ. *unsigned integer number*). Если мы попытаемся сложить с помощью команды процессора два числа типа `unsigned int`, сумма которых больше либо равна 2^{32}, то возникнет **переполнение** и старший единичный 33-й бит результата будет утерян[5].

Для представления отрицательных чисел в виде 32 бит можно один бит выделить под знак — «плюс» или «минус», а в оставшемся 31-м бите записать двоичное представление модуля числа. Это представление называется *обратным кодом*. В нём имеется два различных представления числа 0. В большинстве современных компьютерных архитектурах используется другой принцип, называемый *дополнительным кодом*. Отрицательное число $-a$, $0 < a \leqslant 2^{31}$, имеет представление, которое совпадает с представлением в 32-х битах положительного целого числа $2^{32} - a$. Подробнее этот вопрос будет рассмотрен в лекциях по языку Си.

Подведём итоги:

У каждого исполнителя есть конечный набор элементарных команд (действий), оперирующих элементарными объектами, которых также конечное число.

Входом алгоритма является конечный набор элементарных объектов. Во время работы алгоритма выполняется конечное число элементарных действий и результат алгоритма является конечным набором элементарных объектов.

В компьютерах элементарным объектом является бит. Есть несколько стандартных способов представления чисел (действительных, целых, целых неотрицательных) в виде последовательности бит фиксированной длины.

Алфавит и множество слов

Введём несколько важных понятий.

Пусть B —конечное множество. Назовём его **алфавитом**, а его элементы — буквами. **Множество слов в алфавите** B — это множество конечных последовательностей элементов из B. Множество слов длины n обозначается как B^n. Например, для $B = \{0, 1\}$ множество слов длины 2 это

$$B^2 = \{00, 01, 10, 11\}.$$

Множество B^8 — это множество байт:

$$B^8 = \{00000000, 00000001, 00000010, 00000011, \ldots, 11111111\}.$$

[5]Результат вычисления не может содержать этот бит, но во многих процессорах есть специальный регистр, в котором хранится информация о том, произошло ли переполнение.

Сколько различных слов длины 8 в алфавите $B = \{0, 1\}$?

Ответ: $2^8 = 256$. Множество всех слов длины 8 можно записать как множество всех слов длины 7, к которым в качестве первого элемента сначала дописали 0, а потом — 1. Если число элементов множества X обозначить как $|X|$, то число всех слов длины n есть

$$|B^n| = |B| \cdot |B^{n-1}|,$$

откуда следует формула $|B^n| = |B|^n$.

Множество всех слов состоит из **пустого слова**, обозначаемого как $\varnothing$, слов длины 1, слов длины 2 и так далее. Множество всех слов обозначается как B^*:

$$B^* = \{\varnothing\} \cup B^1 \cup B^2 \cup B^3 \cup \dots$$

Важно понимать, что множество B^* содержит слова сколь угодно большой, но всё же конечной длины.

Входные и выходные данные алгоритмов представляют собой слова в некотором заданном конечном алфавите.

Способы записи алгоритмов

Алгоритмы можно описывать человеческим языком — словами. Так и в математике — все теоремы и утверждения можно записывать без специальных обозначений. Но специальный формальный язык записи утверждений сильно облегчает жизнь математикам: исчезает неоднозначность, появляются краткость и ясность изложения. Всё это позволяет математикам говорить и писать на одном языке и лучше понимать друг друга.

Большинство используемых в программировании алгоритмических языков имеют общие черты. В то же время, не всегда целесообразно пользоваться каким-либо конкретным языком программирования и загромождать изложение несущественными деталями. В этой книге мы будем использовать **псевдокод**, который похож на язык Pascal, но не является таким строгим (в частности, некоторые действия будут описываться просто словами). Запись алгоритма на псевдокоде зачастую нагляднее, она позволяет свободно выбирать уровень детализации, начиная от описания в самых общих чертах и кончая подробным изложением. Псевдокод несложно превратить в работающую программу на каком-либо языке программирования. Можно сказать, что псевдокод сам является программой для некоторого абстрактного **исполнителя псевдокода**.

Мы будем использовать в псевдокоде арифметические операции с целыми и вещественными числами (сложение $+$, умножение $\cdot$, вычитание $-$, деление $/$, остаток от деления $\bmod$), операцию присвоения переменной значения ($\leftarrow$), логические операции сравнения ($>$, $<$, $\leqslant$, $\geqslant$, $=$), а также операторы структурного программирования **for**, **while** и **if**. О них мы расскажем ниже.

На семинарах вы будете работать с конкретным языком программирования и научитесь проходить цепочку: идея алгоритма → псевдокод → программа на языке программирования.

Алгоритмы можно записывать не только в виде текста на каком-либо формальном языке. Существуют и другие способы описания алгоритмов, например, в виде блок-схем (dragon-схемы). Но в конечном счёте, любое описание, в том числе визуальное, так или иначе представлено (или потенциально может быть представлено) в компьютере в виде текста на некотором формальном языке.

Таким образом, запись алгоритма, также как и данные, является словом в некотором конечном алфавите.

Полный набор элементов, с помощью которого можно строго описать алгоритм, следующий:

- Описание исполнителя — математическое описание того, как происходит процесс исполнения программ.
- Программа для этого исполнителя (алгоритм, записанный, формальном языке, тнтепретируемым исполнителем).
- Формат входных данных — способ записи входных данных.
- Формат выходных данных — способ записи выходных данных.

Вычисление делителей числа

Рассмотрим простейший алгоритм (см. псевдокод 1.1), который печатает все делители заданного числа n.

Алгоритм 1.1 Делители числа

```
1: function DIVISORS(n)
2:     for i ∈ {1, 2, ..., n} do
3:         if n mod i = 0 then
4:             print i
5:         end if
6:     end for
7: end function
```

Алгоритм записан в виде функции DIVISORS, которая получает на вход натуральное число n. Логика работы этой функции проста: проверить все натуральные числа от 1 до n на то, делят они число n без остатка или нет, и вывести только те, которые делят. В алгоритме используется *переменная* i. Можно считать, что i — это имя ячейки в памяти исполнителя, в которой может хранится произвольное целое число. Переменная i последовательно принимает значения от 1 до n.

Конструкция псевдокода

```
1: for i ∈ S do
2:     A
3: end for
```

означает, что следует последовательно назначать переменной i значения из множества S и выполнять действие A. Эта конструкция называется **арифметическим циклом**, переменная i — **переменной цикла**, а действие A — **телом цикла**. При этом тело цикла A может быть составным, то есть состоять из последовательности действий, некоторые из которых могут быть арифметическими циклами и другими разрешёнными конструкциями.

Строка «2: **for** $i \in \{1,\ 2,\ \ldots,\ n\}$» означает «последовательно назначать переменной i значения 1, 2, ..., n и на каждом шаге выполнять тело цикла». Один шаг выполнения тела цикла называется **итерацией цикла**.

В теле цикла осуществляется проверка того, что остаток от деления числа n на i равен 0: $n \bmod i = 0$. Для этого используется **условный оператор if**.

Конструкция «**if**(A) **then** B **else** C **end if**» псевдокода означает «если верно A, то выполнить инструкции B, иначе выполнить инструкции C». Часть «**else** C» условного оператора может быть опущена.

Алгоритм Евклида

Рассмотрим *алгоритм Евклида* нахождения наибольшего общего делителя (НОД) двух натуральных чисел. Входными данными алгоритма являются два натуральных числа a и b, а выходными данными будет одно натуральное число — НОД чисел a и b.

Описание алгоритма словами:

1. Если $a = b$, то НОД $= a = b$ и заканчиваем вычисления.
2. Если $a > b$, то из a вычитаем b ($a \leftarrow a - b$). Переходим к 1.
3. Если же $b > a$, то из b вычитаем a ($b \leftarrow b - a$). Переходим к 1.

Формальная запись этого алгоритма представлена в псевдокоде 1.2.

Алгоритм 1.2 Алгоритм Евклида

```
 1: function НОД(a, b)
 2:     while a ≠ b do
 3:         if a > b then
 4:             a ← a − b          ▷ Переменной a присвоить значение a − b.
 5:         else
 6:             b ← b − a          ▷ Переменной b присвоить значение b − a.
 7:         end if
 8:     end while
 9:     return a
10: end function
```

Конструкция «**while**(A) **do** B **end while**» псевдокода означает «повторять последовательность операции B, записанных в **теле оператора while**, пока выполнено условие A». Тело цикла **while** — это все инструкции между **while**(A) **do** и **end while**. Если условие A не выполнено в самом начале, то тело оператора **while** не выполняется ни разу.

Инструкция «**return** a» означает «закончить процесс вычисления с результатом a».

Покажем, что наш алгоритм находит НОД двух чисел a и b.

Действительно, НОД($a,\ b$) = НОД($a - b,\ b$) при $a > b$, поэтому, несмотря на то, что на каждой итерации цикла меняется одно из чисел, значение НОД($a,\ b$) остается неизменным.

Максимальное из чисел a и b на каждом шаге уменьшается, и в какой-то момент одно из них становится равным нулю, а второе — искомому значению НОД. Этому *итеративному алгоритму* (основанному на итерациях цикла) можно сопоставить следующую рекуррентную формулу:

$$\text{НОД}(a,\ b) = \begin{cases} \text{НОД}(b,\ a-b),\ \text{если } a > b, \\ \text{НОД}(a, b-a),\ \text{если } a < b, \\ a,\ \text{если } a = b, \end{cases} \tag{1.1}$$

Задача Л1.2. Докажите, что $\text{НОД}(a,\ b) = \text{НОД}(a-b,\ b)$ для любых неотрицательных целых a и b, таких что $a > b$.

Задача Л1.3. Усовершенствуйте приведённый алгоритм 1.2, используя соотношение $\text{НОД}(a, b) = \text{НОД}(b,\ a \bmod b)$ при положительных a и b (выражение «$a \bmod b$» означает остаток при делении a на b).

Рекурсия. Задача «Ханойские башни»

Одно из важнейших понятий теории алгоритмов — **рекурсия**. Вы наверняка явно или неосознанно использовали метод рекурсии в жизни. Суть метода рекурсии заключается в том, что данная задача сводится к решению подобных задач, но с более простыми входными данными.

Например, пусть необходимо аналитически вычислить производную от композиции двух функций:

$$h(x) = f(g(x)), \quad h'(x) = ?$$

Для этого нужно решить две подобные задачи — найти производные $f'(y)$, $g'(x)$, а в качестве результата вернуть

$$f'(g(x)) \cdot g'(x).$$

Такой рекурсивный алгоритм вы применяете, решая задания по математическому анализу.

Задача «Ханойские башни» является ещё одним ярким примером применения метода рекурсии. Эта задача позволяет увидеть непосредственную связь рекурсии и метода математической индукции.

Задача Л1.4. (Ханойские башни) Есть три стержня и 64 кольца, нанизанных на первый стержень. Все кольца имеют разный диаметр, и меньшие кольца лежат на больших (то есть образуют сужающуюся пирамиду). За ход разрешается взять верхний диск с любого стержня и надеть сверху на один из других стержней. При этом запрещается класть больший диск на меньший. Задача заключается в том, чтобы переместить всю пирамиду с первого стержня на второй.

Доказательство того, что эта задача разрешима, основано на методе математической индукции. Индукция проводится по числу дисков. Для пирамиды из одного кольца всё очевидно — задача решается за один ход (база индукции). Пусть мы умеем перемещать $n-1$ диск (предположение индукции).

Проведём шаг индукции. Пусть дана пирамида из n дисков. Для того, чтобы переместить пирамиду на второй стержень, нам требуется переместить самый нижний большой диск на второй стержень, а для этого необходимо, чтобы все остальные кольца были на третьем стержне. Значит, чтобы переместить n дисков, нам сначала нужно переместить столбик из верхних

$(n-1)$ дисков на третий стержень. По предположению индукции мы можем это сделать. Сделаем это, а затем переместим самый большой диск с первого на второй и, наконец, переместим столбик из $(n-1)$ дисов с третьего стержня на второй. Таким образом, мы провели шаг индукции.

В соответствии со схемой шага индукции, можно сконструировать алгоритм MOVE перемещения произвольного числа дисков с одного стержня на другой. У алгоритма есть три аргумента — число дисков, номер стержня, на котором они находятся, и номер стержня, на который их нужно переместить.

Алгоритм 1.3, записанный на псевдокоде, реализует рекурсивную идею перемещения дисков. Функция $\text{MOVE}(n, x, y)$ перемещает n дисков со стержня с номером x на стержень с номером y. Номера стержней — это числа 1, 2, 3. Нетрудно заметить, что если даны номера x, y двух стержней, то номер третьего стержня вычисляется по формуле $6 - x - y$.

Алгоритм 1.3 Решение задачи «Ханойские башни»

```
1: function MOVE(n, x, y)
2:     if n = 1 then
3:         переместить верхний диск со стержня x на стержень y
4:     else
5:         MOVE(n − 1, x, 6 − x − y)
6:         MOVE(1, x, y)
7:         MOVE(n − 1, 6 − x − y, y)
8:     end if
9: end function
```

Таким образом, решение задачи перемещения n дисков сводится к двукратному перемещению $n-1$ кольца и перемещению одного кольца. Посчитаем количество действий, необходимое для проведения всей процедуры перемещения. Пусть $f(n)$ — необходимое число действий, для переноса пирамиды из n дисков. Для одного кольца ответ равен единице: $f(1) = 1$. Для $f(n)$ получим соотношение

$$f(n) = f(n-1) + 1 + f(n-1) = 2f(n-1) + 1.$$

Решая это рекуррентное соотношение получаем: $f(n) = 2^n - 1$. А значит

$$f(64) > 1\,000\,000\,000\,000\,000\,000.$$

Таким образом, время, необходимое для перемещения пирамидки из 64 дисков, очень велико.

Метод рекурсии является одним из базовых алгоритмических методов. Мы ещё не раз к нему вернёмся на лекциях и семинарах.

Пусть необходимо решить некоторую задачу A с входными данными x.
Метод рекурсии заключается в **сведении** задачи $A(x)$ к аналогичным задачам $A(y_1)$, $A(y_2)$, …, $A(y_m)$ для других входных данных. Сводимость задачи $A(x)$ означает, что решение задачи $A(x)$ может быть получено из решения задач $A(y_1)$, $A(y_2)$, …, $A(y_m)$.

В простейшем случае у задачи A один натуральный параметр n, и существует метод сведения задачи $A(n)$ к задаче $A(n-1)$. При этом необходимо, чтобы решение задачи $A(1)$ было известно. В общем случае сводимость и входные данные x могут быть более сложными.

Важно, чтобы любая цепочка значений входных данных $x_1, x_2, x_3, \ldots$ такая, что для решения задачи $A(x_i)$ требуется решить задачу $A(x_{i+1})$, всегда обрывалась на простых данных, для которых решение задачи A элементарно и может быть легко предъявлено без сведения к другим задачам.

Сформулируем несколько простых алгоритмических задач, которые полезно решить, чтобы освоится с понятием алгоритма и рекурсии.

Задача Л1.5. Напишите, рекурсивный алгоритм вычисления $n! = 1 \cdot 2 \cdot \ldots \cdot n$ на псевдокоде.

Задача Л1.6. Используя соотношение НОД(a, b) = НОД$(a \bmod b, b)$, напишите на псевдокоде рекурсивный алгоритм, вычисляющий НОД двух чисел a и b.

Задача Л1.7. Напишите на псевдокоде рекурсивный алгоритм печати цифр двоичного представления числа, начиная с младшего разряда.

Задача Л1.8. Дано множество прямых на плоскости, никакие три из которых не пересекаются в одной точке. Напишите на псевдокоде рекурсивный алгоритм закраски получившихся многоугольников в черный и белый цвета так, чтобы многоугольники одного цвета не имели общей стороны.

Конечные автоматы

Конечные автоматы — это простейшие преобразователи входных данных в выходные данные. Их обычно рассматривают как *потоковые* преобразователи, то есть такие исполнители, которые потенциально могут работать сколь угодно долго, получая на вход поток символов и печатая на выход преобразованный поток символов.

Конечный автомат может находится в одном из конечного числа состояний. Множество всех возможных состояний обозначим символом K.

Среди состояний есть выделенное начальное состояние $s_0 \in K$ и подмножество состояний останова $F \subset K$. Состояние s_0 является состоянием, в котором автомат находится в момент начала вычислений. В процессе вычислений автомат меняет своё состояние. Как только автомат оказывается в одном из состояний останова, вычисления прекращаются.

Конечный автомат работает по тактам. Программа работы конечного автомата задаётся с помощью *функции переходов между состояниями*. На каждом такте автомат считывает один символ из входа. и в зависимости от его значения и своего состояния переходит в одно из возможных состояний. Функция переходов π для каждой пары (состояние, считанный символ) задаёт следующее состояние[6].

[6]Иногда считают, что функция π может быть определена не на всех парах. С точки зрения математики это вполне штатная ситуация, нужно просто уточнить, как идёт процесс вычисления в случае возникновения пары (состояние, считанный символ), для которой значение π неопределено. Естественно предположить, что в этом случае вычисления останавливаются. С точки же зрения программиста, функции должны быть определены для всех возможных значений параметров, то есть они всегда должны что-то возвращать в качестве результата вычисления. Альтернатива «зависнуть» или вызвать ошибку исполнения рассматривается как грубая ошибка. Конечно, можно ввести специальное значение «неопределённость», и возвращать её для некоторых аргументов, но в нашем случае это не более чем введение нового состояния конечного состояния.

Приведём строгое определение.

Определение 1.1. **Конечный автомат** $\mathcal{M}$ *задаётся множеством из пяти элементов: конечное множество состояний* K*, конечный входной алфавит* B*, начальное состояние* $s_0 \in K$*, подмножество состояний* $F \subset K$*, и функция переходов* $\pi : K \times B \to K$*:*

$$\mathcal{M} = \{B, K, \pi, s_0, F\}, \quad s_0 \in K,\ F \subset K, \quad \pi : K \times B \to K.$$

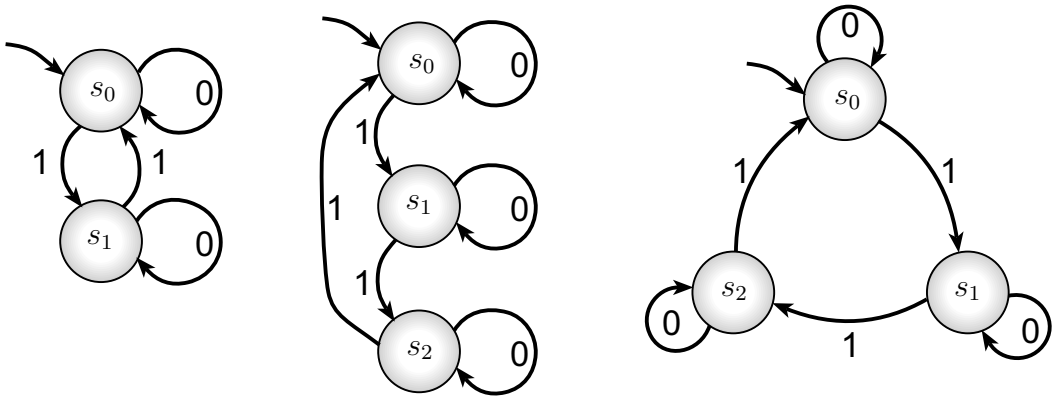
Алгоритм работы исполнителя конечного автомата описывается псевдокодом 1.4.

Алгоритм 1.4 Алгоритм исполнения конечного автомата.

```
1: x ← s_0
2: while x ∉ F do
3:     b ← считать символ
4:     x ← π(x, b)
5: end while
```

Приведём пример конечного автомата, который считает чётность числа введённых единиц. Положим $K = \{s_0, s_1\}$, s_0 — начальное состояние, $B = \{0, 1\}$, а функция π задаётся таблицей[7]:

x	b	$\pi(x, b)$
s_0	0	s_0
s_0	1	s_1
s_1	0	s_1
s_1	1	s_0



(а) (б) (в)

Рис. 1.1: Визуальные представления конечных автоматов. (а) Автомат, определяющий чётность числа единиц. (б) Автомат, вычисляющий остаток от деления на 3 числа единиц. (в) Другое визуальное представление автомата (б).

[7]В этой таблице легко узнается операция сложения по модулю 2 или (если иметь в виду отождествления 1≡**true**, 0≡**false**) булева бинарная операция «исключающее или» (XOR), которая равна истине только тогда, когда ровно один операнд равен истине.

Конечным автоматам удобно сопоставлять визуальное представление. На рисунке 1.1 (а) кружочками S1, S2 представлены состояния автомата, каждой стрелке соответствует строчка в таблице описания функции переходов. Рядом с каждой стрелкой указан символ b, равный '0' или '1'. Стрелка от состояния x в состояние x' отображает, что из состояния x будет осуществлён переход в состояние x', если на вход будет подан символ b. Входящей стрелкой показано начальное состояние автомата.

Автомат находится в состоянии s_0, когда считано чётное число единиц, и в состоянии s_1, если нечётное. Если считанный символ равен '0', то состояние не меняется, а если '1', то меняется.

У этого конечного автомата нет состояний останова, поэтому он будет работать, пока на вход поступают символы. В каждый момент результат вычисления представлен состоянием автомата.

Можно ввести специальный символ завершения ввода '#' и два дополнительных состояния f_0 и f_1 — состояния останова: $F = \{f_0, f_1\}$. А также добавить переходы от s_0 к f_0 и от s_1 к f_1 в случае получения на входе символа '#'.

На рисунках 1.1 (б,в) показаны различные визуальные представления одного и того же конечного автомата, считающего остаток при делении на 3 числа единиц во входных данных.

Рассмотрим более сложную задачу.

Задача Л1.9. На вход поступает двоичная запись натурального числа n, начиная с младшего разряда. Сконструируйте конечный автомат, который вычисляет остаток при делении n на 3.

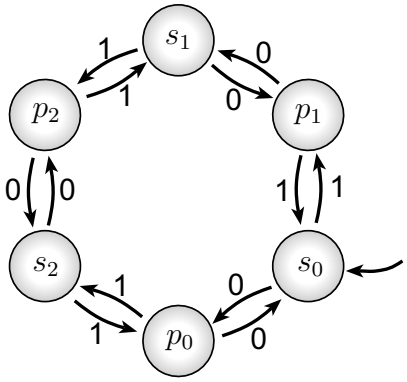


Рис. 1.2: Автомат, который по двоичной записи числа определяет его остаток при делении на 3. Двоичная запись подаётся на вход с младшего разряда.

Введём две группы состояний: $\{s_0, s_1, s_2\}$ и $\{p_0, p_1, p_2\}$. Первая группа соответствует состоянию, когда следующий символ — это цифра нечётного разряда (самого младшего разряда, третьего по младшинству разряда, и т.д). Вторая группа соответствует разрядам с чётными номерами.

На каждый момент автомат считал какое-то количество младших разрядов, их мы интерпретируем как число и для этого числа он уже «знает» ответ. Остаток равен 0, если автомат находится в состояниях s_0 или p_0, 1 — если в состояниях s_1, p_1, и 2 — если в состояниях s_2 или p_2.

Пусть автомат находится в состоянии s_i. Тогда, если следующий символ '1', то считанное число увеличилось либо на 1, либо на $2^2 = 4$, либо на $2^4 = 16$, ..., то есть на 2^{2n}, $n = 0, 1, \ldots$ Нетрудно заметить, что все эти числа при делении на 3 дают остаток 1. То есть при считывании символа '1' в этих состояниях остаток числа увеличивается на 1. Если же будет считан символ '0', то остаток не изменится.

Состояния p_0, p_1 и p_2 соответствуют моментам, когда автомат собирается получить на входе чётный по счёту разряд. При считывании '1' число увеличивается на 2^{2n+1}, $n = 0, 1, \ldots$ Все числа такого вида имеют при делении на 3 остаток 2. Поэтому при считывании '1' в этих состояниях остаток увеличивается на 2, именно поэтому при $b =$ '1' осуществляется переход из p_0 в s_2.

Обобщения конечного автомата

Конечный автомат во время своей работы меняет своё состояние и значение этого состояния, в определённом смысле, является (на каждый момент времени) результатом вычисления.

Для получения более сложных результатов используют обобщения конечных автоматов.

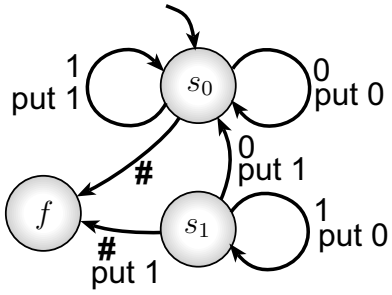
В простейшем случае добавляют возможность привязывать к переходам и состояниям какие-либо действия.

Рассмотрим случай, когда к каждому переходу можно привязать действие «`put X`» — напечатать символ (или строку символов) `X`. Такой конечный автомат мы будем называть **конечным автоматом с возможностью вывода**.

Задача Л1.10. На вход поступает двоичная запись числа n, начиная с младшего разряда. Выведите двоичную запись числа $n + 1$.

На рисунке 1.3 показаны табличное и визуальное представления автомата, который решает поставленную задачу. Состояние s_1 соответствует состоянию, когда есть единица «в запасе», которая переходит на следующий разряд. Это состояние, когда к поступающему числу мы всё еще должны добавить единицу. В состоянии s_0 уже не нужно прибавлять единицу, так как она «осела» на одном из младших разрядов. В состоянии s_0 нужно просто копировать приходящие символы на выход. Символ '#' — это специальный символ конца ввода. Когда он приходит, необходимо закончить вычисление. Состояние f — состояние останова вычислений. При получении символа '#' в состоянии s_1 необходимо не забыть вывести единицу, которая «в запасе».

В таблице в каждой строке описан переход. Кроме состояния, в которое перейдёт автомат из состояния x при считывании символа b, в таблице указано также действие $a(x, b)$, которое будет выполняться при переходе. Соответственно функция перехода π конструируется из двух функций: $\pi(x, b) = (s(x, b),\ a(x, b))$, где $s(x, b)$ — новое состояние, $a(x, b)$ — выполняемое действие.



x	b	$\pi(x, b)$	$a(x, b)$
s_0	0	s_0	putc 0,
s_0	1	s_0	putc 1,
s_0	#	f	–
s_1	0	s_0	putc 1
s_1	1	s_1	putc 0
s_1	#	f	putc 1

Рис. 1.3: Конечный автомат с возможностью вывода, который получает на вход двоичную запись натурального числа n, начиная с младшего разряда, и выводит двоичную запись числа $n + 1$.

Схема работы конечного автомата с действиями описана в псевдокоде 1.5

Задача Л1.11. Докажите, что не существует конечного автомата, который решает задачу увеличения натурального числа на единицу, если входное число записано в двоичной системе счисления и поступает на вход конечного автомата, начиная со старшего разряда.

Алгоритм 1.5 Алгоритм исполнения конечного автомата с действиями.

1: $x \leftarrow s_0$
2: **while** $x \notin F$ **do**
3: $b \leftarrow$ считать символ
4: $(x, a) \leftarrow \pi(x, b)$
5: выполнить действие a
6: **end while**

Конечный автомат со стеком

Довольно частым обобщением конечного автомата является *конечный автомат с дополнительной памятью* — некоторым запоминающим устройством с набором команд для сохранения и извлечения данных.

В качестве такого запоминающего устройства часто рассматривают *стек*.

Определение 1.2. **Стек** — *это абстрактная структура данных, которую удобно представлять как полубесконечную трубку с одним концом и двумя операциями:*

`push(a):` *поместить элемент* `a` *в стек;*

`a=pop():` *извлечь элемент из стека.*

Элементы извлекаются из стека в последовательности, обратной той, в которой помещались. Первым из стека извлекается тот элемент, который был помещён туда последним (LIFO, last in — first out).

Стек означает «стопка». Новый элемент кладётся в стопку сверху, и сверху же берётся элемент, когда извлекается. До нижних элементов стопки можно добраться только после извлечения всех элементов, которые находятся над ними сверху.

Стек предоставляет очень ограниченные возможности хранения данных. По сути доступ открыт только к одной ячейке памяти – к той, в которой хранится верхний элемент стека[8].

Определим автомат, снабжённый стеком для хранения символов, в котором есть возможность во время переходов совершать действия вида `push a` и `pop`.

Определение 1.3. **Конечный автомат со стеком и возможностью вывода** — *это автомат, в котором функция переходов* $\pi : K \times B \times B \to K \times A^*$ *для тройки*

- x *– состояние автомата* $x \in K$*),*
- b *– входной символ* ($b \in B$*),*
- c *– элемент в вершине стека* ($c \in B$*)*

задаёт пару

- $s(x, b, c)$ *– новое состояние автомата* ($s(x, b, c) \in K$*),*
- $a(x, b, c)$ *– последовательность действий* ($a(x, b, c) \in A^*$*).*

[8]Устройство памяти, которое позволяет обращаться к произвольной ячейке памяти по номеру, называется *памятью с произвольным доступом* (random access memory, RAM).

Множество возможных действий A состоит из команд вида `putc z`, `push z` *и* `pop`, *где* `z` *одна из букв алфавита* B.

Значение аргумента `z` каждой команды вида `push z` и `putc z` фиксировано. Значения символов, извлекаемых из стека командой `pop`, не могут использоваться в качестве аргумента `z` операций `push` и `putc`.

Действия, которые можно осуществлять во время переходов между состояниями максимально просты. Недопустимо использовать условные операторы, создавать временные переменные для хранения извлечённых из стека значений, или осуществлять какие-либо другие действия, не указанные в определении.

Рассмотрим следующий конечный автомат со стеком и множеством состояний $K = \{s_0, ok, bad\}$. Среди них состояния останова $F = \{ok, bad\}$, и начальное состояние s_0.

Алфавит входных символов состоит из двух скобок и символа окончания ввода: $B = \{$`(`, `)`, `#`$\}$ Если приходит открывающаяся скобка, то автомат выполняет операцию `push` — помещает эту скобку в стек. Если приходит закрывающаяся скобка, то автомат делает `pop`. Но если стек пуст (признаком этого является $c = \varnothing$), то осуществляется переход в состояние останова *bad*. Если приходит символ конца ввода `#`, то в зависимости от того, пуст автомат или нет, осуществляется переход в состояния *ok* или *bad*.

Приведём таблицу, описывающую функции s и a. В этой таблице мы использовали символ `*` для обозначения произвольного символа. Это позволяет несколько строчек, отличающихся лишь в одном символе, «свернуть» в одну строчку. В таблице добавился столбец c, в котором задаётся значение элемента в вершине стека.

x	b	c	$s(x,b,c)$	$a(x,b,c)$
s_0	`(`	`*`	s_0	`push '('`
s_0	`)`	$\varnothing$	*bad*	–
s_0	`)`	`(`	s_0	`pop`
s_0	`#`	`)`	*bad*	–
s_0	`#`	$\varnothing$	*ok*	–

b	(	(	(	)	)	(	)	)	#
c	∅	(	(	(	(	(	(	(	∅
действия	push (	push (	push (	pop	pop	push (	pop	pop	

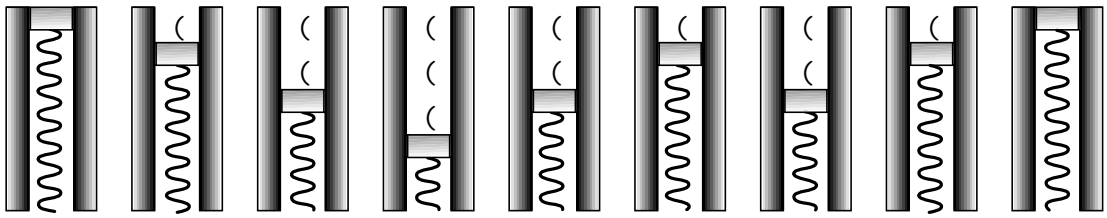


Рис. 1.4: Последовательные состояния стека для входного слова «`(((())())`».

На рисунке 1.4 показаны последовательные состояния стека для случая, когда на вход поступает слово `(((())())`.

Задача Л1.12. Определите конечное состояние этого автомата для входных слов `#`, `()#`, `()()()#`, `())(()#`, `((()())#`, `((()(()))#`, `(((()))))#`, `)(#`.

Задача Л1.13. Существует ли обычный конечный автомат, который эквивалентен (по результату вычислений) приведённому автомату со стеком?

Задача Л1.14. Рассмотрите приведённый на рис. 1.5 конечный автомат со стеком, который переворачивает входное слово в алфавите $B = \{0, 1\}$, то есть выводит слово в обратном порядке символов — от последнего символа к первому. Состояние f является состоянием останова. В его реализации используется новое необычное действие `ungetc c` — вернуть символ `c` в поток входа. Можно ли используя это действие, сконструировать конечный автомат со стеком, который а) два раза повторяет введённое слово в исходном порядке символов; б) три раза повторяет введённое слово в обратном порядке символов.

x	b	c	$s(x,b,c)$	$a(x,b,c)$
s_0	0	*	s_0	`push 0`
s_0	1	*	s_0	`push 1`
s_0	#	*	s_0	`putc c, pop, ungetc #`
s_0	#	$\varnothing$	f	

Рис. 1.5: Описание конечного автомата со стеком к задаче Л1.14.

Заключительные замечания

Итак, мы ввели ряд важных понятий: алгоритм, исполнитель, элементарные действия и элементарные объекты. К этим понятиям мы будем возвращаться ещё не раз. Кроме того, мы рассмотрели двух исполнителей:

Исполнитель псевдокода. Исполнитель, логика работы которого задаётся с помощью псевдокода. Выполняет алгоритмы общего плана, умеет выполнять арифметические операции, имеет сколь угодно большую внешнюю память, понимает конструкции **while**, **for**, **if** и др. Близок к интуитивному понятию человека-исполнителя — человека, способного пошагово исполнять формально записанные алгоритмы.

Исполнитель конечных автоматов. Исполнитель, логика которого задаётся с помощью конечного множества состояний, диаграммы переходов между ними, и (в случае обобщённого конечного автомата) набора действий, привязанных к переходам. Исполнитель может находится в одном из указанных состояний, во время работы осуществляет переходы между ними и осуществляет действия согласно функции перехода и привязанным к переходам действиям. Позволяет осуществлять простейшие преобразования потока данных.

Множество задач, которые можно решить на этих исполнителях, различны. В первом случае — это вычисления и преобразования данных самого общего характера. А с помощью конечных автоматов могут быть запрограммированы лишь простейшие преобразования данных.

В частности, задача увеличения числа, заданного в двоичной системе счисления, решается на конечном автомате с возможностью вывода (без стека и без какой-либо другой дополнительной памяти) лишь в том случае, если разряды числа подаются на вход начиная с младшего разряда. Если же разряды поступают на вход со старшего разряда, то задача становится неразрешимой для исполнителя конечных автоматов. Также на исполнителе конечных

автоматов нельзя решить задачу умножения двух натуральных чисел, заданных в двоичной системе счисления.

На псевдокоде все указанные задачи разрешимы. На исполнителе псевдокода можно «эмулировать»[9] исполнитель конечных автоматов (как обычный, так и обобщённый). Псевдокоды 1.4 (стр. 29) и 1.5 (стр. 32) по сути являются такими *эмуляторами*. Это означает, что все задачи, которые можно решить на исполнителе конечных автоматов, можно решить и на исполнителе псевдокода. Обратное неверно.

Можно показать, что на обобщённом конечном автомате без дополнительной памяти можно реализовать в точности только те преобразования, которые может осуществить исполнитель псевдокода, память которого ограничена некоторым фиксированным конечным числом ячеек, каждая из которых может хранить один из конечного фиксированного набора символов.

Преобразования, которые можно осуществить на исполнителе псевдокода, называются *вычислимыми*. Конечно, это определение недостаточно точно, поскольку мы не дали математически строгого определения исполнителя псевдокода. Оно в значительной степени интуитивно. Более точно класс вычислимых функций мы определим на следующей лекции с помощью строго заданного абстрактного исполнителя машин Тьюринга.

Можно считать, что вычислимые функции — это некоторое подмножество функций вида $f : B^* \to B^*$, где B — некоторый конечный алфавит. Данные произвольной природы можно записать в виде последовательности символов, необходимо только выбрать одну из возможных *формальных нотаций*.

Важно отметить, что не существует такого исполнителя, с помощью которого можно было бы запрограммировать любую функцию вида $f : B^* \to B^*$. Большая часть функций такого вида *невычислима*. Мы разберём это подробнее не следующей лекции.

[9] Подробнее о задаче эмуляции рассказано в лекции 3.

Семинар 1

Алгоритмы и исполнители

Краткое описание: На лекции были определены следующие абстрактные исполнители: исполнитель-человек, конечные автоматы и несколько простых вычислителей. Эти исполнители имеют разное устройство и решают различные вычислительные задачи. Важно научиться действовать в рамках определенного исполнителя и по набору элементарных действий исполнителя видеть его возможности и оптимальные пути решения поставленных задач. Данный семинар посвящён практике программирования абстрактных исполнителей.

Исполнитель псевдокода

Наиболее общим исполнителем является человек — он способен решать широкий класс вычислительных задач, включая задачи, решаемые конечными автоматами и другими абстрактными исполнителями.

Для него программу удобно представлять в виде *псевдокода*. Первые задачи посвящены программированию на псевдокоде.

Задача C1.1. Напишите на псевдокоде функцию, которая вычисляет наименьшее общее кратное двух данных натуральных чисел.

Задача C1.2. Изучите описание функции Is-Prime(n), представленной в псевдокоде 1.6, которая должна определять, простое число n или нет. Иногда эта функция работает неверно (в ней есть по крайней мере две ошибки). Исправьте её так, чтобы она верно работала для всех натуральных чисел n. Оцените максимальное число проверок «$n \bmod i = 0$», которое может быть выполнено при работе данной функции. Эта оценка должна представлять собой функцию от n.

Задача C1.3. Найдите значения, которые вернёт функция F, описанная в псевдокоде 1.7 при $n = 1, 2, 3, 10$. Найдите общую формулу.

Задача C1.4. Напишите на псевдокоде функцию, которая для данных действительных чисел a, b, c выводит, сколько различных корней есть у уравнения $a \cdot x^2 + b \cdot x + c = 0$ и выводит значения этих корней.

Задача C1.5. Напишите на псевдокоде функцию, которая которая находит число способов представить n в виде суммы неповторяющихся положительных целых слагаемых (порядок слагаемых неважен). Подсказка: опишите рекурсивную функцию CountDecompositions(n, m), которая вычисляет число способов представления числа n в виде суммы неповторяющихся слагаемых, каждое из которых меньше m.

Алгоритм 1.6 Алгоритм проверки числа на простоту, содержащий ошибку

```
1:  function Is-Prime(n)
2:      i ← 1
3:      while i · i < n do
4:          if n mod i = 0 then
5:              break              ▷ Досрочный выход из цикла while, если нашли делитель.
6:          end if
7:          i ← i + 2
8:      end while
9:      if i · i < n then
10:         return false           ▷ Данное число составное. Оно делится без остатка на i.
11:     else
12:         return true                                  ▷ Данное число простое.
13:     end if
14: end function
```

Задача С1.6. Напишите на псевдокоде функцию, которая данное натуральное число разлагает в сумму неповторяющихся степеней двойки. А именно, для случая $n = 5$ она печатает последовательность $\{0, 2\}$, так как $5 = 2^0 + 2^2$, для $n = 256$ — последовательность из одного элемента $\{8\}$, так как $256 = 2^8$, а для $n = 255$ — $0, 1, 2, 3, 4, 5, 6, 7$, так как $255 = 2^0+2^1+2^3+2^4+2^5+2^6+2^7$.

Алгоритм 1.7 Функция–загадка

```
1:  function F(n).
2:      k ← n
3:      b ← 3
4:      r ← 1
5:      while k > 0 do
6:          if k mod 2 = 1 then        ▷ Если остаток от деления k на 2 равен 1.
7:              r ← r · b
8:          end if
9:          b ← b · b
10:         k ← k/2                                  ▷ Деление нацело без остатка.
11:     end while
12:     return r
13: end function
```

Задача С1.7. Напишите псевдокод, который выводит простейшие архимедовы тройки, а именно, все тройки тройки натуральных чисел (a, b, c), такие что $a^2 + b^2 = c^2$ и $a \leqslant b \leqslant c \leqslant n$, где n — натуральное число, заданное пользователем. Сколько времени работает ваш код в зависимости от n? Попробуйте написать такой псевдокод, время работы которого (считайте, что время выполнения элементарных команд одинаково и равно одной условной секунде) ограничено квадратичной функцией $C \cdot n^2 + D$, где C и D некоторые константы.

Задача С1.8. Изучите псевдокод 1.8, который выводит все простые числа от 2 до n. Оцените время его работы. Предложите оптимизацию алгоритма, которая уменьшит время его работы

в два раза или более. Будет ли алгоритм работать, если строку «$m \leftarrow k \cdot 2$» заменить на «$m \leftarrow k \cdot k$»? Как примерно растёт время работы алгоритма с ростом n?

Алгоритм 1.8 Решето Эратосфена

```
1: function ERATOSTHENESSIEVE(n).
2:     a ← {0, 0, 0, ..., 0, 0}          ▷ Объявим массив элементов a[1], a[2], ..., a[n].
3:       ▷ В конце работы алгоритма a[i] = 0 только для простых i (i > 1).
4:     k ← 2
5:     while k ⩽ n do
6:         if a[k] = 0 then
7:             print k                    ▷ Выведем очередное простое число k.
8:             m ← k · 2
9:             while m ⩽ n do             ▷ Отметим числа, кратные k, как составное.
10:                a[m] ← 1
11:                m ← m + k
12:            end while
13:        end if
14:        k ← k + 1
15:    end while
16: end function
```

Задача C1.9.* **(самоописывающаяся последовательность)** Последовательность $a_1, a_2, \ldots$ состоит из натуральных чисел и 1) не убывает; 2) a_n в точности равно количеству элементов в последовательности, равных n; 3) $a_1 = 1$. Оказывается, указанные три условия однозначно задают эту последовательность Напишите псевдокод, который вычисляет a_n по заданному n. Оцените, как растёт время работы вашего алгоритма в зависимости от n. Приведём первые элементы последовательности:

$$a_1 = 1,\ a_2 = 2,\ a_3 = 2,\ a_4 = 3,\ a_5 = 3,\ a_6 = 4,\ \ldots a_9 = 5,\ \ldots, a_{12} = 6.$$

Простейшие исполнители

Задача C1.10. Рассмотрим исполнителя, у которого есть одна ячейка памяти x, в которой может храниться целое число из отрезка $[0, 2^{100}]$. Изначально $x = 0$. Исполнитель умеет выполнять две элементарные операции: умножить значение x на 2, и прибавить к значению x число 1:

$$x \leftarrow 2 \cdot x, \quad x \leftarrow x + 1.$$

Напишите кратчайшую программу для этого исполнителя, в результате выполнения которой получается $x = 1000$.

Задача C1.11. Решите предыдущую задачу для расширенного набора элементарных команд:

$$x \leftarrow x + 1, \quad x \leftarrow x - 1, \quad x \leftarrow 2 \cdot x, \quad x \leftarrow 4 \cdot x, \quad x \leftarrow 16 \cdot x.$$

Задача С1.12. Пусть исполнитель имеет одну ячейку памяти x, в которой может храниться одно рациональное число. И пусть он умеет выполнять две элементарные операции: прибавить к числу 1 и обратить число:

$$1)\ x \leftarrow x + 1; \quad 2)\ x \leftarrow 1/x.$$

Изначально в ячейке находится число 0. Укажите последовательность действий для этого исполнителя, которые приводят к x равном
а) 2/5; б) 13/8; в) 13/21; г) 13/23; д) 23/55; е) 610/377.
Каков минимальный размер программы, которая получает указанное значение x?

Задача С1.13. Решите предыдущую задачу для случая расширенного множества элементарных команд:

$$1)\ x \leftarrow 1/x; \quad 2)\ x \leftarrow x + 1;\ 3)\ x \leftarrow x + 2;\ \ldots;\ 1001)\ x \leftarrow x + 1000.$$

Конечные автоматы

Везде в задачах под конечным автоматом мы понимаем либо обычный конечный автомат, либо конечный автомат с действиями вида «puts *строка*», привязанными к переходам.

Задача С1.14. Постройте конечный автомат, который получает на вход слово в алфавите $B = \{0, 1\}$ и возвращает то же самое слово, только инвертированое (нули заменены на единицы и наоборот).

Задача С1.15. Постройте конечный автомат, который получает на вход слово в алфавите $B = \{0, 1\}$ и возвращает то же самое слово, в котором подряд идущие единицы заменены одной единицей.

Задача С1.16. Постройте конечный автомат, который получает на вход двоичную запись натурального числа x, начиная с младшего разряда, и выводит двоичную запись числа
а) $x + 3$; б) $3 \cdot x$; в) $5 \cdot x$; г) $x \bmod 5$;
д) $3x + 2$; е) $x \cdot (x \bmod 3)$; ж) $x/4$ (деление без остатка).

Задача С1.17. Постройте конечный автомат, который получает на вход слово в алфавите $B = \{a, b, c, \ldots, x, y, z\}$ и во время работы каждый раз, как встречается слово «`abrakadabra`» выводит символ `1`. Проверьте работу своего алгоритма на входе `abrabrakadabrabra`.

Задача С1.18. Постройте конечный автомат, который получает на вход слово в алфавите $B = \{a, b, c, \ldots, x, y, z\}$ и во время работы каждый раз, как встречается слово «`abcdabceabcde`» выводит символ `1`, а каждый раз, как встречается слово `abcdea`, выводит символ 2.

Задача С1.19. Постройте конечный автомат, который получает на вход слово в алфавите $B = \{a, b, c, \ldots, x, y, z\}$ и выводит его, удалив подстроки `abab` (в первую очередь) и `abcab` (во вторую очередь). Проверьте работу алгоритма на следующих словах: `abab`, `abcab`, `ababcab`, `abcababcabc` (результат должен быть равен `abccabc`), `aabcababababa` (результат — `aa`), `abcabcabc`.

Задача C1.20. Рассмотрим слова в алфавитах $B_1 = \{A, B, C\}$ и $B_2 = \{0, 1\}$. Напишите конечный автомат, который каким-либо образом кодирует слова из B_1^* словами из B_2^*, а именно, разным словам из алфавита B_1 сопоставляет разные слова из алфавита B_2. Напишите также конечный автомат, который осуществляет *декодирование*, то есть обратное преобразование слов B_2^* в слова B_1^*. Существует ли взаимооднозначное[1] отображение (кодирование) $f : B_1^* \to B_2^*$, такое что f и обратное отображение f^{-1} реализуются в виде конечных автоматов?

Задача C1.21. Существует ли конечный автомат, который
а) получает на вход слово произвольной длины в алфавите $B = \{0, 1\}$ и возвращает то же самое слово, но с обратным порядком букв;
б) получает на вход двоичные записи двух сколь угодно больших натуральных чисел, и возвращает 0 или 1, в зависимости от того, какое из них больше;
в) получает на вход двоичную запись сколь угодно большого числа x, начиная со старшего разряда, и возвращает запись числа $x/3$;
г) выполняет вычисления предыдущего пункта, но при условии, что записи входного и выходного числа начинаются с младшего разряда;
д) получает на вход двоичную запись сколь угодно большого числа x, начиная с младшего разряда, и проверяет простое оно или составное;
д) переводит число из двоичной записи в десятичную (рассмотрите оба возможных порядка цифр);
д) переводит число из двоичной записи в шестнадцатиричную (рассмотрите оба возможных порядка цифр);
д) переводит число из восьмиричной записи в двоичную (рассмотрите оба возможных порядка цифр);
е) получает на вход натуральное число, заданное в десятичной системе счисления, и проверяет, делится оно на 11 или нет (рассмотрите оба возможных порядка цифр)?

[1]Взаимооднозначное отображение $f : A \to B$ — это отображение элементов множества A в элементы множества B, при этом разные элементы из A должны быть отображены в разные элементы из B, и в каждый элемент из B должен быть отображён элемент из A (из предыдущего свойства, этот элемент единственнен). Взаимооднозначные отображения называются **биекциями**.

Лекция 2

Машина Тьюринга

Краткое описание:
На этой лекции будет дано строгое определение понятия *вычислимых функций*. Для этого будет введён абстрактный исполнитель машин Тьюринга. Согласно гипотезе Тьюринга, этот исполнитель может решать тот же класс алгоритмических задач, что и человек, снабжённый сколь угодно большим хранилищем данных.
Кроме того, будет дано альтернативное определение понятия вычислимых функций, основанное на операторах примитивной рекурсии, композиции и оператора взятия минимума аргумента.
Ключевые слова: машина Тьюринга, вычислимые функции, примитивно рекурсивные, частично рекурсивные и общерекурсивные функции, замкнутость множества относительно набора операций.

Вычисления и способы представления данных

Процесс вычислений можно рассматривать как преобразование входных данных в выходные. Данные могут иметь самую различную структуру и представлять собой: натуральное число, набор целых чисел, матрицу действительных чисел, последовательность символов (текст) и др. Структура входных и выходных данных определяется решаемой задачей. Но важно отметить, что входные и выходные данные можно записать (согласно какой-либо нотации) как последовательность символов некоторого алфавита.

Например, наиболее подходящая нотация записи одного натурального числа — это его запись в десятичной или двоичной системе счисления. Для записи последовательности чисел удобно ввести разделительный символ — запятую или пробел. Набор последовательностей чисел можно записать, используя запятую для разделения чисел, и точку с запятой — для разделения последовательностей. Таким образом, входные данные любой природы можно, договорившись о нотации, записать в виде последовательности символов некоторого конечного алфавита.

В свою очередь, каждый символ любого конечного алфавита можно представить как двоичное слово — слово в алфавите $B = \{0, 1\}$. Так, например, в современных компьютерах символы кодируются байтами (бинарными словами длины 8).

Конечные данные любой природы можно записать в виде слова некоторого алфавита B. Поэтому, функции, реализуемые на вычислительных устройствах, можно рассматривать как функции вида

$$f : B^* \to B^*,$$

где B^* означает множество всех слов в некотором конечном алфавите B.

Несложно установить взаимооднозначное соответствие между множеством слов B^* и натуральными числами $\mathbb{N} = \{1,\ 2,\ 3,\ \ldots\}$. Для того, чтобы из слова в алфавите $B = \{0, 1\}$ получить натуральное число, необходимо просто дописать к нему слева единицу и проинтерпретировать результат как двоичную запись натурального числа. Обратное соответствие очевидно. Если дано натуральное число, возьмём его двоичную запись и уберём самую левую единицу. Взаимооднозначное обображение между словами и натуральными числами несложно построить для произвольного конечного алфавита B, причём это отображение будет «легко вычислимым» в обе стороны.

Это позволяет нам считать, что множество рассматриваемых функций (класс задач, которые решают рассмариваемые исполнители) — это множество функций вида $f : \mathbb{N} \to \mathbb{N}$.

Исполнители алгоритмов. Уточнение понятия алгоритма

Мы подошли к важному понятию — вычислимые функции.

Грубо говоря, речь идёт о функциях, которые могут быть запрограммированы на компьютере с потенциально бесконечной памятью. Но для математической строгости необходимо более точно описать *эталонного исполнителя*.

Наиболее известны в литературе три исполнителя: исполнитель машин Тьюринга, исполнитель алгорифмов Маркова (алгорифмы Маркова), и исполнитель машин Поста. Они основаны на трёх различных «парадигмах» вычислений: *автоматное программирование*, *продукционное программирование* и *императивное программирование* соответственно.

Исполнитель машин Тьюринга $\mathbf{I}_\mathrm{T}$ (или коротко — исполнитель Тьюринга) является простейшей вычислительной моделью (абстрактным вычислительным устройством) *автоматного программирования*. Одна из компонент этого устройства – читающая и пишущая головка – работает как конечный автомат с действиями. По сути, логика работы этой головки и называется машиной Тьюринга. Более точное определение будет дано ниже. Исполнитель машин Тьюринга может «исполнять» различные машины Тьюринга. Описание машины Тьюринга на некотором формальном языке является программой для этого исполнителя.

Исполнитель алгорифмов Маркова $\mathbf{I}_\mathrm{M}$ (или коротко — исполнитель Маркова) — это простейший исполнитель, работающий согласно *продукционной парадигме программирования*. Логика работы этого исполнителя задаётся как множество правил подстановки (продукций), согласно которым пошагово преобразуются входные данные. Этот набор правил и называется алгорифмом Маркова. Более точное определение будет дано на следующей лекции. Формальное описание алгорифма Маркова является программой для этого исполнителя.

Исполнитель машин Поста работает согласно *императивной парадигме программирования*. Программа для машины Поста — это описание последовательности действий, связанных условными и безусловными переходами. Принцип работы этого исполнителя довольно близок к принципу работы исполнителя Тьюринга и отдельно нами рассматриваться не будет.

Все три указанных исполнителя *эквивалентны*, то есть все функции, которые можно реализовать на одном из них, можно реализовать и на других. Понятие эквивалентности исполнителей является одной из важных тем данных лекций.

Кроме перечисленных трёх есть и другие абстрактные исполнители. В частности, есть λ-исчисление и соответствующий ему абстрактный исполнитель, работающий согласно *функциональной парадигме программирования*. Упомянутые четыре парадигмы программирования

и λ-исчисление выходят за рамки данного курса. Они частично рассмотрены в дополнительных лекциях.

Машина Тьюринга

Можно считать, что машина Тьюринга — это обобщённый конечный автомат, в который добавили

- специальное запоминающее устройство — бесконечную ленту, состоящую из ячеек памяти, в которых могут храниться символы;
- возможность совершать определённые действия во время переходов между состояниями, а именно, дигаться вдоль ленты, читать содержимое ячеек ленты и записывать в них новые значения.

Может создаться впечатление, что машины Тьюринга были придуманы как обобщение конечных автоматов. Но в действительности, принцип работы машин Тьюринга был придуман Аланом Тьюрингом как *формализация работы человека по точно определённому предписанию над точно заданной исходной информацией*. Алан Тьюринг сделал несколько весьма общих предположений о том, что такое процесс вычисления вообще и какие процессы можно назвать вычислимыми. Перечислим эти предположения в несколько вольном изложении.

- Во-первых, информация должна чётко представляться в виде конечного набора единиц. Единица информации — это бит или какой-либо другой элемент, который может принимать одно из возможных значений. Возможные значения строго заданы и их число конечно. Например, информация может представляться в виде стопки листков, на каждом из которых записан не более N символов из некоторого конечного фиксированного алфавита.
- Во-вторых, человек в один момент может распознать лишь конечное число единиц информации (даже если восприятие его столь мощно, что он единым взором охватывает целый лист, всё равно на этом листе может поместиться лишь конечное число символов).
- В-третьих, все замены, которые он делает в качестве элементарного шага, могут рассматриваться как замены одной единицы информации на другую (написал новую букву вместо старой, или новый лист вместо старого).
- В-четвёртых, хотя в хранилище информации может храниться сколько угодно единиц информации, прямой переход возможен лишь к соседней единице. Это предположение наименее тривиально на первый взгляд. Не только количество информации, обозреваемое в определенный момент времени, конечно, конечен и чётко зафиксирован набор единиц информации, к которым мы можем перейти от текущей.
- И, наконец, сам ум человека может принимать конечное число состояний.

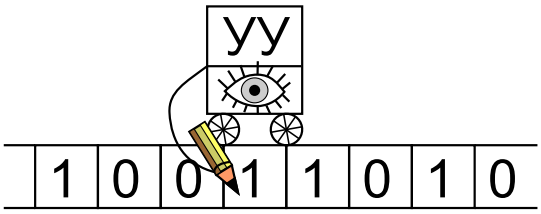


Рис. 2.1: Машина Тьюринга.

Эти предположения находят явное отражение в устройстве машины Тьюринга.

Машина Тьюринга (МТ) состоит из бесконечной в обе стороны **ленты**, разделенной на ячейки, и **управляющего устройства (УУ)**, способного находится в одном из конечного числа состояний (читающая/пишущая головка). Управляющее устройство стоит напротив одной из ячеек ленты. Эта ячейка называется **активной ячейкой**.

В каждой ячейке ленты хранится ровно один символ из заданного конечного набора символов — алфавита B'.

УУ может перемещаться влево и вправо по ленте, читать и записывать в ячейки ленты символы алфавита B' (один символ в каждой ячейке).

Выделяется особый пустой символ #, заполняющий все ячейки ленты, кроме тех из них (конечного числа), в которых записаны входные данные.

УУ работает согласно функции переходов π, которая задаёт алгоритм (логику) работы данной МТ. Функция переходов для текущего состояния s и наблюдаемого в текущей ячейке символа b задаёт, какой новый символ следует записать в эту ячейку, в какое новое состояние перейти и как переместиться — влево, вправо или остаться на месте.

Одно из состояний УУ помечено как начальное. Часть состояний УУ помечены как состояния останова. При переходе в одно из этих состояний работа МТ прекращается и слово, записанное на ленте, рассматривается как результат вычислений.

Входные данные — это слово, записанное на ленте в момент начала работы МТ. Оно записано в алфавите входных данных B, который является подмножеством расширенного алфавита B', с которым умеет работать УУ. Алфавит B' может содержать дополнительные символы, используемые для «служебных целей».

Здесь многократно указана конечность того или иного множества — конечный размер имеет алфавит, множество состояний УУ, и множество возможных перемещений УУ. Единственное, что в машине Тьюринга не имеет конца — это лента с ячейками.

Машины Тьюринга удобно задавать с помощью таблицы. Строчка этой таблицы задаёт состояние головки, а столбец — обозреваемый символ в текущей ячейке ленты. В ячейке таблицы пишется три сущности:

- следующее состояние головки,
- символ из B', который будет записан в ячейку ленты,
- действие — шаг вправо (R), шаг влево (L) или остаться на месте (S).

Следующая таблица задаёт МТ, которая добавляет единицу к натуральному числу, записанному на ленте в двоичной системе счисления.

	0	1	B
a	bR0	bR1	aR#
b	bR0	bR1	cL#
c	cL1	dL0	fS1
d	dL0	dL1	fS#

Число записано на некотором кусочке ленты, а остальные ячейки ленты заполнены специальным символом #. Головка изначально находится над ячейкой, где хранится старший разряд двоичной записи или левее.

Состояния 0, a, b, c, d, e имеют следующие смысловые значения:

- a — начальное состояние ($s_0 = a$);
- b — движение вправо до самого младшего разряда;
- d — осуществляется перенос единички; движение влево к старшему разряду;
- c — перенос единички закончился; движение влево к старшему разряду;
- f — состояние останова ($F = \{f\}$).

Как видите, машины Тьюринга позволяют многократно пробегать по данным. Число раз, которое головка будет находиться над некоторой ячейкой, читать и менять её содержимое может быть сколь угодно большим и зависеть от самих данных, в то время как в конечных автоматах нет возможности вернуться к прочитанным когда-то данным.

Таким образом, машина Тьюринга $\mathcal{M}$ задаётся шестью элементами:

$$\mathcal{M} = \{B, B', K, s_0, F, \pi\},\ B \subset B',\ s_0 \in K,\ F \subset K, \quad \pi : K \times B' \to K \times B' \times A$$

- алфавитом B, посредством которого описываются входные данные,
- расширенным алфавитом B', который включает в себя B, граничный символ # и возможно целый ряд других служебных символов,
- множеством состояний K,
- начальным состоянием s_0,
- множеством состояний останова F.
- функцией перехода $\pi : K \times B' \to K \times B' \times A$, которая по текущему состоянию и наблюдаемому символу определяет тройку: новое состояние, символ, который будет записан в ячейку, и действие из множества $A = \{R, L, S\}$ (шаг вправо, шаг влево, стоять на месте).

Алгоритм работы исполнителя машин Тьюринга $\mathbf{I}_\mathrm{T}$ описан в виде псевдокода 2.9.

Алгоритм 2.9 Схема работы исполнителя машин Тьюринга.

```
1: x ← s_0
2: while x ∉ F do
3:     b ← значение активной ячейки
4:     (x, b, a) ← π(x, b)
5:     значение активной ячейки ← b
6:     выполнить действие a
7: end while
```

Машины Тьюринга могут для некоторых входных данных никогда *не остановиться* и работать потенциально бесконечное время. В таких случаях говорят, что программа «зациклилась» или «зависла». Но следует отметить что при «зависании» УУ вовсе не обязательно начинает по циклу проходить одну и ту же последовательность состояний, то есть бесконечая последовательность последовательных состояний, принимаемых УУ, не всегда является периодической (с некоторым предпериодом). Поэтому термин «зацикливание» не всегда адекватно отображает причину «зависания».

Множество вычислимых функций. Свойство замкнутости

Исторически машина Тьюринга стала эталоном вычислимости — те функции, которые можно реализовать на машине Тьюринга, называются **вычислимыми**, а остальные функции называются **невычислимыми**. Есть эквивалентные по «вычислительным возможностям» исполнители (машина Поста, машина Маркова и др.), которые также могут служить *эталонами вычислимости.*

Тьюринг искал некоторое точное определение алгоритмической вычислимости, которое соответствовало бы интуитивному представлению о этом понятии.

Здесь стоит упомянуть про **тезис Тьюринга-Черча**, который связывает это неформальное интуитивное понятие *эффективно вычислимых функций* и функций, реализуемых в виде машины Тьюринга: «Класс эффективно вычислимых функции совпадает с классом функций, вычислимых на машине Тьюринга». Этот тезис не является ни теоремой, ни аксиомой, ни спорным утверждением, так как содержит неопределяемое интуитивное понятие.

Вспомним, что функции преобразования входных данных в выходные можно рассматривать как отображения вида $f : B^* \to B^*$ (вход и выход — слово в некотором конечном алфавите) или даже как $f : \mathbb{N} \to \mathbb{N}$ (вход и выход — натуральное число).

Под этим имеется в виду, что входные и выходные данные всегда можно закодировать конечной последовательностью единиц и нулей, которой, в свою очередь, легко сопоставляется натуральное число.

Определение 2.1. *Пусть дано некоторое множество объектов X и отображение $\phi : X \times X \to X$, которое паре элементов из X сопоставляет один элемент. Отображение такого вида называется* **бинарным оператором на множестве** X. *Операции сложения, умножения являются бинарными операторами на множестве чисел* ($\mathbb{R}$, $\mathbb{N}$, ...). *Говорят, что подмножество $X' \subset X$* **замкнуто** *относительно бинарного оператора ϕ, если для любых $a,\ b \in X'$ элемент $\phi(a, b) \in X'$, то есть результат применения бинарного оператора ϕ к элементам X' также является элементом X'. Например, множество натуральных чисел замкнуто относительно оператора сложения, но не замкнуто относительно оператора деления.*

При изучении какого-либо подмножества функций, важно определить, замкнуто ли оно относительно различных операторов. Множество вычислимых функций, как подмножество всех функций вида $f : \mathbb{N} \to \mathbb{N}$, в этом смысле обладает хорошими свойствами.

Утверждение 2.1. *Множество вычислимых функций* замкнуто *относительно операторов сложения, умножения и композиции. Это означает следующее: если $f : \mathbb{N} \to \mathbb{N}$ и $g : \mathbb{N} \to \mathbb{N}$ вычислимые функции, то функции*

1) $x \mapsto f(x) + g(x)$ *(обозначается коротко как $f + g$),*
2) $x \mapsto f(x) \cdot g(x)$ *(обозначается коротко как $f \cdot g$),*
3) $x \mapsto f(g(x))$ *(обозначается коротко как $f \circ g$),*

тоже являются вычислимыми. Другими словами, сумма, произведение и композиция двух вычислимых функций есть вычислимая функция.

Доказательство этого утверждения мы приводить не будем. Обратите внимание на то, что в машинах Тьюринга нет элементарных операций, связанных с арифметикой сколь угодно больших натуральных чисел. Но тем не менее, множество вычислимых функций имеют хорошие арифметические свойства — вычислимые функции можно умножать и складывать и снова получать вычислимые функции.

То, что композиция вычислимых функция является снова вычислимой функцией, означает, в частности, следующее. Пусть даны две машины Тьюринга $\mathcal{M}_f$ и $\mathcal{M}_g$ с одним и тем же алфавитом B записи входных и выходных данных, которые реализуют соответственно функции $f : B^* \to B^*$ и $g : B^* \to B^*$. Тогда существует машина Тьюринга $\mathcal{M}_h$, которая реализует функцию $h = f \circ g$, то есть преобразует входные данные так, как они были бы преобразованы в результате работы двух машин — сначала машины $\mathcal{M}_g$, а затем машины $\mathcal{M}_f$. Если во время работы хотя бы одна из этих машин «зависает», то и машина $\mathcal{M}_h$ должна «зависать».

Машина $\mathcal{M}_h$ существует и, более того, существует алгоритм, который по описанию машин $\mathcal{M}_f$ и $\mathcal{M}_g$ получает описание машины $\mathcal{M}_h$. Хорошим упражнением является разработка такого алгоритма.

Частично рекурсивные функции

Есть альтернативное эквивалентное определение вычислимых функций. Это определение основано на *замыкании* некоторого базового набора функций относительно заданного набора операторов.

Рассматриваются натуральнозначные функции от нескольких натуральных аргументов, то есть функции вида $f : \mathbb{N}^k \to \mathbb{N}$ для различных значений k.

Среди них выделяются базовые: функции, которые при всех значениях аргументов равны 0, функции, которые в точности равны одному из своих аргументов, и функция увеличения на единицу $i(x) = x + 1$.

Определение 2.2. *Множество* **частично рекурсивных функций** *определяется как минимальное множество функций, содержащее указанные простейшие функции и, кроме того, замкнутое относительно оператора композиции, оператора примитивной рекурсии, и оператора взятия минимума аргумента.*

Поясним это определение.

Оператор композиции уже нам известен. Необходимо только обобщить его на функции многих переменных. Новая функция получается из двух функций в результате подстановки одной в качестве одного из аргументов другой. Так, имея функции НОД(a, b) и 2^n, можно получить новую функцию $g(n, b) =$ НОД$(2^n, b)$.

Оператор взятия минимума аргумента имеет простой смысл. Пусть дана некоторая функция f от нескольких аргументов. Зафиксируем значения всех аргументов в этой функции, кроме одного. Новая функция ψ определяется как минимальное значение этого аргумента, при котором $f = 0$.

Оператор примитивной рекурсии устроен следующим образом. Пусть есть функции $h(x)$, $g(x, y, n)$. Тогда применение к ним оператора примитивной рекурсии даёт функцию $f(x, n)$, определяемую рекурсивно через своё значение $f(x, n-1)$. Определение состоит из начального условия (I) и шага рекурсии (S):

$$\begin{aligned} &\text{(I)}: \ f(x, 0) = h(x) \\ &\text{(S)}: \ f(x, n) = g(x, \ f(x, n-1), \ n-1) \end{aligned}$$

Отметим, что здесь все переменные принимают целые неотрицательные значения. При этом под x может пониматься несколько аргументов: $x = (x_1, x_2, \ldots, x_m)$.

Если в определении 2.2 исключить оператор взятия минимума аргумента, то получится класс **примитивно рекурсивных функций**.

С точки зрения программистов, примитивно рекурсивные функции — это все функции, которые можно запрограммировать используя арифметические операции, условный оператор **if** и оператор арифметического цикла **for** (циклы, к началу исполнения которых чётко зафиксировано число итераций цикла, и это число конечное) и нельзя использовать оператор **while**.

Оператор примитивной рекурсии аналогичен арифметическому циклу, в котором число итераций заранее известно.

А оператор взятия минимума аргумента естественным образом программируется в виде цикла **while**, для которого заранее не известно, сколько потребуется шагов. Используя в программах оператор **while**, мы можем выйти за пределы примитивно рекурсивных функций.

Отметим, что если какая-либо задача решается с помощью цикла **while**, то это не значит, что она не является примитивно рекурсивной. Функция не является примитивно рекурсивной тогда, когда не существует реализующей её программы, не использующей цикла **while**.

Какие из следующих функций не являются примитивно рекурсивными?
- Вход: натуральное число n. Выход: наибольшее простое число меньшее n.
- Вход: натуральное число n. Выход: наименьшее простое число большее n.
- Вход: натуральное число n. Выход: все делители числа n.
- Вход: многочлен от $P(x)$ с целыми коэффициентами. Выход: все целые корни этого многочлена.

Ответ: все указанные функции являются примитивно рекурсивными.

Для частично рекурсивных функций, также как и в случае функций, вычислимых на машине Тьюринга, есть возможность «зависания». Это возможность связана с применением оператора взятия минимума аргумента. Во время «выполнения» этого оператора ищется минимальное значение x, при котором $f(x) = 0$. Если такого x нет, то результат неопределён. Таким образом, частично рекурсивные функции могут быть определены не при всех значениях аргументов. Частично рекурсивные функции, которые определены для всех значений аргументов, называются **общерекурсивными функциями**.

Примитивно рекурсивные функции по построению определены для всех значений аргументов, то есть не равны **undef** ни при каких аргументах. Но при этом, они не совпадают с множеством общерекурсивных функций. Довольно сложно предъявить общерекурсивную функцию, которая не является примитивно рекурсивной.

Верно следующее утверждение:

Утверждение 2.2. *Множество частично рекурсивных функций вида* $f : \mathbb{N} \to \mathbb{N}$ *с точностью до кодирования входных и выходных данных совпадает с множеством функций вида* $g : B^* \to B^*$*, вычислимых на машине Тьюринга*[1].

Машины Тьюринга на входе и выходе имеют слово в некотором конечном алфавите B, а частично рекурсивные функции определены на наборах натуральных чисел (элементов множества $\mathbb{N}^k$) и возвращают одно натуральное число. Здесь взяты частично рекурсивные функции с $k = 1$.

[1]Класс рекурсивных функций впервые был описан Гёделем (1931 г.) как класс всех числовых функций, выразимых в некоторой формальной системе. Пятью годами позже Чёрч опубликовал свою гипотезу о том, что класс рекурсивных функций совпадает с классом вычислимых функций. Ряд работ по уточнению понятия рекурсивных функций есть у Клини (1936 г.).

Под кодированием имеется в виду одно из простых отображений натуральных чисел в множество слов некоторого конечного алфавита B. В роли такого отображения может выступать представление натуральных чисел в одной из систем счисления или другие представления, которые могут быть получены на машине Тьюринга из двоичного представления.

Элементы множеств $\mathbb{N}^k$, а также $\mathbb{N}^*$ также можно взаимооднозначно закодировать словами (например, записать представления чисел через запятую). Поэтому в данном утверждении можно было не ограничиваться числом $k = 1$.

Задача Л2.1. Пусть дана вычислимая функция $f : \mathbb{N} \to \mathbb{N}$. Определим функцию $g : \mathbb{N} \to \mathbb{N}$, которая на элементе x равна минимальному значению y такому, что $f(y) = x$. Если такого элемента y не существует, то $g(x) = \mathbf{undef}$. Докажите, что функция g также вычислима.
Подсказка: по сути, необходимо показать, что обратная функция вычислимой функции вычислима. Постройте псевдокод, который вычисляет функцию g, вызывая данную функцию f. Придумайте как с помощью операторов взятия минимума аргументов и других операторов, используемых в определении частично рекурсивных функции, построить функцию g, используя при построении функцию f.

Подробнее о рекурсивных функциях можно прочитать в трудах Успенского В.А., Колмогорова А.Н. и Клини С.К.

Проблема останова машины Тьюринга

Возможность «зависания» — неотъемлемое свойство вычислимых функций.

Оказывается, нет алгоритма, который по описанию алгоритма вычисления функции (по псевдокоду, по описанию машины Тьюринга, по описанию построения частично рекурсивной функции) определял бы, «зависает» она или нет для фиксированных входных данных.

Сформулируем проблему останова для машин Тьюринга.

Определение 2.3. *Если на входном слове x машина Тьюринга $\mathcal{M}$ останавливается, то говорят, что* **машина $\mathcal{M}$ применима к** x.

Задача Л2.2. (Проблема останова) Для произвольной машины Тьюринга $\mathcal{M}$ и слова x определить, применима машина $\mathcal{M}$ к x или нет.

Сможет ли человек по описанию машины Тьюринга определить, остановится когда-нибудь эта машина при входных данных x или нет? Есть ряд критериев, позволяющих обнаружить и проанализировать причины возможных «зависаний». Но хватит ли этих критериев? Всегда ли может человек *наверняка* проверить (доказать), что данная машина Тьюринга на указанных входных данных остановится (не остановится)? Если может, то хотелось бы получить формализованное описание алгоритма проверки (генерации доказательства).

Программисты по описанию алгоритма для какого-либо исполнителя часто «видят» места, которые могут привести к «зависанию». В обычных алгоритмах (алгоритмах, записанных псевдокодом) причиной «зависания» может быть бесконечная цепочка рекурсивных вызовов или цикл **while**, условие которого выполнено всегда. У программистов-практиков может возникнуть ощущение, что каждый из случаев использования рекурсии и циклов можно проанализировать и определить, существуют входные данные, которые приводят к «зависанию», или нет. Но оказывается, это не так. Ни в случае исполнителя псевдокода, ни в случае исполнителя машин Тьюринга, алгоритма решения проблемы останова не существует. Эту проблему

можно переформулировать так: нет алгоритма, который по описанию частично рекурсивной функции может определить, является ли она общерекурсивной. Поэтому говорят, что множество общерекурсивных функций *алгоритмически неотделимо*. Подробнее проблему останова мы рассмотрим на следующей лекции.

Семинар 2

Машины Тьюринга

Краткое описание: В семинаре отрабатываются методы построения машин Тьюринга. Основной принцип заключается в разбиении процесса вычисления на этапы и использовании для каждого этапа своего множество состояний.

Как было отмечено в лекциях, результат вычисления можно рассматривать как преобразование входного слова в выходное слово. Слово (строка символов) выбрана в качестве представления аргумента и значения функции по тем соображениям, что любые входные данные (конечный набор чисел, изображения, звуковые данные) можно закодировать в виде последовательности символов.

Напомним вкратце, что представляет собой машина Тьюринга (МТ). Одной из частей МТ является *лента*, на которой записывается аргумент, и с которой читается полученное значение. Лента является бесконечной в обе стороны последовательностью ячеек, в каждой из которых может храниться один символ. Другой важной частью машины является *головка* или *устройство управления* (УУ), которая в каждый момент времени, называемый *тактом*, может находиться только на определенной позиции на ленте и пребывать в одном из нескольких *состояний*. Также УУ может читать и писать символ из той позиции ленты, на которой он находится. Каждый такт УУ перемещается влево или вправо на соседнюю позицию на ленте. Программой для машины Тьюринга называется отображение, которое каждой паре (состояние, читаемый символ) ставит в соответствие тройку (новое состояние, записываемый символ, направление перемещения).

Для описания МТ используют форму записи в виде таблицы. В левом столбце записывают все состояния УУ, а в верхней строчке — все символы алфавита. В ячейках таблицы записывается тройка (записываемый символ, направление перемещения, номер нового состояния).

Алфавит B, указываемый в задачах, — это алфавит, в котором записывается входное слово. Расширенный алфавит B' включает в себя алфавит B и содержит символ пробела ␣, которым заполнены ячейки ленты, не содержащие символов входного слова. В расширенный алфавит B' можно включать произвольное количество дополнительных символов, если в задаче не оговорены специальные условия.

Практика конструирования машин Тьюринга

Единичная нотация целого неотрицательного числа n — это слово, состоящее из цифры 0 и следующих за ней n цифр 1. Например:

число	0	1	2	3	4	5	...
представление	0	01	011	0111	01111	011111	...

В этой нотации запись любого числа начинается с ноля (который называется лидирующим нулём), а затем идёт нужное число единиц.

! Везде далее, если явно не указано множество чисел, имеется в виду множество целых неотрицательных чисел.

Задача C2.1. Реализуйте МТ, которая осуществляет сложение двух чисел, записанных на ленте в единичной нотации. В начальном положении головка указывает на начало первого числа. Ниже представлены примеры входных и выходных слов:

stdin	stdout
0101	011
0110	011
00	0

В данной задачи необходимо просто удалить из входного слова второй ноль (лидирующий ноль второго числа). Это можно сделать, осуществив перенос группы единиц, которая идёт после второго нуля, на одну ячейку влево. Более эффективное решение следующее: заменяем второй ноль на единицу, затем идём вправо до первого пробела, затем первую единицу перед ним заменяем на пробел.

Рассмотрим подробнее решение этой задачи. Для начала нужно сформулировать решение данной задачи в виде комбинации простых действий, которые бы описывались одним-двумя состояниями машины. Например:

A — двигаться вправо по записи первого числа и заменить лидирующий ноль второго числа его на 1;

B — двигаться вправо по единицам второго числа до пробела;

C — шагнуть влево и заменить стоящую там единицу на пробел;

D — встать на начало первого числа;

E — закончить работу.

Теперь нужно составить последовательность прохождения данных блоков таким образом, чтобы конечное состояние каждого блока было начальным состоянием последующего.

Итак, опишем все наши действия «микропрограммами», соответствующими нашим блокам. Для каждой микропрограммы будем использовать свой набор состояний, обозначаемых как имя блока с индексом.

Состояние	#	0	1	Комментарии
A_1		0 R A_2		шагаем вправо через лидирующий ноль первого числа;
A_2		1 R B_1	1 R A_2	двигаемся вправо до лидирующего ноля второго числа и заменяем его на 1:
B_1	# L C_1		1 R B_1	двигаемся вправо до первого пробела;находим пробел и делаем шаг влево;
C_1			# L D_1	заменяем 1 на 0;
D_1		0 L E	1 L D_1	двигаемся влево до лидирующего ноля;

В получившейся таблице некоторые ячейки пустуют. Мы будем допускать такие ситуации, хотя это не согласуется с точным определением машины Тьюринга. Будем считать, что если во время вычисления возникла пара (состояние, символ), которой соответствует пустая ячейка таблицы, то происходит останов выполнения с сообщением об ошибке.

Задача С2.2. Рассмотрите варианты предыдущей задачи с различным начальным положением головки относительно чисел:
а) напротив первого символа записи второго числа;
б) напротив последнего символа записи второго слова.

Задачи С2.3–С2.11: *Описать машину Тьюринга, которая реализует*:

Задача С2.3. (счетчик чётности) Следующую логику работы: вход — слово в алфавите $B = \{\texttt{0}, \texttt{1}\}$; выход равен `0` или `1` в зависимости от того, чётно или нечётно число единиц в данном слове. В начальном состоянии УУ стоит напротив первого левого символа входного слова.

Задача С2.4. (переворачивание слова) Переворачивание слова в алфавите $B = \{\texttt{a}, \texttt{b}\}$. Например, для входа `aabab` должен получится результат `babaa`.

Задача С2.5.* Перемножение двух чисел, заданных в единичной нотации (числа записаны на ленте подряд).

Задача С2.6.* Вычисление квадрата числа, заданного в единичной нотации.

Задача С2.7. Увеличение числа, заданного в двоичной системе счисления, на 3.

Задача С2.8. Сложение двух чисел, заданных в двоичной системе счисления.

Задача С2.9. Перевод двоичной записи числа в единичную нотацию.

Задача С2.10. Перевод единичной нотации числа в двоичную запись.

Задача С2.11.* Перевод десятичной записи числа в двоичную. Предварительно опишите идею алгоритма и оцените требуемое число состояний УУ. Как растёт время работы предложенного алгоритма с ростом длины входного слова?

Задача С2.12. Опишите схему построения МТ, которая получает на вход арифметическое выражение над рациональными числами, записанными в десятичной системе счисления, со скобками и операциями `+`, `-`, `*`, `/` и вычисляет значение этого выражения. На какие шаги (подзадачи) можно разбить построение этой МТ?

Теоретические задачи

Задача С2.13. Реализуйте МТ и укажите для неё входные данные (слово конечной длины), к которым она не применима, и при этом последовательность состояний, принимаемых УУ, ни в какой момент не становится периодической.

Задача С2.14. Пусть даны две машины Тьюринга $\mathcal{M}_f$ и $\mathcal{M}_g$ (программы для исполнителя машин Тьюринга) с входным алфавитом символов $B = \{0, 1, A\}$. Первая реализует функцию $f : B^* \to B^*$, а вторая — $g : B^* \to B^*$. Укажите, как сконструировать машину Тьюринга $\mathcal{M}_h$, которая реализует функцию $h = f \circ g$, то есть такую функцию, которая для входных данных x

возвращает $f(g(x))$. В случае «зависания» $\mathcal{M}_f$ или $\mathcal{M}_g$ считается, что результат равен **undef**[1]. Решив эту задачу, вы докажете, что композиция вычислимых функции является вычислимой функцией.

Задача C2.15. Существует ли МТ, которая определяет простоту заданного числа?

Задача C2.16. Существует ли МТ, которая получает десятичную запись натурального числа и возвращает список всех его делителей, разделённых запятой?

Задача C2.17. Существует ли МТ, которая получает на вход описание другой МТ (в каком-либо виде) и данные для неё и останавливает свою работу только в том случае, если данная машина Тьюринга не применима к указанным входным данным?

Задача C2.18. Существует ли МТ, которая эмулирует работу конечного автомата, а именно, получает на вход описание конечного автомата и входные данные для него и возвращает конечное состояние, в котором будет находится эмулируемый конечный автомат после обработки указанных входных данных?

Задача C2.19. Оцените длину описания МТ (произведение длины алфавита на число состояний), которая складывает два числа в двоичной, восьмиричной и десятичной системах счисления? Какова будет длина программы для сложения чисел, заданных в римской системе счисления?

[1]Для удобства считают, что функции, реализуемые вычислительными машинами, имеют вид $f : G \to G$, где G — это множество всех слов некоторого конечного алфавита дополненное специальным элементом **undef**, соответствующим неопределённому значению. При этом всегда считают, что $f :$ **undef** $\mapsto$ **undef**, то есть $f($**undef**$) =$ **undef**.

Лекция 3

Алгорифмы Маркова. Вычислительная эквивалентность исполнителей

Краткое описание: На этой лекции мы рассмотрим ещё одного абстрактного исполнителя — исполнителя алгорифмов Маркова, принцип работы которого сильно отличается от исполнителя машин Тьюринга. Алгорифмы Маркова осуществляют преобразование данного слова с помощью последовательности простых замен, согласно заданному набор правил подстановок. Возможности исполнителя алгорифмов Маркова и исполнителя машин Тьюринга, на первый взгляд, должны различаться. Но оказывается, эти исполнители эквивалентны, то есть все функции, которые можно реализовать в виде машины Тьюринга, реализуются в виде алгорифма Маркова, и наоборот. Но при этом сложности решения задач на этих двух исполнителях различаются. Некоторые задачи проще решаются на исполнителе машин Тьюринга, другие — на исполнителе алгорифмов Маркова наоборот. Мы приведём ещё одно эквивалентное определение вычислимых функций и в более общем виде сформулируем проблему останова. Разберём схему доказательства алгоритмической неразрешимости проблемы останова и рассмотрим ещё две алгоритмически неразрешимых задачи: проблему эквивалентности программ и проблему определения разрешимости Диафантового уравнения.
Ключевые слова: алгорифмы Маркова и исполнитель алгорифмов Маркова, вычислимые функции, эквивалентность исполнителей, проблема эмуляции исполнителя, проблема останова программы, проблема эквивалентности программ.

Алгорифмы Маркова

Вычисления, основанные на правилах, называются продукционными. Исполнитель алгорифмов Маркова, обозначаемый как $\mathbf{I}_{\mathrm{M}}$, — это простейший исполнитель, осуществляющий продукционные вычисления.

В памяти исполнителя $\mathbf{I}_{\mathrm{M}}$ хранится одно слово X конечной длины, составленное из букв конечного алфавита. Оно изначально равно входному слову, а после останова исполнителя — выходному слову. Преобразование слова X на исполнителе $\mathbf{I}_{\mathrm{M}}$ происходит путём последовательности преобразований — замен одного подслова на другое. Возможные замены описываются **правилами подстановки**.

Упорядоченное множество правил подстановки задаёт логику работы исполнителя $\mathbf{I}_{\mathrm{M}}$. Эти множества правил называются **нормальными алгорифмами Маркова**, но для краткости мы будем называть их алгорифмами.

Пусть b и b' — некоторые слова в алфавите B. Правило «$b \to b'$» означает, что первое слева вхождение подслова b в слове X следует заменить на слово b'.

Некоторые правила могут быть помечены как **конечные**. Для этих правил рядом со стрелкой мы будем ставить точку: «$b \to .\ b'$».

Алгорифм Маркова исполнитель $\mathbf{I}_\mathrm{M}$ исполняет пошагово. На каждом шаге, слово X, записанное на ленте, модифицируется следующим образом. Рассматривается первое правило и проверяется, может ли оно быть применено. **Правило** «$b \to b'$» считается **применимым**, если слово X содержит подслово b. Если первое правило не может быть применено, рассматривается следующее правило, и т.д. Если какое-либо правило было применено, то на следующем шаге исполнитель $\mathbf{I}_\mathrm{M}$ снова начинает с первого правила. Если применяется конечное правило, то работа исполнителя останавливается.

Если из всего списка правил ни одно не может быть применено к текущему слову, то работа исполнителя также останавливается.

Алгоритм работы исполнителя $\mathbf{I}_\mathrm{M}$ формально можно представить в виде псевдокода 3.10. Число n — это длина списка правил подстановок $\{P_0, \ldots, P_{n-1}\}$.

Алгоритм 3.10 Алгоритм работы исполнителя алгорифмов Маркова.

```
i ← 0
X ← X_0
while true do
    if P_i применимо к X then
        X ← P_i(X)
        if P_i конечное then
            завершение работы
        end if
        i ← 0
    else
        i ← i + 1
        if i = n then
            завершение работы
        end if
    end if
end while
```

Рассмотрим задачу сложения двух чисел в «унарной» системе счисления и решающий её алгорифм Маркова.

Задача Л3.1. (Сложение чисел в «унарной» системе счисления) Пусть числа на ленте задаются в следующем виде: $0 =$ `0`, $1 =$ `01`, $2 =$ `011`, $3 =$ `0111`, ... Два числа в исходном слове записаны подряд. Например, для входного слова `0111011` результат должен быть равен `011111`, а для `010` — `01`.

Решение задачи сводится к удалению цифры ноль, с которой начинается запись второго числа. Это можно осуществить путём замены пары символов `10`, на `1`. Соответствующий алгорифм будет содержать одно правило `10` $\to .$ `1`.

Действительно, `10` может встретиться только один раз на границе чисел. Это стоп-правило и после его применения работа исполнителя завершится. Но необходимо ещё рассмотреть случай, когда первое число равно 0 и записано как `0`. Для этого особого случая следует записать ещё одно правило `00` $\to .$ `0`, и в итоге получится алгорифм из двух правил:

```
1. 10  →.  1
2. 00  →.  0
```

Рассмотрим более сложный пример — уже известную нам задачу увеличения данного неотрицательного целого числа на 1.

Задача Л3.2. Напишите алгорифм Маркова, который увеличивает данное натуральное число, записанное в двоичной системе счисления, на единицу. Вход: двоичная запись целого неотрицательного числа n (слово в алфавите $B = \{0, 1\}$). Выход: двоичная запись натурального числа $n + 1$.

Для решения нам придётся расширить входной алфавит до алфавита $B' = \{\mathtt{0}, \mathtt{1}, \mathtt{A}, \mathtt{B}\}$.

Буква `A` будет служить нам курсором. Увеличивать число на единицу мы будем обычным образом, начиная с младшего разряда (самого правого). Для начала допишем в конец слова букву `A`. Для этого нехитрого действия на исполнителе $\mathbf{I}_\mathrm{M}$ необходимо целых три правила:

```
1. A1  →  1A
2. A0  →  0A
3. ∅   →  A
```

Здесь ∅ обозначает пустое слово, то есть слово, в котором нет ни одной буквы. Возьмём для примера слово `1011`. С помощью этих правил оно будет подвергнуто следующим пошаговым преобразованиям:

```
A1011    // сработало правило 3 (в начало добавилась буква A),
         // пустое слово "присутствует" вначале каждого слова;
1A011    // сработало правило 1: подслово A1 заменилось на 1A;
10A11    // сработало правило 2: подслово A0 заменилось на 0A;
101A1    // сработало правило 1;
1011A    // сработало правило 1;
```

Если продолжить пошаговые вычисления, то снова сработает правило 3 и в начале слова появится буква `A`, которая на следующих шагах под действием правил 1 и 2 будет перемещаться в конец слова. Затем снова вначале появится буква `A`, снова и так далее. Чтобы этого не происходило, допишем следующие правила.

Нам необходимо двигаться в право, заменяя единицы на нули, а как только появится первый ноль, обратить его в единицу и закончить вычисления. Для этого можно было бы записать такие правила:

```
1A  →   A1
0A  →.  1
```

Но вот вопрос: куда их поместить — до, или после уже имеющихся трёх правил. Оказывается, оба варианта плохи. В первую очередь, необходимо заменить тип курсора — заменить букву `A` на другую букву, например, букву `B`. Буква `B` также будет играть роль курсора и соответствовать второй стадии процесса вычислений. Полный набор правил выглядит так:

```
1. 1B  →   B0
2. 0B  →.  1 // это стоп-правило
3. A1  →   1A
4. A0  →   0A
5. A   →   B
6. ∅   →   A
```

Правильный порядок расположения правил часто сильно расходится с порядком, в котором они применяются. В данном случае сначала срабатывает правило 6, потому что первые пять

правил не могут быть применены. В правой части первых пяти правил есть либо буква A, либо буква B, и поэтому они неприменимы к слову, состоящему из единиц и нулей. Правило 6 добавляет в начало слова букву A. Затем начинают срабатывать правила 3 и 4 (буква A перемещается в конец), затем срабатывает правило 5 (буква A заменяется на B), затем начинает работать правило 1 (заменяем все последние единицы на нули «перепрыгивая» через них буквой B), и, в конце концов, срабатывает завершающее правило 2 (заменяем первый справа ноль на единицу):

```
1011  // данное слово
A1011 // сработало правило 6;
1A011 // сработало правило 3;
10A11 // сработало правило 2;
101A1 // сработало правило 3;
1011A // сработало правило 3;
1011B // сработало правило 3;
101B0 // сработало правило 1;
10B00 // сработало правило 1;
1100  // сработало стоп-правило 2, вычисления прекратились.
```

Правильный ли результат получит данный алгорифм Маркова для входных слов 0, 1, 111111111, 111111000? Добавьте несколько правил, чтобы алгорифм всегда работал правильно.

Ответ на данный вопрос следующий: для слов, состоящих из одних единиц приведённый алгорифм работает неверно — он на них «зависает», то есть неприменим к ним. Для того, чтобы это исправить, необходимо в добавить в начало списка правил правило AB $\to$. 1.

Итак, с задачей увеличения числа на единицу мы справились. Насколько широк класс вычислительных задач, которые можно решить с помощью алгорифмов Маркова? Оказывается он совпадает с классом задач, решаемых с помощью машины Тьюринга.

Эквивалентность и полнота различных исполнителей

Везде далее будем считать, что задан некоторый формальный язык описания алгорифмов Маркова, и что эти описания являются *программами* для *исполнителя алгорифмов Маркова* $\mathbf{I}_{\mathrm{M}}$. Перед началом работы в исполнитель $\mathbf{I}_{\mathrm{M}}$ закладывается программа, являющаяся описанием алгорифма Маркова, после чего он начинает работать так, как указанный алгорифм Маркова.

Аналогично, машины Тьюринга будем отображать в программы для *исполнителя машин Тьюринга* $\mathbf{I}_{\mathrm{T}}$. Эти программы являются описаниями машин Тьюринга на некотором формальном языке. Логику работы машины Тьюринга можно записывать в виде таблицы, описывающей функцию π переходов между состояниями, а эту таблицу можно представить в виде одного слова в некотором конечном алфавите, используя специальные разделительные символы для отделения ячеек и строк таблицы. Более точно, программа для исполнителя $\mathbf{I}_{\mathrm{T}}$ должна представлять собой описание пяти элементов $\{K, B, B', \pi, s_0, F\}$. Какую из возможных формальных нотаций взять не так важно. Важно, чтобы эта нотация однозначно описывала

указанные пять элементов и была достаточно простой (задача интерпретации этого описания должна быть разрешимой).

В переходе от алгорифмов Маркова и машин Тьюринга к программам для соответствующих исполнителей есть тонкий момент, связанный с описанием алфавитов B и B'. Дело в том, что у исполнителей алфавит должен быть конечным и фиксированным. Поэтому необходимо либо заранее ограничится некоторым алфавитом B' и не рассматривать другие алфавиты, либо выбрать некоторый конечный алфавит B_0 и договорится о способе отображения (кодирования) букв произвольного алфавита B' в слова в алфавите B_0. Будем считать, что выбран второй вариант[1], но это не принципиально.

После сделанного замечания можно считать, что на обоих исполнителях реализуются функции вида

$$f : B^* \to B^* \cup \{\mathbf{undef}\},$$

то есть функции, которые получают на вход слово в алфавите B, и возвращают слово в алфавите B или **undef**. Значению **undef** соответствует случай, когда программа «зависает». Алфавит B может быть любым непустым конечным множеством.

Для удобства будем считать, что неопределённое значение **undef** может присутствовать и на входе. При этом по определению положим, что $f(\mathbf{undef}) = \mathbf{undef}$. Множество всех функций вида

$$f : B^* \cup \{\mathbf{undef}\} \ \to \ B^* \cup \{\mathbf{undef}\}$$

условно обозначим как $\mathcal{A}$.

Определение 3.1. **Исполнители $\mathbf{I}_1$ и $\mathbf{I}_2$ эквивалентны**, *если у них совпадают множества всех задач и подмножества решаемых задач. Исполнители, которые эквивалентны исполнителю машин Тьюринга* $\mathbf{I}_\mathrm{T}$ *, называются* **Тьюринг-эквивалентными**.

Множество всех задач исполнителей машин Тьюринга и алгорифмов Маркова совпадают — это задачи реализации функций $\mathcal{A}$. Лишь часть функций из множества $\mathcal{A}$ можно реализовать

[1]В исполнителях количество элементарных объектов должно быть конечно, и, в частности, алфавит, в котором описываются алгорифмы Маркова и машины Тьюринга для соответствующих исполнителей, должен быть конечным.

Но алфавит B записи входных данных и расширенный алфавит B' алгорифмов Маркова и машин Тьюринга не зафиксированы и могут быть произвольными. Указанные алфавиты являются частью их описания.

Алфавит, «понимаемый» исполнителем, нельзя сделать таким, чтобы он включал все возможные алфавиты.

Чтобы разрешить указанную проблему следует договориться о способе записи букв произвольного абстрактного алфавита в виде слов некоторого конечного алфавита. Так в компьютерах буквы русского алфавита кодируются бинарными словами. Описание алфавитов B и B' будет заключаться в указании размера алфавита B и количества дополнительных символов алфавита B'. Буквы абстрактного алфавита B' можно произвольным образом упорядочить и i-й букве назначить кодовое слово, состоящее из специальной буквы `X`, и двоичной записи числа i. Входные и выходные данные будут записываться в виде последовательности не самих букв, а кодов букв алфавита. Алфавит $B_0 = \{\mathtt{X}, \mathtt{0}, \mathtt{1}\}$ при этом будет универсальным алфавитом, в котором кодируются входные и выходные данные для произвольных алгорифмов Маркова и машин Тьюринга.

на исполнителе $\mathbf{I}_\mathrm{T}$. Обозначим такие функции как $\mathcal{C}_\mathrm{T}$. Лишь часть функций из множества $\mathcal{A}$ можно реализовать на исполнителе $\mathbf{I}_\mathrm{M}$. Обозначим такие функции как $\mathcal{C}_\mathrm{M}$. Оказывается, $\mathcal{C}_\mathrm{T} = \mathcal{C}_\mathrm{M}$, то есть исполнители $\mathbf{I}_\mathrm{M}$ и $\mathbf{I}_\mathrm{T}$ эквивалентны.

1. Исполнители $\mathbf{I}_\mathrm{T}$ и $\mathbf{I}_\mathrm{M}$ эквиваленты. Реализуемые на этих исполнителях функции называются **вычислимыми** и обозначаются как $\mathcal{C}$.

2. Лишь часть функций из множества $\mathcal{A}$ являются вычислимыми, то есть $\mathcal{C} \subset \mathcal{A}$ и $\mathcal{C} \neq \mathcal{A}$.

Кроме терминов «вычислимые» и «невычислимые функции» часто используют термины «**алгоритмически разрешимые**» и «**алгоритмически неразрешимые задачи**».

Мы не будем приводить доказательства первого утверждения. Для его доказательства достаточно сконструировать *эмулятор* исполнителя $\mathbf{I}_\mathrm{M}$ алгорифмов Маркова на исполнителе $\mathbf{I}_\mathrm{T}$ машин Тьюринга и эмулятор исполнителя $\mathbf{I}_\mathrm{T}$ на исполнителе $\mathbf{I}_\mathrm{M}$[2]. Это технически сложная, но осуществимая задача. О том, что такое эмуляция и эмулятор мы расскажем ниже.

Второе утверждение называют иногда **свойством неполноты исполнителя машин Тьюринга**. Полным принято называть такой набор операций или такой набор возможностей исполнителя, с помощью которого можно реализовать всё множество функции выбранного вида. Возможностей исполнителей $\mathbf{I}_\mathrm{T}$ и $\mathbf{I}_\mathrm{M}$ не хватает для того, чтобы реализовать все функции $\mathcal{A}$. Более того, не существует исполнителей с конечным числом элементарных объектов и элементарных действий, которые были бы полны в классе $\mathcal{A}$. Исполнитель машин Тьюринга задаёт определенный «потолок», через который нельзя «перепрыгнуть» конечным исполнителям (исполнителям с конечным числом элементарных объектов и элементарных действий). Исполнителей, которые эквивалентны машине Тьюринга, называют **Тьюринг-полными** или **Тьюринг-эквивалентными**.

В том, что существуют невычислимые функции, несложно убедится с помощью сравнения мощностей множества $\mathcal{A}$ всех функций вида $f : B^* \cup \{\mathbf{undef}\} \to B^* \cup \{\mathbf{undef}\}$ и множества программ исполнителя $\mathbf{I}_\mathrm{T}$ (или $\mathbf{I}_\mathrm{M}$). Гораздо сложнее предъявить конкретную функцию, которую нельзя реализовать в виде алгорифма Маркова или машины Тьюринга.

Докажем, что существуют невычислимые функции. Множество $\mathcal{A}$ несчётно. Действительно, множество $B^* \cup \{\mathbf{undef}\}$ счётно, поэтому мощность множества $\mathcal{A}$ совпадает с мощностью множества функций вида $f : \mathbb{N} \to \mathbb{N}$, то есть множеством всех бесконечных натуральнозначных последовательностей. Таких последовательностей несчётное количество[3]. В то же время, множество вычислимых функций счётно. Действительно, каждой вычислимой функции соответствует машина Тьюринга. Как

[2]Программа для исполнителя $\mathbf{I}_\mathrm{T}$, которая эмулирует исполнителя $\mathbf{I}_\mathrm{M}$, получает на вход программу A для исполнителя $\mathbf{I}_\mathrm{M}$ и входное слово x, а возвращает результат вычислений, который вернул бы алгорифм Маркова, соответствующий программе A, на входных данных x. При этом если этот алгорифм Маркова «зависает» при на этом входных данных то и программа-эмулятор должна «зависнуть». Ясно, что если существует эмулятор исполнителя $\mathbf{I}_1$ на исполнителе $\mathbf{I}_2$ и есть программа, решающая некоторую задачу на исполнителе $\mathbf{I}_1$, то автоматически получаем решение этой задачи на исполнителе $\mathbf{I}_2$.

[3]Это доказывается на курсе дискретной математики. Самое популярное доказательство основано на диагональном методе Кантора. Бесконечных последовательностей нулей и единиц уже несчётное количество, а множество натуральнозначных последовательностей его содержит.

уже отмечалось, машины Тьюринга можно отобразить в их описания — программы для исполнителя $\mathbf{I}_\mathrm{T}$. И тогда каждой машине Тьюринга будет поставлено в соответствие конечное слово в некотором конечном алфавите. Множество всех слов в конечном алфавите счётно.

Таким образом, мощность множества всех функций $\mathcal{A}$ больше мощности множества машин Тьюринга, и существуют невычислимые функции — функции, которые нельзя реализовать в виде машины Тьюринга (нельзя запрограммировать на исполнителе $\mathbf{I}_\mathrm{T}$).

Итак, множество $\mathcal{A}$ всех функций делятся на две части — вычислимые ($\mathcal{C}$) и невычислимые ($\mathcal{A} \setminus \mathcal{C}$), при этом мощность множества невычислимых больше мощности множества вычислимых. Невычислимых функций несчётное множество, тогда как вычислимых — счётное.

Одна из невычислимых функции — это *функция статуса останова*. Но прежде, чем переходить к рассмотрению этой функции и связанной с ней проблемы останова, рассмотрим задачу эмуляции одного исполнителя на другом.

Проблема эмуляции исполнителей

Сформулируем общую проблему эмуляции одного исполнителя на другом:

Задача Л3.3. Дано два исполнителя $\mathbf{I}_1$ и $\mathbf{I}_2$. Напишите программу $\mathcal{M}_2$ для исполнителя $\mathbf{I}_2$, которая получает на вход программу $\mathcal{M}_1$ для исполнителя $\mathbf{I}_1$ и входные данные x, а возвращает результат, который был бы получен исполнителем $\mathbf{I}_1$ при выполнении программы $\mathcal{M}_1$ на входных данных x.

Пусть функция $\mathrm{U}_1(\mathcal{M}_1, x)$ равна результату исполнения программы $\mathcal{M}_1$ на исполнителе $\mathbf{I}_1$ на входных данных x. Проблема эмуляции заключается в реализации этой функции на втором исполнителе $\mathbf{I}_2$.

Определение 3.2. *Программа $\mathcal{M}_2$ называется* **эмулятором**[4] **исполнителя $\mathbf{I}_2$ на исполнителе $\mathbf{I}_1$**. *Эмулятор исполнителя Тьюринга на исполнителе Тьюринга называется* **универсальной машиной Тьюринга (УМТ)**.

[4]Следует отметить, что эмуляция одного исполнителя на другом исполнителе (другого типа или того же самого) — довольно частая практическая задача. В современных компьютерных технологиях широко представлены виртуальные машины — вычислительные машины, которые *эмулируются* на реальных вычислительных машинах. В основном, это виртуальные машины для исполнения Java байт-кода или .NET байт-кода (байт-код — команды процессора виртуальной машины), в который компилируются программы на языке Java и *C♯*. В том, что программы компилируются не в машинный код процессора реальной машины, а в байт-код виртуальной машины, есть свои преимущества. Это позволяет создавать *платформонезависимые* приложения. Для использования существующих приложений на новых или нестандартных вычислительных машинах не потребуется заново подправлять программный код и собирать приложения специально под эту архитектуру. Достаточно будет написать под эту архитектуру виртуальную машину, исполняющую байт-код.

Кроме того, эмуляция исполнителей используется для запуска программ в защищённом режиме, когда все команды, перед исполнением, проходят проверку на «вредоносность». Надо сказать также, что в в самой архитектуре современных процессоров присутствует идея виртуализации (эмуляции), воплощением которой является защищенный режим работы процессора, позволяющий исполнять несколько процессов (исполнителей) на одном процессоре.

Итак, УМТ — это машина Тьюринга (в наших терминах — программа для исполнителя $\mathbf{I}_\mathrm{T}$), которая получает на вход описание двух объектов: описание машины Тьюринга a (программу a для исполнителя $\mathbf{I}_\mathrm{T}$) и описание входных данных x. В результате выполнения УМТ должно получится то, что получилось бы при выполнении a на входных данных x. УМТ — это машина Тьюринга, решающая задачу эмуляции произвольной другой машины Тьюринга.

Один из важнейших результатов заключается в следующем:

Универсальная машина Тьюринга существует.
Более того, проблема эмуляции разрешима для произвольной пары Тьюринг-эквивалентных исполнителей.

Универсальную машину Тьюринга (одну из возможных) сконструировал сам Алан Тьюринг. Реализация УМТ приведена в книге Минского [8].

Кроме исполнителя алгорифмов Маркова и исполнителя машин Тьюринга можно рассматривать исполнителей каких-либо языков программирования, например, исполнителей языка Pascal, Java, Си, Python и др. Для некоторых языков X и Y сформулированное выше утверждение соответствует тому, что на языке Y можно написать интерпретатор языка X.

Практически для всех возможных пар языков (X, Y) это утверждение верно. Создание интерпретаторов и компиляторов одного языка на другом является хорошо известной задачей для программистов.

В случае языка Java имеем следующее утверждение: на языке Java можно написать интерпретатор программ на языке Java[5].

Оказывается, наличие эмулятора исполнителя на нём самом влечёт неразрешимость задачи останова[6].

Проблема останова программы

На предыдущей лекции мы сформулировали проблему останова. Уточним её формулировку. Для этого введём понятие *функции статуса останова*:

Определение 3.3. **Функция** Is-Applicable$_1(\mathcal{M}, x)$ **статуса останова исполнителя** $\mathbf{I}_1$ — *это функция, которая паре*

1) *программа* $\mathcal{M}$ *для исполнителя* $\mathbf{I}_1$,

2) *входные данные* x *для программы* $\mathcal{M}$

[5]В случае языка Си аналогичное утверждение не совсем верно. Это связано явно подразумеваемой конечностью оперативной памяти в вычислительной модели языка Си. На практике это не играет никакой роли, и более того, именно на языке Си пишутся интерпретаторы и большинства интерпретируемых языков.

При переходе к языку Java доступная память не становится бесконечной. Но в вычислительной модели языка Java отсутствует понятия адреса и понятие размера адресного пространства, размер доступной памяти в этом языке не ограничен самим языком, поэтому соответствующий этому языку исполнитель является Тьюринг-полным.

[6]Кроме наличия программы эмулятора исполнителя на нём самом необходимо выполнение ещё ряда условий.

сопоставляет 1 *или* 0, *в зависимости от того, останавливается программа* $\mathcal{M}$ *для исполнителя* $\mathbf{I}_1$ *на входных данных* x *или нет.*

Функция статуса останова определяет, применима данная программа к данному входу или нет.

Задача Л3.4. (Проблема останова) Необходимо реализовать функцию статуса останова исполнителя $\mathbf{I}_1$ на исполнителе $\mathbf{I}_2$, то есть необходимо написать программу $\mathcal{M}_2$ для исполнителя $\mathbf{I}_2$, которая для пары $(\mathcal{M}_1, x)$ определяет, остановится ли исполнитель $\mathbf{I}_1$ при выполнении программы $\mathcal{M}_1$ на входных данных x или нет.

Проблема останова неразрешима для произвольной пары Тьюринг-эквивалентных исполнителей.

Проблема останова оказывается неразрешимой, если в качестве исполнителей $\mathbf{I}_1$ и $\mathbf{I}_2$ брать исполнителей $\mathbf{I}_\mathrm{T}$, $\mathbf{I}_\mathrm{M}$, других вычислительно эквивалентных им исполнителей, или различные их комбинации. Это один из важнейших результатов теории вычислимых функций.

Давайте попробуем понять, почему проблема останова алгоритмически неразрешима. Если исполнитель останавливается, то эмуляция вычислений позволяет нам определить, на каком именно шаге он остановится. А если нет? Если он не остановился на 1000 шаге? Можно отработать ещё 1000, и ещё 10^9 шагов. Сколь бы много шагов исполнителя мы не сэмулировали, это не гарантирует того, что исполнитель не остановится никогда. Конечно, если исполнитель начинает по циклу пробегать одно и то же множество состояний[7], то сразу можно сделать вывод, что он никогда не остановится. Но существуют случаи, когда состояния не повторяются, а исполнитель никогда не завершает своей работы. Примером такого случая может быть алгорифм Маркова, состоящий из одного правила: $\mathcal{M} = \{\mathtt{A} \to \mathtt{AA}\}$.

Чтобы развеять иллюзию существования решения проблемы останова, приведём простой пример — алгоритм (схему алгоритма) 3.11 проверки теоремы Ферма, которая звучит так: «Не существует таких натуральных чисел a, b, c и $n > 2$, таких что $a^n + b^n = c^n$».

Описанный алгоритм остановится только в том случае, если теорема Ферма неверна. Таким образом, алгоритм, решающий проблему останова, в данном случае должен был бы доказать теорему Ферма и вывести, что программа не остановится. Утверждение, что данная программа остановится, всегда можно записать в виде формального математического утверждения, и доказательство этого утверждение может быть довольно сложным. Общего алгоритма доказательства или опровержения таких утверждений нет.

Проблема останова алгоритмически неразрешима — нельзя написать алгоритм, который на вход получает описание алгоритма и определяет, остановится ли данный алгоритм на фиксированных входных данных.

Можно сказать, что основная причина неразрешимости задачи останова — это оператор цикла **while**, который работает неопределенное заранее число шагов (в отличие от стандартного оператора цикла **for**, который принято использовать для перебора элементов множества известного размера).

[7]В понятие «состояние исполнителя» входит как состояние памяти исполнителя, так и состояние управляющего устройства исполнителя.

Алгоритм 3.11 Алгоритм проверка теоремы Ферма

```
i ← 5;
while true do
    for c ∈ {1, ..., (i − 1)} do;
        for a ∈ {1, ..., (c − 1)} do;
            for b ∈ {1, ..., a} do;
                for n ∈ {3, ..., i} do;
                    if a^n + b^n = c^n then
                        print ''Теорема Ферма не верна.''
                        закончить работу
                    end if
                end for
            end for
        end for
    end for
    i ← i + 1
end while
```

Именно оператор цикла **while** является причиной «зависания» многих программ. Поэтому, прежде, чем использовать оператор цикла **while**, тщательно проанализируйте условия выхода из этого цикла. Необходимо, чтобы условия выхода из цикла выполнялись через некоторое количество шагов при любых корректных входных данных (корректном состоянии хранимых данных).

Приведём схему доказательства того, что проблема останова алгоритмически неразрешима. Для этого мы широко будем использовать возможность эмуляции.

Назовём подмножество натуральных чисел **разрешимым**, если функция f_A, которая равна единице на элементах этого множества, и нулю – на остальных, вычислима:

$$A \subset \mathbb{N},\ A \text{ разрешимо } \equiv\ f_A \text{ — вычислима, где } f : \mathbb{N} \to \{0,1\},\ f(x) = \begin{cases} 1, & \text{если } x \in A, \\ 0, & \text{если } x \in \overline{A}. \end{cases}$$

Функция f_A называется **характеристической функцией множества** A.

Функция $U(n,x)$, $n,x \in \mathbb{N}$ равная 0 или 1, такая что для любого разрешимого множества A найдётся m: $f_A(x) = U(m,x)$, называется **универсальной функцией разрешимых множеств**. Другими словами, $U(n,x)$ представляет собой параметризованное (параметром n) семейство характеристических функций всех разрешимых множеств. Существует множество различных универсальных функций. Но есть ли среди них вычислимые? Оказывается — нет, и это ключевой момент в приводимом ниже доказательстве.

Доказательство проводится методом от противного и состоит из четырёх шагов:

1. Пусть проблема останова разрешима.

2. Тогда существует *вычислимая* универсальная функция разрешимых множеств $U(n,x)$.

3. Тогда функция $d(x) = 1 - U(x,x)$ тоже вычислима. Но функция d является характеристической функцией некоторого множества B. Несложно заметить, что не существует

такого n, что $d(x) = U(n, x)$ (функции $d(x)$ нет среди семейства функций $U(n, x)$, запараметризованных параметром n). Действительно, для любого n имеем, $d(x) \neq U(n, x)$ при $x = n$.

4. Таким образом, семейство функций $U(n, x)$, где $n \in \mathbb{N}$, не содержит $d(x)$, хотя по определению оно должна содержать все характеристические функции разрешимых множеств. Значит универсальная функция разрешимых множеств не может быть вычислимой. Получили противоречие с утверждением шага 2.

Самым сложным шагом здесь является шаг 2. На этом шаге необходимо явно сконструировать алгоритм вычисления функции $U(n, x)$, исходя из предположения, что задача останова разрешима. Сделаем это в несколько действий:

2.1. Можно написать программу, которая перебирает все слова из некоторого алфавита B, из которого составляются тексты программ. Сначала она перебирает однобуквенные слова, затем двубуквенные, и так далее. Эта программа не останавливается.

2.2. Можно написать программу, которая перебирает все корректные программы. Для этого достаточно взять программу с предыдущего шага и добавить этап проверки слова на то, является ли слово корректной программой. Это шаг вычислим.

2.3. Можно написать программу, которая перебирает только такие программы, которые останавливаются. Для этого достаточно взять программу с предыдущего шага и использовать процедуру проверки программы на останов (такая процедура по предположению существует). Кроме того можно сделать ещё дополнительную проверку на то, что результатом выполнения программы является 0 или 1. Для этого достаточно из всех команд вывода разрешить только команды вывода 0 и 1, совмещённые с командой завершения вычислений.

2.4. Можно написать программу, которая получает на вход натуральное число n и возвращает текст программы, который будет выведен программой с предыдущего шага n-м по счёту.

2.5. Можно написать программу, реализующую функцию $U(n, x)$. Для этого необходимо взять программу с предыдущего шага и подать ей на вход n. Текст программы, который она вернёт в качестве результата, и входные данные x следует направить на вход эмулятору. Результат вычисления эмулятора следует вернуть в качестве результата вычисления функции $U(n, x)$.

Заметим, что данное доказательство приведено не для исполнителя машин Тьюринга, а для некоторого исполнителя, для которого возможно сделать все упомянутые действия, в частности, эмуляцию этого исполнителя на нём самом и композицию программ (по двум данным программам a и b составление программы c, которая реализует функцию, равную композиции функций, реализуемых программами a и b).

Более полное доказательство алгоритмической неразрешимости проблемы останова можно найти в книге [12].

Проблема эквивалентности программ

Приведём пример ещё одной невычислимой функции. Для этого введём понятие эквивалентных программ.

Определение 3.4. *Пусть дано два исполнителя* $\mathbf{I}_1$, $\mathbf{I}_2$ *и программы* $\mathcal{M}_1$, $\mathcal{M}_2$ *для них. Программы называются эквивалентными, если они решают одну и ту же задачу (реализуют одну и ту же функцию).*

Задача Л3.5. (Эквивалентность двух программ) Даны формальные описания $\mathcal{M}_1$ и $\mathcal{M}_2$ программ для исполнителей $\mathbf{I}_1$ и $\mathbf{I}_2$. Функция Eq распознаёт, реализуют они одинаковые вычислимые функции или нет, а именно, $\mathrm{Eq}(a, b)$ равно 1, тогда и только тогда, когда данные программы применимы к одному и тому же множеству слов и для любого входного слова из этого множества получают один и тот же результат. В остальных случаях Eq должно быть равно 0. Задача — написать программу $\mathcal{M}_3$ для исполнителя $\mathbf{I}_3$, которая реализует данную функцию.

Если в качестве исполнителей $\mathbf{I}_1$, $\mathbf{I}_2$, $\mathbf{I}_3$ брать исполнителей $\mathbf{I}_\mathrm{T}$, $\mathbf{I}_\mathrm{M}$ или другие Тьюринг-эквивалентные исполнители, то эта задача *неразрешима.*

Проблемы останова и эквивалентности неразрешимы (или практически неразрешима) для программ, записанных на большинстве современных языков программирования. Есть понятие Тьюринг-полного языка программирования — языка программирования, на котором можно реализовать эмулятор исполнителя машин Тьюринга, то есть написать программу, которая получает на вход формальное описание машины Тьюринга и начальное состояние ленты, после чего в точности эмулирует все шаги работы машины Тьюринга. Для таких языков проблемы останова и эквивалентности точно неразрешимы. Но есть языки программирования, не являющиеся Тьюринг-полными, в частности, языки, в которых подразумевается конечность памяти, доступной программе (конечность адресного пространства ячеек памяти)[8]. На таких языках нельзя смоделировать в полной мере бесконечную ленту машины Тьюринга. Но и для них, проблему останова на практике нельзя решить, так как это потребует очень большого времени.

Проблемы останова и эквивалентности разрешимы лишь для простых исполнителей. Так, например, проблема эквивалентности конечных автоматов алгоритмически разрешима на исполнителях, вычислительно эквивалентных машине Тьюринга.

[8]Конечность адресного пространства ячеек памяти явно отображена, например, в языке программирования Си. Поэтому язык Си нельзя назвать Тьюринг-полным. Языки программирования, в которых просто нет понятия адреса, часто являются Тьюринг-полными, хотя, конечно, это не может добавить им «вычислительной мощи» на практике. Определить, остановится данная программа на Си или нет, практически невозможно. Действительно, на компьютерах с 32-битной архитектурой программам выделяется 2^{32} байт памяти, и именно столько различных значений могут принимать адреса в языке Си. Конечность адресного пространства позволяет просмотреть всё конечное множество состояний памяти и определить те, которые приводят к зацикливанию. Но размер множества всех возможных состояний памяти очень велик, и эта идея практически неосуществима.

Предложите, как можно проблему останова свести к проблеме эквивалентности и наоборот, то есть как, имея функцию статуса останова Is-Applicable, реализовать функцию эквивалентности Eq и наоборот.

Пусть у нас (каким-то чудом) появилась реализация функции останова Is-Applicable исполнителя $\mathbf{I}_1$ на исполнителе $\mathbf{I}_3$. Тогда сконструируем программу A для исполнителя $\mathbf{I}_3$, которая перебирает все возможные входы и запускает эмуляторы исполнителей $\mathbf{I}_1$ и $\mathbf{I}_2$ с программами $\mathcal{M}_1$ и $\mathcal{M}_2$ для этих входов. Программа A будет останавливаться, только если результаты выполнения этих программ ($\mathrm{U}_1(\mathcal{M}_1, x)$ и $\mathrm{U}_2(\mathcal{M}_2, x)$) различны. Тогда Is-Applicable(A, $\{\mathcal{M}_1, \mathcal{M}_2\}$) равно 1 тогда и только тогда, когда программы $\mathcal{M}_1$ и $\mathcal{M}_2$ не эквивалентны:

$$\begin{aligned} &A = \text{перебирать все возможные } x \text{ и закончить вычисления,} \\ &\qquad \text{если } \mathrm{U}_1(\mathcal{M}_1, x) \neq \mathrm{U}_2(\mathcal{M}_2, x); \\ &\mathrm{Eq}(\mathcal{M}_1, \mathcal{M}_2) = 1 - \text{Is-Applicable}(A, \{\mathcal{M}_1, \mathcal{M}_2\}). \end{aligned}$$

В действительности сводимость несколько сложнее. Прежде чем запускать вычисления $U_1(\mathcal{M}_1, x)$ и $U_2(\mathcal{M}_2, x)$ необходимо убедится, что программы–эмуляторы исполнителей $\mathbf{I}_1$ и $\mathbf{I}_2$ не «зависнут» во время вычисления. Для этого достаточно подать эти эмуляторы с указанными аргументами на вход функции Is-Applicable. Если один из них «зависает», то и другой должен «зависать», иначе они не эквивалентны и программа A завершает работу.

Обратная сводимость функции Is-Applicable(A, x) к функции Eq очевидна. Данную программу (с зафиксированными входными данными x) необходимо проверить на эквивалентность программе, которая всегда зависает.

Проблема разрешимости Диафантового уравнения

Приведём ещё одну проблему, алгоритмически неразрешимую на Тьюринг-эквивалентных исполнителях.

Задача Л3.6. (Определение разрешимости Диафантового уравнения) Напишите программу для исполнителя **I**, которая получает на вход описание многочлена P с рациональными коэффициентами от нескольких переменных, и возвращает 1 или 0 в зависимости от того пусто множество решений уравнения $P = 0$ в рациональных числах или нет.

Проблема существования решения этой задачи была сформулирована Д. Гильбертом в 1900 г. и решена Ю. Матиясевичем в 1970 г. Гильберт сформулировал проблему без упоминания Тьюринг-эквивалентного исполнителя (тогда машина Тьюринга ещё не была придумана). Он использовал понятие чётко специфицированной процедуры (алгоритма), исполняемой человеком (математиком).

Заметим, что задача определения разрешимости системы алгебраических уравнений (уравнений вида $P = 0$, где справа от знака «равно» стоит многочлен с рациональными коэффициентами) в комплексных числах алгоритмически разрешима. Найден решающий её алгоритм, время работы которого ограничено многочленом от числа уравнений и их размера. Это является важнейшим результатом компьютерной алгебры.

Сегодня в самых разных областях математики (языки и грамматики, алгебра, теория чисел, помехоустойчивые коды), найдено много интересных алгоритмически неразрешимых задач.

Заключительные замечания

1. Можно построить и *универсальный алгорифм Маркова*, который мог бы интерпретировать любой алгорифм, включая самого себя.

2. Умение работать в рамках различных исполнителей является важным. Несмотря на некоторую искусственность машины Тьюринга и машины Маркова (непригодность для решения практических задач) они дают представление об автоматном и продукционном программировании (хотя на лекциях мы их рассматривали как средство определения понятия вычислимости). В современных языках программирования широко распространены регулярные выражения Perl и преобразования строк на основе регулярных выражений (Perl regular expressions), которые по сути являются расширением алгорифмов Маркова. Представление об автоматах и машине Тьюринга играют ключевую роль в задачах создания электронных вычислительных устройств.

3. *Сложности решения*[9] различных задач на различных исполнителях различаются: какие-то задачи проще решить на одном исполнителе, какие-то — на других. Здесь под сложностью решения можно понимать как время работы исполнителя, так и трудоёмкость написания программы для исполнителя. И это крайне важно при решении практических задач — когда перед программистом встаёт задача, первое, что ему следует сделать, — это выбрать наиболее подходящую формальную вычислительную модель (парадигму программирования, язык программирования) и архитектуру вычислительной машины[10]. Более того, иногда эффективным решением является разработка такой формальной модели, в рамках которой удобно решать поставленную задачу. На практике этой означает, что иногда имеет смысл разрабатывать свои языки программирования, наиболее подходящие для решения типичных задач заданной предметной области. Это один из примеров мета-подходов в программировании, где часто возникают задачи вида «создать инструмент для создания инструментов для создания инструментов ... решения класса задач X».

4. Идея построения эмулирующих программ и устройств широко используется в компьютерах. Почти никогда не пишутся программы (алгоритмы) в системе понятий и правил (системе команд) реального исполнителя (процессора компьютера). Большинство программ сегодня пишется для некоторой абстрактной машины, которая эмулируется программой-эмулятором на реальном компьютере. Причём эта конструкция может быть многоступенчатой, то есть состоять из цепочки программ, которые производят другие программы для других исполнителей и т.д., при этом многократно производится преобразование информации из одного представления в другое. Как ни странно, эти сложные построения происходят именно из желания человека упростить и стандартизовать процесс решения практических задач. Эмуляция (виртуализация) вычислительных машин и различных устройств является важным шагом на этом пути.

5. Реальный компьютер и реальный интерпретатор имеют конечную память. У исполните-

[9] Различные аспекты вычислительной сложности разобраны в дополнительной лекции 21.

[10] Под архитектурой вычислительной машины имеется ввиду её внутреннее устройство: компонентный состав, принцип организации памяти и передачи данных между компонентами, набор инструкций, «понимаемых» центральным вычислительным устройством (если таковое имеется), и др. Выбор архитектур на современном рынке не так велик — различных специфических архитектур множество, но все они устроены похожим образом, и нет существенного различия в типе задач, эффективно решаемых на различных архитектурах.

лей с конечной памятью появляется новый тип останова — нехватка памяти. Для исполнителей с конечной памятью проблема останова разрешима. В частности, если ленту исполнителя машин Тьюринга ограничить, то проблема его останова станет разрешимой. Обозначим $\mathbf{I}_{\mathrm{T}}\,(< N)$ исполнителя машин Тьюринга, который имеет конечную ленту из N ячеек и аварийно заканчивает свою работу, если выполняется команда смещения УУ за пределы ленты. Проблема останова исполнителя $\mathbf{I}_{\mathrm{T}}\,(< N)$ разрешима на исполнителе $\mathbf{I}_{\mathrm{T}}$. Это можно доказать следующим образом. Если у исполнителя ограничена память, то число возможных состояний этого исполнителя (в случае $\mathbf{I}_{\mathrm{T}}\,(< N)$ это множество совокупных состояний пары «УУ + лента») конечно, а значит, «зависающие» алгоритмы — это алгоритмы, которые приводят к периодическому повторению состояний машины. Решение задачи останова сведётся к эмуляции исполнителя и хранению списка состояний, в которых побывал эмулируемый исполнитель. При повторении одного из состояний можно выводить ответ «не остановится», если же эмулируемый исполнитель остановится, то следует вывести ответ «остановится». Из конечности числа состояний следует, что одна из указанных альтернатив обязательно будет иметь место.

6. Если задача алгоритмически разрешима, то это не означает, что она разрешима на практике. *Вычислительная сложность* оказывается слишком большой для целого ряда важных практических задач (например, для задачи прогноза погоды, численного решения задач квантовой механики, моделирования и анализа физических процессов в двигателях, в океанах, на солнце и др.). Среди аспектов вычислительной сложности выделяют *сложность конструирования* (комбинационная сложность, сложность описания), время вычислений (алгоритмическая сложность) и требуемую память. Память и сложность конструирования подвластны человеку (программистов много, методы организации крупных программистских проектов известны и отработаны, память тоже «дело наживное»). Наиболее важным аспектом вычислительной сложности является время. Не имеет смысла писать программу расчёта траектории ракеты, если за время вычисления ракета уже попадёт в цель и её нельзя будет поразить средствами противоракетной обороны. Факт временно́й сложности используется, например, в системах защиты информации. В криптографии любой шифр может быть потенциально раскрыт, но если дешифровка требует 0.5 млн. лет суммарной работы всех компьютеров в мире, то шифр можно считать надёжным — через 0.5 млн. лет зашифрованное письмо просто утратит всякий смысл.

7. Всегда, составляя программу для машины, следует оценивать различные аспекты вычислительной сложности решаемой задачи. По крайней мере, нужно иметь хотя бы грубые оценки. Большая часть написанного на данный момент кода не используется и не будет использоваться, так как разработчики, перед тем как кодировать, забыли оценить объём кода и сложность его поддержки и/или число отдельных сущностей и количество связей между ними, и/или требуемую память и/или время работы придуманных алгоритмов.

Семинар 3

Алгорифмы Маркова

Краткое описание: Практика построения алгорифмов Маркова. На примере задачи о скобочных выражениях и увеличения числа на единицу мы изучим основные принципы, облегчающие задачу решения алгоритмических задач на исполнителе алгорифмов Маркова.

Исполнитель алгорифмов Маркова $\mathbf{I}_\mathrm{M}$ — это исполнитель, логика работы которого задаётся набором правил преобразования входного слова. Входное слово состоит из букв конечного алфавита B, а правила указывают какие следует делать замены: «заменить подслово X на слово Y». Слова X и Y являются словами расширенного алфавита B', который содержит B как подмножество. Некоторые правила помечены как стоп-правила.

Наборы правил (программы для этого исполнителя) называют **алгорифмами Маркова**.

Правила упорядочены. Исполнитель $\mathbf{I}_\mathrm{M}$ сначала пробует применить первое правило. Если оно не применимо к текущему слову, то берётся второе правило и так далее, пока не встретится правило, которое можно применить к слову. После применения какого-либо правила исполнитель $\mathbf{I}_\mathrm{M}$ снова переходит к первому правилу. Вычисления заканчиваются, когда встречается правило, помеченное как стоп-правило.

Алфавит B, указываемый в задачах, — это алфавит, в котором записывается входное слово. Расширенный алфавит B' включает в себя алфавит B. В расширенный алфавит B' можно включать произвольное количество дополнительных символов, если в задаче не оговорены специальные условия.

Практика построения алгорифмов Маркова

Обозначим символом «$\varnothing$» пустое слово, а символом «␣» — пробел.

Рассмотрим следующий простой алгорифм Маркова $\mathcal{M}$ в алфавите $B = \{\texttt{(}, \texttt{)}\}$:

$$\mathcal{M} = \{\texttt{()} \to \varnothing.$$

Он состоит из одного правила, по которому во входном слове первое вхождение двух подряд идущих открывающей и закрывающей скобок будет заменено на пустое слово. Этот алгорифм распознаёт правильные скобочные выражения (структуры), а именно, правильные скобочные выражения он преобразует в пустое слово. Рассмотрим как он отработает на входном слове `(()())()`:

$$\texttt{(}\underline{\texttt{()}}\texttt{())()} \to \texttt{(}\underline{\texttt{()}}\texttt{)()} \to \underline{\texttt{()}}\texttt{()} \to \underline{\texttt{()}} \to \varnothing.$$

Получили в качестве результата вычислений пустое слово $\varnothing$.

Для скобочного выражения `())(()` результат преобразований не равен пустому слову:

$$\underline{()})(() \to)(\underline{()} \to)(.$$

Задача С3.1. Докажите, что при выполнении описанного алгорифма Маркова в результате получится пустое слово тогда и только тогда, когда на входе дано правильное скобочное выражение. Правильное скобочное выражение определяется рекурсивным образом:

- пустое слово является правильным скобочным выражением;
- если `S` — правильное скобочное выражение, то `(S)` — тоже правильное скобочное выражение;
- если даны два правильных скобочных выражения, то если их написать рядом, то снова получится правильное скобочное выражение.

Данное определение можно записать в виде *правил вывода*:

```
S ::= ∅              // пустое слово является правильным
S ::= '(' S ')'      // любое правильное слово, окруженное скобками, является правильным
S ::= S S            // рядом стоящие правильные слова (возможно, разные) образуют правильное слово
```

Правила вывода не следует путать с правилами подстановки алгорифмов Маркова. Первые описывают *грамматику*[1], и с их помощью конструируются слова грамматики. Вторые — это правила преобразования входного слова.

Следующий алгорифм Маркова просто дописывает в начало слова символ `1`:

$$\mathcal{M} : \{\varnothing \to . \ 1.$$

Здесь пустое слово стоит справа от знака стрелки. Это правило применяется к любому слову, так как пустое слово как бы находится вначале каждого слова. Обратите внимание на то, что стрелка в этом правиле помечена точкой. Э Это значит, что это стоп правило и как только оно будет применено, вычисления останавливаются, и значение слова в памяти исполнителя рассматривается как результат вычислений.

Задача С3.2. Напишите алгорифм Маркова с алфавитом $B = \{$`(`, `)`$\}$ и расширенным алфавитом $B' = \{$`(`, `)`, `N`, `Y`$\}$, который проверяет правильность скобочного выражения. В случае правильного выражения результат должен быть равен слову `Y`, а в случае неправильного — слову `N` (без каких-либо дополнительных символов справа или слева).

Задача С3.3. Рассмотрите алгорифм Маркова $\mathcal{M}_1$ с алфавитом $B = \{$`0`, `1`, `a`$\}$:

$$\mathcal{M}_1 = \begin{cases} \texttt{1} & \to & \texttt{0}, \\ \texttt{0} & \to & \texttt{1}. \end{cases}$$

Верно ли, что этот алгоритм заменит в слове все единицы на нули, а все нули на единицы? Что произойдёт, если на вход подать слово `01101`? Укажите множество слов, к которым применим данный алгорифм.

[1] Подробнее о грамматиках рассказано на в лекции 4 на стр. 76.

Задача С3.4. Рассмотрите алгорифм Маркова $\mathcal{M}_2$ (рис. 3.1 на стр. 73). Опишите, что он делает. Какой будет результат вычислений на входном слове
а) `1`; б) `0`; в) `0110110`; г) `10101100`?

Задача С3.5. Рассмотрите задачу Л3.2 (стр. 57), разобранную на лекции 3. Решите эту задачу для случая, когда цифры входного слова записаны в обратном порядке — от младшего разряда к старшему.

Задача С3.6. Выясните, что делает алгорифм Маркова $\mathcal{M}_3$ (рис. 3.1 на стр. 73), который на вход получает слово в алфавите $B = \{\mathtt{0}, \mathtt{1}\}$ и имеет расширенный алфавит $B' = \{\mathtt{0}, \mathtt{1}, \mathtt{x}, \mathtt{A}, \mathtt{B}\}$. Какой будет результат вычислений, если на вход подать слово
а) `00`; б) `00111`; в) `0101`?
Опишите множество слов, к которым применим данный алгоритм. Попробуйте написать *эквивалентный* алгорифм Маркова с меньшим числом правил.

Задача С3.7. Напишите алгорифм Маркова с алфавитом $B = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$, который приписывает букву `Z` к входному слову справа. Можно ли решить эту задачу при $B' = B$?

Задача С3.8. Напишите алгорифм Маркова с алфавитом $B = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$, который приписывает букву `Z` к входному слову справа при условии, если входное слово содержит символ `Z`. Можно ли решить эту задачу при $B' = B$?

Задача С3.9. Напишите алгорифм Маркова, который реализует перевёртывание слова в алфавите $B = \{\mathtt{a}, \mathtt{b}\}$. В частности слово `abaabbb` он должен преобразовать в слово `bbbaaba`. Можно ли решить эту задачу, имея в B' лишь один дополнительный символ?

Задача С3.10. Модифицируйте алгорифм из задачи С3.9 таким образом, чтобы он переворачивал слова в алфавите $B = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Обратите внимание на число правил подстановок. Как растёт число подстановок (актов применения) от длины входного слова? Существует ли способ построения алгорифма, в котором число правил подстановок ограничено линейной функцией от размера алфавита B? Если да, то приведите алгоритм конструирования алгорифма, если нет, то докажите почему.

Задача С3.11. Напишите алгорифм Маркова, который реализует удвоение числа, данного на входе в единичной записи.

Задача С3.12. Напишите алгорифм Маркова, который реализует удвоение числа, заданного в двоичной системе счисления. В частности, слово `0` должно остаться `0`, а слово `1011` должно преобразоваться в слово `10110`.

Задача С3.13. Напишите алгорифм Маркова, который к целому неотрицательному числу, заданному в двоичной системе счисления, добавляет 1. В частности, слово `0` должно преобразоваться в `1`, слово `11` — в слово `100`, а слово `10111` — в слово `11000`.

Задача С3.14. Напишите алгорифм Маркова, который получает на вход слово, состоящее из одних единиц, и возвращает в качестве результата слово из единиц вдвое большей длины.

Задача С3.15. Напишите алгорифм Маркова, который реализует умножение двух целых неотрицательных чисел. Вход представляет собой слово, состоящее из записанных подряд единичных записей чисел.

Задача С3.16. Напишите алгорифм Маркова, который получает на вход бинарное слово и возвращает слово из единиц, длина которого равна числу нулей во входном слове.

Задача С3.17.* Можно ли сконструировать алгорифм Маркова, который получает на вход натуральное число n, записанное в двоичной системе счисления, и возвращает слово из n единиц? Если да, сконструируйте его, если нет — докажите, что нельзя.

Задача С3.18.* Напишите алгорифм Маркова, который получает на вход слово состоящее из n единиц и возвращает в качестве результата слово из
а) n^2 единиц; б) 2^n единиц.

$$\mathcal{M}_2 = \begin{cases} \texttt{11} \to \texttt{0x0x}, \\ \texttt{10} \to \texttt{0x1x}, \\ \texttt{01} \to \texttt{1x0x}, \\ \texttt{00} \to \texttt{1x1x}, \\ \varnothing \to. \ \varnothing. \end{cases}$$

$$\mathcal{M}_3 = \begin{cases} \varnothing \to \texttt{x}, \\ \texttt{x0} \to \texttt{1x}, \\ \texttt{x1} \to \texttt{0x}, \\ \texttt{x} \to. \ \varnothing. \end{cases}$$

$$\mathcal{M}_4 = \begin{cases} \texttt{␣} \to \varnothing, \\ \texttt{(0)} \to \texttt{0}, \\ \texttt{(1)} \to \texttt{1}, \\ \texttt{0 \& 0} \to \texttt{0}, \\ \texttt{0 \& 1} \to \texttt{0}, \\ \texttt{1 \& 0} \to \texttt{0}, \\ \texttt{1 \& 1} \to \texttt{1}, \\ \texttt{0 | 0} \to \texttt{0}, \\ \texttt{0 | 1} \to \texttt{1}, \\ \texttt{1 | 0} \to \texttt{1}, \\ \texttt{1 | 1} \to \texttt{1}. \end{cases}$$

Рис. 3.1: Описания алгорифмов Маркова

Задача С3.19. Разберите алгорифм Маркова $\mathcal{M}_4$, который на вход получает слово в алфавите $B = \{\texttt{␣}, \texttt{(}, \texttt{)}, \texttt{|}, \texttt{\&}, \texttt{0}, \texttt{0}\}$, представляющее собой логическое выражение со скобками и булевыми операциями `|` (логическое ИЛИ) и `&` (логическое И), и возвращает результат вычисления данного логического выражения. Приоритет операции `|` меньше, чем у операции `&`. Символ `1` обозначает *истину*, а символ `0` — *ложь*. Опишите множество слов, к которым применим данный алгорифм. Доработайте его так, чтобы в случае некорректного выражения на входе результат был равен `ERROR`. Проверьте правильность работы алгорифма на словах `1|0&(0|0)`, `1&(0|0)&1|1`.

Теоретические задачи

Задача С3.20. В алфавите $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ заданы алгорифмы P, Q и R:

$$P : \{\texttt{ca} \to \texttt{ac}\ , \quad Q : \{\texttt{cb} \to \texttt{bc}\ , \quad R : \begin{cases} \texttt{ca} \to \texttt{ac}, \\ \texttt{cb} \to \texttt{bc}. \end{cases}$$

Верно ли, что функция, реализуемая алгорифмом R, является композицией функций, реализуемых алгорифмами P и Q. Ответ обоснуйте.

Задача С3.21. Назовём алгорифм Маркова **укорачивающим**, если в каждом правиле подстановки длина левой части больше длины правой части. Докажите, что для укорачивающих алгорифмов проблема останова алгоритмически разрешима.

Задача С3.22. Назовём алгорифм Маркова **неудлиняющим**, если в каждом правиле подстановки длина левой части больше либо равна длине правой части. Докажите, что для неудлиняющих алгорифмов проблема останова алгоритмически разрешима.

Лекция 4

Языки и метаязыки

Краткое описание: Мы рассмотрим несколько простых формальных языков: язык правильных скобочных выражений, три языка арифметических выражений (префиксная, инфиксная и постфиксная). Кроме того, мы изучим абстрактную структуру данных «стек». Она будет использоваться при построении алгоритма вычисления арифметических выражений, записанных в обратной польской нотации. На этой лекции мы также познакомимся с языком описания формальных языков (метаязыком), основанном на форме (нотации) Бэкуса-Наура. Он будет активно использоваться в дальнейших лекциях.
Ключевые слова: формальный язык, грамматика, форма Бэкуса-Наура, префиксная (польская), инфиксная (алгебраическая) и постфиксная (обратная польская) нотации, стек.

Формальные языки — это материал, с которым регулярно приходится работать программисту. Система записи арифметических выражений с помощью цифр, знаков арифметических операций и скобок — это формальный язык, знакомый всем со школы. Языки программирования также являются формальными языками. Синтаксис и правила построения корректных программ чётко определены для каждого языка программирования (формально описаны).

На этой лекции мы познакомимся с несколькими важными формальными языками.

Язык правильных скобочных выражений. Определение грамматики

Правильные скобочные выражения — это слово из скобок, которое может остаться от арифметического выражения, если из него убрать все знаки арифметических операций и числа. Например, от выражения $((1 \cdot 2) + (3 \cdot 4))$ останется `(()())` — это одно из правильных скобочных выражений.

Приведём примеры правильных и неправильных скобочных выражений:

правильные	неправильные
`()`, `()()`, `(()())`, `()()()()`, `(())`	`)(`, `(()`, `())(()`, `(((()`

Приведём строгое определение множества правильных скобочных выражений.

Правильные скобочные выражения — это некоторое подмножество слов в алфавите $B = \{$`(`, `)`$\}$, состоящем из двух скобок — закрывающей и открывающей.

Ясно, что правильное скобочное выражение состоит из чётного числа символов, так как для каждой открывающей скобки должна быть парная закрывающая. Условия равенства числа

открывающих скобок и числа закрывающих скобок не достаточно, чтобы выражение было правильным.

Приведём одно из возможных определений множества правильных скобочных выражений.

Определение 4.1. *Правильные скобочные выражения – это слова в алфавите* $B = \{(,)\}$, *удовлетворяющие двум свойствам:*
1) По мере движения справа налево по символам правильного скобочного выражения число прочитанных открывающих скобок всегда больше либо равно числу закрывающих.
2) По достижении конца выражения число открывающих скобок точно равно число закрывающих.

Это определение легко пояснить: каждая закрывающая скобка должна иметь парную открывающую скобку, стоящую левее неё.

Приведём другое определение, основанное на *правилах вывода*.

Определение 4.2. *Рассмотрим* **правила вывода***:*

```
S ::= ∅
S ::= ( S )
S ::= S S
```

Эти правила следует интерпретировать следующим образом. Первое правило означает, что пустое слово является правильным выражением. Второе правило означает, что если некоторое правильное выражение окружить скобками, то получится снова правильная. И третье правило: если записать две правильные скобочные выражения рядом, то получится правильное выражение. Все слова, которые можно получить (вывести) с помощью указанных правил, по называются **правильными скобочными выражениями**.

Например, если подставить пустое слово вместо S в правую часть второго правила, то получим слово `()`, значит слово `()` *выводимо* из символа S. Если в это же правило подставить слово `()`, то получим слово `(())`. Таким образом, пользуясь вторым правилом, можно вывести слова вида `(((...)))`. Воспользуемся теперь третьим правилом. Вместо первого символа S подставим одно из выведенных слов, и вместо второго символа S тоже подставим одно из выведенных слов (возможно, другое). Так, например, можно взять `()` и `((()))` и получить слово `()((()))`.

Набор правил вывода называется **грамматикой**[1]. Про язык, который описывается правилами вывода, говорят, что **язык порождается грамматикой**.

Более подробно о грамматиках рассказывается на курсе «Теория и реализация языков программирования».

Замечание. Круглые скобки используются в ряде нотации как специальные символы группировки. Чтобы подчеркнуть, что скобки здесь не являются специальным символом, мы заключим их в одинарные кавычки:

```
S ::= ∅
S ::= '(' S ')'
S ::= S S
```

Везде далее символы в описаниях грамматик мы будем, по возможности, заключать в одинарные кавычки.

Приведём точное определение грамматики.

Грамматика — это четыре элемента (V_T, V_N, P, S), где
V_T — алфавит терминальных символов (в примере со скобочными выражениями это два символа — открывающая и закрывающая скобки),
V_N — алфавит нетерминальных символов, не пересекающийся с V_T (в нашем примере это один символ S),
P — конечное множество правил; каждое правило есть пара (α, β) и записывается в виде $\alpha \to \beta$, где α и β непустые слова в алфавите $B = V_T \cup V_N$,
S — начальный символ (цель) грамматики, $S \in V_N$.

С каждым нетерминальным символом связывается множество выводимых из него слов в алфавите V_T.

Везде далее мы будем рассматривать грамматики, у которых в каждом правиле слева стоит один нетерминальный символ.

Такие грамматики называются **контекстно-свободными**.

Опишем процесс вывода в контекстно-свободных грамматиках. Он определяется рекурсивным образом. Слово w в алфавите V_T выводимо из символа X, если есть правило вида $X \to ...$, в котором каждый нетерминальный символ в правой части можно заменить на одно из выводимых из него слов и получить в итоге нужное слово w. Применяя различные правила и осуществляя различные замены нетерминальных символах в правых частях правил, можно последовательно расширять множество слов, выведенных из различных нетерминальных символов. Все слова, которые можно вывести из символа S называются словами, выводимыми из грамматики или языком, который задаёт грамматика.

Задача Л4.1. Докажите эквивалентность двух приведённых определений правильных скобочных выражений.

Задача Л4.2. Покажите, что следующие два правила также задают язык правильных скобочных выражений.

```
S ::= ∅
S ::= '(' S ')' S
```

Утверждение последней задачи может покажется очевидным. Но будьте осторожны — в теории языков есть множество «подводных камней» и «неочевидных очевидностей».

Есть три важные задачи, связанные с языком правильных скобочных выражений.

Задача Л4.3. (Определение правильных скобочных выражений) Напишите алгоритм, который распознаёт правильные скобочные выражения, то есть получает на вход слово из скобок и возвращает `YES` или `NO` в зависимости от того, является ли введённое слово правильной скобочным выражением или нет.

Задача Л4.4. (Подсчёт количества правильных скобочных выражений) Напишите алгоритм, который по числу n выводит число правильных скобочных выражений длины $2n$.

Подсказка: Посмотрите на рисунок 4.1. Там показано несколько скобочных выражений, каждой из которых сопоставлен путь из стрелочек. Правила соответствия следующие: если скобка открывается, то идём по диагонали вверх, а если скобка закрывающая — то по диагонали вниз. Все возможные правильные скобочные выражения длины 6 соответствуют всем возможным путям по стрелочкам в графе[2] G.

[2]Подробнее о графах будет говорится в лекции 17. Здесь под графом можно понимать набор точек на плоскости, соединённых стрелочками.

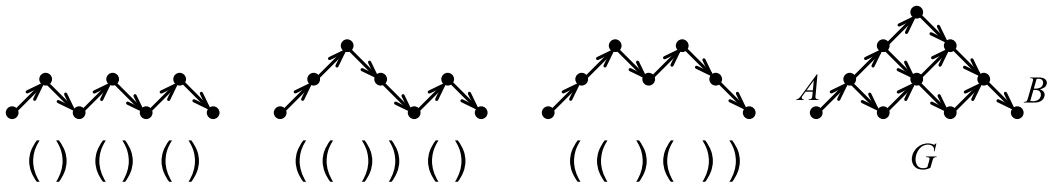


Рис. 4.1

Для каждой вершины этого графа можно посчитать число способов, которыми можно добраться до неё из вершины A.

И третья важная задача про скобочные выражения следующая:

Задача Л4.5. (Перебор правильных скобочных выражений) Напишите алгоритм, который по числу n выводит все правильные скобочные выражения длины $2n$.

Заметим, что все три задачи можно решить, используя *рекурсию*. Но при этом для второй задачи про число правильных скобочных выражений рекурсивное решение не эффективно, и не позволит вам найти число скобочных выражений большой длины (рекурсивный алгоритм будет работать слишком долго).

В этих задачах удобно рассматривать двухпараметрическое множество скобочных выражений (и соответствующее двухпараметрическое множество задач): *скобочные выражения длины n в которых присутствуют m открывающих скобок, для которых не нашлось парных закрывающих скобок.*

Форма Бэкуса-Наура

Наиболее распространённой формальной системой для описания языка с помощью правил вывода в удобном для человека и компьютера виде является форма Бэкуса-Наура (БНФ, Backus-Naur Form, BNF), которая была разработана для описания языка Algol 60.

В 1981 году Британский институт стандартов (British Standards Institute) опубликовал стандарт EBNF (Extended Backus Naur Form). По сути, это стандарт описания правил вывода, с дополнительными возможностями, которые позволяют записывать эти правила коротко.

Мы будем использовать расширение BNF, основанное на регулярных выражениях.

Вот пример описания языка целых чисел:

```
Integer         ::= Sign? UnsignedInteger
UnsignedInteger ::= NZDigit Digit+
Sign            ::= '+' | '-'
NZDigit ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Digit   ::= '0' | NZDigit
```

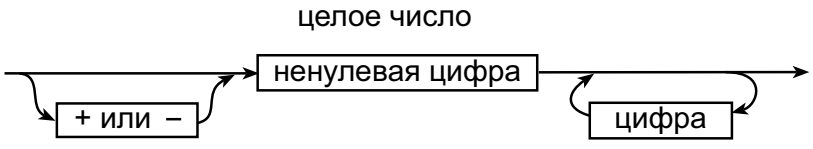


Рис. 4.2: Синтаксическая диаграмма языка целых чисел.

Последовательность трёх символов `::=` используется для разделения имени выводимого нетерминального символа правой части правила. Вертикальная черта «`|`» соответствует союзу «или». Терминальные символы и строчки терминальных символов записываются в одинарных кавычках.

Круглые скобки используют для группировки. Специальный символ «`*`» ставится после элемента, который в данном место может присутствовать любое число (0 раз, 1 раз, 2 раза, ...). Символ «`?`» ставится после объекта, который может отсутствовать (опциональное включение – 0 раз или 1 раз). И наконец, символ «`+`» ставится после объекта, который может присутствовать любое число раз больше 0 (1 раз, 2 раза, ...).

Например, из символа `S` по правилу

```
S  ::= 'a' ('b' | 'B')? ('c' | 'C')*
```

можно вывести слова

```
a, ab, aB, ac, aC, abc, aBc acc, abcc, accc, abccc, acCcc, abCccC, ...
```

Рассмотрим следующее правило EBNF:

```
A ::= B D? E* F
```

Оно может быть записано в виде пяти обычных правил вывода, без использования специальных знаков вывода:

```
A  ::= B DO ER F
DO ::= ∅
DO ::= D
ER ::= ∅
ER ::= ER E
```

Таким образом, операцию опционального вхождения «`?`» и операцию многократного вхождения «`*`» можно записать с помощью двух обычных правил. В последнем случае одно из правил является рекурсивным. Специальные символы «`*`», «`+`» и «`?`» позволяют записывать правила более кратко.

Арифметические выражения в обратной польской нотации

Язык арифметических выражений — это ещё не язык программирования. Но на задаче разработки программы-калькулятора можно обозначить много ключевых моментов теории языков программирования вообще.

Рассмотрим несколько необычную запись (нотацию) арифметических выражений, в которой сначала идут два операнда, разделённые пробелом, а затем знак арифметической операции. Например,

```
 5 6 *                    --   5 * 6
(5 6 *)  (2 3 *) +        --   (5 * 6) + (2 * 3)
 1 2 3 4 5 * + - /        --   1  / (2 - (3 + (4 * 5)))
```

Алгоритм 4.12 Калькулятор выражений в обратной польской нотации.

```
 1: function CalculateRPN
 2:     while есть следующий символ c do
 3:         switch(c)
 4:             c есть цифра  ⇒
 5:                 считать все последующие цифры и
 6:                 интерпретировать их все как число X;
 7:                 push (X);
 8:             (c = ' ') ⇒ continue;
 9:             (c = '*') ⇒ push( pop() * pop() );
10:             (c = '+') ⇒ push( pop() + pop() );
11:             (c = '-') ⇒ push(-pop() + pop() );
12:             (c = '=') ⇒ напечатать число pop();
13:         end switch
14:     end while
15: end function
```

b	1	2	3	*	+	4	-	=
действия	push 1	push 2	push 3	y=pop, x=pop, push $x{\cdot}y$	y=pop, x=pop, push $x{+}y$	push 4	y=pop, x=pop, push $x{-}y$	x=pop, putc x

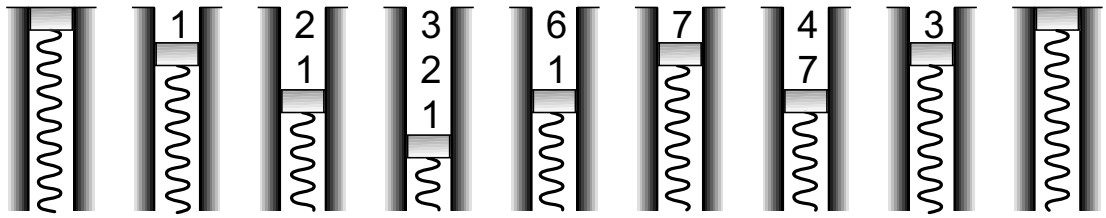


Рис. 4.3: Последовательные состояния стека калькулятора выражений в обратной польской нотации для входного выражения «1 2 3 * + 4 - =».

Эта нотация называется обратной польской нотацией. Заметьте, что во втором примере *скобки можно опустить* — выражение по-прежнему будет интерпретироваться однозначно.

Транслятор-вычислитель таких выражений естественно построить на основе *стека*.

Каждое считываемое число помещается в стек, а как только встречается арифметическая операция, из стека считываются два элемента (`a = pop()`, `b = pop()`), над ними производится соответствующая операция (условно обозначим её звёздочкой «*»), а результат заносится в стек (`push(a * b)`). Этот алгоритм вычисления представлен в виде псевдокода 4.12.

Задача Л4.6. Переведите арифметическое выражение «1 2 * 3 + 4 - 5 /» из обратной польской нотации в привычную алгебраическую нотацию.

Следует отметить, что в данном случае от записи алгоритма на псевдокоде до работающей программы пока далеко. Как считывать действительные числа? Что делать, если в выражении встречаются отрицательные числа? Было бы удобно, если бы на вход поступал не поток символов, а поток объектов, и число рассматривалось как один объект. Обычно так и строят программы лексического анализа текстов. Исходный поток символов разбивают на поток

более крупных единиц, которые называют **токенами**, а программа анализатор уже работает с потоком токенов.

Опишем множество правил, которые задают множество корректных арифметических выражений в обратной польской нотации:

```
exp ::= NUMBER
exp ::= exp ' ' exp ' +'
exp ::= exp ' ' exp ' -'
exp ::= exp ' ' exp ' *'
exp ::= exp ' ' exp ' /'
```

Эти правила следует дополнить ещё правилами, описывающими символ NUMBER. Множество слов, выводимых из символа NUMBER могут быть целыми или действительными числами. Есть несколько нотаций записи чисел. Попробуйте самостоятельно описать с помощью правил язык записи действительных чисел. Обратите внимание на символ пробела, который присутствует между двумя выражениями `exp`. Он необходим для того, чтобы соседние числа «не слипались» в одно число.

Как интерпретировать запись «`2 3 -4 - 5 * /`» – как $(2/((3-(-4))\cdot 5)$, или как $(2-3)/(4\cdot(-5))$? Можно ли оба минуса интерпретировать как унарные минусы?

Разработаны специальные системы, которые по множеству правил вывода создают программы — *синтаксические анализаторы* текстов, написанных на соответствующем языке. Эти программы могут распознавать слова, выводимые из заданных правил, и делать более сложные действия.

В частности в системе `bison` можно задать указанные выше пять правил, и поставить каждому правилу в соответствие *действие* которое следует применять к *значениям* элементов правой части правила, чтобы получить *значение* символа, который стоит в левой части правила.

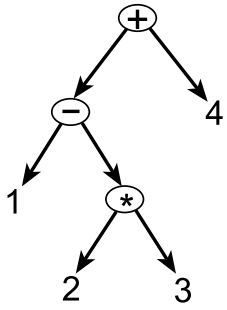
Системы для создания синтаксических анализаторов — отдельная обширная тема. Более подробно про эти системы вы можете узнать в документации системы `bison` или `yacc`.

Приведём здесь лишь простой пример правил, описывающие калькулятор выражений в обратной польской нотации:

```
exp:      NUMBER           { $$ = $1;         }
        | exp exp '+'      { $$ = $1 + $2;    }
        | exp exp '-'      { $$ = $1 - $2;    }
        | exp exp '*'      { $$ = $1 * $2;    }
        | exp exp '/'      { $$ = $1 / $2;    }
```

Получив на вход эти правила, система `bison` сгенерирует программу на Си (Си++), которая будет готовым калькулятором. Более подробно этот пример рассатривается в дополнительной лекции 23 на странице 378.

Такой подход — автоматическая генерация программ — стал неотъемлемым этапом разработки сложных систем. Это простой пример отображает мета-переход при решении задач — переход от программ, решающих задачу, к программам, которые генерят программы, которые решают задачу. Мета-переходы сегодня играют важную роль в индустрии компьютерных систем.



- $((1 - (2 * 3)) + 4)$
- $(+\ (-\ 1\ (*\ 2\ 3))\ 4)$
- $1\ 2\ 3\ *\ -\ 4\ +$

Рис. 4.4: Дерево синтаксического разбора выражения «$((1 - (2 * 3)) + 4)$».

На рисунке 4.4 нарисовано дерево синтаксического разбора арифметического выражения $(1 - (2 * 3)) + 4$. Справа от дерева приведены три нотации записи этого выражения: 1) алгебраическая (инфиксная); 2) польская (префиксная); 3) обратная польская (постфиксная).

Первая, алгебраическая, наиболее нам привычна. Она получается, если все объекты с дерева спроецировать вниз на горизонталь и окружить скобками группы объектов из одного поддерева.

Польская нотация наиболее удобна для программистов, так как по сути представляет собой удобную для разбора запись этого дерева. Описание дерева начинается открывающейся скобкой, затем идёт идентификатор вершины дерева, а затем следуют описание поддеревьев в аналогичном формате, а затем идёт закрывающая скобка:

```
tree     ::= NUMBER
tree     ::= '(' operator tree tree ')'
operator ::= '+' | '-' | '*' | '/'
```

На рисунке 4.5 показана соответствующая синтаксическая диаграмма записи арифметических выражений в польской нотации. Согласно польской нотации построен синтаксис языка Lisp (scheme). Используя польскую нотацию, удобно форматировать документы (см. рис. 4.6).

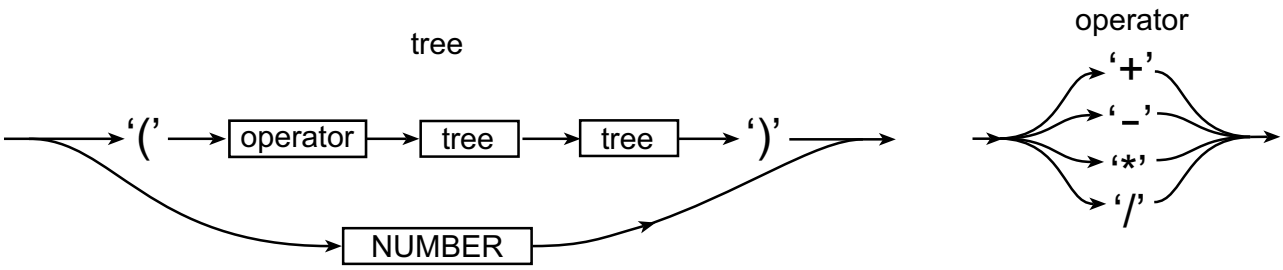


Рис. 4.5: Синтаксическая диаграмма записи арифметических выражений в префиксной форме.

Польская нотация на примере языка форматирования документов:

```
(paragraph
    На уроке решали следующие задачи
    (list
       (li 1 Числа Фибоначчи)
       (li 2  НОД)
    )
    Получили следующие результаты
    (table
       (head (hi       ) (hi 1) (hi 2))
       (row  (ri Иванов) (ri -) (ri +))
       (row  (ri Петров) (ri +) (ri +))
       (row  (ri Котов ) (ri +) (ri -))
    )
)
```

Польская нотация на примере языка арифметических выражений:
Запись

```
(+
    (* 2 3 )
    (+
        (* 6 7)
        (* 8 9)
    )
)
```

соответствует выражению

```
(2*3)+((6*7)+(8*9))
```

Рис. 4.6: Примеры использования польской (префиксной нотации) а) для структурирования документов; б) для записи арифметических выражений.

Обратная польская нотация также активно применяется. Согласно этой нотации устроен язык описания векторной графики PostScript.

Выражения в обратной польской нотации (постфиксной) вычисляются быстрее, нежели выражения в польской (префиксной) или алгебраической (инфиксной). Поэтому обратную польскую нотацию использую там, где одно и то же выражение (программу) необходимо вычислить (выполнить) для большого числа значений аргументов, и где скорость работы критична. Например при запросе к базам данных, может быть указано логическое выражение, которому должны удовлетворять запрашиваемые записи. Логическое выражение переводится в постфиксную форму, затем вычисляется последовательно для всех записей, что позволяет немного ускорить процесс отбора нужных записей.

Постфиксная запись удобна для машинной обработки, но часто неестественна для человека. Она соответствует случаю, когда глаголы ставится в конце предложения: «задание выполнить», «хлеб магазин в купить», «Иван задача_N2 решение поручить», ...

Семинар 4

Распознавание языков с помощью абстрактных исполнителей

Краткое описание: Завершающий семинар по абстрактным исполнителям посвящён задачам на конструирование алгорифмов Маркова и машин Тьюринга, которые распознают языки, порождённые простыми грамматиками.

Задачи по алгорифмам Маркова

Задача С4.1. Напишите алгорифм Маркова, который получает на вход слова из нулей и единиц и применим только к словам, в которых нечётное число единиц.

Задача С4.2. Напишите алгорифм Маркова, в котором не более пяти правил подстановки и который применим ко всем словам в алфавите $A = \{\mathtt{a}, \mathtt{b}\}$, кроме слова `aba`.

Задача С4.3. В алфавите $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ описать алгорифм Маркова, который выдаёт в качестве результата пустое слово, если буквы `a` и `b` входят во входное слово в равном количестве, и любое непустое слово — иначе.

Задача С4.4. Назовём слово *хорошим*, если выполнено одно из условий:

- Оно равно слову «`0`».
- Оно начинается на символ «`1`», а затем следует *хорошее* слово.
- Оно начинается на символ «`2`», а затем следует два записанных друг за другом *хороших* слова.
- Оно начинается на символ «`3`», а затем следует три записанных друг за другом *хороших* слова.

Это определение *рекурсивно*. Напишите алгорифм Маркова, который применим только к хорошим словам. Примеры хороших слов: `10`, `200`, `3000`, `21010`, `2010`, `3020010`, `11111110`, `30011111110`. Множество *хороших* можно определить формально с помощью *правил вывода*:

```
S ::= '0'
S ::= '1' S
S ::= '2' S S
S ::= '3' S S S
```

Эти четыре правила являются формальной записью данного словесного описания.

Задача С4.5. Можно ли сконструировать алгорифм Маркова, который получает на вход натуральное число, записанное в десятичной системе счисления, и возвращает слово `YES` для простых чисел и слово `NO` — для составных?

При описании правил вывода мы будем использовать круглые скобки для группировки, символ `?` — для указания того, что объект, стоящий слева, может быть пропушен (повторение 0 или 1 раз), символ `+` — для указания того, что объект, стоящий слева, может быть повторён любое число раз (повторение 1 или более раз), символ `*` — для указания того, что объект, стоящий слева, может быть повторён любое число раз или пропущен (повторение 0 или более раз), символ `|` — для обозначения союза «или».

Задача С4.6. Напишите алгорифм Маркова, который применим только к словам, выводимым из символа `A`:

```
A ::=  B+ | C+  | B+ 'a' C+
B ::= 'b' | 'B'
C ::= 'c' | 'C'
```

Задача С4.7. Напишите алгорифм Маркова с алфавитом $B = \{0, 1, (,), [,]\}$, который применим только к словам, выводимым из символа `S`:

```
S ::= A | B | '1' | '0'
A ::= '(' B  B ')'
B ::= '[' A  A ']'
```

Задача С4.8. Напишите алгорифм Маркова, который применим только к словам, выводимым из символа `A`:

```
A ::= ('a' (B | C)? )+
B ::=  'b' | 'B'
C ::=  'c' | 'C'
```

Задача С4.9. Напишите алгорифм Маркова, который применим только к словам вида
а) `AB`, `AABB`, `AAABBB`, `AAAABBBB`, ...;
б) `ABC`, `AABBCC`, `AAABBBCCC`, ...
Можно ли данные языки задать с помощью грамматик?

Задача С4.10. Напишите алгорифм Маркова, который распознает корректно записанные целые числа, задаваемые грамматикой

```
Integer         ::= Sign? UnsignedInteger
UnsignedInteger ::=  NZDigit Digit+  | '0'
Sign    ::=  '+' | '-'
NZDigit ::=  '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Digit   ::=  '0' | NZDigit
```

Для слов, выводимых из символа `Integer`, алгорифм должен получать слово «`YES`», а для остальных слов — «`NO`».

Задачи по машинам Тьюринга

Задача C4.11. Напишите машину Тьюринга, которая получает на вход скобочное выражение и возвращает слово `YES`, если оно правильное, и `NO` — иначе.

Решите задачи C4.1–C4.10 для машин Тьюринга.

Теоретические задачи

Задача C4.12. Приведите пример задач, которые проще решаются на исполнителе алгорифмов Маркова, чем на исполнителе машин Тьюринга, и задач, которые проще решаются на исполнителе машин Тьюринга. Предложите решения этих задач для обоих исполнителей и оцените скорость роста времени работы исполнителей с увеличением размера входных данных. Считайте, что одна подстановка при выполнении алгорифмов Маркова осуществляется за константное время, независимое от длины хранимого слова и длин слов, участвующих в подстановке, и относительного смещения найденной подстроки относительно начала.

Задача C4.13.
а) Существует ли алгорифм Маркова (машина Тьюринга), который получает на вход описание некоторого алгорифма Маркова (машина Тьюринга) и определяет (возвращает слово `YES` или `NO`), существует ли для этого алгорифма вход, на котором он не останавливается?
б) Существует ли алгорифм Маркова (машина Тьюринга), который получает на вход описание некоторого алгорифма Маркова (алгоритма для машины Тьюринга) и применим только к таким алгорифмам (машинам), для которых существует вход, на котором они останавливаются?
в) Существует ли алгорифм Маркова (машина Тьюринга), который получает на вход описание некоторого алгорифма Маркова (машины Тьюринга) и применим только к таким алгорифмам (машинам), для которых существует вход, на котором они не останавливаются?

Задача C4.14. Докажите, что в множестве всех слов алфавита $B = \{0,\ 1\}$ существуют неразрешимые подмножества. Приведите пример неразрешимого множества.

Задача C4.15. Докажите, что пересечение, объединение и разность двух разрешимых множеств является разрешимым множеством.

Следующие задачи (C4.16–C4.18) имеют повышенную сложность и подразумевают программирование на некотором алгоритмическом языке. Прежде, чем решать эти задачи, следует разработать формальные языки описания конечных автоматов, машин Тьюринга и алгорифмов Маркова.

Задача C4.16.* Напишите программу-эмулятор исполнителя **I**. Эта программа должна получать в качестве аргумента имя файла, в котором хранится текст программы для исполнителя **I**. На стандартный поток ввода поступают входные данные. Программа должна отработать действия, которые выполнил бы исполнитель **I**. По окончании работы программа должна вывести на стандартный поток вывода результат вычислений.
а) $\mathbf{I}$ = исполнитель конечных автоматов с действиями;
б) $\mathbf{I} = \mathbf{I}_{\mathrm{T}}$; в) $\mathbf{I} = \mathbf{I}_{\mathrm{M}}$.
В случае а) следует реализовать возможность вывода символов и совершения действий со сте-

ком. В случае б) входные данные включают в себя описание состояния ленты и начального положения управляющего устройства. Добавьте возможность включения режима отладки, в котором программа-эмулятор выводит описание каждого шага исполнения эмулируемого исполнителя.

Задача С4.17.* Напишите программу, которая получает на вход текст программы для исполнителя $\mathbf{I}_1$ и возвращает текст эквивалентной программы для исполнителя $\mathbf{I}_2$:
а) $\mathbf{I}_1 = \mathbf{I}_\mathrm{T}$, $\mathbf{I}_2 = \mathbf{I}_\mathrm{M}$; б) $\mathbf{I}_1 = \mathbf{I}_\mathrm{M}$, $\mathbf{I}_2 = \mathbf{I}_\mathrm{T}$.

Задача С4.18.* Напишите программу, которая получает на вход описания двух конечных автоматов и возвращает 0 или 1 в зависимости от того, являются они вычислительно эквивалентными или нет.

Часть II

Язык программирования Си

Лекция 5

Базовые понятия языка программирования Си

Краткое описание: В этой лекции мы начнём изучение алгоритмического языка программирования Си. Будут рассмотрены базовые понятия: функции, переменные, операторы структурного программирования, сборка (компиляция) программ.
Ключевые слова: файл, программа, исполняемый код, машинный код, функция, переменная, тип переменной, размер типа, объявление переменной, объявление функции, оператор, препроцессинг, компиляция, компоновка, библиотека.

Программа и сборка программ

Напомним, что программа, в общем случае, — это описание логики действия для некоторого исполнителя. Соответственно, программу на языке Си также можно рассматривать как описание логики действий некоторого абстрактного исполнителя — исполнителя программ на языке Си. Этот исполнитель абстрактный, в действительности такого исполнителя нет, его не воплотили в физическое устройство и не собираются это делать.

Программы, написанные на языке Си, проходят через стадию преобразования в последовательность инструкций для реальной вычислительной машины — **исполняемый код**, который непосредственно исполняется вычислительной машиной, а точнее, её центральным вычислительным устройством — процессором. Исполняемый код ещё называют **машинным кодом**.

Указанное преобразование происходит в два этапа – **компиляция** и **компоновка** – которые осуществляются соответственно двумя программами: **компилятором языка Си** и **компоновщиком**. Во время компиляции, каждый файл, содержащий часть текста программы, преобразуется в промежуточный **объектный** файл. Затем компоновщик собирает исполняемый код из объектных файлов. Более подробно процессы компиляции и компоновки будут рассмотрены нами позже.

Язык Си — **императивный**. Это означает, что программы на языке Си представляют собой описание последовательности действий. При этом язык позволяет определять новые действия (макродействия), которые складываются из последовательности элементарных действий или других макродействий. Роль таких макродействий в языке Си играют **функции**.

Абстрактный исполнитель программ на Си предоставляет возможность пользоваться памятью. Элементарной единицей памяти является байт. В программах на Си можно создавать именованные блоки байт для хранения чисел, символов или их массивов. Также можно хранить произвольные данные, кодируя их последовательностью байт. Именованные блоки байт называются **переменными**.

Переменные

В алгоритмических языках состояние исполнителя задаётся (описывается) в виде конечного набора переменных. Каждая переменная представляет собой хранилище информации определенной ёмкости (размера). **Размером переменной** называют число выделенных под переменную байт. Входные данные алгоритма обычно размещают в переменных, которые называют **входными параметрами** или **входными переменными**. Количество переменных, их размер и интерпретацию (определение того, что каждая переменная *означает*) выбирают исходя из решаемой задачи. Процесс выбора переменных называют **параметризацией задачи**.

Например, если мы решаем задачу определения минимального расстояния от точки до прямой на плоскости, то нашими параметрами являются координаты точки, и коэффициенты уравнения прямой. Всего в этой задаче 5 параметров одинаковой природы. Для каждого параметра выделяют переменную. Также иногда выделяют отдельную, так называемую **выходную переменную**, которая принимает значение результата решения — в нашем случае — расстояния.

Помимо входных и выходных переменных, которые являются частью состояния исполнителя, выделяют также **внутренние переменные**, которые дополняют набор переменных и соответственно увеличивают множество возможных состояний исполнителя.

Типы переменных

Как упоминалось выше, переменные могут иметь различную природу и размер, что в совокупности называют **типом переменной**. Большинство языков имеют дело с числами. Числа могут иметь свою природу: целые, натуральные, действительные, комплексные. В языке Си каждый тип имеет свое имя. В нём различают целые знаковые, целые беззнаковые и действительные числа, которые имеют стандартные имена: `int`, `unsigned`, `float` соответственно. Также в языке Си имеются дополнительные типы: `char` — для хранения символов, `long`, `unsigned long` — для хранения больших целых чисел, `double` — для хранения вещественных чисел двойной точности[1].

[1]Ёмкость базовых типов переменных в большинстве алгоритмических языков ограничена и для каждого компилятора чётко зафиксирована. Размер переменной типа `float` обычно равен 32 бита. Это значит, что переменная типа `float` может принимать $2^{32} \approx 4 \cdot 10^9$ возможных значений. Соответствующие этим значениям действительные числа некоторым образом распределены по числовой прямой. Можно говорить о среднем расстоянии между соседними числами, представимыми типом `float`, в окрестности некоторого действительного числа (плотность распределения меняется вдоль числовой прямой). Можно также говорить о количестве действительных чисел на некотором отрезке $[a, b]$, представимых типом `float`. Есть понятия самого маленького положительного числа и самого большого положительного числа, представимого типом `float`. Конечность типа `float` ограничивает точность вычислений над переменными этого типа — все вычисления делаются с точностью до расстояния между соседними числами, представимыми типом `float`. При сложных последовательных вычислений с одними и теми же числами ошибка, связанная с машинным представлением действительных чисел, может накапливаться и сильно искажать результат. Тип `double` имеет размер 64 бита. Этот тип охватывает больший отрезок на действительной оси и плотность чисел, представимых типом `double` примерно в 2^{28} раз больше, чем у типа `float` (между двумя соседними числами `float` находятся $2^{28} - 1$ чисел типа `double`).

Для определения размера переменной определённого типа в языке Си используется встроенный оператор `sizeof(x)`, где `x` может быть либо именем переменной, либо именем типа переменной. Он возвращает количество байт, использованных под указанную переменную (переменную указанного типа). На семинаре мы рассмотрим пример кода на языке Си, который выводит размеры переменных различных типов.

Рассмотрим тип `int`. На компьютерах 32-разрядной архитектуры его размер равен 4 байта, то есть 32 бита. Неотрицательные числа меньшие 2^{31} представляются в виде двоичной записи. Старший 32-й бит в их представлении равен нулю. Отрицательное число $-a$, модуль a которого меньше либо равен 2^{31}, записывается в 32 битах как двоичная запись положительного числа $2^{32} - a$. Старший бит для отрицательных чисел равен 1, так как $2^{31} \leqslant 2^{31} - a < 2^{32}$ при $1 < a \leqslant 2^{31}$.

Таким образом, в типе `int` можно представить целые числа из отрезка $[-2^{31},\ 2^{31} - 1]$. Например, отрицательное число -11 будет записано в памяти как беззнаковое представление числа $2^{32} - 11$:

$$\begin{aligned} 2^{32} - 11 &= 1\ 00000000\ 00000000\ 00000000\ 00000000_2 - 1011_2 = \\ &= 11111111\ 11111111\ 11111111\ 11111111_2 - 1010_2 = \\ &= 11111111\ 11111111\ 11111111\ 11110101_2 \end{aligned}$$

Здесь во второй строчке мы для удобства уменьшили на 1 уменьшаемое и вычитаемое.

Примеры представления отрицательных чисел:

-1	`11111111 11111111 11111111 11111111`
-128	`11111111 11111111 11111111 10000000`
-256	`11111111 11111111 11111111 00000000`
-255	`11111111 11111111 11111111 00000001`
$-(2^{10} + 2^3 + 1)$	`11111111 11111111 11111011 11110111`
$-(2^{10} + 2^3)$	`11111111 11111111 11111011 11111000`

Таким образом, чтобы получить представление отрицательного числа $-a$ в 32 битах, необходимо взять его модуль a, вычесть из него единицу, записать результат в двоичном виде, дописав слева нули так, чтобы всего было 32 цифры, а затем инвертировать все биты.

Такое представление отрицательных целых чисел называется *дополнительным кодом*. Есть также представление, называемое *обратным кодом*, в котором один бит используется под знак, а в остальных битах записывается двоичная запись модуля числа. При таком представлении есть два различных представления числа 0: «+0» и «−0». Этого недостатка нет у дополнительного кода. Кроме того, дополнительных код имеет важное преимущество: если складывать или умножать отрицательные числа, интерпретируя их представление как беззнаковое, то результат (интерпретируемый уже как знаковый) будет верный.

Например, для сложения чисел $-a$ и $-b$ достаточно сложить их беззнаковые представления $2^{32} - a$ и $2^{32} - b$. Получим, $2^{33} - a - b$. Если $a + b$ не слишком большие, а именно, $a + b \leqslant 2^{31}$, то младшие 32 разряда (бита) числа $2^{33} - a - b$ точно такие же, как и у числа $2^{32} - a - b$, которое соответствует знаковому представлению числа $-a - b$.

Объявление переменных

Переменные задаются в алгоритмических языках с помощью специальной синтаксической конструкции **объявления переменных** (variable declaration). В Си объявление переменной состоит из имени типа переменной и имени самой переменной или имён переменных, разделенных запятыми. Приведём пример объявления нескольких переменных в языке Си:

```
int ivar;
double a, b;
long x;
```

В первой строке мы объявили переменную с именем `ivar` целого типа `int`. Во второй — две переменные `a` и `b` вещественного типа двойной точности `double`. В третьей строке мы объявили переменную с именем `x` целого типа `long`.

Точное определение синтаксиса объявления переменной в расширенной нотации BNF выглядит следующим образом:

```
var_declaration_statement  ::= var_decl ';'
var_decl ::= type_name sp var_name sp ? (',' sp var_name)*
var_name ::= identifier
type_name ::= identifier
identifier ::= alpha ( alpha | number | '_' )*
sp ::= (<пробел> | <знак табуляции> | <перевод строки>)+
```

Везде далее мы *не* будем в описании синтаксиса использовать символ `sp`, обозначающий некоторое количество пробельных символов. Пробельные символы никак не учитываются компилятором, их можно в любом количестве вставлять между двумя элементами языка, и мы это будем неявно подразумевать.

Функции

Важным понятием в алгоритмических языках является понятие **функции**. Функция представляет собой формализацию понятия задачи или подзадачи, на множество которых разбивается задача. Функция имеет **имя**, **набор входных параметров** (входных переменных), которые задаются аналогично переменным, тип возвращаемого значения (результата) и **тело** — описание внутреннего устройства (логики работы функции).

Функцию можно **объявить**, **определить** и **вызвать**, причём вызвать функцию можно только после её объявления

Объявление функции

Объявление функции (function declaration) включает в себя лишь её имя, тип возвращаемого значения и объявление входных параметров. Этого вполне достаточно компилятору, чтобы сделать проверку корректности вызовов данной функции. Определение функции дополняет объявление описанием тела функции. А вызов состоит только из имени функции и значений параметров, передаваемых функции. Рассмотрим каждую конструкцию и её предназначение по порядку.

Вот пример объявления функции, принимающей на вход три параметра вещественного типа двойной точности (`double`) и возвращающей значение целого типа.

```
int overlap(double lb, double le, double x);
```

Вначале описывается тип возвращаемого значения, затем имя функции, затем – в круглых скобках — перечисленные через запятую объявления параметров. Следует обратить внимание на символ ';' в конце объявления. Такого объявления, вполне достаточно языку Си для того, чтобы можно было *вызвать* функцию `overlap`. Объявление также называют *декларацией функции*.

Ниже приведено формальное описание **синтаксиса объявления функции**:

```
function_declaration_statement ::= function_decl ';'
function_decl ::= type_name function_name '(' var_decl * ')'
function_name ::= identifier
```

Определение функции

В определении функции, помимо всех частей, указанных в объявлении, присутствует *тело* — описание алгоритма, выполняемого в данной функции. В языке Си, тело функции в определении следует в фигурных скобках сразу за перечислением входных параметров.

```
int overlap(double lb, double le, double x) {
    // начало тела функции
    if ( x < lb ) {
        return -1;    // возвращаем результат -1, если x < lb;
    } else if ( x > le ) {
        return 1;     // возвращаем результат 1, если x > le;
    } else {
        return 0;     // иначе возвращаем результат 0;
    }
}
```

В примере мы использовали комментарии — поясняющий текст, который идёт после двух символов слэш (наклонная вправо черта) до конца строки. Этой возможностью мы будем пользоваться ещё не раз.

Формальное описание **синтаксиса определения функции** выглядит следующим образом:

```
function_definition ::= function_declaration body
body ::= ('{' statement* '}') | statement
statement ::= (expression | operator | variable_declaration) ';'
```

В стандарте ANSI C объявления переменных и, возможно, присвоение им начальных значений внутри блока должны идти первыми, до других команд. Например:

```
int level(int x, int y) {
    int c = 1;    // объявляем переменную c и инициализируем
                  // её значением 1;
    int a = 2, b = 3, z; // объявляем три переменные a, b, z
    z = a*x + b*y+c;     // присваиваем переменной z
```

```
                        // значение ax+by+c;
    return z / 2;       // возвращаем значение z/2 в качестве
                        // результата вычисления функции level;
}
```

Вызов функции

Вызов функции осуществляется путём задания имени функции и перечисления значений параметров в круглых скобках. Например, вызов функции `overlap` будет выглядеть следующим образом:

```
int is_overlap;
is_overlap = overlap(0.0, 1.0, -1.2);
```

Перед вызовом, мы объявили переменную `is_overlap` и затем присвоили ей значение результата вызова функции `overlap` для последующей обработки. Следует заметить, что порядок следования параметров вызова функции, должен совпадать с порядком следования параметров в объявлении.

Разделение объявления и определение также делает непротиворечивым использование рекурсивных вызовов, а именно, в теле функции могут содержаться вызовы этой же функции, поскольку формально, её объявление было дано до тела.

```
function_call ::= function_name
                  '(' (expression (',' expression)* )? ')'
```

Приоткроем завесу над тайной разделения описания функции на объявление и определение. Причиной тому — поддержка модульной архитектуры программ.

Каждый модуль рассматривается как самостоятельный элемент программы, компилирующийся отдельно. Поскольку сущность модуля составляют тела функций, то в одном модуле вполне может не оказаться тел некоторых вызываемых в модуле функций — они могут находится в других модулях.

Само применение модулей решает две задачи: поддержка логического разделения функций на обособленные группы (для удобства неоднократного применения в различных проектах) и упрощение процесса разработки (удобство компиляции и разработки программы по частям). Сложность современных компьютерных систем велика и ясно, что единственный способ преодоления сложности — это конструирование их из отдельных кирпичиков.

Разделение на модули, также позволяет избежать повторной компиляции всех функций программы, в случае если изменилась только одна. Это особенно критично для проектов, где компиляция всего кода занимает несколько часов.

Для компиляции отдельного модуля не требуется, чтобы все используемые функции имели известные тела. Достаточно, чтобы каждая вызываемая функция имела известный компилятору *прототип* (function prototype), то есть была объявлена. Имея прототип функции, компилятор сможет проверять, правильно ли вызывается эта функция и правильно ли читается результат (проверяется соответствие типов). Компилятор будет «знать» типы аргументов и тип возвращаемого результата и сможет правильным образом оформить вызов функции.

Операторы структурного программирования

Условный оператор `if`

Оператор `if` позволяет выбрать один из нескольких путей дальнейших действий исполнителя в зависимости от условия, выражаемого через текущее состояние исполнителя. Поэтому оператор `if` называют **условным оператором**, а место его появления в алгоритме — **ветвлением**.

Приведём пример использования оператора `if` в языке Си.

```
if ( a > b ) {   // если a > b
    b = a % b; // в b поместить остаток от деления a на b
} else {       // иначе
    a = b % a; // в b поместить остаток от деления b на a
}
```

Как видно, общая синтаксическая конструкция выглядит достаточно просто: сначала пишется имя самого оператора `if`, затем, в скобках записывается условие в виде вычисляемого выражения. Далее, в фигурных скобках следует описание ветви алгоритма, выполняющейся в случае, если результат вычисления условного выражения окажется ненулевым, то есть истинным[2]. Альтернативная ветвь, которая должна быть выполнена в случае ложного результата условного выражения, определяется после ключевого слова `else`. Выполнение действий после выполнения соответствующей ветки оператора `if` продолжается с того места, где заканчивается описание последней ветки.

Формальное определение условного оператора:

```
if_op ::= 'if' '(' expression ')' body ('else' body)?
```

Обратите внимание, что описание альтернативной ветки со словом `else` можно опустить.

Арифметический цикл `for`

Если какое-то действие (или известный набор действий) требуется выполнить известное конечное число раз, то на помощь приходит конструкция **арифметический цикл**, реализуемый с помощью оператора `for`. Рассмотрим простой пример:

```
int i;
for( i = 0; i < 1000; i++ ) {
    printf("%d\n", i * i);
}
```

Этот код 1000 раз выполнит строку `printf("%d\n", i)` и в результате выведет квадраты первой 1000 неотрицательных целых чисел. При этом переменная `i` будет последовательно принимать значения 0, 1, 2, ..., 999. Значение 1000^2 напечатано не будет, переменная `i` достигнет значения 1000 и цикл прервётся.

В языке Си конструкция `for` состоит из имени оператора `for`, трёх выражений в круглых скобках `A`, `B`, `C`, разделённых точкой с запятой, и тела цикла `D` (см. рис. 5.1 (а)).

[2]В стандарте языка Си нет выделенного логического (булевского) типа, который может принимать два значения: *истина* и *ложь*. Вместо этого используется целочисленный тип `int`. Причём истинным считается любое ненулевое значение, а ложным – нулевое.

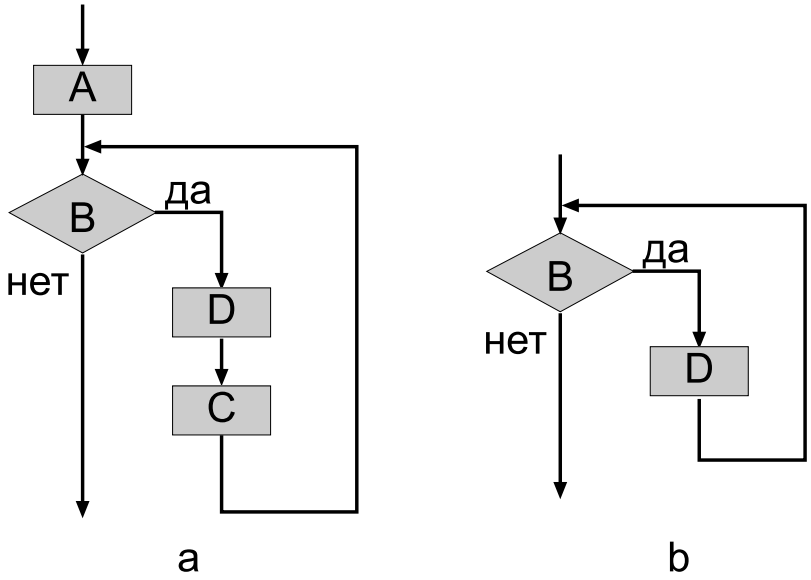


Рис. 5.1: Схемы циклов: а) `for(A; B; C ) D`; б) `while (B) D`.

Выражения `A`, `B`, `C` — это выражение инициализации, условие цикла и инкрементальное выражение цикла, соответственно.

В нашем примере, выражением инициализации является выражение `i = 0`, условием цикла — выражение `i < n`, а инкрементальным выражением — `i++`.

Тело цикла выполняется несколько раз, пока условие цикла истинно. При этом, если условие цикла изначально ложно, то тело цикла не выполнится ни разу. Каждый шаг выполнения тела цикла называется **итерацией цикла**.

Инкрементальное выражение `i++` вычисляется после каждой итерации цикла. Можно считать, что оно дописано в конец тела цикла.

Конструкцию типа `for(i = 0; i < n ; i++){ ... }` принято использовать тогда, когда переменные `n` и `i` не изменяют своих значений внутри тела цикла. Хотя менять их значение внутри цикла не запрещено.

Формальное определение синтаксиса оператора `for`:

```
for_op ::= 'for' '(' init ? ';' cond ? ';' iter ? ')' body
init   ::= expression
cond   ::= expression
iter   ::= expression
```

Приведём примеры арифметических циклов

```
// Вывод чисел 10, 9, 8, ..., 1, 0
for( i = 10; i >= 0; i-- ) {
    printf( "%d\n", i );
}
// Вывод чисел 1, 2, 4, 8, 16, ..., 1024
for( i = 1; i < 2000; i = i*2 ) {
    printf( "%d\n", i );
}
```

```
// Снова вывод чисел 1, 2, 4, 8, 16, ..., 1024
for( j = 1, i = 0; i <= 10; i++, j = j*2 ) {
    printf( "%d\n", j );
}
```

В последнем примере использовалась возможность объединения выражения с помощью запятой. При вычислении выражения «`X, Y`» вычисляются последовательно выражения `X` и `Y`, а результат равен значению последнего выражения `Y`.

Цикл `while`

Конструкция цикла `while` похожа на конструкцию цикла `for` (см. рис. 5.1 (б)), но не содержит выражения инициализации `A` и выделенного инкрементального выражения `C`.

Пример использования оператора `while` для организации цикла:

```
double x = 0.1;
while ( x < 0.99999 ) {
    x = 4 * x * (1 - x);
    printf("%lf\n", x);
}
```

Цикл `while` используется в алгоритмах, когда требуется повторять некоторую последовательность действий, пока не будет достигнут необходимый результат. Причём, количество повторений, необходимых для этого, заранее неизвестно, а определяется исходя из условий достижения результата.

В языке Си, конструкция цикла `for` является наиболее общей, но, на практике `for` используется именно для подчеркивания арифметичности цикла, а `while` — наоборот,и для подчеркивания неарифметичности. В некоторых языках предусмотрены специальные конструкции для арифметических циклов, например, циклы `for` в языках Бейсик или Паскаль, или конструкция `foreach` во многих других языках, в то время как в Си, такого выделенного средства для организации арифметических циклов — нет.

Формальное определение синтаксиса оператора `while`:

```
while_op ::= 'while' '(' condition ')' body
```

Этапы сборки программы

Процесс сборки программы на языке Си проходит в два этапа: компиляция и компоновка. Перед компиляцией каждого файла с исходным кодом осуществляется **препроцессинг** исходного кода.

Препроцессинг

На этапе препроцессинга происходит выполнение директив препроцессора. В частности, выполняются директивы `#include`, `#define`, директивы условной компиляции `#ifdef`, `#else`, `#endif` и ряд других.

Директива `#include` используется для включения в код программы содержимого другого файла. Например, директива `#include <stdio.h>` включает в текст вашей программы содержимое файла `stdio.h`.

Директива `#define` используется для задания макроопределений. Бывают макроопределения с параметрами и без:

```
macro_def  ::= #define identifier letters
macro_def  ::= #define identifier '(' arg_list ')' letters
arg_list   ::= identifier ( ',' identifier )*
```

Например,

```
#define N 100
#define ERROR_MSG "Ошибка выполнения программы!"
#define begin  {
#define end    }
#define foreach(i,n) for(i = 0 ; i < n ; i++)
```

Макроопределения записываются в одну строчку. После строки `#define` идет идентификатор, обозначающий *имя макроопределения*, а далее идёт любой набор символов, раскрывающий *значение макроопределения*.

Препроцессор действует согласно простому алгоритму — везде, где в тексте встречается имя макроопределения, он заменяет его на его значение.

Подробнее с директивами препроцессора мы познакомимся на семинаре.

Пока важно отметить наличие этого первого этапа обработки текстового файла на языке Си. Полученный в результате текстовый файл поступает на вход компилятору.

Компиляция

При компиляции файла на языке Си происходит трансляция функций в машинный код, который затем будет непосредственно исполнятся процессором вычислительной машины. Машинный код — это последовательность байт, обозначающих различные инструкции для процессора, это «родной язык» вычислительной машины.

В простейшем случае, каждый файл `имя_файла.c` с кодом на языке Си, предварительно обработанный препроцессором, будет *скомпилирован* в объектный файл `имя_файла.o` (в операционных системах семейства Unix) или `имя_файла.obj` (в операционных системах семейства Windows).

Грубо говоря, объектные файлы состоят из тел функций в машинном коде и двух специальных карт имён: карта имён определённых функций и глобальных переменных и карта имён вызываемых внешних функций и внешних переменных. Первая карта ставит в соответствие имени функции или переменной её адрес, а вторая — наоборот, месту вызова внешней для данного файла функции или обращения к внешней переменной – её имя.

Компилятор вызывается следующей командой:

```
> gcc -c module.c -o module.o
```

которая означает следующее: «скомпилировать исходный файл `module.c`, результирующий объектный файл назвать `module.o`»[3].

[3]Общепринятой практикой, за исключением особых случаев, является использование того же имени для объектного файла, что и для исходного.

Компоновка

Компоновка является завершающим этапом сборки исполняемого кода. В компоновке участвуют все объектные файлы, полученные на этапе компиляции. При компоновке можно подключать **библиотеки** — наборы скомпилированных функций и глобальных переменных. Например, файл `libm.so` содержит скомпилированные тела математических функций и переменных, объявленных в заголовочном файле `math.h`.

Компоновщик «смотрит», какие функции вызываются в каждом объектном файле, ищет их тела в других объектных файлах и библиотеках и подставляет в местах их вызова адреса из соответствующих карт. То же самое происходит и с глобальными переменными.

Во время компоновки конструируется исполняемый файл — все объектные файлы, поданные на вход компоновщику, преобразуются в один.

Но даже если программа состоит из одного объектного файла, то компоновщик не бездействует. Он проверяет, что все вызываемые функции имеют тела, иначе он возвращает ошибку «вызов неопределённой функции xxx» («undefined reference 'xxx'»). Ошибка возникнет и в том случае, если у какой-либо функции будет два тела — «функция xxx дважды определена» («multiple definition of 'xxx'»).

То же самое касается глобальных переменных. Компоновщик следит, чтобы глобальные переменные объявлялись и каждая их них объявлялись ровно один раз.

Если среди всех объектных файлов компоновщиком не будет обнаружена функция `main`, то он выдаст соответствующее предупреждение и откажется создавать исполняемый файл.

Компоновщик (как и компилятор) вызывается командой `gcc`. Например, для компоновки программы, содержащихся в нескольких объектных файлах, необходимо выполнить следующее:

```
> gcc -o program module1.o module2.o module3.o
```

где `module1.o`, `module2.o`, `module3.o` — имена объектных файлов, подлежащих компоновке, а `program` — имя исполняемого файла.

Подключение библиотеки `libm.so` осуществляется с помощью опции `-lm` компоновщика, например:

```
> gcc -lm module.o -o program
```

Подключение произвольной библиотеки `libxxx.so` осуществляется с помощью опции `-lxxx`.

В файле `libc.so` находятся тела функций, заданных в стандарте ANSI C, в частности, функций `printf`, `scanf`, `exit`, и других функций, объявленных в файлах `stdio.h`, `stdlib.h`, `string.h`. Эта библиотека подключается по умолчанию.

Наиболее простым способом получения исполняемого файла является следующий вызов:

```
> gcc -lm -o program module1.c module2.c module3.c
```

В этом случае, компиляция и компоновка выполняются одной командой. Компоновщик при этом понимает, что нужно предварительно вызвать компилятор для каждого входного файла с исходным кодом. Мы предлагаем на первых занятиях использовать именно этот способ сборки.

Семинар 5

Примеры простых программ

Краткое описание: На этом семинаре приобретаются навыки создания, сборки и выполнения простых программ. Для сборки программ мы будем использовать компилятор GNU C (программу `gcc`). Разберём примеры простых программ. Изучим как устроены программы на Си, как производить считывание и вывод чисел, как организовывать циклы и условные операторы. В конце рассмотрим простейшие директивы препроцессора.
Ключевые слова: сборка программ, компиляция, переменные, операторы, функции, директивы препроцессора.

Компиляция программ

Программа на Си это один или несколько текстовых файлов. Программы можно писать в любом текстовом редакторе, но существует несколько удобных сред разработки программ, которые позволяют оперативно запускать, отлаживать программы, автоматически создавать часть кода и специальных файлов.

Но пока мы будем рассматривать программы состоящие из одного файла. После того, как программа написана, нужно создать запуускаемый файл. Если ваша программа есть один файл `hello.c`, то для получения запускаемой программы нужно выполнить команду:

```
> gcc hello.c -o hello
```

В результате получится исполняемый файл `hello`, который можно запускать (execute).

Здравствуй, мир!

Первая программа, которую мы рассмотрим, — это «Hello, world» — программа 5.1, которая печатает на экран строчку «Hello world!» и заканчивает своё выполнение.

Первая строчка `#include<stdio.h>` означает «включить файл `stdio.h`». В этом файле находятся объявления функций, констант и переменных, связанных с вводом и выводом данных. Аббревиатура STDIO означает «STanDard Input/Output». Расширение `h` (буква «h» после точки) означает «header». Такое расширение имеют **заголовочные файлы**. В заголовочных файлах описывается, какие функции предоставляет соответствующая библиотека.

Программа 5.1: «Здравствуй, мир»

```
#include <stdio.h>
int main () {
```

```
    printf ("Hello, world!\n");
    return 0;
}
```

Далее идёт функция `main`. Её описание начинается со строчки

```
int main()
```

что следует интерпретировать следующим образом:

«Функция с именем `main` возвращает целое число
(число типа `int`) и не имеет аргументов.»

Далее открываются фигурные скобки, идёт описание **тела функции**, и фигурные скобки закрываются. Функция `main` — это главная функция программы, именно она начинает выполняться, когда программа запускается.

Между фигурных скобок находится тело функции. Тело функции представляет собой последовательность инструкций. Сначала исполняется первая инструкция —

```
printf("Hello, world!\n");
```

Это инструкция осуществляет вызов функции `printf` с аргументом `"Hello, world!\n"`. Функция `printf` описана в заголовочном файле `stdio.h`. Программа выведет строчку с текстом «`Hello, world!`». Обратите внимание на комбинацию «`\n`», она задает специальный **символ возврата каретки** (символ новой строчки), который в действительности не является печатным символом, а интерпретируется текстовым терминалом как действие — перейти на следующую строчку.

Затем идёт инструкция «`return 0;`». Эта инструкция завершает выполнение функции, возвращая в качестве результата число 0 (заканчивает работу с кодом возврата 0). Функция `main` должна возвращать 0, если выполнение программы прошло успешно.

Лишние пробелы и пустые строчки в программах на языке Си не играют роли.

Учимся складывать

Разнообразные вычисления — моделирование, решение алгебраических и дифференциальных уравнений — это то, для чего создавались первые вычислительные машины. Давайте научимся использовать компьютер для вычислений.

Начнём со сложения двух чисел.

В нашей программе будет две целочисленные переменные — a и b — две ячейки памяти, в которых могут хранится целые числа из определённого диапазона.

Локальные переменные, которые будут использоваться в функции, объявляются сразу же после открывающей фигурной скобки. Объявления начинаются со слова, указывающего тип переменных.

Программа 5.2: Сумма двух целых чисел

```
#include <stdio.h>
int main () {
    int a, b;
    printf ("Введите два числа: ");
```

```
    scanf  ("%d%d", &a, &b);
    printf ("%d\n", a + b);
    return 0;
}
```

Посмотрите на программу 5.2. Функция `scanf`, также как и `printf`, объявлена в файле `stdio.h`. Эта функция считывает данные, которые пользователь (тот, кто запустит вашу программу) вводит с клавиатуры. Слово «scan» означает «считывать данные», а «print» — «печатать данные». Буква «f» на конце взята от английского слова «formated», то есть `scanf` и `printf` суть функции *для форматированного ввода и вывода данных*.

Первый аргумент у функции `scanf` (то, что идет после открывающей круглой скобки до запятой) — это описание формата ожидаемых входных данных. В данном примере ожидается, что пользователь введёт два целых числа. Символ «`%`» служебный, с него начинается описание формата. Обычно, после него идет один или два символа, определяющих природу входных данных.

Формат «`%d`» соответсвует целому числу записанному в десятичной системе счисления и применяется к переменным типа `int`.

Приведённая программа умеет складывать только целые числа. Если нужно складывать действительные числа, то программу нужно несколько модифицировать. Ниже приведена программа, которая считывает два действительных числа и выводит результат четырех арифметических операций: сложения, вычитания, умножения и деления. Причём, программа выводит результаты вычислений два раза — сначала в обычном виде, а потом со специальным форматированием. Формат «`%15.10lf`» соответствует выводу числа типа `double`, при котором под запись числа выделяется ровно 15 позиций (если это возможно), а после запятой пишется ровно 10 цифр. Выравнивание числа осуществляется по правому краю.

Программа 5.3: Арифметические операции над действительными числами.

```
#include <stdio.h>
int main () {
    double a, b;
    printf ("Введите два числа: ");
    while(scanf ("%lf%lf", &a, &b) == 2 ) {
        printf ("%lf %lf %lf %lf\n",
            a + b, a - b, a * b, a / b );
        printf ("%15.10lf %15.10lf %15.10lf %15.10lf\n",
            a + b, a - b, a * b, a / b );
    }
    return 0;
}
```

Подробнее о форматах вывода и ввода прочитайте в документации функции `printf`.

Задача C5.1. Попробуйте скомпилировать и запустить эту программу. Введите два числа и посмотрите результат. Что произойдёт, если второе число равно 0?

Вычисление максимума

Одна базовых конструкций алгоритмических языков — это оператор цикла `for`. Конструкция

```
for(i = 0 ; i < n ; i++) {
    ....
}
```

используется для того, чтобы инструкции, записанные в фигурных скобках, выполнить n раз. Например, программа 5.4 выведет слово `Hello` 1000 раз. Переменная `i` называется переменной цикла. В данном примере она объявляется сразу после открывающейся скобки в теле функции `main` с помощью инструкции `int i;`. В цикле она принимает последовательно значения 0, 1, ..., 999. Затем она принимает значение 1000 и тело цикла уже не выполняется.

Программа 5.4: Пример использования оператора цикла `for`

```
#include<stdio.h>
int main() {
    int i;
    for(i = 0 ; i < 1000 ; i++) {
        printf("Hello");
    }
}
```

Рассмотрим программу 5.5. Она устроена следующим образом. Сразу после открывающей скобки идет объявление переменных, которые используются в программе — переменные `max` и `n` целого типа. Причём переменной `max` присваивается начальное значение 0. После объявления переменных идут инструкции. Сначала печатается приглашение «`Введите количество чисел:`». Затем считывается целое число и результат помещается в переменную `n` — инструкция `scanf("%d", &n)`. Потом пишется приглашение «`Введите %d чисел:`», где вместо символов «`%d`» будет подставлено значение переменной `n`. Затем начинается оператор цикла, в котором тело цикла выполняется `n` раз. А именно, `n` раз считывается очередное число `a` (`scanf("%d", &a)`) и это число сравнивается с текущим максимумом (оператор `if`). Если оно больше, чем текущий максимум, хранящийся в переменной `max`, то значение переменной `max` обновляется. Более подробно операторы `for` и `if` описаны в лекции.

Программа 5.5: Вычисление максимума n введённых чисел

```
#include <stdio.h>
int main () {
    int i, n, a, max;
    printf ( "Введите количество чисел: " );
    scanf  ( "%d", &n );
    printf ( "Введите %d чисел: ", n );
    scanf  ( "%d", &max );
    for(i = 1; i < n ; i++ ) {
        scanf ( "%d", &a );
        if( a > max ) max = a;
    }
    printf ( "%d\n", max );
```

```
    return 0;
}
```

Программа 5.6: Вычисление максимума n введённых чисел (2-й вариант)

```
#include <stdio.h>
int main () {
    int i, n, a, max = 0;
    printf ( "Введите количество чисел: " );
    scanf  ( "%d", &n );
    printf ( "Введите %d чисел: ", n );
    for( i = 0; i < n ; i++ ) {
        scanf ("%d", &a);
        if( a > max ) max = a;
    }
    printf ( "%d\n", max );
    return 0;
}
```

Задача С5.2. Укажите пример входных данных, при котором программа 5.6 находит неправильный максимум. Сравните логику работы программы 5.6 и программы 5.5. Найдите случай, когда эти программы выводят разный ответ.

Таблица умножения

Рассмотрим задачу вывода таблицы умножения. Эта задача разбивается на следующие шаги: «напечатать 1-ю строчку таблицы», «напечатать 2-ю строчку таблицы», ..., «напечатать n-ю строчку таблицы». Каждый из этих шагов, в свою очередь, разбивается на шаги печати отдельных элементов в строчке.

Эта задача решается в программе 5.7 с помощью конструкции **цикл в цикле**.

Внешний цикл начинается со строчки

```
for(i = 1; i <= n ; i++)
```

Переменная `i` соответствует номеру строчки. Она последовательно принимает значения 1, 2, ..., n. Внутри тела этого цикла находится ещё один цикл, который начинается со строчки

```
for(j = 1; j <= n ; j++)
```

Переменная `j` соответствует номеру столбца и также пробегает значения 1, 2, ..., n. Для каждого отдельного значения `i` запускается внутренний цикл по переменной `j`. Тело внутреннего цикла содержит строчку

```
printf ("%5d", i * j);
```

Эта инструкция интерпретируется как «напечатать число, получающееся в результате умножения `i * j`, в формате `%5d`». Формат `%5d` означает «напечатать целое число, выделив под него 5 мест, равнение делать по правому краю и незадействованные места заполнить пробелами».

Тело внешнего цикла кроме внутреннего цикла содержит также инструкцию «`printf("\n");`», которая переводит «печатающую каретку» на новую строчку.

Все используемые переменные имеют целый тип `int` и объявляются в строчке

```
int i, j, n;
```

В программах можно писать комментарии. Короткий комментарий начинают с двух слэшей «`//`» — всё что идёт после них до конца строчки пропускается компилятором. Длинный комментарий на несколько строчек начинают комбинацией «`/*`» и заканчивают комбинацией «`*/`».

Программа 5.7: Таблица умножения

```
/*  Программа table.c
    Считывает целое число n.
    Выводит таблицу умножения n x n.
*/
#include <stdio.h>
int main() {
    int i, j, n;
    printf ("Введите n: ");
    scanf ("%d", &n);
    for(i = 1; i <= n ; i++) { // i = номер строчки
        for(j = 1; j <= n ; j++) { // i = номер столбца
             printf ("%5d", i * j);
        }
        printf("\n"); // переход на новую строчку
    }
    return 0;
}
```

Простые числа

Программа 5.8: Программа, проверяющая число на простоту. В программе допущена ошибка (см. задачу С5.3).

```
#include <stdio.h>
int main() {
    int n, i;
    scanf("%d", &n);
    for(i = 2 ; i*i < n; i++)
        if(n % i == 0)
            break
    if(i*i < n && i != 2) {
        printf("not prime\n");
    } else {
        printf("prime\n");
    }
```

```
    return 0;
}
```

Задача С5.3. Изучите программу 5.8. Она должна считывать число и выводить «`prime`», если число простое, и «`not prime`» — если составное. Найдите ошибку в программе и исправьте её.

Задача С5.4. Напишите программу, которая выводит все простые числа меньшие 10000.

Задача С5.5. Напишите программу, которая вычисляет функцию $\alpha(n)$ — число простых чисел меньших n. Реализуйте для этого алгоритм Эратосфена (см. псевдокод 1.8 на стр. 38). Проверьте предположение, что эта функция стремится к $n/(\log n + C)$, где C — некоторая константа. Как примерно растёт время работы программы с ростом n?

Подключение математической библиотеки

Существует стандартная библиотека математических функций. Функции, которые она предоставляет объявлены в файле `math.h`, поэтому, если вы хотите пользоваться такими функциями как `sin`, `cos`, `log`, `exp` (экспонента), `asin` (арксинус), `acos` (арккосинус), `sqrt` (квадратный корень), необходимо вначале программы вставлять строчку

```
#include <math.h>
```

Задача С5.6. Создайте файл `sin.c` с программой 5.9. Скомпилируйте его с помощью команды

```
 > gcc sin.c -lm - o aaa
```

Эта команда означает «скомпилировать файл `sin.c` в программу `aaa`, при компоновке подключив библиотеку `libm`». Запустите файл `aaa` и введите действительное число.

Программа 5.9: Вычисление синуса

```
#include <stdio.h>
#include <math.h>
int main() {
    double x;
    scanf("%lf", &x);
    printf("sin(%lf)=%8.8lf\n", x, sin(x));
    return 0;
}
```

Библиотека `libm` (подключаемая с помощью опции `-lm`) содержит откомпилированные математические функции, которые объявляются в заголовочном файле `math.h`. Если вы используйте функции из этой библиотеки (такие как `log`, `sin`, `cos`, `exp`, `sqrt` и др.), то не забывайте включать её заголовочный файл с помощью директивы `#include<math.h>`.

Подробную информацию об опциях компилятора `gcc` можно получить с помощью следующих команд:

```
 > man gcc
 > info gcc
```

*Препроцессинг

Перед компиляцией программа подвергается *препроцессингу*. На этапе препроцессинга происходит исполнение команд препроцессора, которые также называют *директивами препроцессора*.

Директивы препроцессора начинаются с новой строки и символа решетки `#`. Наиболее часто используемые — это директивы `#include`, `#define`, а также набор `#ifdef`, `#else` `#endif`.

Рассмотрим их подробнее.

Директива `#include`

Директива `#include имя_файла` имеет простой смысл — строка с этой командой заменяется на содержимое файла, имя которого указано в угловых скобках или двойных кавычек. Двойные кавычки используют для файлов, которые относятся к данному проекту, а угловые — для стандартных файлов (в основном, для заголовочных файлов стандартных библиотек). Обычно, заголовочные файлы содержат только **прототипы функций**, предоставляемых соответствующей библиотекой, то есть список объявлений функций, где каждое объявление содержит имя функции с указанием аргументов и типа возвращаемого значения.

Директива `#define`

Директива `#define` используется для задания макроопределений. Она имеет два варианта синтаксиса:

```
#define идентификатор  любой набор символов
#define идентификатор(список аргументов) любой набор символов
```

Рассмотрим первый вариант. Директива

```
#define N 100
```

приведёт к тому, что везде ниже идентификатор `N` будет заменён на `100`. Рассмотрим код

```
int N=1;
#define N 100
int showN(int N) {
    printf("N=%d", N);
    return N;
}
```

После обработки препроцессором он будет выглядеть следующим образом:

```
int N = 1;
int showN(int 100) {
    printf("N=%d", 100);
    return 100;
}
```

Этот результат пойдёт на вход компилятору, который сообщит об ошибке компиляции, так как аргументы у функции `showN` заданы не корректно.

С помощью директивы `#define` идентификаторам можно назначать любую последовательность символов, в том числе не являющуюся корректным куском кода. Рассмотрим код:

```
#define N 100
int ar[N];
#define ARGS    name, a, b
#define ARGSF  "%s %d %d"
#define CHECK   if (a < 0){printf("Некорректное значение a");}
#define ASSERT(e) if (!(e)){printf("Ошибка в строке %d",__LINE__);}
#define max(a,b)  ((a>b)?a:b)
#define foreach(i,n)   for(i = 0; i < n ; i++ )
CHECK;
ASSERT( max(a+b,a*b) < 1000 );
printf(ARGSF, ARGS);
foreach(a, N) {
    printf("%d\n", ar[a]);
}
```

Он будет преобразован препроцессором в следующий код:

```
int ar[100];
if (a < 0){print("Некорректное значение a");};
if (!(((a+b>a*b)?a+b:a*b) < 1000)) {
             printf("Ошибка на строчке %d\n", 9);}
printf("%s %d %d", name, a, b);
for(a = 0; a < 100 ; a++ ) {
    printf("%d\n", ar[a]);
}
```

В данном примере также была использована константа `__LINE__`, равная номеру строчки в файле. За подстановку значения этой константы также отвечает препроцессор.

В данном примере `max` является не функцией, а макроопределением.

Различие между функцией и макроопределение заключается в том, что макроопределения *раскрываются* (заменяются на их значения) перед компиляцией, а функции компилирутся и вызываются уже во время работы программы.

Выражение `max( a+b, a*b )` препроцессор заменит на `((a+b>a*b)?a+b:a*b)`, и текст, в котором проделаны уже все замены, передаст на вход компилятору.

При этом большее из выражений `a+b` и `a*b` будет вычисляться дважды. Это может привести к неприятным последствиям.

У макроопределений есть свои плюсы — макроопределения можно использовать для чисел любого типа, а в функциях нужно чётко указывать типы аргументов и возвращаемого значения.

Задача C5.7. В коде 5.10 представлена программа с макроопределением `max` и функцией `maxi`. Скомпилируйте программу, выполните и объясните результат.

Программа 5.10: Код к задаче C5.7

```
#include <stdio.h>

#define max(a,b) ( (a > b)? a : b)
```

```
int maxi(int a, int b) {
   if(a > b)
       return a;
   else
       return b;
}
int main() {
    int a, b;
    a = 5; b = 6;
    printf("%d\n", max(a, b));
    printf("%d\n", max(a, b));

    printf("%d\n", max (--a, --b));
    a = 5; b = 6;
    printf("%d\n", maxi(--a, --b));

    printf("%lf\n",  max(1.5, 2.7));
    printf("%lf\n", 1.0L * maxi(1.5, 2.7));

    // Здесь запись 1.0L обозначает действительное число 1
    // типа double. При умножении числа типа double и числа
    // типа int результат получается типа double.

    return 0;
}
```

Директивы #ifdef и #ifndef

Директива #ifdef используется совместно с директивами #else и #endif. Она позволяет включать (исключать) куски кода в зависимости от заданных макроопределений. Это, например, используется для включения (исключения) отладочного кода.

Пусть в некоторой программе число a должно быть положительным. Тогда можно в различных частях программы писать проверочный код

```
#ifdef DEBUG
   if (a <= 0) {
       printf("ОШИБКА: a <= 0, a=%d.\n", a);
       exit(2);
   }
#endif
```

Этот код будет включён в код программы после препроцессинга только в том случае, если в программе (где-нибудь в начале) будет стоять директива #define DEBUG. Значение макроопределения DEBUG может быть просто пустым.

Отладочный код может проверять корректность хранимых данных и работать работать достаточно долго. Кроме того, в отладочный код можно поместить вывод значений ключевых

переменных, чтобы следить за ходом их изменения по мере выполнения программы. Естественно включать его только в отладочных версиях программы. Для получения конечной рабочей программы необходимо закомментировать строчку `#define DEBUG` и собрать программу заново.

Ещё один приём заключается в задании макроопределения `CHECK` которое равно отладочному коду только в случае, когда задано макроопределение `DEBUG`:

```
#ifdef DEBUG
#define CHECK  if (a <= 0) { \
       printf("ОШИБКА! Число a <= 0. a=%d\n", a);\
       exit(2);\
   }
#else
#define CHECK
#endif
```

Задание макроопределения должно быть одной строчкой. Если это неудобно, то следует перенос строчки *экранировать* символом \ (бэкслэш), то есть просто ставить этот символ в конце каждой строчки пока не закончится макроопределение.

Задавать макроопределения можно прямо в строчке сборки программы. Например, команда

```
> gcc -DDEBUG -DCHECKALL a.c -o a.exe
```

осуществит сборку программы `a.c` задав макроопределения `DEBUG` и `CHECKALL`.

Другой пример использования директивы `#ifdef` связан с разработкой *мультиплатформенных* программ, то есть программ, которые компилируются и работают в разных операционных системах на разных компьютерных архитектурах.

Дело в том, что функции для работы с файлами, временем, процессами, и другие функции, связанные с *системными вызовами*, различаются в разных операционных системах. Не для всех случаев есть универсальный код, работающий во всех операционных системах. Удобно поддерживать не несколько, а одну версию программного кода, но при этом использовать директивы `#ifdef` для адаптации кода под операционную систему и особенности программного обеспечения и архитектуры.

Например, при компиляции в операционной системе Windows задано макроопределение `_WIN32_`. Это позволяет писать следующий код:

```
#ifdef _WIN32_
  // код, который работает в операционной системе Windows
#else
  // код, который работает в других операционных системах
#endif
```

Есть также похожая директива `#\ifndef XXX`, которая включает следующий за ней код в программу, если макроопределение XXX не задано.

На практике активно используется метод окружения внутренностей заголовочных файлов конструкцией `#ifndef XXX`,`#define XXX` ...`#endif`.

```
#ifndef XXX
#define XXX
```

```
 /* Код, который не может быть включён более одного раза */
 int n;
#endif /* XXX */
```

Если указанный кусок кода присутствует в файле xxx.h, то две директивы

```
#include "xxx.h"
#include "xxx.h"
```

не приведут к ошибке компиляции. При выполнении второй директивы макроопределение `XXX` уже будет определено и кусок кода между директивами `#ifndef` и `#endif` не будет включён в в файлы проекта более одного раза. Опасность повторения директивы `#include` существует, так как заголовочные файлы сами в свою очередь могут включать другие заголовочные файлы и все зависимости по включению иногда сложно проследить.

В заключении скажем, что препроцессор являются дополнительным средством, *отдельным от языка Си*. Часто директивы препроцессора можно не использовать и ограничится лишь средствами языка Си. Но ясно также, что без директивы `#include` задача разработки крупных проектов сильно бы усложнилась.

Лекция 6

Выражения языка Си. Массивы. Потоки

Краткое описание: На этой лекции мы расскажем про основу всех вычислений — выражения, рассмотрим их типы и способы построения. Уделим особое внимание понятию L-выражения. Также в этой лекции мы осветим вопросы организации взаимодействия программы с внешней средой. Познакомимся с потоками и файлами и научимся с ними работать.

Арифметические и логические выражения

До этого момента мы активно пользовались **выражениями**, но не давали их точного определения или способа построения. Арифметические выражения для многих языков программирования достались в наследство от математики и способ записи выражений очень похож на способ записи математических формул. Переменные, вызовы функций и константы сами по себе являются выражениями. Все остальные выражения могут быть построены по следующим правилам:

- константа скалярного типа является выражением.
- Имя переменной скалярного типа является выражением.
- Вызов функции, возвращающей скалярный тип является выражением.
- Выражение, заключенное в круглые скобки является выражением.
- Два выражения, разделенные бинарным оператором, являются выражением.
- Выражение, перед которым стоит унарный оператор, является выражением.
- Выражение, перед которым стоит префиксный оператор, является выражением.
- Выражение, после которого стоит постфиксный оператор, является выражением.
- Три выражения, разделенные двумя частями тернарного оператора являются выражением.

Стоит заметить, что самыми элементарными частями, из которых конструируется выражение должны быть строго скалярного типа, будь то числовая константа, переменная, вызов функции или более сложные конструкции (которые также являются выражениями, но не являются арифметическими), о которых мы расскажем позже.

Бинарные операторы включают в себя большинство арифметических и логических операторов:

```
- + / * % && || < > == >= <= != & | ^ << >> ,
```

Это операторы, которые применяются к двум операндам, называемым левым и правым операндом.

Первые четыре — это операторы вычитания, сложения деления и умножения. Оператор `%` применяется к двум целочисленным выражениям, и результат равен остатку от деления первого на второе. Если первый операнд отрицателен, то и результат тоже отрицательный. Вот примеры верных тождеств.

```
 17 %  5  ==  2
-17 %  5  == -2
-17 % -5  == -2
 17 % -5  ==  2
```

Бинарный оператор `==` является логическим оператором и означает «тождественно равно», и его результатом является «истина» или «ложь». Операторы `< > >= <= !=` соответствуют логическим «меньше», «больше», «больше либо равно», «меньше либо равно», «не равно». Операторы `&& ||` тоже являются логическими операторами и обозначают логические «И» («AND») и «ИЛИ» («AND»). Для логических операторов можно считать, что «истина» есть значение типа `int`, равное 1, а «ложь» — значение типа `int`, равное 0.

```
(7 > 3) && (4 <= 4)    == 1
(7 < 3) || (4 >= 3)    == 1
(7 < 3) && (4 >= 3)    == 0
(7 < 3) || (4 <= 3)    == 0
9 && 4     ==  1 -- здесь два ненулевых операнда
               интерпретируются  как 'истина' и результат
               тоже равен 'истина', то есть 1.
```

Кроме логических «И» и «ИЛИ» есть также *побитовые логические операторы* `&`, `|` и `^`, применимые ко всем целочисленным типам (коротким и длинным, знаковым и беззнаковым). Можно представлять себе это так — берутся компьютерные представления чисел в виде набора бит фиксированной длины (необходимо, чтобы оба операнда (их типы) имели один размер, а если нет — то один из операндов приводится к типу большего размера). Представление первого операнда подписывается под представлением второго, и в каждом столбце осуществляется логическая операция. Например,

x	0	1	1	0	1	1	0	1
y	1	1	0	1	1	1	1	1
x & y	0	1	0	0	1	1	0	1
x | y	1	1	1	1	1	1	1	1
x ^ y	1	0	1	1	0	0	1	0

Каждый бит результата зависит только от двух стоящих над ним бит. В случае операции `&` «И» в бите результата стоит 1 только тогда, когда два стоящих над ним бита равны 1. Для побитовой операции `|` «ИЛИ» наоборот — бит результата равен 0 тогда и только тогда, когда соответствующие биты операндов равны 0. Биты результата операции `^` равны 1 в тех местах, где биты операндов различны.

В случае положительных целых чисел смысл побитовых операций `&`, `|` и `^` можно пояснить так. Представим операнды `X` и `Y` как сумму неповторяющихся степеней двойки (такое представление единственно). Тогда `X & Y` есть число, составленное из слагаемых, которые присутствуют и в разложении `X`, и в разложении `Y`. Выражение `X | Y` есть число, составленное из

слагаемых, каждое из которых присутствует хотя бы в одном из разложений. И наконец, выражение `X ^ Y` есть число, составленное из слагаемых, которые присутствуют ровно в одном разложении — либо в разложении `X`, либо в разложении `Y`:

```
9 & 4          ==   0,  т.к. разложение 9=8+1 не содержит 4.
9 | 4          ==   13, т.к. 13==8+4+1.
13 & 25        ==   9,  т.к. 13==8+4+1, 25==16+8+1, 9==8+1.
13 | 25        ==   29, т.к. 29==16+8+4+1.
13 ^ 25        ==   20, т.к. 20==16+4.
```

С помощью бинарного оператора «`,`» (запятая) можно несколько выражений объединять в одно. При этом эти выражения будут вычисляться последовательно слева направо, а результат выражения будет равен значению последнего выражения. Оператор «запятая» можно использовать, например, так:

```
for( j = 1, i = 0; i <= 10; i++ )printf("%d\n",j), j=j*2;
// или
for( j = 1, i = 0; i <= 10; i++,printf("%d\n",j), j=j*2 );
// или
for( j = 1, i = 10;  printf("%d\n", j), j = j*2, i-- ; );
```

Но так делать не рекомендуется, так как это затрудняет понимание логики кода.

Рассмотрим **унарные операторы**:

```
+ - ! ~
```

Унарный плюс и унарный минус имеют очевидный смысл. Унарный оператор `!` применяется к целочисленным и логическим выражениям и означает *отрицание*. Пусть `X` есть выражение с целым значением. Тогда выражение `!X`, равно 0, если `X` не равно 0, и равно 1, если `X` равно 0.

Унарные операторы «`*`» и «`&`» не относятся к арифметике или логическим операциям. Это операции взятия адреса и разыменования указателя. Об *указателях* мы подробно поговорим на следующей лекции[1]. Если говорить коротко, то указатель — это адрес места, где хранятся данные. Можно считать, что это порядковый номер байта в адресном пространстве. Пусть объявлена переменная `x`. Тогда выражение `x` — это значение переменной `x`, а выражение `&x` — адрес в памяти, где хранится `x` (адрес первого байта переменной). Унарный оператор `*` обратен оператору взятия адреса. Выражение `*(a)` есть значение, которое находится по адресу `a`. В частности, `*(&x)` можно отождествить с самой переменной `x`. Операция взятия адреса не может быть применена к произвольному выражению. Например, `&(3+x)` некорректное выражение, так как выражение в скобках не хранится в памяти и не имеет адреса. Это временное числовое значение, появляющееся по ходу вычисления, которое формально нигде не хранить-ся. Операция взятия адреса применяется только к переменным и к выражениям, которые могут играть роль переменной (см. ниже про L-выражения).

Префиксные и постфиксные операторы `++` и `--` – специфические для языка Си конструкции. Постфиксная запись `i++` означает «увеличить значение переменной `i` на единицу». Тоже

[1]При конструировании языков программирования ощущается явная нехватка разнообразных символов. Приходится использовать одни и те же символы для разных целей. Это не только затрудняет понимание программы человеком (необходимо думать о том, какую роль в данном контексте имеет та или иная запись), но и усложняет алгоритмы компиляции программ.

самое означает и префиксная форма оператора: `++i`. Различие между префиксной и постфиксной формами заключается в том, чему будет равно результирующее значение выражения. При постфиксной записи, результатом является прежнее (то, что было до выполнения операции) значение операнда, а при префиксной — новое значение. Например, после выполнения данной строчки

```
i = 1; a = i++;
```

(переменные `i` и `a` объявлены как `int`) переменная `a` примет значение `1`, а при выполнении строчки

```
i = 1; a = ++i;
```

переменная `a` окажется равной 2.

Следующий важный случай выражения — **присваивание**. В принципе, присваивание является бинарным оператором, обозначаемым символом `'='`. Обычно используется запись вида

```
<имя переменной> = <выражение>;
```

При вычислении выражения типа «присваивание» происходит вычисление выражение справа от знака `=`, и результат заносится в переменную, имя которой указано слева. Как и любое выражение, оно имеет результат, а именно, присвоенное значение. Поэтому нередко в программах на Си можно встретить запись

```
a = b = c = 0;
```

Она означает: присвоить переменным `a`, `b` и `c` значение 0.

Но левый операнд может быть не только именем переменной. Левый операнд должен быть **L-выражением**, частным случаем которого является имя переменной:

```
<L-выражение> = <выражение>;
```

L-выражение — это выражение, результат которого имеет определённый адрес в памяти.

Выражения «`1`», «`x+y`», «`x-1`» не являются L-выражениями.

Кроме имён переменных, есть и другие случаи L-выражений. Они будут описаны на лекции 7. Забегая вперёд, скажем, что L-выражениями являются элемент массива (например, `a[0]`, `a[i+j]`) и разименование указателя.

Постфиксные и префиксные операторы `++` и `--` также применимы только к L-выражениям. Это естественно, так как в них неявно присутствует присваивание. Например, если `x` есть переменная типа `int`, то

«`(x++)`» равносильно «`(x=x+1, x-1)`».
«`(++x)`» равносильно «`(x=x+1, x)`».

Тернарный оператор — это оператор, состоящий из трёх частей, разделённых символами '?' и ':', который используется как «быстрый `if`», например, `a < b ? a : b` — выражение, которое вычисляет минимум из значений двух выражений (`a` и `b`).

Приведение типов

Язык Си — это строго типизированный[2] язык. Строгая типизация означает, что проверка типов переменных перед использованием в выражениях и вызовах функций осуществляется на этапе *компиляции*. Такое поведение компилятора позволяет обнаруживать большое число ошибок на стадии разработки программы и, тем самым, дать некоторую гарантию правильности выполнения программы — компиляция программы без ошибок уже многое будет означать, но не избавляет от вероятности появления ошибок на этапе *исполнения*. Компилятор следит за тем, чтобы функциям передавались аргументы нужных типов, а тип возвращаемых значений соответствовал бы типу переменных, которым эти значения присваиваются.

Но в практике программирования часто встречается ситуация, когда данные одного типа требуется *преобразовать* в данные другого типа. Это преобразование называется **приведением типов** или **кастингом** (cast). Различают **явные** (явно указанные программистом) и **неявные** (подразумеваемые) приведения типов.

Неявные приведения типа

В некоторых строго типизированных языках запрещается неявное приведение даже примитивных типов. Но язык Си реализует несколько иную политику. В Си примитивные типы могут быть преобразованы неявным образом. Чаще всего, неявное преобразование производится в арифметических выражениях и в присваивании.

Рассмотрим фрагмент программы на языке Си:

```
int a = 1;
double x = 2 * a;
```

Здесь выражение `2*a` имеет целый тип, так как является произведением двух выражений целого типа — целой константы `2` и целой переменной `x`. Но поскольку слева от знака равно стоит переменная типа `double` результат умножения приводится к типу `double`.

Аналогично, в выражении `double z = 10.0; int x = z/3;` операция деления `z/3` производится с вещественными числами двойной точности (тип `double`), а затем результат преобразуется в целое число и присваивается переменной `x`. Если компилятору в арифметическом выражении приходится оперировать числами различных типов, то результат выражения приводится к типу с максимальной точностью.

Во время присваивания, точность значения выражения, стоящего справа от знака равенства, может быть выше точности выражения, стоящего слева, при этом компилятор старается выбрать наиболее подходящий способ приведения типа числа. Например, если переменной типа `int` присвоить значение типа `double`, то компилятор произведёт операцию выделения целой части числа (если она не превышает максимального значения, допустимого в `int`, в противном случае, результатом будет 0) и отбросит дробную. А в случае, если переменной типа `char` присваивается значение типа `long`, компилятор возьмёт в качестве значения только младший

[2]Многие могут не согласиться с особой строгостью типизации в Си и мотивируют это тем, что в нём можно достаточно своевольно распоряжаться информацией о типах с помощью такого инструмента как приведение типов.

байт результата.[3]

Явное приведение типа

Процессом приведения типов можно управлять. Для этого в языке Си предусмотрена следующая конструкция, называемая **явным приведением типов**. Например, чтобы подсказать компилятору, что какое-либо выражение должно быть приведено к типу `long`, достаточно поставить имя типа в скобках перед приводимым выражением: `(long)( 10 * 5000 )`.

Чему равны выражения `(double)(2/3)` и `(2/(double)3)`? Объясните причину различия результатов.

Ответ: первое выражение равно `0.0`, а второе — `0.666...6667`.

Если оба операнда имеют целый тип, то и результат бинарного оператора (в данном случае деления) тоже будет целым. Поэтому в первом выражении в скобках получается число `0`, которое затем приводится к типу `double`.

Отдельного рассмотрения требует случай преобразования знаковых и беззнаковых целых. При преобразовании из знаковых в беззнаковые или наоборот и при совпадении размеров типов, модификации представления чисел не происходит. Это значит, что преобразуемое и преобразованное значения имеют одинаковые представления в памяти — после преобразования значения хранимых бит переменной остаются прежними. Например, результатом преобразования `(unsigned char)(-1)` будет число `255`, а результатом `(char)(128)` — число `-128`.

Статические массивы

Переменные могут хранит один объект фиксированного типа. Есть возможность создавать хранилища для большого числа элементов одного типа и обращаться к ним по номеру. Для этого используются массивы.

Массивы являются особым видом переменных, которые используются для представления упорядоченного множества элементов одного типа, например, массив 10 целых чисел, или массив 256 символов.

Статические массивы используются в случае, когда при написании программы заранее известно максимально возможное число элементов. У массива, также как у переменных, есть имя. Для объявления массива 100 элементов типа `int` используется запись

```
int arr[100];
```

Данный массив имеет имя `arr`. Он содержит элементы `arr[0]`, `arr[1]`, `arr[2]`, ..., `arr[99]`. Общий размер памяти, занимаемой массивом `arr`, равен $100 \cdot$ `sizeof(int)`.

[3] Данное утверждение справедливо только для платформ с т.н. Little-Endian представлением целых чисел (старший разряд – последний). Другой способ представления целых чисел называется Big-Endian и используется в большинстве RISC-процессоров. Процессоры архитектуры x86 используют первый тип представления.

Обращение к элементу массива

Чтобы обратиться к определённому элементу массива, достаточно написать имя массива и в квадратных скобках — номер элемента массива, называемого также **индексом**. В языке Си принято отсчитывать индексы с нуля. Вот пример обращения к третьему элементу массива `arr`.

```
arr[2]
```

Результатом данного выражения является значение 3-го элемента массива `arr`. Выражение имеет тип `int` и является L-выражением, то есть может быть использовано в качестве левого операнда оператора присваивания. Например:

```
arr[2] = 1 + arr[1];
```

То есть выражение обращения к элементу массива ничем не отличается от простой (скалярной) переменной. Приведём ещё один пример работы с массивами, в котором покажем, что в качестве индекса элемента может выступать не только константа, но и любое выражение, имеющее целый тип.

```
int i;
for( i = 0; i < 31; i++ ) {
    arr[ 17*i % 32 ] = i;
}
```

При объявлении массивов их можно сразу же инициализировать. Например,

```
int b[10] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
```

Забежим несколько вперёд. Переменные и массивы, используемые в программе, располагаются в адресном пространстве программы. Адресное пространство программы — это конечная последовательность ячеек памяти типа байт. Они пронумерованы от 0 до $2^{32} - 1$ в компьютерах с 32-разрядной архитектурой.

Адрес переменной можно получить с помощью унарного оператора `&`. Так например выражение `&x` есть адрес переменной `x`.

В случае с массивами, само имя массива соответствует *адресу* начала области памяти, в которой расположены элементы массива, то есть номеру первого байта первого элемента массива. Элементы массива расположены подряд друг за другом, без пропусков.

Строки

В языке Си нет специально выделенного типа, обозначающего строку, для этого используется массив элементов типа `char`. Например:

```
char str[100] = "Hello world!";
```

В этой строке объявлен массив из 100 элементов типа `char`. Но этот массив можно интерпретировать как строка, которая может содержать не более, чем 99 символов. Длина строки ограничена 99, так как последний символ (байт) должен быть равен 0.

Строка `"Hello world!"` разместится по 13 элементам массива str:

0	1	2	3	4	5	6	7	8	9	10	11	12	...
'H'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	0	...

Нулевой байт, обозначаемый как `'\0'`, служит признаком конца строки. Компилятор автоматически добавил его в конец при инициализации строки. Безусловно, строки можно инициализировать и как обычные массивы:

```
char str[100]={'H','e','l','l','o',' ','w','o','r','l','d','!',
                0, 0,  /* пропущен значительный кусок */ ,0};
```

Но это не всегда удобно.

В языке Си нет специальных операторов для работы со строками. В частности, нет специальных операторов сравнения, копирования, слияния строк. Сравнение двух строк при помощи обычных арифметических операторов сравнения не даст нужного результата. Так, например, следующее сравнение *неверно*, хотя компилятор не выдаст сообщения о ошибке:

```
char str[100] = "Иванов";
if ( str > "Сидоров" ) printf("ok");
```

В данном случае в операторе `if` будут сравниваться не строки, а их адреса, которые раз от раза могут различаться, и результат сравнения не будет иметь ничего общего с лексикографическим (алфавитным) порядком строк.

Строковая константа "Сидоров"размещена компилятором в памяти, а значение этой строковой константе будет равно адресу первого байта, в котором разместилась первая буква `С`!. Результат сравнения вернёт истину, если этот адрес имеет большее значение, нежели адрес первого элемента массива символов str.

Для работы со строками разработана стандартная библиотека `string.h`, которая присутствует практически во всех пакетах компиляторов языка Си. Для лексикографического сравнения строк необходимо использовать функцию `strcmp` из библиотеки `string.h`, которая сравнивает строки посимвольно:

```
#include <string.h>
char str[100] = "Иванов";
if ( strcmp(str, "Сидоров") > 0 ) printf("ok");
```

Функция `strcmp` возвращает 1, если первая строка следует за второй, -1 — если вторая следует за первой, и 0 — если они совпадают.

Оператор присваивания также не работает (или работает не так как, возможно, хотелось бы) для строк. Следующий код просто не будет компилироваться:

```
char name1[100] = "Иванов";
char name2[100];
name2 = name1;
```

Для копирования строк следует организовывать цикл или использовать функцию `strcpy` из библиотеки `string.h`.

Подробнее о функциях, предоставляемых библиотекой `string.h` можно узнать из документации, доступ к которой можно получить, например, по команде «`man string`».

Заметим, что строки можно объявлять как переменные типа `char*`:

```
char str[] = "Иванов";
```

В этом случае переменная `str` хранит *адрес*. При инициализации ей присваивается значение *адреса* области памяти, в которой компилятор расположил строку `"Иванов"` и будет, аналогично массиву символов, соответствовать адресу первого символа данной строки. Размер этого массива равен длине строки плюс 1. Этот метод удобен тем, что при объявлении строковых констант не нужно самому считать их длину.

Компилятор не запретит писать выражения `str[20]` или `str[100]`, но их использование может привести к ошибке исполнения (`Runtime error`), а именно, `Segmentation fault`.

При обращении к элементу массива индекс может быть равен произвольному целому числу. Компилятор не проверяет выход значения индекса за допустимые границы. Но в момент выполнения программы может возникнуть ошибка «`Segmentation fault`», которая означает, что в программе произошло обращение к участку адресного пространства, недоступному программе.

Если объявлен массив `int a[100]`, то элемент `a[-1]` равен целому числу, которое получится если проинтерпретировать четыре байта, находящиеся в адресном пространстве перед первым элементом массива, как значение типа `int`. Обращение к элементу `a[-1]` может и не привести к ошибке исполнения, поскольку это могут быть доступные байты какой-то другой переменной или массива.

Статические многомерные массивы

Специального типа *многомерный массив* в языке Си не существует, однако, возможно объявление массива массивов или массива массива массивов и т.д. Например, строка

```
double array_2d[20][100];
```

задаёт для переменной `array_2d` область памяти, в которой помещается 20 массивов по 100 элементов типа `double`.

Обращение к элементам такого «двумерного» массива происходит аналогичным образом:

```
double x = array_2d[20][100];
array_2d[19][99] = 1.3e-10;
```

Несложно догадаться как объявляются 3-х и более -мерные массивы:

```
double a3[20][20][20];
int a4[20][20][20][20];
```

Терминал. Потоки ввода и вывода.

Один из способов взаимодействия с пользователем — это стандартные потоки ввода и вывода данных, которые в языке Си обозначаются как `stdin` и `stdout`. Слово *стандартный* означает, что эти потоки доступны любой программе. В простейшем случае поток ввода — это клавиатура, а поток вывода — это экран. Именно с этими потоками работают функции `scanf` и `printf`. Они осуществляют форматированный ввод/вывод данных.

Стандартные потоки ввода-вывода.

Программа, которая позволяет печатать на экран данные, выводимые программой в стандартный поток ввода, и считывать данные, вводимые с клавиатуры, называется терминалом[4](или консолью).

Есть простейшие функции для посимвольного ввода и вывода: `getc` и `putc`, которые считывают из потока и записывают в поток ровно один символ. Поток может иметь конец. Для обозначения конца потока используется специальный символ `EOF`.

Рассмотрим простую программу.

```
// Программа echo.c
#include <stdio.h>
int main () {
    int c;
    while ( (c = getc(stdin)) != EOF ) {
        putc(c, stdout);
    }
    return 0;
}
```

Она считывает символы из стандартного потока ввода и печатает из в стандартный поток вывода. Если вы запустите эту программу и введёте набор символов, то программа просто повторит их. Обратите внимание, что программа работает с входными данными *построчно*. А именно, если вы запустите программу, введёте строку `Hello world` и нажмёте клавишу <Enter>, то увидите как программа повторит её:

```
> ./echo
Hello world
Hello world
```

Но, исходя из логики программы, более ожидаемым результатом был бы следующий:

```
> ./echo
HHeellllloo  wwoorrlldd
```

Действительно, явного ожидания перевода строки в коде не присутствует. Это связано с работой самого терминала. Терминал оптимизирует работу с потоками ввода и вывода и копит символы в специальном буфере, осуществляя ввод и вывод только по достижению конца строки (символа `'\n'`) или по мере заполнения буфера. Данная программа будет повторять строки, которые вы вводите, пока на вход не поступит сигнал конца потока `EOF`. Его можно ввести с клавиатуры с помощью комбинации <Ctrl+D> в Unix-системах и <Ctrl+Z> в Windows. Важно понимать, что *потоки ввода и вывода независимы*, несмотря на то, что набираемые вами символы и символы, выводимые программой, отображаются рядом в одном окне.

[4]Вначале компьютеры были простыми и представляли собой терминалы, то есть они предоставляли возможность взаимодействия с пользователем через два потока, которым соответствовали клавиатура и экран. Сегодня же терминалами называют программы, эмулирующие работу в таком режиме.

Потоки и Файлы

Поток — это некоторая последовательность символов, из которой нам доступен только один конец. Его можно сравнить с концом длинной трубы, другой конец которой нам не виден. С потоком можно производить следующие действия: открыть (у трубы есть крышка), положить символ, взять символ, протолкнуть и закрыть. Каждый раз, когда мы кладём символ в поток, все остальные символы продвигаются к другому концу потока, но, по достижении ими конца не вываливаются из него.

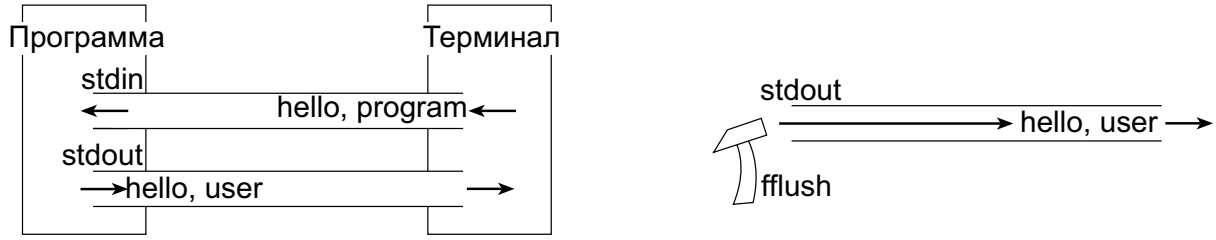


Рис. 6.1: Поток — это один конец трубы, используемый либо для ввода, либо для вывода данных, либо для того и другого одновременно. Потоки `stdin` и `stdout` являются стандартными потоками ввода и вывода программы. Команда `fflush(stdout)` вызывает «проталкивание» введённых данных до конца «трубы». Обычно потоки настроены так, что при появлении в входе символа перевода строки автоматически происходит «проталкивание»(auto flush).

При этом «труба» может «забиться», и придётся немного подождать, пока «труба» не освободиться для ввода следующих символов. И наоборот, забирая символ из потока, остальные символы продвигаются к концу.

Для обозначения потока в стандартной библиотеке ввода/вывода языка Си имеется специальный тип `FILE*`. Мы уже использовали стандартные потоки ввода/вывода – `stdin` и `stdout`. Это глобальные переменные типа `FILE*`. Они всегда открываются автоматически для любой запускаемой программы и их нет особой нужды закрывать. Гораздо интереснее дела обстоят с обычными файлами, которые также представляются в виде потоков.

Можно открыть поток ввода или вывода в файл, так, как если бы это был терминал или клавиатура. Для этого используется функция `FILE *fopen(char *name, char *mode)`. В аргументе `name` этой функции указывается имя файла, который нужно открыть, а в аргументе `mode` указывается способ взаимодействия с файлом. `fopen` возвращает значение открытого файла или значение `NULL`, в случае, если при открытии произошла ошибка, например, файла не существует или у программы не достаточно прав для его открытия.

Способ взаимодействия с файлом, передаваемый в аргументе `mode` – это строка (`char *`), которая может начинаться с одной из следующих последовательностей символов:

- `"r"` — Открыть текстовый файл для чтения. Чтение начинается с начала файла.
- `"r+"` — Открыть для чтения и записи. Чтение или запись начинаются с начала файла.
- `"w"` — Создать текстовый файл и открыть его для записи. Запись начинается с начала файла. Если файл с таким именем уже существует, он урезается до нулевой длины (его содержимое теряется).
- `"w+"` — Открыть для чтения и записи. Файл создается, если до этого его не существовало, в противном случае он урезается. Чтение или запись начинаются с начала файла.

- `"a"` — Открыть для дописывания (записи в конец файла). Файл создается, если до этого его не существовало. Запись осуществляется в конец файла.

- `"a+"` — Открыть для чтения и дописывания (записи в конец файла). Файл создается, если до этого его не существовало. Чтение или запись производятся с конца файла.

Строка `mode` может также включать в себя символ `"b"` в качестве последнего символа или символа, входящего в любые описанные выше двухсимвольные комбинации[5].

После того, как поток успешно открыт, мы можем продолжить работу с ним. Например, осуществлять чтение из него или запись:

```
FILE *stream;
int x = 300;
stream = fopen("my_calcs.txt", "w+");
if ( stream != NULL ) {
    fprintf(stream, "MyResult=%d\n", x / 200);
    fclose(stream);
} else {
    printf("Can't open file my_calcs.txt for writing.\n");
}
```

Для того, чтобы положить один символ `c` в поток `stream` используется функция `int putc(int c, FILE *stream)`, а чтобы прочитать — `int getc(FILE *stream)`. Функция `getc` возвращает код прочитанного из потока символа, либо специальное значение `EOF`, обозначающее конца потока. Функция `putc` возвращает записанный символ, в случае успешного помещения символа в поток, либо `EOF` в случае ошибки. Все остальные функции взаимодействия с потоком могут быть выражены через функции `getc` и `putc`, поскольку именно они оперируют с минимальной порцией информации — байтом, которая может быть записана или прочитана из потока. Существует большее количество функций для операций с потоками. Их описание можно найти в документации к библиотеке стандартного ввода-вывода `stdio.h`.

После того, как мы завершили работу с потоком, его необходимо закрыть. Для закрытия потока существует функция `int fclose(FILE *f)` В аргументе которой передается значение, которое было ранее получено с помощью функции `fopen`.

Обычные файлы операционной системы, с которыми мы привыкли иметь дело, немного отличаются от стандартных потоков ввода вывода тем, что в файле можно записывать и читать байты из любого места файла. Т.е. текущее положение потока ввода/вывода можно помещать в произвольное место файла. Для управления положением «начала» потока в файле используются две функции: `int fseek(FILE *stream, long offset, int whence)` и `long ftell(FILE *stream)`. Первая помещает начало потока `stream` по смещению `offset` (в байтах). Если параметр `whence` равен значению `SEEK_SET`, то смещение отсчитывается от начала файла. Если `whence` равен `SEEK_CUR`, то смещение считается от текущего положения, а если `SEEK_END` — то относительно конца файла. При чтении или записи в поток, ассоциированный с файлом, положение потока смещается на количество прочитанных или записанных байт. Если

[5]Символ `"b"` игнорируется во всех POSIX-совместимых системах, включая Linux. Другие системы могут иначе обращаться к текстовым и бинарным файлам, и добавление `"b"` может оказаться полезным, если осуществляется ввод-вывод в двоичный файл. Возможно, ваша программа будет когда-нибудь работать с не-Unix окружением.

файл открыт на запись, и положение потока достигло конца файла, его размер автоматически увеличивается на необходимую для записи величину. Файлы, которые позволяют произвольно изменять положение потока называются файлами с произвольным доступом, а файлы без такой возможности — файлы с последовательным доступом. Обычно, все файлы доступные простому пользователю являются файлами с произвольным доступом.

Вот пример работы с файлом с произвольным доступом:

```
long pos;
FILE *stream = fopen("myfile.txt", "r+");
fseek(stream, 100L, SEEK_SET);  // перейти к 101-у байту файла
                               // (первый байт имеет смещение 0);
putc(stream, 'x');              // записать в него 'x';
fseek(stream, 0L, SEEK_END);    // перейти в конец файла;
putc(stream, 'y');              // дописать в конец 'y';
fseek(stream, -20L, SEEK_END);  // перейти к 20-у с конца байту;
c=getc(stream);                 // прочитать его значение;
fseek(stream, -40L, SEEK_CUR);  // отсчитать 40 байт к началу;
putc(stream, 'z');              // записать в текущий байт 's';

fseek(stream, 0L, SEEK_END);
pos = ftell(stream);            // узнать смещение конца файла
                                // относительно начала,
                                // то есть размер файла;
pos /= 2;
fseek(stream, pos, SEEK_END);   //
putc(stream, 't');              // записать в середину 't'
fclose(stream);
```

При работе с файлом с произвольным доступом, в случае если он открыт одновременно на чтение и на запись, каждый раз, когда мы собираемся выполнять операцию записи после операции чтения или наоборот, необходимо заново отпозиционировать поток функцией `fseek`.

Семинар 6

Типы данных и их внутреннее представление

Краткое описание: На этом семинаре мы изучим тип `char`, решим несколько задач на системы счисления, научимся работать со статическими массивами, приобретём навыки работы потоками ввода и вывода.
Данный семинар включает большой набор задач на операторы структурного программирования, типы переменных, вычисление выражений, и машинную точность. Эти задачи могут служить материалом для самостоятельных и контрольных работ.

Размеры базовых типов переменных

Задача С6.1. Разберите программу 6.1, которая печатает на экране размеры переменных различных типов. Скомпилируйте и запустите её. Оцените сверху сколько различных значений может принимать переменная типа `double`.

Программа 6.1: Вывод размеров элементарных типов

```
#include <stdio.h>
int main() {
   double x;
   int y;
   long z;
   unsigned long uz;
   float a;
   printf("s(double)=%d,s(x)=%d\n", sizeof(double),sizeof(x));
   printf("s(int)=%d,s(y)=%d\n",    sizeof(int),   sizeof(y));
   printf("s(long)=%d,s(z)=%d\n",   sizeof(long),  sizeof(z));
   printf("s(unsigned long)=%d, sizeof(uz)=%d",
                sizeof(unsigned long), sizeof(uz));
   printf("s(float)=%d,s(a)=%d\n",  sizeof(float), sizeof(a));
   return 0;
}
```

Задача С6.2. Найдите максимальную степень тройки, представимую типом `int`.

Символы. Тип `char`

Символы в языке Си представляются с помощью типа `char`. Но в действительности, тип `char` это целочисленный знаковый тип размера 1 байт. Переменные типа `char` принимают целые значения из промежутка $[-128, 127]$.

Значениям от 0 до 127 стандарт ASCII[1] сопоставляет определенные символы. Символы, соответствующие значениям $[-128, -1]$ являются специальными и зависят от выбранной кодировки.

Если `a` является переменной (или выражением) одного и целочисленных типов, то соответствующий его значению символ можно напечатать в стандартный поток вывода, обозначаемый как `stdout`, с помощью команды `printf("%c", a)` или `putc(a, stdout)`.

Для считывания одного символа из стандартного потока ввода используются команды `scanf("%c", &a)` или `a = getc(stdin)`.

Задача С6.3. Изучите программу 6.2. Что она делает? Напишите программу, которая выводит таблицу со всеми символами и их ASCII кодами. Чему равны ASCII коды символов `'0'`, `'1'`, `'2'`, ..., `'9'`?

Программа 6.2: Определение ASCII кодов

```
#include <stdio.h>
int main () {
    int c;
    do {
        c = getc(stdin);
        printf ("Вы нажали '%c'. ASCII код = %d\n", c, c);
    } while (c != '\n'); // пока не нажата клавиша <Enter>
    return 0;
}
```

Задача С6.4. На основании приведённой ниже функции `to_lower` напишите программу, которая считывает строку символов и все большие латинские буквы заменяет на маленькие.

```
char to_lower(char c) {
    if (c >= 'A'  && c <= 'Z') {
        return c - 'A' + 'a';
    } else {
        return  c;
    }
}
```

Задача С6.5. Напишите функцию с прототипом `int is_word_letter(char ch)`, которая возвращает 1 если данный символ `ch` является латинской буквой, цифрой, или подчёркиванием, а в остальных случаях — 0. Просмотрите документацию функций `isdigit`, `isspace`, `isalpha` и других функций, объявленных в файле `ctype.h`.

[1] ASCII – сокращение от American Standard Code for Information Interchange – Американский Стандартный Код Обмена Информацией.

Задачи на системы счисления

Программа 6.3: Код к задаче С6.6

```
#include <stdio.h>
int main () {
    int n;
    scanf ("%d", &n);
    while(n) {
        printf("%d", n%2);
        n /= 2; // равносильно n = n / 2
    }
    printf("\n");
    return 0;
}
```

Задача С6.6. Изучите программу 6.3. Что она делает? Напишите на её основе программу, которая переводит введённое число (в десятичной записи) в двоичную запись.
а) Для этого заведите массив для хранения цифр, и вместо печати символов помещайте их в этот массив. Затем выведите содержимое массива в нужном порядке. б) Решите эту задачу, используя не массив, а рекурсивную функцию.

Задача С6.7. (Число единиц) Напишите программу, которая определяет число единиц в двоичной записи введённого числа N, $0 \leqslant N \leqslant 2^{31}$.

Программа 6.4: Функции чтения и печати чисел в двоичном представлении.

```
int read_binary() {
    int res = 0;
    while (1) {
        c = getc( stdin );
        if ( c != '0' && c != '1' ) {
            ungetc( c, stdin );
            break;
        }
    }
    return res;
}
void write_binary(int n) {
    if ( n > 0 ) {
        write_binary( n / 2 );
    }
    putc( '0' + n%2, stdout );
}
```

Задача С6.8. (Сложение в двоичной системе) Изучите код 6.4. Функции `scan_binary` и `print_binary` считывают и выводят целое положительное число в двоичной записи. Обратите внимание, что функция `write_binary` является рекурсивной, то есть в её определении встречается вызов её самой. Модифицируйте их так, чтобы они работали и для отрицательных

чисел. Напишите на основе данных функций программу сложения двух чисел. Входные числа и результат записываются в двоичной системе счисления. Число цифр во входных числах не превосходят 30.

Задача C6.9. (Системы счисления) Напишите функции

```
int scan_number(int q);
void print_number(int n, int q);
```

которые считывают и выводят числа, записанные в системе счисления с основанием `q`. Напишите программу, которая переводит данное число N, $0 < N < 2^{31}$ из Q-ричной системы в P-ричную систему счисления.
Вход. В первой строке входа даны десятичные записи чисел P и Q. Во второй строке входа — Q-ричная запись числа N. $1 < P, Q \leqslant 10$, $-2^{31} \leqslant N < 2^{31}$.
Выход. P-ричная запись числа N.

Задача C6.10. Изучите форматы ввода и вывода данных `"%o"` и `"%x"`. Напишите, используя `scanf` и `printf`, программу для перевода чисел из восьмиричной в шестнадцатиричную систему счисления.

Задача C6.11. (Сумма степеней двойки) Напишите программу, которая для данного натурального числа N предъявляет его разложение на сумму неповторяющихся степеней двойки.

Формат входа. Натуральное число N, $0 < N < 2^{31}$.

Формат выхода. Степени двойки, в сумму которых разлагается N, перечисленные через пробел.

stdin	stdout
31	1 2 4 8 16
10	2 8
256	256
255	1 2 4 8 16 32 64 128

Задача C6.12. (Сумма степеней тройки) Напишите программу, которая для данного натурального числа N, $0 \leqslant N \leqslant 2^{31}$, предъявляет его разложение на сумму неповторяющихся степеней тройки, взятых со знаком $+$ или $-$.

Задача C6.13. (Сумма больших двоичных чисел) Напишите программу, которая суммирует большие натуральные числа, заданные в двоичной системе счисления.

Формат входа. Две строки, содержащие двоичные записи натуральных чисел a и b. Число цифр в каждой записи не более 1000.

Формат выхода. Двоичная запись числа $a + b$.

Во входе двоичные записи могут перед значащими цифрами иметь несколько нулей.

stdin	stdout
11111 1	100000
10 11	101
10000 01111	11111

Задачи на операторы и машинную точность

Задача С6.14. (Сумма цифр) Напишите программу, которая находит сумму цифр данного натурального числа N.

Формат входа. Натуральное число N, $0 \leqslant N \leqslant 2^{31}$.

Формат выхода. Сумма цифр числа N.

stdin	stdout
1234	10
111111111	9

Задача С6.15. (Три максимальных) Напишите программу, которая среди данных целых чисел a_i, $i = 1, \ldots, N$ находит три различных самых больших. Решите задачу однопроходным алгоритмом, без использования массивов и сортировки.

Формат входа. Натуральное число N, $0 < N < 100$, и N целых чисел.

Формат выхода. Три различных целых числа.

Задача С6.16. (Делители числа) Напишите программу, которая находит все делители введённого натурального числа N.

Формат входа. Натуральное число N. $1 \leqslant N < 2^{31}$.

Формат выхода. Делители числа N в порядке возрастания, включая 1 и N.

Решите задачу с дополнительным условием: время работы программы должно расти с N примерно как $\sqrt{N}$.

Задача С6.17. (Таблица остатков степеней) Выведите таблицу остатков от деления b^n на p, где p — фиксированное простое число, $1 \leqslant b \leqslant p-1$, $0 \leqslant n \leqslant p-1$. Пусть значения n соответствуют строчкам таблицы, а значения b — столбцам.

Формат входа. Натуральное число p, $0 < p \leqslant 100$.

Формат выхода. Таблица чисел, состоящая из p строк, в каждой строке $p-1$ число, а именно, в k-й строке находятся остатки от деления чисел 1^k, 2^k, 3^k, $\ldots (p-1)^k$ на p.

Задача C6.18. (Тождество Гаусса) Обозначим число несократимых дробей с знаменателем n через $\phi(n)$. По определению положим $\phi(1) = 1$. Напишите программу, которая находит $\sum_d \phi(d)$, где d пробегает все делители числа N

Формат входа. Натуральное число N, $1 \leqslant N < 2^{31}$.

Формат выхода. Искомая сумма. Убедитесь, что эта сумма равна N.

Задача C6.19. (Функция Мёбиуса) Функция Мёбиуса $\mu(n)$ равна 0 для всех n, содержащих среди своих делителей хоть один квадрат целого числа, а для других n равна $(-1)^k$, где k — число всех простых делителей числа n.
а) Напишите программу, которая вычисляет функцию Мёбиуса.
б) С максимально возможной точностью проверьте тождество

$$\sum_{n=1}^{\infty} \mu(n)/n^2 = 6/\pi^2.$$

Задача C6.20. (Сумма обратных квадратов) Что выведет следующий фрагмент кода?

```
int i;
double s = 0;
for( i = 1 ; i < 1000000; i++) {
    s += 1/(i*i);
}
printf("%lf\n", s);
```

Исправьте его так, чтобы он выводил значение $\sum_{n=1}^{10^5} 1/n^2$ с точностью до 10 знаков после десятичной точки. Сравните результат с $\sum_{n=1}^{\infty} 1/n^2 = \pi^2/6$. Улучшится ли точность результата если просуммировать слагаемые в обратном порядке?

Задача C6.21. (Сумма ряда синуса) Напишите программу, которая находит частичную сумму ряда $a_n = (-1)^n \frac{x^{2n+1}}{(2n+1)!}$:

$$S(n,x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots \pm \frac{x^n}{(2n+1)!!}.$$

Формат входа. Строка, содержащая натуральное число N и действительное число x. $1 \leqslant N \leqslant 10^6$, $-10 < x < 10$.

Формат выхода. Значение $S(N,x)$ с точностью до 6 знаков после десятичной точки.

Задача С6.22. (Суммирование гармонического ряда) Напишите программу, которая суммирует первые миллион слагаемых гармонического ряда сначала с первого по последний элемент, а потом наоборот — с последнего по первый:

$$A_1 = \sum_{i=1}^{10^6} \frac{1}{i}, \quad A_2 = \sum_{i=10^6}^{1} \frac{1}{i}.$$

Попробуйте использовать сначала тип `float`, а затем `double`. Сравните результаты. Какое из двух чисел A_1 или A_2 ближе к истинному значению? Объясните, почему.

Задача С6.23. (Квадратное уравнение) Напишите программу, которая решает квадратные уравнения.

Формат входа. Действительные числа a, b, c, разделённые пробелом.

Формат выхода. В первой строке дано число корней k, а во второй перечислены значения корней.

Проверьте, правильно ли работает программа на входных данных «`1 -5 6`», «`1 -2 1`», «`0 0 0`», «`0 -2 1`», «`1 -2 0`».

Задача С6.24. (Сумма трёх квадратов) Напишите программу, которая определяет, можно ли данное число N представить в виде суммы трёх квадратов целых чисел.

Формат входа. Натуральное число N, $0 < N < 2^{31}$.

Формат выхода. `YES` или `NO`.

Задача С6.25. (Архимедовы тройки) Напишите программу, которая находит все тройки натуральных чисел (a, b, c), такие что $0 < a \leqslant b \leqslant c \leqslant N$ и $a^2 + b^2 = c^2$. Такие тройки называются архимедовыми.

Формат входа. Натуральное число N, $0 < N < 2000$.

Формат выхода. Несколько строк, в каждой строчке три числа a, b и c, разделённые пробелом. В конце в отдельной строке выведите количество найденных архимедовых троек.

Как растёт время работы вашей программы с ростом N? Можно ли написать программу, время работы которой растёт пропорционально количеству выведенных троек?

Задача С6.26. (Полные квадраты) Напишите программу, которая среди ввёдённых чисел находит числа, являющиеся полными квадратами. Напишите и используйте функцию `int is_square(int n)`, которая возвращает 1, если натуральное число n является квадратом некоторого натурального числа k, а иначе возвращает 0.

Формат входа. Натуральное число N, $0 < N < 100$, и N натуральных чисел меньших 2^{31}.

Формат выхода. Выведите те введённые числа, которые являются полными квадратами.

Задача C6.27. (Степени K) Напишите программу, которая среди ввёдённых чисел находит числа, являющиеся степенями числа k. Напишите и используйте функцию `int is_power_of(int n, int k)`, которая возвращает 1, если натуральное число n является степенью числа k, а иначе возвращает 0.

Формат входа. Натуральное число N, $0 < N < 100$, и N натуральных чисел меньших 2^{31}.

Формат выхода. Выведите те введённые числа, которые являются степенями числа k.

Задача C6.28. (Максимальная представимая степень) Найдите самую большую степень числа K, представимую типом `unsigned int`. Напишите программу, которая находит это число.

Формат входа. Натуральное число K, $0 < K < 2^{31}$

Формат выхода. Максимальная степень числа K, представимая типом `unsigned int`.

stdin	stdout
2	2147483648
3	3486784401

Задача C6.29. (Явное приведение типов) Найдите, чему равны выражения `(unsigned char)(-1)`, `(unsigned char)(256)`, `(unsigned char)(-1000)`, `(unsigned int)(-1)`, `(int)(-1.1234)`, `(int)(1.7)`, `(char)(-1.7)`, `(double)(3/2)`, `(double)(3.0/2.0)`, `(float)(1.5E+500)`, `(int)(1.5E+500)`, `(int)(1.5E+8)`, `(int)(1.25 * 20.75)`, `(int)(4 * 20.75)`.

Задача C6.30. (Округление) Напишите программу, которая округляет введённое действительное число в сторону ближайшего целого.

Задача C6.31. (Предел) Найдите с помощью компьютера значение выражения

$$\frac{\sin(\tan x) - \tan(\sin(x))}{\arcsin(\arctan(x)) - \arctan(\arcsin(x))}$$

при $x = 1,\ 1/2,\ 1/4,\ 1/8,\ 1/10,\ 1/100$ (x измеряется в радианах). К чему стремится значение этого выражения по мере того, как x стремится к 0?

Задача C6.32. (Башня степеней) Рассмотрим функцию $f(x) = x^{x^{x^{\cdots}}}$. Эта функция представляет собой бесконечную башню степеней. Строгое определение этой функции даётся через предел:

$$f(x) = \lim_{n\to\infty} f_n(x), \quad f_0(x) = 1,\ f_{n+1}(x) = x^{f_n(x)}.$$

При некоторых x этот предел существует, а при некоторых – нет. Нарисуйте графики функций $f_{100}(x)$ и $f_{101}(x)$ на промежутке $x \in (0, 3)$. Существует ли предел $f(x)$ при $x = 0.1,\ 0.7,\ 1,\ 1.1,\ 1.5,\ 2,\ 3$? При каких a верно утверждение «$x = a^{1/a} \Rightarrow f(x) = a$»? Используйте функции `pow` и `log`, объявленные в файле `math.h`. При компиляции программой `gcc` не забудьте указать опцию `-lm`.

Задачи на массивы

Задача С6.33. (Максимально удалённые точки) Напишите программу, которая во множестве введёных точек на плоскости находит пару максимально удалённых друг от друга точек. Для хранения точек используйте динамически выделенную память.

Формат входа. В первой строке дано число точек N, $1 < N < 1000$. Затем следуют N строчек, в каждой из которых дана пара вещественных координат точек.

Формат выхода. Выведите номера двух искомых точек. Точки нумеруются от 1 до N в соответствии с порядком, в котором они задавались во входе. Есть максимально удаленных пар точек несколько, выведите одну из них. Выведите также значение расстояния между ними.

stdin	stdout
4 1.5 2 -3 4 4 6 6 -1	4 2 10.2956

Задача С6.34. (Сумма чисел в столбцах) Напишите программу, которая для данной таблицы находит сумму цифр в каждом стобце.

Формат входа. В первой строке дано натуральное число N — размер таблицы, $0 < N < 50$. В следующих N строках дано по N вещественных чисел, разделённых пробелом. Каждое число по модулю менее 10^{100}.

Формат выхода. Строка, содержащая N чисел.

Задача С6.35. (Транспонирование) Напишите программу, которая транспонирует введённую таблицу целых чисел. Память для хранения чисел выделяйте динамически.

Формат входа. В первой строке дано натуральное число N — размер таблицы. $0 < N < 50$. В следующих N строках дано по N чисел, разделённых пробелом. Каждое число по модулю менее 2^{31}.

Формат выхода. Таблица чисел, в которой число, стоящее в i-й строке в j-м столбце равно числу, стоящее в j-й строке в i-м столбце входной таблицы.

Задача С6.36. (Таблица частот) Напишите программу, которая выводит список частот символов, встретившихся во входных данных.

Формат входа. Текст, содержащий латинские буквы, цифры и знаки препинания. Ввод заканчивается символом конца потока `EOF`. Длина текста менее 10000.

Формат выхода. Несколько строк, в каждой из которых стоит символ, встретившийся в тексте, и, через пробел, количество раз, которое он встретился.

stdin	stdout
ababca	a 3 b 2 c 1

Задача С6.37. (Биномиальные коэффициенты) Напишите программу, которая для данного N выводит коэффициенты разложения $(1+x)^N = \sum_{k=0}^{N} C(N,k) \cdot x^k$. Для этого напишите рекурсивную функцию `int C(int n, int k)` которая вычисляется согласно рекуррентной формуле $C(n,k) = C(n-1,k-1) + C(n-1,k)$, $C(n,n) = C(n,0) = 1$. Используйте метод запоминания вычисленных значений, а именно, объявите массив `Cd[25][25]` для хранения вычисленных значений, который изначально заполните нулями. В функции `int C(int n, int k)` реализуйте следующую логику действий:
1) перед тем как вернуть результат, поместите его в элемент `Cd[n][k]`;
2) вначале функции проверьте, нет ли ненулевого вычисленного значения в массиве `Cd`, и если есть — возвращайте его как результат.

Формат входа. Целое число N, $0 \leqslant N < 25$.

Формат выхода. Список коэффициентов $C(N,0)$, $C(N,1)$, ..., $C(N,N)$.

Задача С6.38. (Число разложений) Напишите программу, которая считает количество разложений $P(N)$ данного натурального числа N на неупорядоченные натуральные слагаемые. Например, для $N = 5$ есть 7 различных разложений: $5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1$. Реализуйте функцию $p(n,k)$ равную количеству разложений числа n на слагаемые, которые меньше либо равны k. Заметьте, что для $1 \leqslant k \leqslant n$ верна формула $p(n,k) = p(n-k,k) + p(n,k-1)$, и $p(n,1) = 1$. На основе этих соотношений напишите рекурсивную функцию `p` с запоминанием вычисленных значений. Для хранения вычисленных значений $p(n,k)$ используйте двумерный глобальный массив. $P(N) = p(N,N)$.

Формат входа. Натуральное число N, $0 < N < 100$.

Формат выхода. Число $P(N)$.

Задача С6.39. (Максимальный квадрат) Напишите программу, которая находит на данной квадратной карте $N \times N$ белый квадрат максимального размера. Карта разбита на ячейки 1×1. Каждая ячейка имеет чёрный или белый цвет. Реализуйте рекурсивную функцию `int max_square(int x, int y)` с запоминанием вычисленных значений, которая возвращает размер максимального белого квадрата, с правой нижней вершиной в ячейке с координатами (x,y) (координаты увеличиваются вниз (y) и вправо (x)). Подсказка: выразите `max_square(x,y)` через `max_square(x-1,y)`, `max_square(x,y-1)`, `max_square(x-1,y-1)` и цвет ячейки с координатами (x,y).

Формат входа. Натуральное число N, $0 < N < 100$, а затем N строчек по N символов. Символ `'#'` обозначает чёрный цвет, а символ `'.'` — белый.

Формат выхода. Размер стороны максимального белого квадрата и координаты его правой нижней вершины. Координаты отсчитываются с нуля. Если есть несколько вариантов ответа, выведите один из них.

stdin	stdout
5 #..#. ..... ##... #.... #.###	3 4 3

Потоки ввода и вывода

Задача C6.40. (Фильтр символов) Изучите программу 6, приведённую на странице 121. Эта программа выводит введенные символы, то есть считывает символы, поступающие на стандартный поток ввода, и выводит их в стандартный поток вывода. Программа 6.5 делает то же самое, только все введённые цифры заменяет на символ `'#'`. Напишите программу, которая получает на стандартный поток ввода текст и выводит его в стандартный поток вывода и при этом

а) удаляет все знаки препинания (`'.'`, `','`, `'!'`, `'?'`, `';'`, `':'`);
б) удаляет все цифры;
в) заменяет последовательности подряд идущих пробельных символов (пробела `' '`, табуляции `'\t'` и `'\n'`);
г) заменяет все специальные символы (не являющиеся буквами, знаками препинания или пробельными символами) на знак вопроса.

Программа 6.5: Код к задаче C6.40.

```
#include <stdio.h>
int main() {
    int c;
    while ( (c = getchar()) != EOF ) {
        if ( c <= '9' && c >= '0' ) {
            putchar( '#' );
        } else {
            putchar( c );
        }
    }
}
```

Задача C6.41. («Обратное эхо») Изучите код функции `recho`, приведённой ниже. Что она делает? Объясните принцип её работы.

Программа 6.6: Код к задаче C6.41.

```
void recho() {
    int c = getchar();
    if ( c != EOF ) {
        recho();
        putchar(c);
    }
}
```

Задача C6.42. (Таблица частот) Напишите программу, которая выводит список частот символов, встретившихся во входных данных.

Формат входа. Текст, содержащий латинские буквы, цифры и знаки препинания. Ввод заканчивается символом конца потока `EOF`. Длина текста менее 10000.

Формат выхода. Несколько строк, в каждой из которых стоит символ, встретившийся в тексте, и, через пробел, количество раз, которое он встретился.

stdin	stdout
ababca	a 3 b 2 c 1

Лекция 7

Работа с памятью

Краткое описание: В лекции рассматриваются понятия **адреса** и **указателя**, которые активно используются при программировании на языке Си. Практически все именованные объекты языка, к которым, в частности, относятся переменные и функции, связаны с адресом памяти. В языке Си есть возможность объявлять переменные и работать с выражениями, значение которых равно адресу. Выражения, равные адресу, называют указателями.
Мы узнаем разницу между динамическим и статическим выделением памяти и научимся пользоваться функциями `malloc` и `free`.
Ключевые слова: адрес, указатель, арифметика указателей, разыменование указателей, статическое и динамическое выделение памяти.

Адреса и указатели

В абстрактных исполнителях входные данные для алгоритма представляются в виде слова, записанного на ленте в определённом алфавите. В алгоритмических языках существует более удобное представление данных в виде множества типизированных переменных. К переменным мы обращались по имени, которое задавали исходя из смыслового значения переменной.

В действительности данные в компьютере также расположены на абстрактной ленте, которая называется линейным адресным пространством, но лента эта имеет конечную длину. В отличие от машины Тьюринга и машины Маркова, современные компьютеры обладают возможностью **абсолютной адресации** позиции на ленте и могут совершать действия одновременно с несколькими ячейками ленты.

Ячейкой ленты является байт. Абсолютным адресом ячейки называется натуральное число, обозначающее номер её позиции относительно начала ленты. В компьютерах с 32-битной архитектурой длина ленты равна $2^{32} = 4\,294\,967\,296$. Поэтому, можно сказать, что адрес — это число от 0 до $2^{32} - 1$ включительно.

В языке Си имена переменных и функций являются обозначением адресов памяти, где расположены их значения и тела (представление в машинном коде) соответственно.

Операции взятия адреса и разыменования

До некоторых пор, можно было вполне пользоваться переменными, как хорошей абстракцией и ничего не знать об адресах и об устройстве памяти. Однако, программы на языке Си, в которых не используются операции с адресами, скорее исключение, чем правило. Оперирование адресом является мощным инструментом для решения многих задач. Для начала рассмотрим простые операторы работы с адресами и поясним их смысл.

Операция **взятия адреса** обозначается унарным оператором `&` и может быть применена к любому именованному объекту или L-выражению. Но результат операции взятия адреса L-выражением уже не является (т.е. фактически, нельзя изменить адрес переменной).

Операция разыменования обозначается унарным оператором '`*`'. Выражение вида `*X` может быть интерпретировано как «значение, находящееся по адресу `X`». В этом случае, выражение `X` должно иметь указательный тип (как объявляются указательные типы мы рассмотрим ниже).

Указатели

В языке Си возможно объявить переменную, которая бы содержала адрес других переменных. Вот пример объявления такой переменной.

```
char *str;
```

Данная запись означает, что переменная `str` имеет значение адреса, по которому расположено значение типа `char`. В этом случае говорят, что переменная `str` *указывает* на тип `char` или имеет **указательный** тип. Саму переменную называют при этом **указателем**.

Рассмотрим простой код:

```
int a = 1;
int *p;
p = &a;
*p = 2;
printf("Значение переменной = %d\n", a);
printf("Адрес переменной = %p\n", &a);
```

В результате его выполнения будут выведены следующие строки:

```
Значение переменной = 2
Адрес переменной = 0xbffff7a4
```

Значение адреса может быть другим, это зависит от компилятора, архитектуры компьютера, времени выполнения программы и прочих факторов. В этом коде значение переменной `p` равно *адресу* переменной `a` (присваивание происходит в строчке «`p = &a;`»). А значение выражения `*p` равно значению, лежащему по адресу, указанному в `p`, то есть значению переменной `a`. Важно отметить, что выражение `*p` может встречаться слева от знака равно и выступать в качестве имени принимающей ячейки, т.е. является L-выражением.

Чтобы показать мощь работы с указателями, рассмотрим задачу:

Задача Л7.1. Проанализируйте следующую программу и ответьте, что будет выведено на экран:

```
void func(int *a) {
    *a = 1;
}
int main() {
    int x = 10;
    func(&x);
```

```
    printf("%d\n", x);
}
```

Указатель достаточно часто используется в программах на языке Си. Более того, ни одна уважающая себя программа не обходится без использования указателей. Но указатель всегда таит в себе скрытую опасность. Например, если сравнить указатель с почтовыми адресами людей, то вполне можно представить ситуацию, когда письмо приходит «не по адресу» или вообще не доходит. Например, это может произойти из-за того, что адресат переехал и живет в другом доме, или – хуже того – умер. А возможно, дом уже снесли и адрес потерял всякий смысл. Похожие ситуации происходят и с указателями на области памяти в языке Си. Т.е. никто не предоставляет гарантии, что по указанному адресу храниться именно то, что нам нужно. Чтобы максимально избежать подобных случаев, принято инициализировать переменные, которые ни на что не указывают специальным значением `NULL`. Значение `NULL` является константой и эквивалентна адресу 0.

Продемонстрируем «безопасный» вариант функции `func` из предыдущего примера:

```
void func(int *a) {
    if ( a != NULL ) {
        *a = 1;
    }
}
```

В этой реализации функции `func` мы предварительно проверяем, действительно ли `a` является инициализированным указателем. И только в этом случае меняем значение, на которое он указывает. Естественно, использование `NULL` является конвенциональным. Это означает, что все проверки на `a` имеют смысл в том случае, если вызывающая сторона заботится о правильной инициализации указателей. Например, следующая программа, несмотря на наличие проверки, может выдавать ошибку исполнения:

```
int main() {
    int *y;
    func(y);
    printf("%d\n", *y);
}
```

Переменная `y` является неинициализированным указателем и, в зависимости от компилятора и среды, может указывать на любую область памяти, работа с которой может привести к ошибке.

Арифметика указателей

Указатели являются достаточно мощным средством, и язык Си обеспечивает максимальное удобство работы с ними. В частности, с указателями возможно проводить арифметические операции. Например, пусть объявлена переменная типа `double *x`. Выражение вида `x + 1` будет иметь значение адреса `x`, но увеличенного на `sizeof(double)`, т.е. на размер типа, на который указывает переменная `x`. Таким образом, все операции работы с массивом можно свести к операциям с указателем. Например, следующие строки с точки зрения компилятора эквивалентны:

```
x[i] = 3;
*(x + i) = 3;
```

В данном случае, переменную `i` называют *смещением* относительно адреса `x`.

Использование арифметики указателей позволяет писать более компактный код. Например, для копирования строки `b` в строку `a` достаточно написать

```
while (*a++ = *b++) ;
```

вместо

```
for (i = 0; b[i] != 0; i++) a[i] = b[i];
```

Типичным примером применения арифметики указателей служит функция подсчета длины строки:

```
int strlen(char *str) {
    char *start = str;
    while (*str) str++;
    return (str - start);
}
```

Здесь, начало строки передается в параметре `str` и сохраняется в переменной `start`. Затем, в цикле `while` ищется конец строки (нулевой символ), при этом текущая просматриваемая позиция находится в переменной `str`. Длина строки вычисляется как разница между адресом нулевого символа, оканчивающего строку, и началом строки. Обратите внимание на то, что функция `strlen` работает время, пропорциональное длине строки.

Переменные и выделение памяти

Компилятор языка Си выделяет память для всех объявленных переменных в соответствии с их размером. Если переменная объявлена внутри какой-либо функции или блока, то память для этой переменной выделяется лишь на время выполнения данной функции либо блока. Такие переменные называют **локальными**. После выхода из блока или функции, адресное пространство, выделенное для локальных переменных, уничтожается.

Переменные, объявленные вне каких-либо блоков, называются **глобальными**. Глобальные переменные имеют «постоянное место жительство» в адресном пространстве программы и не уничтожаются во время работы программы.

Рассмотрим пример:

```
int count_swaps = 0;
void swap(int *a, int *b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
    count_swaps++;
}
```

Функция `swap` меняет местами два целых числа, расположенных по указанным адресам `a` и `b`. Переменная `tmp` является локальной — она объявлена внутри блока, описывающего тело

функции `swap`. Она требуется для временных нужд, и «живёт» в адресном пространстве лишь во время выполнения функции `swap`.

Глобальная переменная `count_swaps` считает количество выполненных обменов.

Для имён переменных существует также такое понятие как **область видимости**. Переменные, объявленные внутри некоторого блока будут доступны из всех блоков, вложенных в данный. Вполне возможно объявить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из «внешних» переменных. В таком случае, любое обращение к переменной с таким именем будет однозначно интерпретироваться как обращение к локальной переменной. Для локальной переменной будет выделена область памяти, отличная от области памяти, отведенной под «внешнюю» переменную.

Для пояснения, рассмотрим следующую задачу:

Задача Л7.2. Не прибегая к программированию, определите, какие значения будут выведены на экран:

```
int k = 10;
int f( int k ) {
    printf("4: %d\n", k);
}
int g( int n ) {
    printf("4: %d\n", k);
}

int main() {
    printf("1: k = %d\n", k);
    if ( 1 ) {
        int k = 11;
        printf("2: k = %d\n", k);
        k = 12;
    }
    printf("3: k = %d\n", k);
    {
        k = 13
        f( 14 );
        g( k );
    }
    return 0;
}
```

Локальные переменные размещаются в специальной области памяти программы, называемой **программным стеком**. Метод размещения локальных переменных в программном стеке называется автоматическим. Ещё существуют статический, внешний и регистровый методы размещения переменных. При статическом методе размещения переменной адрес для переменной фиксируется на всё время выполнения программы. Для того, чтобы объявить переменную как статическую, требуется добавить слово `static` перед ее типом. Например:

```
static int x = 0;
```

Хорошим тоном является явная инициализация статических переменных перед объявлением. Рассмотрим следующую функцию:

```
int increment_counter(int delta) {
    static int counter = 0;
    counter += delta;
    return counter;
}
```

Здесь статическая переменная `counter` инициализируется в момент старта программы и продолжает «жить» в адресном пространстве вне зависимости от количества и последовательности вызовов функции `increment_counter`. Статическая переменная ведёт себя как глобальная переменная, но видна она только внутри блока, в котором объявлена.

Внешний метод размещения используется для взаимодействия модулей программы. Переменная, объявленная как внешняя, доступна за пределами модуля (файла) в котором она объявлена. Сложные программы могут состоять из нескольких модулей и внешние переменные становятся общими для различных модулей программы. Объявляется внешняя переменная с использованием ключевого слова `extern`:

```
extern double rub2usd;
```

Однако, этого объявления не достаточно, для того, чтобы понять, в каком именно модуле хранится эта переменная. Поэтому, в модуле-экспортёре эта переменная объявляется дважды (с ключевым словом `extern` и без него), а в модуле-импортёре — только с `extern`. Далее, мы встретимся с внешними переменными и внешними функциями при обсуждении библиотек.

Регистровый способ размещения — это указание компилятору, чтобы для хранения данной переменной был использован процессорный регистр (самая быстрая память), а не обычна оперативная память. Сделать переменную регистровой имеет смысл в случае, если она используется в алгоритмах, очень требовательных ко времени исполнения, для ускорения вычислений. Объявляется регистровая переменная при помощи ключевого слова `register`. Рассмотрим следующий пример «быстрой» реализации функции, вычисляющей n-е число Фибоначчи с использованием регистровых переменных:

```
long fib(int n) {
    register long a = 1;
    register long b = 1;
    while ( n-- ) {
        b = b + a;
        a = b - a;
    }
    return b;
}
```

Динамическое выделение памяти

В предыдущих случаях мы видели, что компилятор сам заботится о выделении памяти для размещения значений переменных и массивов в том случае, если их размер известен в

момент компиляции. Но как быть, например, с массивами, длина которых определяется только уже в момент вычислений? В этом случае поступают двояким образом. Если возможно как-то определить максимально возможные (и разумные) границы массива, то их объявляют как глобалные или локальные переменные. Если таковое невозможно, то используют другую технику выделения: *динамическую память*.

Принцип работы динамической памяти достаточно прост. Программа запрашивает у т. н. *среды исполнения* некоторое количество байт, которое ей потребуется для размещения данных. Требуемое количество байт может быть получено уже при работе программы и быть неизвестной на момент написания программы и её компиляции.

Если существует область памяти, достаточная для непрерывного размещения запрошенного количества байт, то программа получает *указатель* на эту область. Далее, эта область памяти может быть использована им записи и чтения. После того, как выделенная область перестала быть нужной программе, её требуется *освободить*.

В чём приемущества использования динамической памяти? Во-первых, динамическая память выделяется по необходимости в необходимом количестве и нет нужды заранее выделять память сверх меры, если этого не требуется. Во-вторых, выделенная область продолжает быть доступна до момента её освобождения для различных участков программы, вне зависимости от того, на каком уровне вложенности блоков она была выделена (разумеется, если возможно обеспечить передачу указателя на область между этими участками).

В стандартной библиотеке существует набор функций для работы с динамическим выделением памяти. Это функции `malloc` и `free`. Имя первой функции образовано от слов *Memory Allocation* и эта функция реализует запрос на выделение области памяти заданного размера. Вот её объявление и пример использования:

```
void *malloc(size_t bytes);

size_t size = 20000;

void *x = malloc(size);
```

В качестве аргумента, функции `malloc` передается количество байт, которое требуется выделить. А возвращаемое значение — адрес начала области памяти заданного размера. Если выделить память запрошенного размера не удалось, функция возвращает значение `NULL`. Следует обратить внимание на тип возвращаемого значения. По символу `*` можно догадаться, что это — указательный тип, только на что он указывает? Тип `void` означает «пустой» тип. Указатель на тип `void` используется для того, чтобы подчеркнуть, что область памяти просто является последовательностью байт и используется во многих системных функциях общего назначения[1]. Для дальнейшей работы с выделенной областью памяти, указатель на `void` требуется *привести* к нужному типу.

Чтобы освободить выделенную функцией `malloc` память, для возможного повторного использования, требуется вызвать функцию `free` с единственным параметром — адресом начала выделенной области:

[1]По стандарту ANSI C 99 арифметические операции с указателем на `void` трактуются как операции с массивом байт.

```
void *x = malloc(100);
free(x);
```

Способ работы динамической памяти тесно связан с принципами действия операционной системы (ОС), что во многом определяет удобство работы с нею, поэтому кратко рассмотрим основы работы ОС.

Изначально, для работающей программы (процесса), ОС выделяет некоторое количество памяти, которое, по её мнению, было бы достаточно для того, чтобы загрузить код, глобальные переменные и константы, а также существовало достаточно места для стека[2]. ОС предоставляет программам возможность увеличивать размер задействованного адресного пространства либо за счет увеличения стека, либо за счет увеличения *кучи* (heap) — свободной области памяти, отведённой программе специально для удовлетворения возникающих в процессе вычислений потребностей в памяти. С точки зрения операционной системы, затраты на выделение программе пространства под стек или под кучу приблизительно одинаковые, однако, с программной точки зрения, выделение памяти под стек происходит автоматически, а увеличение кучи требует определённого запроса к операционной системе (системного вызова). Как всякий системный вызов, запрос на увеличение размера кучи требует определенных ресурсов ОС, и, как следствие, процессорного времени, поэтому динамическое выделение памяти считается достаточно дорогим удовольствием.

Разработчики ОС балансируют между скоростью выделения памяти и экономией самой памяти программы, поэтому, алгоритмы выделения памяти (например те, которые используются в реализациях функции `malloc`) стараются повторно использовать уже освобождённую динамическую память, чтобы не делать дорогостоящих вызовов ОС для запроса увеличения используемого адресного пространства. Однако, это приводит к другой проблеме — *фрагментации* динамической памяти — ситуации, когда (из-за частых запросов на выделение и освобождения маленьких кусочков памяти) занятые области состоят из множества маленьких кусков перемежающихся маленькими свободными областями, то есть образуют фрагментарную («пористую») структуру. Это затрудняет (замедляет) работу функции `malloc` по поиску свободной области подходящего размера.

[2]Программный стек — это специальная область памяти, предназначенная для организации вызовов вложенных процедур, хранения их локальных переменных и аргументов.

Семинар 7

Типы данных. Работа с памятью

Работа со указателями

Краткое описание: Мы рассмотрим различные примеры задач, в которых оказывается удобным применение указателей.
Отработаем процедуру динамического выделения памяти для одномерных и двумерных массивов.
Используя указатели и приведение указательных типов, изучим внутреннее представление переменных типа `double`.

Задача С7.1. В коде 7.1 даны два определения функций `swap1` и `swap2`. Опишите результаты вызовов этих функций. Добавьте команды вывода значений переменных `x` и `y` для проверки вашей догадки.

Программа 7.1: Код к задаче С7.1.

```
void swap1(int *a, int *b) {
    int *tmp;
    *tmp = *a; *a = *b ; *b = *tmp;
}
void swap2(int a, int b) {
    int tmp;
    tmp = a; a = b ; b = tmp;
}
int main() {
    int x = 1, y = 2;
    swap1( &x, &y);
    swap2( x, y);
    return 0;
}
```

Задача С7.2. Студент Заза Сичинава решил освоить работу с указателями. Для этого он решил доработать предыдущую программу 7.1 и написал функцию `getaddr` (см. код 7.2). Что выведет данная программа и выведет ли вообще? В чём, на ваш взгляд, кроется проблема?

Программа 7.2: Код к задаче С7.2.

```
int * getaddr( int a) {
    return &a;
```

```
}
int main() {
    int x = 1, y = 2;
    swap(getaddr(x), getaddr(y));
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Задача С7.3. Напишите функцию

```
int stats(double *a, int size,
             double *min, double *max, double *avg);
```

которая вычисляет минимальное, максимальное и среднее значение элементов массива `a`. Результат она должна разместить соответственно по адресам `min`, `max`, `avg`.

Работа со строками

Задача С7.4. Изучите описания функций `count_letters1` и `count_letters2`. Эти функции подсчитывают число латинских букв в заданной строке. В первой функции используется индекс `i`, который на каждой итерации инкрементируется. А во второй функции переменной индекса нет, вместо него используется указатель, который также инкрементируется на каждой итерации. В первой функции переменная `str` не изменяется.

```
int count_letters1(char *str) {
    int res = 0;
    int i;
    for(i = 0 ; str[i] ; i++ ) {
        if ( str[i] >= 'a' && str[i] <= 'z' ||
            str[i] >= 'A' && str[i] <= 'Z' ) {
            res++;
        }
    }
}
int count_letters2(char *str) {
    int res = 0;
    while ( *str ) {
        if ( *str >= 'a' && *str <= 'z' ||
            *str >= 'A' && *str <= 'Z' ) {
            res++;
        }
        str++;
    }
}
```

Задача С7.5. Ознакомьтесь с кодом функции `substring`. Что делает данная функция? Перепишите эту функцию без использования арифметики указателей. Напишите программу, которая

тестирует работу функции.

```
char* substring(char *str, char *substr) {
    char *p;
    while ( *str ) {
        char *s = substr;
        p = str;
        while ( *s && *(p++) == *(s++) );
        if ( !*s ) return str;
        str++;
    }
    return 0;
}
```

Задача С7.6. Ознакомьтесь с документацией стандартной функции `strcpy` из библиотеки `string.h`. Попробуйте написать её полный аналог, взяв за основу приведённую ниже функцию `copy_string`.

```
void copy_string(char *dst, char *src) {
    while ( *src ) {
        *(dst++) = *(src++);
    }
}
```

Задача С7.7. Используя функции из библиотеки `string.h`, напишите программу, которая:

а) читает число вхождений одной подстроки в другую;
б) делает тоже самое, но без учета регистра;
в) разбивает текст на строки, шириной не больше заданной в местах разделения между словами(пробельные символы); слова, превышаюшие заданную ширину, должны быть разорваны и перенесены.

Задача С7.8. Напишите функцию

```
int shrink_spaces(char *str, int *size, int *words, int *chars);
```

которая получает указатель на строку и модифицирует её: все последовательности подряд идущих пробельных символов заменяет на один пробел и удаляет пробельные символы с начала и с конца строки. Пробельные символы – это символ пробела `' '`, символ табуляции `'\t'` и символ перевода строки `'\n'`. При реализации используйте арифметику указателей (инкрементируйте указатели, а не целочисленный индекс). Функция должна возвращать новую длину строки (`size`), число слов (`words`), число непробельных символов (`chars`).

Динамическое выделение памяти

Для выделения памяти, «время жизни» которой выходит за пределы времени жизни функции, или размер которой изначально не определен, используется

динамическое выделение. Для работы с динамически выделяемой памятью существуют функции `void *malloc(size_t size)`, `void * realloc(void *mem, size_t size)`, `void free(void *mem)`. Первая возвращает указатель на область памяти, размером `size` байт. Вторая изменяет размер выделенной ранее памяти и возвращает указатель на новую область памяти. Информация, хранящаяся в прежней области, при этом перемещается в новую. Обе эти функции возвращают значение `NULL`, в случае, если выделение невозможно (использована вся память). Третья функция `free` принимает в качестве аргумента адрес области, полученный ранее одной из первых двух функций и освобождает ее. После освобождения, память по старому адресу будет недоступна и чтение/запись в освобожденной области памяти может привести к ошибке.

Задача С7.9. Изучите следующую программу, которая проверяет введенные числа на уникальность и выводит их. Перепишите программу так, чтобы `realloc` вызывался не для каждого нового элемента, а реже. Для этого выделяеёте память с запасом и объявите переменную `allocated_size`, для хранения значения выделенной памяти.

```
#include <stdlib.h>
int main() {
    int *x = NULL;
    int number, i, found;
    int count = 0;

    while ( scanf("%d", &number) == 1 ) {
        found = 0;
        for ( i = 0; i < count; i++ ) {
            if ( number == x[i] ) {
                found = 1;
                break;
            }
        }
        if ( ! found ) {
            x = realloc( x, ++count * sizeof(int) );
            if ( x == NULL ) {
                fprintf(stderr, "Ошибка выделения памяти");
                return 1;
            }
            x[count-1] = number;
        }
    }
    for ( i = 0; i < count; i++ ) {
        printf("%d\n", x[i]);
    }
    free( x );
    return 0;
}
```

Задача С7.10. Напишите функцию, которая возвращает указатель на строку-шифр, которая получается из заданной строки путём вставки после первой буквы каждого слова его части, находящейся между первой и последней буквой. Если слово состоит из одной или двух букв, оно не меняется. Например: «Пора уже выучить новый язык» перейдёт в «Порора ужже выучитыучить новыовый язызык». Воспользуйтесь функциями `isspace`, `ispunct` для определения границ слов.

Двумерные массивы

Задача С7.11. Пусть задана матрица 4×4 вещественных чисел: `double matrix[4][4]`. Напишите программу, которая запрашивает у пользователя значения элементов матрицы и, затем, считает её детерминант.

Задача С7.12. Запустите следующие две программы 7.3 и 7.4. Чем они различаются? Объясните, каким образом работает каждая из них. В чём преимущество каждой из реализаций?

Программа 7.3: Код №1 к задаче С7.12.

```
#include <stdio.h>
int matrix[5][5];
int main() {
    int i, j;
    for ( i =0; i < 5; i++ ) {
        for(  j = 0; j < 5; j++ ) {
            matrix[i][j] = (i * j) % 5;
        }
    }
    for ( i = 0; i < 5; i++) {
        for( j = 0; j < 5; j++ ) {
            printf("%5d ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("%lf\n", val);
    return 0;
}
```

Программа 7.4: Код №2 к задаче С7.12.

```
#include <stdio.h>
#include <malloc.h>
int *matrix;
int main() {
    int i;
    matrix = (int *) malloc(5*5*sizeof(int));
    for ( i = 0; i < 5 * 5; i++ ) {
```

```
        matrix[i] = (i / 5) * (i % 5) % 5;
    }
    for ( i = 0; i < 5*5; i++ ) {
       printf("%5d ", matrix[i]);
       if (i % 5 == 0 ) printf("\n");
    }
    free(matrix);
    return 0;
}
```

Задача С7.13. Объясните, почему программа 7.5 не будет компилироваться.

Программа 7.5: Код с ошибкой к задаче С7.13.

```
#include <malloc.h>
int *matrix;
int main() {
    int i, j;
    matrix = (int *) malloc(5*5*sizeof(int));
    for ( i =0; i < 5; i++ ) {
        for(  j = 0; j < 5; j++ ) {
            matrix[i][j] = (i * j) % 5;
        }
    }
    free(matrix);
    return 0;
}
```

Программа 7.6: Динамическое выделение двухмерного массива (вариант 1).

```
int** matrix_create(int n, int m) {
    int **res = (int**) malloc( n * sizeof(int*) );
    if (res) {
        int i;
        for( i = 0 ; i < n ; i++ ) {
            res[i] = (int*) malloc( m * sizeof(int) );
            if ( !res[i] ) {
                for( i-- ; i >=0 ; i-- ) free( res[i] );
                return 0;
            }
        }
        return res;
    } else {
        return 0;
    }
}
void matrix_delete(int **data, int n) {
    if ( data ) {
```

```
        int i;
        for( i = 0 ; i < n ; i++ ) {
            if ( data[i] ) free( data[i] );
        }
        free( data );
    }
}
```

Программа 7.7: Динамическое выделение двумерного массива (вариант 2).

```
int** matrix_create(int n, int m) {
    int **res = (int**)malloc( n * sizeof(int*) );
    if ( res ) {
        int i;
        res[0] = (int*) malloc( n * m * sizeof(int) );
        if ( res[0] ) {
            for( i = 1 ; i < n ; i ++) {
                res[i] = res[i-1] + m;
            }
            return res;
        } else {
            free( res );
        }
    }
    return 0;
}
void matrix_delete(int **data, int n) {
    if ( data ) {
        if ( data[0] ) free( data[0] );
        free( data );
    }
}
```

Задача С7.14. (Произведение матриц) Изучите коды 7.6 и 7.7. В них представлены две различные реализации функций `matrix_create` и `matrix_delete`. Первая создаёт двумерный массив элементов типа `int` указанного размера, а именно, она создает массив из `n` указателей, которые указывают на `n` массивов размера `m`. Функция `matrix_delete` освобождает память, выделенную под двумерный массив. В первой реализации (7.6) для каждого из `n` одномерных массивов делается отдельный вызов функции `malloc`. Во второй реализации (7.7) один раз вызывается `malloc`, в котором сразу запрашивается весь необходимый объём памяти (`n*m*sizeof(int)`) который, потом в цикле распределяется между `n` массивами. Используя одну из двух реализаций, напишите программу, которая получает на вход две матрицы размера $n \times n$, и возвращает их произведение. Реализуйте в коде функцию

```
int **mmultiply(int **a, int **b, int size);
```

которая осуществляет вычисление произведения матриц, заданных указателями `a` и `b`.

Формат входа. Натуральное число N, $N < 100$, а затем описание двух матриц. Описание каждой матрицы состоит из N строк, в каждой строке перечислены через пробел N целых чисел, каждое из которых по модулю меньше 10000.

Формат выхода. Описание матрицы, равной произведению двух матриц, данных на входе.

Задача C7.15. Перепишите функции `matrix_create` и `matrix_delete` так, чтобы их можно было использовать для выделения (освобождения) памяти двумерных массивов элементов произвольного типа. Для этого в эти функции придётся добавить третий аргумент — размер типа элемента.

Изучение внутреннего представления типа `double`

Программа 7.8: Изучение внутреннего представления типа `double`.

```
#include <stdio.h>
typedef unsigned char BYTE; // ввели короткое обозначение BYTE
                            // для типа unsigned char
void print_byte(BYTE b) {
    int i;
    for (i = 7; i >= 0; i--) {
        printf ("%d", b & (1 << i) ? 1 : 0 );
    }
    printf(" ");
}
void print_bytes (void *mem, int n) {
    int i;
    for (i = n-1; i >=0; i--) {
        print_byte ( * (((BYTE*)mem) + i) );
    }
    puts("");
}
int main() {
    double x;
    while(1) {
        scanf ("%lf", &x);
        print_bytes(&x, sizeof(x));
    }
    return  0;
}
```

Задача C7.16. (Представление типа `double`) Изучите код 7.8. Изучите с его помощью, как представляются в памяти числа 1, -1, 2, -2, 0.5, 0.25, 0.125. Опишите формат хранения чисел вида $\pm(1+\varepsilon)2^n$, где $n \in \mathbb{Z}$, $0 \leqslant \varepsilon < 1$. Подсказка: при хранении числа x в переменной типа `double` один бит S уделяется под знак числа, группа из m бит хранит неотрицательное целое

число M, и группа из e бит хранит неотрицательное целое число E, так что значение числа x равно

$$x = (-1)^S \cdot (1 + M \cdot 2^{-m}) \cdot 2^{(E-K)},$$

где K – некоторая константа. Чему равны константы e, m и K?

Задача С7.17. (Плотность типа `double`) Действительных чисел, представимых типом `double` конечное число, и это число ограничено $2^{8 \cdot \text{sizeof}(double)}$. На числовой оси $(-\infty, +\infty)$ эти числа располагаются с различной плотностью. Плотность в точке x определим как отношение числа точек в некоторой окрестности числа x и размера этой окрестности:

$$\rho(x) = \frac{\text{количество точек на отрезке } [x-\varepsilon, x+\varepsilon]}{2\varepsilon}$$

Альтернативное определение:

$$\rho(x) = \frac{1}{\text{среднее расстояние между числами, представимыми типом } \texttt{double} \text{ в окресности числа } x}.$$

Можно написать функцию, которая вычисляет логарифм от плотности чисел, представимых типом `double` в окрестности данного числа x. Изучите код 7.9. Проверьте, что зависимость логарифма плотности от логарифма числа x линейная. Чему равен линейный коэффициент в этой зависимости?

Программа 7.9: Изучение плотности чисел, представимых типом `double`.

```
#include <stdio.h>
#include <math.h>
int density_log(double x) {
    int count = 0;
    double c = x, l, r;
    l = x - 1.0; r = x + 0.5;
    while(r - l > 0) {
        c = (l + r) / 2;
        if(count % 2 == 0) {
            l = c;
        } else {
            r = c;
        }
        count++;
    }
    return count;
}
int main() {
    double x;
    for(x=1E-20; x < 1E+20; x *= 2.5555) {
        int count = density_log(x);
        printf("log(density(%7.4lf)) = %3d\n", x, count);
    }
```

```
    return 0;
}
```

Задача C7.18. Какие из указанных чисел представимы типом double:
$0,\ -1345,\ 1,\ 1.5,\ 2/3, 3/2,\ 1.33,\ 1.125,\ 1/70,\ 2^{100},\ 2^{-128},\ 2^{300},\ 10^{10},\ 10^{22},\ 111111111$?

Задача C7.19.* Сравните внутреннее представление типов `float` и `double`. Оцените, сколько чисел типа `double` лежит между двумя соседними числами, представимыми типом `float`.

Задача C7.20. Напишите программу, которая для данного натурального числа a, $1 \leqslant a \leqslant 2^{32}$, находит максимальное число вида a^n, абсолютно точно (без машинной погрешности) представимое типом `double`.

Лекция 8

Сложные типы данных и библиотеки функций

Краткое описание: В лекции рассматриваются операции с типами данных, сложные типы — структуры и объединения. А также назначение, способы построения и использования библиотек функций.
Ключевые слова: структуры (`struct`), объединения (`union`), оператор `typedef`, статически и динамически подключаемые библиотеки.

Явное приведение типа указателей

В Си часто пользуются явным приведением типа указателей. Указательный тип имеет всегда известный размер, поскольку является *адресом*. При приведении указателя к другому типу изменяется только тип (то, как интерпретирует этот адрес компилятор), а сам адрес остаётся прежним. Тип указателя требуется знать компилятору только при выполнении операций разыменования указателя и арифметических действий с указателем.

Рассмотрим следующий пример:

```
int x = 65535;
char z = * (char *)&x;
char *b = (char*)&x;
```

Последнее выражение следует читать так: «взять значение типа `char`, находящееся по тому же адресу, где расположена переменная `x`».

В результате, переменная `z` примет значение, равное значению первого байта (поскольку тип `char` имеет размер 1 байт) представления целого числа `x`. Четыре байта `b[0]`, `b[1]`, `b[2]`, `b[3]` — это байты представления числа `x`.

Использование кастинга указателей является достаточно опасным. Размеры типов отличаются, и если программист об этом забыл, могут возникнуть неприятные и трудно обнаружимые ошибки. Пример «опасного» кастинга:

```
char z = 30;
int *p;
p = (int *)&z;
*p = 7754;
```

В данном случае, значение 7745 будет записано область памяти, расположенную по тому же адресу, что и `z`, но размером, соответствующим типу `int`, который превышает размер типа `char`. Переменная `z` располагается где-то в памяти и занимает 1 байт. А присваивание `*p=7754` помещает представление числа 7754 в четыре байта, находящихся по этому адресу. Это может, в лучшем случае, привести к модификации значений соседних (по объявлению) переменных,

а в худшем – порче стека или выходу за пределы страницы памяти (segmentation fault), что проявляется как «падение» программы. Часто говорят, что кастинг указателей «снимает типизацию», т.е. лишает компилятор возможности контролировать тип выражения на стадии компиляции. Вся ответственность за ошибки, связанные с кастингом при этом возлагается на программиста.

Как показывает практика, обойтись без кастинга указателей невозможно. Но избежать опасностей, связанных с кастингом – вполне реально. Вот несколько рекоммендаций:

1. Избегайте явного и неявного кастинга.
2. Если кастинг неизбежен, следите за тем, чтобы размер приводимого типа был не меньше, чем приведённого.
3. Если и это невозможно, то используйте дополнительные механизмы, сохраняющие и передающие информацию о размере области памяти, указатель на которую преобразовывается.
4. Явно выделяйте данные, требующие кастинга. Стандартным стало соглашение использовать для этих целей указатель на тип `void`. Таким образом обозначают указатель на данные, тип которых неизвестен на этапе компиляции. Это автоматически заставляет программиста производить явный кастинг, поскольку никакие арифметические выражения с типом `void` невозможны, а также заставляет задуматься о размере области памяти, скрывающейся за `(void *)`. Примером могут служить многие функции из стандартной библиотеки (`malloc`, `free`, `qsort`, ...), а также многие системные вызовы.

Сложные типы данных

Мы уже встречались со сложными типами данных, такими как массив, в котором каждый его элемент имеет либо примитивный тип, либо также является массивом. Однако, массив объединяет элементы только одного типа и каждый элемент адресуется порядковым номером.

На практике часто удобно оперировать не массивами и не примитивными типами переменных, а некоторой совокупностью переменных, имеющих, возможно, различный тип, но логически объединенных в единое целое.

Примером сложных типов могут быть моделируемые объекты реального мира и различного рода абстракции: материальная точка в трехмерном пространстве, информация о студенте, геометрическая фигура, окно программы, и т.д. Каждый моделируемый объект обладает набором характеристик, которые каким-то образом должны быть отражены в программе в виде чисел, строк или каких-то других объектов.

Представление сложных объектов в языке Си возможно путём конструирования новых типов данных. «Конструкторами» в этом случае служат специальные абстракции языка Си: *структуры* и *объединения*.

Структуры

Структура — это составной тип данных, объединяющий в себе несколько элементов других типов. При этом каждый элемент структуры имеет имя и может быть модифицирован или прочитан по этому имени. Размер структуры равен сумме размеров типов входящих в нее элементов.

Объявление структур

Объявление структуры является объявлением нового типа данных, который затем может быть использован как и любой другой тип: для объявления переменных, типов возвращаемых значений и аргументов функций. Структура объявляется при помощи ключевого слова `struct`. Рассмотрим пример объявления:

```
struct complex {
    double re;
    double im;
};
```

Таким образом мы объявили новый тип данных, который в последствии будет называться `struct complex`, содержащий 2 *поля* типа `double`: `re` и `im`.

После объявления типа, мы можем создавать переменные этого типа, объявлять функции, возвращающие и принимающие этот тип данных в качестве аргумента и даже создавать массивы.

```
struct complex z;
struct complex multiply(struct complex x, struct complex y);
struct complex polar2complex(double rho, double phi);

struct complex points[200];
```

Обращение со структурами

Для обращения к полям структур используется оператор '.' (точка). Он ставится после выражения, имеющего тип структуры, а после точки ставится имя поля структуры. Результатом выполнения оператора обращения к полю является L-выражение типа поля. Вот пример обращения к полю структуры:

```
struct complex z;

z.re = 0.15;
z.im = z.re * 2;
```

Выражения структурного типа могут являться L-выражениями, однако арифметические операторы применять к структурам не удастся, поскольку компилятор не может произвести преобразование структурного типа к примитивному. Например, данный кусок кода после компиляции выполнит ожидаемые действия:

```
struct complex a, z;

z.im = 10;
z.re = 20;

a = z;
```

Значения полей переменной `z` полностью скопируются в переменную `a`. А следующий фрагмент кода вызовет ошибку компиляции:

```
if ( a == z ) printf ("ok!");
```

На практике часто используются указатели на структуры для передачи объектов функциям в качестве параметров , чтобы не копировать каждый раз содержимое полей (при передачи параметров, значения аргументов копируются). Для доступа к полям структуры, переданной по указателю можно сначала *разыменовать* указатель на структуру с помощью оператора *, затем, при помощи оператора ., обратиться к полю структуры:

```
void conjugate(struct complex *z ) {
    (*z).im = -(*z).im;
}
```

Для подобных операций существует также специальный оператор '->', называемый «разыменованием указателя». Слева от него пишется выражение типа указателя на структуру, а справа – имя поля. Вот пример использования данного оператора:

```
void conjugate(struct complex *z) {
    z->im = - z->im;
}
```

Объединения

Объединение (union) — это составной тип данных, объединяющий в себе несколько элементов других типов. При этом каждый элемент объединения также имеет имя и может быть модифицирован или прочитан по этому имени. Размер объединения равен максимальному из размеров типов входящих в него элементов. Элементы объединения разделяют один и тот же участок памяти, а точнее первый байт элементов имеет один и тот же адрес.

```
union variant {
    double double_value;
    int    integer_value;
    long   long_value;
    char  *string;
};
```

Обращение со объединениями происходит точно таким же образом, что и со структурами.

Использовать конструкцию union удобно в тех случаях, когда некоторая функциональность должна быть реализована для нескольких возможных известных входных данных, но слабо от них зависит. Либо в качестве обозначения неопределенного типа данных, возвращаемых функцией, конкретный тип которых определяется в процессе реализации.

Если бы не было конструкции union, ввиду строгости типизации языка Си, для каждого используемого типа пришлось бы писать свою реализацию функции.[1]

Например, существует некоторый алгоритм, который можно применить как для целых чисел, так и для чисел с плавающей точкой. Для простоты возьмём удвоение числа:

```
struct number {
    int type;
```

[1]Способность одной и той же функции быть применимой к различным типам называют также *полиморфизмом*. В современных объектно-ориентированных языках поддержка полиморфизма более развита и связана с более сложным способом создания новых типов – *наследованием*. Наследование можно организовать и в языке Си, но описание данного способа выходит за рамки данной книги.

```
    union {
        double real;
        int integer;
    } value;
}
struct number times2(struct number n) {
    struct number result;
    result.type = n.type;
    if ( n.type == 1 ) {
        result.value.integer = 2 * n.value.integer;
    } else {
        result.value.real = 2 * n.value.real;
    }
    return result;
}
```

К сожалению, компилятор не обладает возможностью определить, какого именно типа было записано значение в union, поэтому информацию о типе приходится передавать отдельно. Так мы и поступили в примере с организацией структуры number, в которую добавили поле `type` и сам `union`[2].

Оператор `typedef`

Оператор `typedef` используется для задания коротких имён типов. Строка

```
typedef struct complex complex_t;
```

позволяет вам везде далее вместо `struct complex` использовать идентификатор `complex_t`. Общее правило применения оператора `typedef` следующее: сначала пишется ключевое слово `typedef`, затем следует определение типа или его имя (например, `struct complex` или `union number`), а затем — новое имя типа. Последующее использование имени `ccomplex_t` будет распознано компилятором как `struct complex`.

Ниже приведены примеры использования оператора `typedef`:

```
typedef unsigned int UINT;
UINT x = 10;

typedef char[32] String;

String str;
strcpy(str, "my string");

typedef struct {
    double re;
    double im;
```

[2]Заметьте, что мы не написали имени после слова `union`, создав таким образом безымянное объединение в структуре `number`. Поскольку данное объединение будет использовано только внутри структуры, нет смысла его именовать отдельно.

```
} complex;

complex z = {20.0, 0.1};

complex j;

j.re = 0;
j.im = 1;
```

Библиотеки функций

Разбиение программы на модули

Одним из принципов структурного программирования является разбиение задачи на подзадачи в соответствии со структурой решения. Точно также доказательства теорем разбиваются на более мелкие теоремы и леммы. Программа — это задача, которая разбивается на функции — подзадачи. Поскольку уже накоплен достаточный опыт построения программ, многие сходные подзадачи часто встречаются в различных программах. Такие подзадачи принято выделять в **библиотеки** функций, тем самым упрощается жизнь программистов, которым не требуется повторно решать те задачи, которые были уже когда-то решены. Достаточно использовать нужную библиотеку. Так, математикам тоже нет необходимости повторно доказывать известные теоремы (разве только что в учебных целях).

Внешние функции и переменные. Ключевое слово `extern`

Каждый адресуемый элемент, будь то переменная или функция имеет область видимости. По умолчанию, все глобальные переменные и функции имеют область видимости уровня *модуля*. Т.е. вне модуля имена функций и переменных, объявленных в нём, теряют всякий смысл. Чтобы реализации функций были доступны другим модулям, их объявления помечают ключевым словом `extern`. Например, упоминание

```
extern int myfunc(int a);
```

в заголовочном файле даёт компоновщику понять, что символ `myfunc` является глобальным, то есть его область видимости простирается за пределы модуля, в котором он определен. Это даёт возможность использовать функцию `myfunc` в других модулях.

Статические и динамические библиотеки

Библиотеки бывают двух типов: **статические** и **динамические библиотеки**. Они устроены сходным образом, но различаются способом использования.

Реализации функций из статической библиотеки подключаются к основному, импортирующему модулю на стадии компоновки. В результате, конечный исполняемый файл уже содержит внутри себя реализации необходимых функций из библиотеки. При этом, в конечный файл попадают только те функции библиотеки, которые в нём реально используются.

При компоновке исполняемого модуля с динамической библиотекой, включения реализаций библиотечных функций в исполняемый код не происходит. В месте вызова библиотечных

функций у результирующего исполняемого файла стоят специальные «заглушки», которые заменяются на вызовы реальных функций в момент запуска исполняемого файла.

Если компоновка происходит со статической библиотекой, то код реализаций используемых функций библиотеки просто включается в исполняемый модуль конечной программы.

Таки образом, динамическая библиотека должна быть подгружена в память перед запуском программы, которая её использует, поскольку код функций находится именно в ней. В то время как статическая библиотека отдала код реализаций своих функций программе и не требует предварительной загрузки.

Функции статических библиотек включаются в программный код на стадии компоновки.
Функции динамических библиотек загружаются в память перед исполнением программы или во время исполнения программы.

Пример создания библиотеки

Приведём пример создания простой библиотеки функций и её использования в программе. Для того, чтобы создать библиотеку требуется всего лишь две вещи:

- заголовочный файл с декларациями функций библиотеки;
- файлы с реализациями функций библиотеки.

Допустим, у нас существует 2 файла – `mylib_module1.c` и `mylib_module2.c` – с реализациями функций библиотеки и файл `mylib.h` с декларациями этих функций. Естественно, файлы с реализациями должны включать (при помощи директивы препроцессора `#include`) файл с декларациями. Сначала файлы с реализациями функций проходят процесс компиляции. При этом образуются файлы `mylib_module1.o` и `mylib_module2.o` с объектным кодом. В этих файлах присутствуют реализации функций и таблица соответствия адресов и имён переменных и функций:

```
> gcc -c -o mylib_module1.o mylib_module1.c
> gcc -c -o mylib_module2.o mylib_module2.c
```

Опция `-c` запускает компилятор. Опция `-o` — заставляет компилятор положить результат в указанной за ней файл.

Затем, получившиеся объектные файлы компонуются в файл динамической библилотеки `libmylib.so`:

```
> gcc -shared -o libmylib.so mylib_module1.o mylib_module2.o
```

Имена файлов динамических библиотек в Unix-системах обычно начинаются с `lib` и имеют расширение `.so`.

Чтобы получить статическую библиотеку, нужно просто собрать все объектные файлы в один при помощи архиватора `ar`. Файлы статических библиотек используют расширение `.a`:

```
> ar -qs libmylib.a mylib_module1.o mylib_module2.o
```

Использование библиотеки начинается с включения её заголовочного файла в файл программы (назовём её `myprog.c`). Файл программы может быть скомпилирован отдельно, вместо вызовов библиотечных файлов будут стоять «заглушки», а компилятор проверит правильность вызовов на основании заголовочного файла:

```
> gcc -o myprog.o myprog.c
```

Затем настаёт очередь компоновщика, которому мы должны указать, где будет искаться файл библиотеки и как она называется:

```
> gcc -o myprog myprog.o -L./ -lmylib
```

В опции `-L` мы указываем путь, где необходимо искать нужную нам библиотеку. В данном случае, это текущая директория[3]. В опции `-l` мы указываем имя библиотеки, с которой нужно скомпоновать нашу программу[4]. При этом, произойдёт статическая компоновка со статической библиотекой (должен существовать файл `libmylib.a`). Чтобы скомпоновать программу с динамической библиотекой `libmylib.so`, нужно добавить опцию `-shared`:

```
> gcc -shared -o myprog myprog.o -L./ -lmylib
```

[3]По умолчанию в Unix, библиотеки ищутся компоновщиком в стандартных директориях `/lib`, `/usr/lib`, `/usr/local/lib` и текущая директория в этот список не входит, поэтому её приходится задавать явно

[4]Заметьте, что мы после `-l` ввели имя библиотеки, отбросив префикс `lib` — компоновщик подставит его сам при поиске.

Семинар 8

Реализация структур данных «стек» и «очередь»

Краткое описание: На этом семинаре мы рассмотрим абстрактные структуры данных «стек» и «очередь» и получим их реализации в виде набора функций языка Си.
С помощью структуры данных «стек» мы решим задачу распознавания правильных скобочных выражений, в которых присутствуют несколько типов скобок.
Также, на примере функций работы со стеком мы научимся создавать и использовать статические и динамические библиотеки функций.

Структура данных «стек»

Стек — это абстрактная структура данных для хранения однотипных элементов, поддерживающая две команды: `push(a)` и `pop()`:

`push(a)`: поместить элемент `a` в стек;

`pop()`: извлечь элемент элемент из стека.

Стек можно представлять как стопку (см. рис. 8.1) — класть можно только на верх стопки, и брать тоже только сверху. Тот элемент, который положен в стопку последним, будет первым извлечен из стека командой `pop()`.

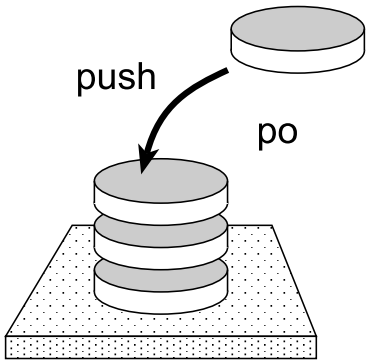


Рис. 8.1: Стек.

Рассмотрим случаи, когда элементы, хранящиеся в стеке есть целые числа типа `int`. Проще всего стек реализовать с помощью глобального массива (`int data[N]`) и переменной (`int last`), которая хранит индекс первой незаполненной ячейки массива:

Программа 8.1: Простейшая реализация стека.

```
#define N 1000
int data[N];
int last = 0;
void push(int a) {
    data[last++] = a;
}
int pop() {
    return data[--last];
}
```

В строчке «`data[last++] = a;`» в квадратных скобках используется операция «`last++`» суффиксного инкремента переменной `last`. Эта операция возвращает значение переменной `last` и *после этого* увеличивает значение переменной на 1. Операция префиксного декремента «`--last`» сначала уменьшает значение переменной на 1, а потом возвращает как результат получившееся значение переменной.

Приведённый код очень прост, но имеет следующие недостатки:

- В этой программе нет «защиты от дурака», а именно, функцию `pop()` можно вызвать и в случае, когда стек пустой. Переменная `last`, используемая как индекс, станет отрицательной и, по сути, программа начнёт обращаться к «чужой» памяти или памяти, где хранится её собственный код или другие данные. Запись чего-либо в эту память с помощью последующих вызовов функции `push` приведёт к непредсказуемым результатам. Также под элементы стека изначально выделен массив из 100 ячеек, но проверки на то, что добавляемый элемент не попадёт за границы этого массива, нет.
- Размер стека ограничен числом `N`. Какое бы значение для `N` мы не выбрали, может случится так, что в стек будет помещено более `N` элементов. Можно сделать проверку на число элементов в стеке и не позволять добавлять более `N` элементов. Но это плохой стиль — не следует ограничивать пользователя каким-либо размером. Правильно — по мере необходимости, выделять памяти под стек столько, сколько нужно пользователю. Для решения этой проблемы необходимо динамически выделять память с помощью функции `malloc`, объявленной в файле `malloc.h`, а при заполнении выделенной памяти делать запрос на перевыделение памяти большего размера с помощью функции `realloc`.
- В такой реализации возможно пользоваться только одним единственным стеком, поскольку массив элементов стека объявлен глобально. Для устранения этого недостатка необходимо *инкапсулировать* переменные `data` и `last` в структуру данных, и добавить функции `new_stack()` и `delete_stack()` для создания и удаления экземпляров структуры данных «стек»;

Устранение указанных недостатков приводит к коду 8.2 большего размера.

Программа 8.2: Реализация стека (вариант 2).

```
#include <malloc.h>
/* Структура с указателем на массив int и служебной информацией.
 */
typedef struct stack {
    int *data;
    int last;
    int data_size;
} stack_t;

/* Конструктор стека.
 * Принимает начальный размер стека
 * и возвращает указатель на новый стек
 */
stack_t* new_stack(int initial_size) {
    stack_t *stack = (stack_t*) malloc(sizeof(stack_t));
    stack->data_size = initial_size;
```

```
    if ( stack->data_size <= 0 ) stack->data_size = 100;
    stack->last = 0;
    stack->data = (int*)malloc(sizeof(int)*stack->data_size);
    return stack;
}

/* Деструктор стека.
 * Принимает на вход указатель на стек
 * и освобождает занятую им память.
 */
void delete_stack(stack_t *stack) {
    free(stack->data)
    free(stack);
}

/* Операция push.
 * Принимает указатель на стек и добавляемый элемент.
 * При необходимости увеличивает количество памяти для
 * массива элементов.
 */
void push(stack_t *stack, int a) {
    if ( stack->last == stack->data_size ) {
        stack->data_size = (stack->data_size * 3 + 1) / 2;
        stack->data = (int*)realloc(stack->data,
        stack->data_size);
    }
    stack->data[stack->last++] = a;
}

/* Операция pop. Принимает указатель на стек и адрес значения
 * верхнего элемента.  Возвращает 1 в случае, если стек был
 * непуст и 0 -- в противном случае.
 */
int pop(stack_t *stack, int *a) {
    if ( stack->last > 0 ) {
        *a = stack->data[--stack->last];
        return 1;
    } else {
        fprintf(stderr,
            "Попытка извлечь элемент из пустого стека!\n");
        return 0;
    }
}
```

Для увеличения надёжности этого кода, полезно в каждой функции проверять, что `stack` и `stack->data` не равны нулю, и если один из них равен нулю, выдавать предупреждение об ошибке и заканчивать выполнение программы. После каждого вызова функций `malloc` и

realloc принято писать проверку на то, что их результат не равен 0. Если результат равен нулю, значит в системе нет памяти и ваш запрос на динамическое выделение памяти не может быть выполнен. В таких случаях нужно выдавать предупреждение "Нет памяти" в поток вывода ошибок stderr и заканчивать работу программы с помощью команды exit(1):

```
    fprintf(stderr, "Нет памяти\n");
    exit(1);
```

Реализация «стека» с помощью односвязного списка

Односвязный список — это способ представления упорядоченного множества однотипных элементов, в котором каждому элементу ставится в соответствие следующий элемент. Сделать это можно следующим образом.

Допустим, что элементами списка являются целые числа, тогда, для элементов организуется структура (мы назовем её list, а соответствующий тип — list_t):

```
typedef struct list {
    int value;
    struct list *next;
} list_t;
```

Таким образом, мы сделали структуру, содержащую наш элемент (поле value) и указатель на такого же типа структуру (поле next), содержащее следующий элемент списка[1].

Далее, мы можем преобразовать наши функции работы со стеком таким образом, чтобы они использовали односвязный список, вместо динамического массива:

Программа 8.3: Реализация стека с помощью списка.

```
#include <malloc.h>
/* Структура с указателем на массив int
 * и служебной информацией
 */
typedef struct stack {
    list_t *head;
} stack_t;
/* Конструктор стека.
 * возвращает указатель на новый стек
 */
stack_t* new_stack() {
    stack_t *stack = (stack_t*) malloc (sizeof(stack_t));
    stack->head = NULL;
    return stack;
}
/* Деструктор стека.
 * Принимает на вход указатель на стек
```

[1]Стоит заметить, что мы не можем использовать новое имя типа list_t для обозначения поля next, а пользуемся длинной конструкцией struct list, поскольку этот тип еще не определен.

```
 * и освобождает занятую им память.
 */
void delete_stack(stack_t *stack) {
    list_t *next, *elem = stack->head;
    while ( elem ) {
        next = elem->next;
        free(elem);
        elem = next;
    }
    free(stack);
}
/* Операция push. Принимает указатель на стек и добавляемый
 * элемент. При необходимости увеличивает количество памяти
 * для  массива элементов.
 */
void push(stack_t *stack, int a) {
    list_t *elem = (list_t *)malloc(sizeof(list_t));
    elem->value = a;
    elem->next = stack->head;
    stack->head = elem;
}
/* Операция pop. Принимает указатель на стек и указатель на
 * переменную типа int,  в которую нужно поместить результат.
 * Возвращает 1 в случае, если стек был непуст,
 * и 0 -- в противном случае.
 */
int pop(stack_t *stack, int *a) {
    list_t *elem;
    if ( stack && stack->head ) {
        *a = stack->head->value;
        elem = stack->head->next;
        free(stack->head) ;
        stack->head = elem;
        return 1;
    } else {
        fprintf(stderr,
            "Попытка извлечь элемент из пустого стека!\n");
        return 0;
    }
}
```

Заметьте, что в данной реализации не вызывается функция `realloc`, а также не требуется вводить начальное количество элементов. Эта реализация является наиболее гибкой с точки зрения пользователя и наиболее экономной — памяти выделяется всегда столько, сколько нужно. Однако, использование вызова `malloc` на каждый добавляемый элемент делает такую реализацию неприемлемой для программ, которые часто используют стек (и динамическое выделение памяти вообще), поскольку частые вызовы `malloc/free` *фрагментируют память*,

и, с каждым разом, вызов `malloc` становится всё более длительным.

Структура данных «очередь»

Очередь — это абстрактная структура данных для хранения однотипных элементов, поддерживающая две команды: `push(a)` и `pop()`:

`enqueue(a)`: поместить элемент `a` в очередь;

`dequeue()`: извлечь элемент элемент из очереди.

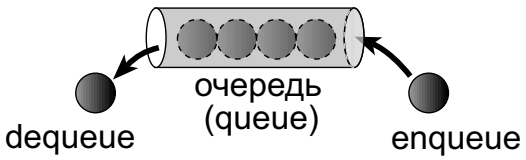


Рис. 8.2: Очередь.

Очередь можно представлять как трубу с двумя концами (см. рис. 8.2), внутри которой последовательно расположены элементы. Класть новый элемент можно только в правый конец трубы, а брать только из левого. Тот элемент, который положен в стопку первым, будет первым извлечен из очереди командой `dequeue()`. Очередь также называют *контейнером типа FIFO* (first in – first out).

Реализация «очереди» с помощью односвязного списка

Задача С8.1. Реализуйте очередь с помощью односвязного списка.

Задача С8.2. Рассмотрим абстрактную структуру данных, которая соответствует трубе и предоставляет возможность класть и брать для обоих концов трубы:

```
int add1(tube_t *t, int a);
int add2(tube_t *t, int a);
int extract1(tube_t *t, int *a);
int extract2(tube_t *t, int *a);
```

Предложите реализации этой структуры. Можно ли её реализовать с помощью односвязного списка? Какой тип списка удобнее использовать?

Распознавание правильных скобочных выражений с использованием стека

Задача С8.3. Разберите программу 8.4, которая распознает правильные скобочные выражения. Прочитайте документацию по функции `getchar()` (используйте команду «`man 3 getchar`»).

Программа 8.4: Распознавание правильных скобочных выражений

```
#include <stdio.h>
int main () {
    int c, a;
    printf ("Введите слово из скобок: ");
    do {
        c = getchar();
        if( c == '(' ) a++;
```

```
        else if( c == ')' ) if( --a < 0 ) break;
    } while( c != '\n' );
    if( a == 0 )
        printf ("Правильное выражение\n");
    else
        printf ("Не правильное выражение\n");
    return 0;
}
```

Задача С8.4. (сложные скобочные выражения) Напишите программу, проверяющую правильность скобочного выражения, состоящей из трёх типов скобок: `( ) { } [ ]`. Используйте стек, в который будут помещаться приходящие открывающие скобки. Если очередной символ `c` — закрывающаяся скобка, то выполняйте команду «`a=pop()`» и проверяйте совпадение типов скобок, находящихся в переменных `c` и `a`.

Задача С8.5. Напишите программу, которая перебирает все скобочные выражения с одним типом скобок длины $2n$, проверяет их на правильность и выводит только правильные скобочные структуры.

Задача про «хорошие» слова

Задача С8.6. В задаче С4.4 (стр. 84) было описано множество «хороших» слов.
а) Напишите программу на языке Си, которая получает на вход слово в алфавите $B = \{0, 1, 2, 3\}$ и определяет, «хорошее» оно или нет. Подсказка: напишите рекурсивную функцию `read_good_word`:

```
char* read_good_word (char *word);
```

Она получает указатель типа `char *` и пробует прочитать «хорошее» слово начиная с указанного символа. Функция возвращает указатель на место того символа, до которого она «дочитала». Если во время чтения встречается ошибка (присутствует символ, не принадлежащий алфавиту), функция должна вернуть указатель равный данному в аргументе (не смогла прочитать ни одного символа).
б) Решите эту же задачу однопроходным алгоритмом, не используя массивов. Для чтения символов из потока ввода пользуйтесь функцией `getchar`. Напишите функцию

```
int read_good_word ();
```

которая считывает из потока ввода только хорошее слово, а лишние символы, которые идут за хорошим словом оставляет в потоке. Если из потока удалось прочитать хорошее слова, функция возвращает 1, а иначе — 0.

Калькулятор арифметических выражений в обратной польской нотации

Обратная польская запись устроена следующим образом: сначала перечисляются аргументы функции, а затем пишется имя функции. В частности, для операции сложения двух чисел

«a+b» (функции от двух аргументов) получаем следующую запись: «`a b +`», а для сложного выражения «`a * b + c`» получаем `a b * c +`. Вот примеры арифметических выражений и соответствующих им записей в обратной польской нотации:

`2 * 3 + 1`	`2 3 * 1 +`
`1 + (2 + (3 + 4))`	`1 2 3 4 + + +`
`2 * 3 + 4 / 5 * 6`	`2 3 * 4 5 / 6 * +`
`a + (b * c)`	`a b c * +`

Обратная польская запись называется также *постфиксной*.

Задача С8.7. (Калькулятор выражений в постфиксной нотации) Разберите программу 8.5, которая вычисляет арифметические выражения в обратной польской нотации согласно алгоритму 4.12, описанному на стр. 80. Умеет ли она работать с отрицательными числами? Добавьте к ней возможность вычисления операции степени, которую обозначьте символом «^» (запись «`x y ^`» соответствует x^y). Посмотрите помощь по функции `pow`, выполнив команду «`man 3 pow`». Что произойдёт, если на вход программы 8.5 подать строчку из 10 символов «+»? Модифицируйте её так, чтобы при некорректно заданном выражении она не «падала», а выдавала сообщение об ошибке во входном выражении.

Программа 8.5: Калькулятор выражений в постфиксной нотации

```
#include <stdio.h>
#include <stack.h>
double stack[100];
void push(double a) {
    stack[pt++] = a;
}
double pop() {
    return stack[--pt];
}
int main() {
   int i;
   while( !feof( stdin ) ) {
      int c = getchar();
      int x;
      switch ( c ) {
         case EOF: break;
         case ' ' : break;
         case '\n': break;
         case '=' : printf("Result = %d\n", pop()); break;
         case '+' : push(  pop() + pop() ); break;
         case '-' : push( -pop() + pop() ); break;
         case '*' : push(  pop() * pop() ); break;
         default:
            ungetc( c, stdin );
            if( scanf( "%lf", &x ) != 1 ) {
               fprintf( stderr, "Can't read number\n" );
               return 1;
            } else {
```

```
                push( x );
            }
            break;
        }
    }
    return 0;
}
```

Задача С8.8. (Перевод из постфиксной в инфиксную нотацию) Напишите программу, которая получает на вход выражение в префиксной нотации, а возвращает то же самое выражение в инфиксной нотации. При переводе в инфиксную нотацию потребуется добавлять скобки. Реализуйте такой алгоритм, время работы которого растёт линейно с длинной входного выражения.

Подсказка: Алгоритм конструирования инфиксного выражения из постфискного подобен алгоритму вычисления постфиксного выражения. Следует использовать стек, но хранить в нём не числа, а строки — инфиксные выражения, соответствующие прочитанным кусочкам префиксного выражения. Например, при считывании знака `'+'` из стека извлекаются инфиксные выражения $a =$ `pop()` и $b =$ `pop()`, осуществляется слияние скобки `'('`, выражения b, знака `+`, выражения a, скобки `')'`, и результат слияния помещается в стек командой `push`. Рассмотрите при этом два варианта:

- инфиксные выражения в стеке хранятся как строки и имеют тип `char*`; слияние строк происходит с помощью функции `strcat`;
- инфиксные выражения хранятся как списки «атомов» (знаков операторов и чисел), в время вычислениях вызывается операция слияния списков; и лишь в конце вычисления конечный список «атомов» преобразуется в строку.

Какой из вариантов будет работать быстрее? Ответ поясните примерами. Для хранения атомов придумайте специальную структуру, содержащее поле `type`, которое может быть равно либо константе `ATOM_NUMBER`, либо константе `ATOM_BINARY_OP`, либо константе `ATOM_UNARY_OP`. Второй элемент структуры должен быть объединением (`union`).

Создание и использование библиотеки stack

Научимся создавать свои библиотеки функций.

Мы уже неоднократно пользовались библиотекой стандартных функций ввода-вывода `stdio.h` и математической библиотекой `math.h`. Попробуем создать свою библиотеку полезных функций.

Пусть это будет библиотека функций для работы со стеком. Сначала создаётся заголовочный файл `stack.h`, который будет содержать в себе объявления функций, используемых новых типов и глобальные переменные. Ключевое слово `extern` в декларации переменных и функций означает, что их имена будут видны в программах, подключающих данную библиотеку.

Программа 8.6: Заголовочный файл библиотеки stack.

```
// Файл stack.h
/* Наличие макроопределения _STACK_H_ будет отображать,
```

```
 * что заголовочный файл stack.h подключен.
 */
#ifndef _STACK_H_
#define _STACK_H_
/* Хорошим тоном является комментирование каждой функции,
 * переменной и типа, объявляемых в заголовочном файле.
 */

/* Тип данных стек. */
typedef struct stack stack_t;

/* Заметьте, что объявление структуры в заголовочном
 * файле давать не обязательно, если предполагается,
 * что пользователю библиотеки знание о внутреннем
 * представлении стека не потребуется и все операции
 * с новым типом будут производиться только функциями
 * данной библиотеки и только с использованием указателя
 * на него.
 */

/* Создать новый стек */
extern stack_t *new_stack();

/* Удалить стек */
extern void delete_stack(stack_t *stack);

/* Положить значение на вершину стека */
extern void push(stack_t *stack, int value);

/* Взять значение с вершины стека */
extern int pop(stack_t, int *value);

#endif /* _STACK_H_ */
```

Этот заголовочный файл необходимо включить в файл с реализацией данных функций stack.c:

Программа 8.7: Исходный код функций библиотеки stack.

```
// Файл stack.c
#include "stack.h"
stack_t *new_stack(int initial_size) {
...
}
void delete_stack(stack_t *stack) {
...
}
void push(stack_t *stack, int value) {
```

```
...
}
int  pop(stack_t *stack, int *value) {
...
}
```

Следует следить за тем, чтобы имена и сигнатуры (количество и тип аргументов и возвращаемого значения) функций при деклараций функций и их реализаций совпадали. Ключевое слово `extern` повторять не нужно.

Два файла `stack.h` и `stack.c` составляют исходный код нашей библиотеки.

Чтобы использовать нашу библиотеку в программе `usestack.h` достаточно включить файл `stack.h` при помощи директивы препроцессора `#include`:

Программа 8.8: Пример программы, использующей библиотеку stack.

```
// Файл usestack.c
#include "stack.h"
int main() {
    stack_t *s;
    int x;
    s = new_stack();
    push(s, 10); push(s, 20); push(s, 30);

    while ( pop (s, &x) ) {
        printf("pop=%d\n", x);
    }
    delete_stack(s);
    return 0;
}
```

Использование библиотеки в исходных кодах

Один из способов подключения библиотеки к проекту — это включение исходных кодов библиотеки в проект. Для компиляции файла `usestack.c` вместе с файлом библиотеки `stack.c` используется команда

```
> gcc usestack.c stack.c -o usestack
```

В результате получается запускаемый файл `usestack`, который можно использовать. Этот метод включения библиотеки имеет ряд недостатков. Во-первых, если библиотека большая, компиляция исходных файлов библиотеки может занять значительное время. Кроме того, при компиляции файлов библиотеки может возникнуть ошибка. И наконец, исходные файлы библиотеки могут быть интеллектуальной собственностью, и автор не откроет их для использования. Гораздо надёжнее и проще предоставлять библиотеку в виде заголовочного файла библиотеки и скомпилированного файла библиотеки.

Есть два типа библиотек — статически подключаемые и динамически подключаемые.

Создание и использование статически подключаемой библиотеки

Для создания статически подключаемой библиотеки следует указывать опцию `-c`:

```
> gcc -c -o libstack.a stack.c
```

Опция `-c` команды `gcc` означает «только скомпилировать, не создавая запускаемого файла». В результате получится файл библиотеки `libstack.a`. Статически подключаемая библиотека подключается на этапе компоновки запускаемого файла, использующего библиотеку. Для этого необходимо указать опцию `-lstack`:

```
> gcc usestack.c -lstack  -o usestack
```

В результате получится готовый к использованию запускаемый файл `usestack`.

Создание и использование динамически подключаемой библиотеки

Динамически подключаемую библиотеку следует компилировать с опцией `-shared`:

```
> gcc -shared -o libstack.so stack.c
```

В результате компиляции получится файл библиотеки `libstack.so`. Его нужно поместить в директорию, в которой хранятся подключаемые библиотеки.

А при компиляции программы, необходимо будет включить файл библиотеки при помощи опции компоновщика (линковщика) `-lstack`. Данная опция заставляет компоновщик искать библиотеку с именем `libstack.so` в местах стандартного расположения библиотек.

```
> gcc -o usestack -L. -lstack usestack.c
```

Чтобы программа `usestack` успешно выполнилась, необходимо, чтобы файл `libstack.so` находился в одной из стандартных директорий библиотек[2]. Кроме того, с помощью переменной среды `LD_LIBRARY_PATH` можно указать системе, где искать подключаемые библиотеки. Эта переменная представляет собой список директорий, разделённых символом двоеточия. С помощью команды

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.:/home/student/lib
```

к текущему значению переменной дописываются слева символы «:.:/home/student/lib», то есть к списку директорий добавляется ещё две директории — текущая директория, обозначаемая одной точкой, и директория `/home/student/lib`. Динамически подключаемые библиотеки подключаются во время запуска программ, их использующих.

[2]В операционной системе Linux это директории `/usr/lib` и `/usr/local/lib`.

Часть III

Алгоритмы и структуры данных

Лекция 9

Рекурсия и итерации

Тему «Алгоритмы и структуры данных» мы начнем с изучения алгоритмов решения простых задач — вычисления максимума введённых чисел и вычисления чисел Фибоначчи.

Однопроходные алгоритмы

Рассмотрим класс задач, в которых входом является последовательность некоторых объектов, потенциально сколь угодно большого размера. В этот класс задач, попадают, например, следующие задачи:

- вычисление максимума (минимума) набора чисел;
- вычисление среднего арифметического набора чисел;
- упорядочивание по возрастанию чисел из данного набора;
- вычисление наибольшего общего делителя набора целых чисел;
- вычисление выпуклой оболочки набора точек на плоскости;
- вычисление медианы чисел (медиана набора чисел — это число, которое окажется в середине последовательности, если данный набор чисел упорядочить по возрастанию).

При конструировании алгоритмов для таких задач возникает естественный вопрос о том, как растёт среднее и максимальное время работы алгоритма от размера последовательности объектов, а также как растёт память, требуемая алгоритму. Для первых двух задач есть простые алгоритмы 9.13 и 9.14.

Алгоритм 9.13 Однопроходный алгоритм вычисления максимума чисел

```
1:  function MAX-STREAM(input stream)
2:      m ← READ;
3:      while not end of input do
4:          x ← READ
5:          if x > m then
6:              m ← x
7:          end if
8:      end while
9:      WRITE(m)
10: end function
```

Алгоритм 9.14 Однопроходный алгоритм вычисления среднего-арифметического чисел

```
1: function AVERAGE-STREAM(input stream)
2:     n ← 0;
3:     s ← 0;
4:     a ← 0;
5:     while not end of input do
6:         x ← READ
7:         n ← n + 1
8:         s ← s + x
9:         a ← s/n
10:    end while
11:    WRITE(s)
12: end function
```

ОПРЕДЕЛЕНИЕ 9.1. *Алгоритм называется однопроходным, если каждый элемент входной последовательности считывается алгоритмом один раз и размер требуемой алгоритму памяти чётко зафиксирован, то есть не растёт с ростом размера последовательности.*

Видно, что представленные алгоритмы MAX и AVERAGE являются однопроходными. Для первого алгоритма требуется память только под переменную m — текущее значение максимума. Во-втором алгоритме память требуется под три переменные n (количество считанных чисел), s (сумма считанных чисел), a (текущее значение среднего-арифметического считанных чисел).

Задача Л9.1. Существует ли однопроходный алгоритм, который вычисляет 10-е по величине число данного набора чисел? Если существует, опишете его.

В данных алгоритмах нет необходимости хранить в памяти все считанные числа, достаточно хранить в памяти набор величин S, по которому можно вычислить результат r_n на n-м шаге работы алгоритма:

$$r_n = \text{RESULT}(S_n)$$

Кроме того, в задачах, решаемых однопроходными алгоритмами, есть возможность получить новое значение величин S_n по старому значению S_{n-1} и новому считанному элементу последовательности x_n:

$$S_n = \text{UPDATE-STATE}(S_{n-1}, x_n).$$

Задача Л9.2. Пусть дана функция $\text{GCD}(a, b)$ (greatest common divisor), которая вычисляет наибольший общий делитель двух целых чисел. Напишите на псевдокоде однопроходный алгоритм GCD-STREAM, который находит наибольший общий делитель произвольного количества вводимых целых чисел.

Однопроходные алгоритмы — очень важный объект в программировании. По сути, это такие алгоритмы, которые могут быть реализованны в виде конечного автомата. Они работают быстро и не требуют много памяти. Особенно популярны однопроходные алгоритмы сжатия данных, которые можно представлять как черный ящик, в который входит поток байт и вы-

ходит поток байт[1].

Различные преобразователи сигналов (данных) из одного формата в другой также обычно являются однопроходными алгоритмами. Программисты считают большой удачей, когда какую-либо задачу обработки потока данных (кодирование, преобразование в другой формат, проверку корректности данных) удаётся решить с помошью однопроходного алгоритма или явно записать в виде конечного автомата.

Обратимся к перечисленным вначале задачам. Не все из них, могут быть решены однопроходным алгоритмом. В частности, однопроходным алгоритмом не могут быть решены задачи упорядочивания (сортировки) чисел по возрастанию, вычисления выпуклой оболочки точек на плоскости и задача вычисления медианы

Алгоритмы вычисления a^n

На примере задачи вычисления $f(a, n) = a^n$ рассмотрим два принципиально различных метода конструирования алгоритмов — итерации и рекурсия.

Первый алгоритм Power-Simple-Iterations$(a,\ n)$ (псевдокод 9.15) состоит из одного арифметического цикла, тело которого выполняется n раз. В теле этого цикла переменная r, изначально равная 1, каждую итерацию домножается на a.

Алгоритм 9.15 Итеративный алгоритм вычисления a^n

```
1: function Power-Simple-Iterations(a, n)
2:     r ← 1;
3:     for i ∈ {1, ..., n} do
4:         r ← r · a
5:     end for
6:     return r
7: end function
```

Число итераций цикла равно n, в теле цикла выполняется элементарное действие, и поэтому время работы этого алгоритма растет пропорционально n.

Второй простой способ вычисления a^n — это способ, основанный на рекурсии. Рассмотрим рекуррентное соотношение

$$f(a, 0) = 1, \quad f(a, n) = a \cdot f(a, n - 1).$$

Оно задает функцию $f(a, n) = a^n$. На основе этого соотношения можно написать рекурсивный алгоритм 9.16.

В строчке 5 происходит рекурсивный вызов функции Power-Simple-Recursion с параметрами $(a,\ n - 1)$. Это приводит к тому, что запускается процесс вычисления функции Power-Simple-Recursion (обычно говорят «осуществляется вызов функции») с меньшим значением второго аргумента, при этом можно считать, что исполнитель текущего алгоритма «ожидает» окончания вычисления Power-Simple-Recursion$(a,\ n - 1)$ и лишь потом

[1]В случае наличия сжатия, число байт на выходе должно быть меньше, чем число байт на входе. Необходимо также наличие обратного преобразователя, который из сжатого потока байт восстанавливает исходный поток байт.

Алгоритм 9.16 Рекурсивный алгоритм вычисления a^n

```
1: function Power-Simple-Recursion(a, n)
2:     if n == 0 then
3:         return 1
4:     end if
5:     return a · Power-Simple-Recursion(a, n − 1)
6: end function
```

продолжит свою работу. Значение аргумента n для исполнителя, вычисляющего функцию Power-Simple-Recursion$(a,\ n-1)$ будет на 1 меньше, нежели у исполнителя, его породившего.

Если $n > 1$, то при вычислении Power-Simple-Recursion$(a,\ n-1)$ произойдёт то же самое — в строке 5 снова будет осуществлён вызов функции, при этом значение второго аргумента будет еще на 1 меньше. Рекурсивные вызовы будут осуществляться каждый раз, когда второй аргумент больше 0. Таким образом, при вычислении a^{100} будет осуществлено ровно 100 рекурсивных вызовов.

Рекурсивный вызов — это элементарная операция. Если считать, что операция умножения также элементарна, то время работы рекурсивного алгоритма пропорционально n.

Существуют ли более оптимальные алгоритмы, чем два представленных? Да, существуют. В частности, если n является степенью двойки, то a^n можно получить последовательно возводя в квадрат зачение переменной r, изначально равной a:

$$a \to a^2 \to a^4 \to a^8 \to a^{16} \to a^{32} \to$$

В частности, по этой цепочке всего за 10 шагов можно получить a^{1024}. Эту идею можно воплотить в более оптимальный рекурсивный алгоритм 9.17 для произвольного значения n (не только для степеней двойки). Алгоритм основан на рекуррентном соотношении

$$f(a,n) = \begin{cases} a \cdot f(a, n-1), & \text{при нечётном } n, \\ f(a, n/2) \cdot f(a, n/2), & \text{при чётном } n. \end{cases}$$

Алгоритм 9.17 Рекурсивный алгоритм вычисления a^n (способ 2)

```
1: function Power-Complex-Recursion(a, n)
2:     if n == 0 then
3:         return 1
4:     end if
5:     if n делится на 2 нацело then
6:         return Power-Complex-Recursion(a · a, n/2)
7:     else
8:         return a · Power-Complex-Recursion(a, n − 1)
9:     end if
10: end function
```

Задача Л9.3. Опишите цепочку рекурсивных вызовов при вычислении Power-Complex-Recursion$(a, 100)$. Запишите все передаваемые в этой цепочке значения второго параметра в двоичной системе счисления.

Ответ: $a^{100} \to a^{50} \to a^{25} \to a^{24} \to a^{12} \to a^6 \to a^3 \to a^2 \to a^1 \to a^0$.

n	двоичная запись	представление в виде суммы степеней двойки n
100	1100100	64+32+4
50	110010	32+16+2
25	11001	16+8+1
24	11000	16+8
12	1100	8+4
6	110	4+2
3	11	2+1
2	10	2
1	1	1

Задача Л9.4. Докажите, что число вызовов функции Power-Complex-Recursion при вычислении a^n не превосходит удвоенного числа знаков в двоичной записи числа n.

Алгоритм 9.18 Итеративный алгоритм вычисления a^n (способ 2)

```
1: function Power-Complex-Iterations(a, n)
2:     r ← 1;
3:     b ← a;
4:     while n > 0 do
5:         if n mod 2 == 1 then
6:             r ← r · b
7:         end if
8:         b ← b · b
9:         n ← n div 2
10:    end while
11:    return r
12: end function
```

Число знаков в q-ричной записи числа n равно числу $\log_q n$, округлённому до ближайшего целого числа в большую сторону. Поэтому число вызовов можно ограничить сверху числом $2\log_2 n + 1$. Таким образом, время работы алгоритма Power-Simple-Recursion растёт с n примерно логарифмически. Это записывают так: «время работы алгоритма равно $O(\log_2 n)$». Подробнее смысл выражения $O(f(n))$ мы обсудим в следующей лекции.

А сейчас рассмотрим еще один итеративный алгоритм, в котором по сути, также используется идея разложения числа n на сумму степеней двойки.

Построим таблицу, указывающую значения переменных на каждой итерации перед выполнением строчки 8:

номер итерации	n	b	r
1	100	a	1
2	50	a^2	1
3	25	a^4	a^4
4	12	a^8	a^4
5	6	a^{16}	a^4
6	3	a^{32}	a^{4+32}
7	1	a^{64}	$a^{4+32+64}$

Алгоритмы вычисления чисел Фибоначчи

В математике для описания функций часто используются рекуррентные соотношения, в которых значение функции определяется через её значение при других (обычно меньших) аргументах. Наиболее известным примером является последовательность Фибоначчи 1, 1, 2, 3, 5, 8, 13, ..., определяемая следующими соотношениями

$$F_0 = 1, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

Используя это рекуррентное соотношение, можно построить рекурсивный алгоритм вычисления чисел Фибоначчи:

Алгоритм 9.19 Числа Фибоначчи

```
1: function FIBO(n)
2:     if n = 0 or n = 1 then
3:         return 1
4:     end if
5:     return FIBO(n − 1) + FIBO(n − 2)
6: end function
```

Наибольший интерес в этом алгоритме представляет строчка 5. Она означает следующее: «Запустить процесс вычисления FIBO$(n-1)$, затем запустить процесс вычисления FIBO$(n-2)$, результаты вычислений сложить и вернуть в качестве результата». Можно считать что в этой строчке исполнитель нашего алгоритма просит нового исполнителя вычислить FIBO$(n-1)$, а сам ждёт, когда тот закончит вычисления. Узнает у него результат, просит вычислить FIBO$(n-2)$ и снова ждёт результата. Два полученных результата складывает, возвращает значение суммы как результат и заканчивает работу. дополнительный исполнитель. Новый исполнитель действует по такому же алгоритму — если его аргумент больше 1, то он вызывает очередного исполнителя для вычисления нужных ему значений. Получается серия вызовов функции FIBO, которые представляют собой **дерево рекурсивных вызовов**.

На рисунке 9.1 представлено **дерево рекурсивных вызовов**, возникающее при вычислении F_6. Это дерево демонстрирует как функция сама себя использует при вычислении. Например, при вычислении F_6 были вызваны функции вычисления F_5 и F_4. Для вычисления F_5 понадобились F_4 и F_3, и так далее.

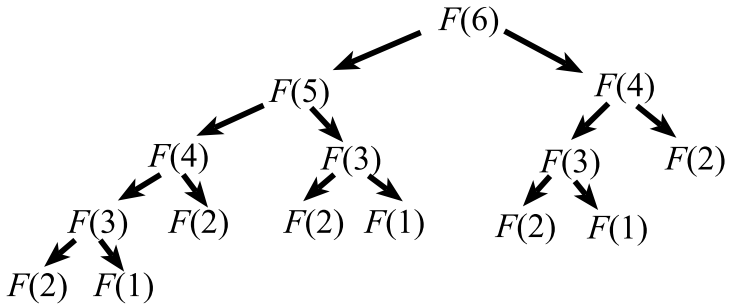


Рис. 9.1: Дерево рекурсивных вызовов для F_6.

Для того, чтобы рекурсивный алгоритм заканчивал свою работу, необходимо, чтобы дерево рекурсивных вызовов при любых входных данных обрывалось и было конечным. В данном примере дерево рекурсивных вызовов обрывается на F_0 и F_1, для вычисления которых не используются рекурсивные вызовы.

Довольно часто «зависание» компьютеров связано с использованием плохо реализизованных рекурсивных идей. Наш пример тоже плох. Попробуйте подставить отрицательный аргумент n и убедитесь, что приведённый алгоритм «зависает».

Задача Л9.5. Сколько раз вызывались вычисления F_0 и F_1 при вычислении F_6? Нарисуйте дерево рекурсивных вызовов для F_7 и определите, сколько раз будут вызваны F_0 и F_1. Найдите общую формулу для числа вызовов F_0 и F_1 при вычислении F_n.

Пользоваться рекурсивными алгоритмами нужно осторожно, так как они могут быть неэффективными с точки зрения времени работы.

Попробуем оценить количество операций, требуемое рекурсивному алгоритму, для того, чтобы вычислить n-й член последовательности Фибоначчи (здесь под операцией мы понимаем строчку в программе, подробнее о времени работы алгоритмов и сложности вычислительных задач мы поговорим при изучении сложности задачи сортировки). Обозначим это число как $T(n)$.

Для вычисления $\textsc{Fibo}(n)$ нам потребуется вызвать $\textsc{Fibo}(n-1)$ и $\textsc{Fibo}(n-2)$, поэтому $T(n) \geqslant T(n-1) + T(n-2)$. Используя это соотношение и то, что $T(1) = T(2) > 1$ можно по индукции доказать, что $T(n) \geqslant F_n$. Но числа Фибоначчи возрастают достаточно быстро ($F_{50} = 20365011074$).

Давайте посчитаем, сколько операций сложения (строка 5) будет осуществлено при вычислении числа Фибоначчи F_{50}.

При вычислении F_0 и F_1 строка 5 не выполняется и число операций сложения равно 0. При вычислении F_2 осуществляется одно сложение. При вычислении F_2 осуществляется два сложения. Обозначим число операций сложения при вычислении F_n как G_n. Тогда $G_0 = G_1 = 0$, $G_2 = 1$, $G_3 = 2$, ... Несложно увидеть, что выполняется следующее рекуррентное соотношение:

$$G_n = G_{n-1} + G_{n-2} + 1.$$

Последовательно вычисляя первые 50 чисел, получим $G_{50} = 20365011073$. Эта последовательность растёт достаточно быстро, в частности, $G_{100} \approx 5.7 \cdot 10^{20}$. Можно показать, что G_n растёт примерно как a^n, где $a = (\sqrt{5}+1)/2 \approx 1.61803$.

Время, необходимое описанному рекурсивному алгоритму, примерно пропорционально вычисляемому числу Фибоначчи. Это следует из того, что число Фибоначчи F_n в рекурсивном алгоритме получается в результате сложения F_n единиц. Необходимое число операций сложения G_n ровно на 1 меньше самого числа F_n.

Поэтому даже на мощном компьютере с помощью этого алгоритма нам не удастся вычислить больше, чем первые несколько десятков членов последовательности Фибоначчи. Вы, наверное, уже догадываетесь, в чём здесь проблема. В нашей программе очень много избыточных вызовов — в дереве рекурсивных вызовов много повторений. Например $\textsc{Fibo}(n-2)$ будет вызвана два раза: сначала из $\textsc{Fibo}(n)$, а потом из $\textsc{Fibo}(n-1)$, и оба раза будут проводится одни и те же вычисления. Простой «человеческий» алгоритм вычисления чисел Фибоначчи работает существенно быстрее: нужно помнить последние два числа Фибоначчи, вычислять следующее число и повторять этот шаг нужное число раз. Приведём его описание на псевдокоде:

Алгоритм 9.20 Числа Фибоначчи: нерекурсивный алгоритм

```
1:  function FiboNR(n)
2:      if n < 3 then
3:          return 1;
4:      else
5:          b ← 1                          ▷ Предпоследнее вычисленное число Фибоначчи
6:          a ← 1                              ▷ Последнее вычисленное число Фибоначчи
7:          for i ∈ 3 . . . n do
8:              c ← a + b
9:              b ← a
10:             a ← c
11:         end for
12:         return c
13:     end if
14: end function
```

Алгоритм 9.20 вычисления F_n выполнит $n-2$ итераций цикла **while**. Соответственно время работы этого алгоритма растёт линейно с n.

Как мы видим, данный рекурсивный алгоритм оказался существенно менее эффективным (дольше работающем при больших n) нежели нерекурсивный алгоритм.

Но это не значит, что использовать рекурсию не надо. От экспоненциального роста времени вычисления рекурсивных алгоритмов легко избавится с помощью запоминания вычисленных значений. А именно, в памяти хранят вычисленные значений, а в начале функции помещается проверка на то, что требуемое значение уже вычислено и хранится в памяти. Если это так, то это значение возвращается как результат, а вычисления и рекурсивные вызовы осуществляются лишь в том случае, когда функция с такими аргументами ещё ни разу не вызывалась. Подробнее этот метод мы рассмотрим при изучении динамического программирования.

Задача Л9.6. Сколько раз в рекурсивном алгоритме вычисления $\textsc{Fibo}(10)$ будет вызвана процедура вычисления $\textsc{Fibo}(1)$?

Задача Л9.7. Сколько раз в рекурсивном алгоритме вычисления $\textsc{Fibo}(n)$ будет вызвана процедура вычисления $\textsc{Fibo}(m)$?

Задача Л9.8. Продолжите функцию F_n на отрицательные значения n. Измените рекурсивный алгоритм вычисления Fibo так, чтобы он работал и при отрицательных n.

Задача Л9.9. Дана рекуррентная последовательность $a_n = 2 \cdot a_{n-1} - a_{n-2} + 1$, $a_0 = a_1 = 1$.

Напишите рекурсивный и нерекурсивный алгоритмы вычисления n-го элемента этой последовательности.

Задача Л9.10. Рассмотрим следующее рекуррентное соотношение для функции $f(n) = a^n$:

$$a^0 = 1, \quad a^n = \begin{cases} a^{n-1} \cdot a, \text{ если } n \text{ – нечётное,} \\ (a^{n/2})^2, \text{ если } n \text{ – чётное.} \end{cases}$$

Нарисуйте дерево рекурсивных вызовов для $f(1000)$ (подсказка: это дерево не ветвится и является цепочкой вызовов).

Семинар 9

Рекурсия и итерации

Однопроходные алгоритмы: вычисление максимума и средних значений

Задача С9.1. Изучите программы 9.1 и 9.2. Напишите программу, которая находит минимальное, максимальное, среднее арифметическое и среднее квадратичное введённых действительных чисел (используйте тип `double`) и располагает эти четыре числа в порядке возрастания.

Программа 9.1: Максимальное число.

```
#include <stdio.h>
int main () {
    int i, n, a, max;
    printf ("Введите количество чисел: ");
    scanf  ("%d", &n);
    printf ("Введите %d чисел: ", n);
    scanf  ("%d", &max);
    for(i = 1; i < n ; i++) {
        scanf ("%d", &a);
        if(a > max) max = a;
    }
    printf ("%d\n", max);
    return 0;
}
```

Программа 9.2: Сумма чисел.

```
#include <stdio.h>
int main () {
    int i, n;
    double a, sum = 0;
    scanf  ("%n", &n);
    for(i = 0; i < n ; i++) {
        scanf ("%lf", &a);
        sum += a;
    }
    printf ("%15.10lf\n", sumn);
    return 0;
```

```
}
```

Задача C9.2. Напишите программу, реализующую однопроходный алгоритм вычисления 4-х самых больших различных чисел из данного набора чисел.

Алгоритмы вычисления a^n

Программа 9.3: Возведение в степень (итерационный алгоритм 1)

```
#include <stdio.h>
double power(double a, long n) {
    int i;
    double r = 1;
    if (n < 0){
        n = -n;
        a = 1 / a;
    }
    for(i = 0 ; i < n ; i++) {
        r *= a;
    }
    return r;
}
int main() {
    double x;
    long n;
    while (scanf ("%lf %ld", &x, &n) == 2) {
        printf("%lf\n", power (x, n));
    }
}
```

Программа 9.4: Возведение в степень (рекурсивный алгоритм 1).

```
double power(double x, long n) {
    if(n == 0) return 1;
    if(n < 0)  return power (1 / x, -n);
    else       return x * power (x, n - 1);
}
```

Программа 9.5: Возведение в степень (рекурсивный алгоритм 2).

```
double power(double x, long n) {
    if( n == 0 ) return 1;
    if( n < 0  ) return power ( 1 / x, -n);
    if( n % 2  ) return x * power (x, n - 1);
    else         return power(x * x, n / 2);
}
```

Программа 9.6: Возведение в степень (итерационный алгоритм 2).

```
double power(double a, long n) {
    double b = a;
    double r = 1;
    while(n != 0) {
        if(n % 2 == 1) r *= b;
        b *= b;
        n /= 2;
    }
    return r;
}
```

Алгоритмы вычисление чисел Фибоначчи

В программе можно определять свои функции. Рассмотрим это на примере функции, вычисляющей числа Фибоначчи. Функция `fib` получает на вход целое число (`int n`) и возвращает в качестве результата также целое число — число Фибоначчи F_n.

Задача С9.3. Изучите программу 9.7. Скомпилируйте её и запустите. Обратите внимание на строчку «`return fib(n-1) + fib(n-2);`» в описании функции `fib`. Она означает «вернуть в качестве результата сумму двух чисел, которые получаются при вычислении `fib(n-1)` и `fib(n-1)`». Это соответствует определению чисел Фибоначчи:

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 1,\ F_1 = 1.$$

При $n > 1$ числа Фибоначчи определяются рекурсивно, то есть через другие числа Фибоначчи, а при $n = 1$ и $n = 0$ они по определению равны 1. Модифицируйте программу так, чтобы она выводила первые 50 чисел Фибоначчи. Сколько примерно времени вычисляется `fib(50)`?

Задача С9.4. Изучите программу 9.8, скомпилируйте и запустите. Логика её работы заключается в следующем: в переменных a и b хранятся последние вычисленные числа Фибоначчи. На каждой итерации вычисляется следующее число Фибоначчи (переменная c) и происходит обновление значений a и b. Модифицируйте её так, чтобы она выводила 50 первых чисел Фибоначчи. Сравните время работы этой программы программой, полученной в задаче С9.3. Верно ли вычислены F_{49} и F_{50}?

Программа 9.7: Числа Фибоначчи (рекурсивный алгоритм).

```
#include<stdio.h>
int fib(int n) {
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
int main() {
    int n;
    scanf("%d", &n);
    printf("%d\n", fib(n));
    return 0;
}
```

Программа 9.8: Числа Фибоначчи (итерационный алгоритм).

```
int fib(int n) {
    int a, b, c, i;
    a = b = c = 1;
    for(i = 1; i < n ; i++) {
        c = a + b;
        b = a;
        a = c;
    }
    return c;
}
```

Анализ рекурсивного алгоритма вычисления чисел Фибоначчи

Задача С9.5. Запустите программу 9.9. Найдите рекуррентную формулу для количества вызовов функции `fib` (`fib_calls`). Измените программу так, чтобы она считала количество вызовов функции `fib` с аргументом 1 или 0.

```
F( 0) =         1  fib_calls( 0) =         1
F( 1) =         1  fib_calls( 1) =         1
F( 2) =         2  fib_calls( 2) =         3
F( 3) =         3  fib_calls( 3) =         5
F( 4) =         5  fib_calls( 4) =         9
F( 5) =         8  fib_calls( 5) =        15
F( 6) =        13  fib_calls( 6) =        25
F( 7) =        21  fib_calls( 7) =        41
F( 8) =        34  fib_calls( 8) =        67
F( 9) =        55  fib_calls( 9) =       109
F(10) =        89  fib_calls(10) =       177
F(11) =       144  fib_calls(11) =       287
...
```

Напишите программу, которая ищет отношение `fib_calls(n+1)/fib_calls(n)`. Измените тип переменной `count` на `double`.

Повторение: Глобальные и локальные переменные

Переменные, которые объявляются в теле функции, «видны» только внутри этого тела (доступны только для тех инструкций, которые находятся внутри тела функции). Можно объявлять переменные с одинаковыми именами внутри различных функций. Эти переменные никак не будут связаны друг с другом. Память под локальные переменные и переменные, являющиеся аргументами функций, выделяется только на время выполнения функций. «Время жизни» этих переменных равно времени вычисления функции.

Есть возможность завести общие переменные, которые будут «видны» из всех функций. Они называются **глобальными переменными**. Глобальные переменные объявляются вне всех функций. обычно перед определениями функций сразу после окончания секуции с командами препроцессора.

В приведённой здесь программе 9.9 переменная `count` глобальная. Она используется для подсчета количества обращений к функции `fib`. Переменная `n` в функции `main` локальная. Она не видна из функции `fib`. Но у функции `fib` в аргументах есть переменная с тем же именем `n`. Аргументы в языке Си следует рассматривать также, как и обычные локальные переменные.

Программа 9.9: Анализ рекурсивного алгоритма вычисления чисел Фибоначчи

```
#include<stdio.h>
int count = 0; // глобальная переменная
int fib(int n) {
    count++;
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
int main() {
    int n;      // локальная переменная функции main
    for(n = 0 ; n < 20; n++) {
        count = 0;
        printf("F(%2d) = %8d  ", n, fib(n));
        printf("fib_calls(%2d) = %8d\n", n, count);
    }
    return 0;
}
```

Задача С9.6. Любое число можно представить в виде суммы неповторяющихся чисел Фибоначчи. Более того, можно сделать так, что в этой сумме отсутствуют пары соседних чисел Фибоначчи. Задача: представить введённое число n в виде такой суммы. Пример входа/выхода:

stdin	stdout
25	21 + 3 + 1

Задача С9.7. Различаются ли результат и скорость работы следующих двух функций?

```
double f1(unsigned int n) {
    if (n==0) return 0;
    if (n==1) return 0.1;
    return  1.9*f1(n - 1) - 0.95*f1(n - 2);
}
double f2(unsigned int n) {
    if (n==0) return 0;
    if (n==1) return 0.1;
    return  0.95*f2(n - 1) + 0.95*(f2(n - 1) - f2(n - 2));
}
```

Реализуйте идею «рекурсии с запоминанием», а именно, объявите глобальные массивы «`double fdata[1000]`» и «`int is_calculated[1000]`» и инициализируйте массив `is_calculated`. нулями. Запоминайте вычисленные значения функции в глобальном массиве `fdata`. Определение фунции `f` может выглядеть так:

```
double f(unsigned int n) {
```

```
    if (is_calculated[n]) {
        return fdata[n];
    } else {
        double result;
        if (n == 0) result =  0;
        else if (n == 1) result = 0.1;
        else  result = 1.9*f(n - 1) - 0.95*f(n - 2);
        fdata[n] = result;
        is_calculated[n] = 1;
        return result;
    }
}
```

Сравните время вычисления `f(50)` и `f1(50)`. Объясните результат сравнения.

Задача С9.8. Напишите рекурсивную функцию `F`, вычисляющую числа Фибоначчи, основывающую на рекуррентных формулах $F_{2n+1} = 2F_nF_{n+1} - F_n^2$ и $F_{2n} = F_nF_{n-1} + 2F_{n-1}^2 + (-1)^n$ (убедитесь с помощью численного эксперимента, что эти формулы верны). Как растет число рекурсивных вызовов в зависимости от n? Нарисуйте дерево рекурсивных вызовов получающееся при вычислении F_{50}. Сравните время вычисления функции для двух способов рекурсивного вызова:

```
... // Способ 1 рекурсивного вызова
return 2*F(n)* F(n+1) - F(n)*F(n);

... // Способ 2 рекурсивного вызова
double fn = F(n);
double fn1 = F(n+1);
return 2*fn* fn1 - fn*fn;
```

Задача С9.9. Решите предыдущую задачу, при условии, что для нечётных n используются рекуррентные формулы $F_{2n+1} = 2F_n(F_n + F_{n-1}) - F(n)^2$ и $F_{2n} = F_nF_{n-1} + 2F_{n-1}^2 + (-1)^n$, а для чётных n формулы $F_{2n+1} = 2F_nF_{n+1} - F(n)^2$ и $F_{2n} = (F_{n+1} - F_n)(2F_{n+1} - F_n) + (-1)^n$.

Задача С9.10. Для последовательности векторов $g_n = \|F_n, F_{n-1}\|^T$, координаты которыхо равны двум соседним числам Фибоначчи, верно следующее соотношение:

$$g_n = A \circ g_{n-1}, \text{ где } A \left\| \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix} \right\|, \text{ то есть } \left\| \begin{matrix} F_n \\ F_{n-1} \end{matrix} \right\| = \left\| \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix} \right\| \circ \left\| \begin{matrix} F_{n-1} \\ F_{n-2} \end{matrix} \right\|.$$

Реализуйте алгоритм вычисления чисел Фибоначчи, основанный на возведении матрицы A в n-ю степень. Используйте быстрый алгоритм возведения в степень.

Лекция 10

Сортировка, оценка времени работы алгоритмов

Краткое описание: Рассматривается одна из наиболее важных алгоритмических задач — задачу сортировки объектов по некоторому признаку. Приводятся и анализируются несколько алгоритмов сортировки — сортировка, основанная на многократном вычислении максимума, сортировка методом «пузырька» и быстрая сортировка
Приводятся верхняя и нижняя граница сложности задачи сортировки массива элементов — скорости роста времени работы самого оптимального аогоритма в зависимости от размера массива..
Ключевые слова: сортировка методом пузырька, быстрая сортировка, $O(f)$ и $\Theta(f)$.

Одна из простейших алгоритмических задач формулируется следующим образом:

Задача Л10.1. Дан массив из N чисел. Найдите алгоритм, который сортирует числа массива (упорядочивает числа по возрастанию).

У этой задачи есть решение, работающее время, пропорциональное

$$N \cdot \log N.$$

Сведение задачи сортировки к многократному поиску максимума

Мы умеем находить максимум для данного набора чисел. Это простой однопроходный алгоритм. Максимум из введённых чисел будет последним числом в отсортированном массиве. Давайте среди оставшихся чисел снова найдём максимум — это будет предпоследнее число в отсортированном массиве, и т. д. Так мы можем $N-1$ раз найти максимум, исключая после каждого шага найденное максимальное число. Найденные максимумы будем записывать по порядку в отдельный массив B. Таким образом, мы получим упорядоченный по убыванию или по возрастанию массив B, в зависимости от того, с какого конца мы будем его заполнять числами.

Для нахождения максимума из K чисел требуется время, пропорциональное K. Поэтому, для нашего алгоритма потребуется время пропорциональное

$$1 + 2 + \ldots + (N-1) = \frac{N(N-1)}{2}.$$

Это квадратичная функция от N. При больших N увеличение значения N в m раз приводит к тому, что время работы увеличивается примерно в m^2 раз. Про такие алгоритмы говорят, что они работают **квадратичное по N время** или что **асимптотика времени работы равна** $O(N^2)$.

Предложенный алгоритм не самый лучший метод сортировки, но самый простой для понимания.

Задача Л10.2. Напишите псевдокод алгоритма сортировки методом многократного нахождения максимума.

Сортировка «пузырьком»

Сортировка «пузырьком» осуществляется следующим образом. «Пробежим» по элементам массива, сравнивая каждый из них со следующим. После каждого сравнения будем менять соседние элементы местами, если они расположены в неправильном порядке. Потом «пробежим» снова и снова, пока массив не окажется упорядоченным.

Задача Л10.3. Докажите, что через некоторое количество таких «пробегов» массив станет упорядоченным.

Описанная идея представленна в виде псевдокода 10.21. Алгоритм состоит из двух вложенных циклов, условного оператора и операции обмена местами двух элементов массива.

Алгоритм 10.21 Сортировка методом пузырька

```
function SORT(массив a)
    L ← длина массива a                ▷ Массив имеет элементы a[0], a[1], ... a[L − 1]
    for j = 1, 2, ..., L − 1 do
        for i = 1, ..., L − j do
            if a[i − 1] > a[i] then
                tmp ← a[i]
                a[i] ← a[i − 1]
                a[i − 1] ← tmp
            end if
        end for
    end for
end function
```

Например, пусть у нас есть массив $a = \{5,\ 4,\ 3,\ 2,\ 1\}$. Тогда, на первой итерации мы сравним $a[0] = 5$ с $a[1] = 4$ и поменяем их местами. В итоге получим:

$$a = \{4,\ 5,\ 3,\ 2,\ 1\}.$$

Далее мы смещаемся на один элемент вправо и сравниваем $a[1] = 5$ с $a[2] = 3$ — снова порядок нарушен и нам нужно их менять местами:

$$a = \{4,\ 3,\ 5,\ 2,\ 1\}.$$

Затем мы переходим к элементу $a[2]$ и сравниваем его с $a[3]$ и так далее. После ещё двух обменов получим

$$a = \{4,\ 3,\ 2,\ 1,\ 5\}.$$

Таким образом, после первого «пробега» число 5 оказалась в конце массива. Она и была «пузырьком» который «всплыл» в конец массива. Во время второго «пробега» всплывать будет число 4, оно встанет прямо перед числом 5.

Какой бы начальный массив мы не взяли, всегда после первого «пробега» нашего алгоритма самый большой элемент окажется в конце массива, после второго «пробега» следующий по величине элемент окажется перед ним, после третьего «пробега» третий по величине элемент окажется на своём месте и так далее. Из этих рассуждений следует:

Утверждение 10.1. *В алгоритме сортировки методом «пузырька» массива из N элементов достаточно сделать $(N-1)$ «пробег».*

Изучите текст программы 10.1. Обратите внимание на то, что каждый следующий «пробег» заканчивается на один элемент раньше, нежели предыдущий. Идея сортировки методом «пузырька» похожа на идею сведения к поиску максимума и работает тоже квадратичное от размера массива время.

Быстрая сортировка

Описание алгоритма быстрой сортировки

Быстрая сортировка работает в среднем быстрее, чем сортировка методом «пузырька». Она основана важной идее построения алгоритмов:

Разделяй и Властвуй.

Лучший способ решить сложную задачу — это разделить её на несколько простых и «разделаться» с ними по отдельности. По сути, это один из важных инструментов мышления при решении задач.

Функция QuickSort сводит сортировку данного ей массива к разделению этого массива на две группы элементов и сортировке этих двух групп по отдельности, причём для сортировки этих групп *вызывает саму себя*.

Пусть нам нужно отсортировать участок массива A, начиная с p-го по q-й элемент включительно. Будем называть этот участок подмассивом $A[p..q]$. Тогда

> **ШАГ 1:** Возьмем элемент $A[p]$ и «раскидаем» остальные элементы $A[(p+1)..q]$ по разные стороны от него стороны — меньшие влево, большие — вправо, то есть переставим элементы подмассива $A[p..q]$ так, чтобы вначале шли элементы меньше либо равные $A[p]$ потом элементы, больше либо равные $A[p]$. Назовём этот шаг разделением (partition).
>
> **ШАГ 2:** Пусть r есть новый индекс элемента $A[p]$. Тогда, если $q-p>2$, вызовем функцию сортировки для подмассивов $A[p..(r-1)]$ и $A[(r+1)..q]$.

Этот алгоритм представлен в виде псевдокода 10.22.
На семинаре вы рассмотрите реализацию алгоритма быстрой сортировки на языке Си.

Задача Л10.4. Пусть все элементы массива $A[p..q]$ равны. Какое значение вернёт функция Partition?

Задача Л10.5. Покажите, что время работы функции Partition(A,p,q) составляет $\Theta(n)$, где $n=q-p+1$.

Алгоритм 10.22 Алгоритм быстрой сортировки

```
function QuickSort(A, p, q)
    if p < q then
        r = Partition(A, p, q)
        QuickSort(A, p, r − 1)
        QuickSort(A, r + 1, q)
    end if
end function
```

```
function Partition(A, p, q)
    x ← A[p]
    i ← p − 1
    j ← q + 1
    while TRUE do
        repeat j ← j − 1
        until A[j] ⩽ x
        repeat i ← i + 1
        until A[i] ⩾ x
        if i < j then
            поменять A[i] ↔ A[j]
        else
            return j
        end if
    end while
end function
```

Оценка эффективности алгоритма быстрой сортировки

Время работы алгоритма быстрой сортировки зависит от того, как разбивается массив на каждом шаге. Если разбиение происходит на примерно равные части, время работы составляет $\Theta(n \log n)$ (см. задачу Л10.5) . Если же размеры частей сильно отличаются, сортировка может занимать время $\Theta(n^2)$, как при сортировке методом «пузырька».

Мы рассмотрим две функции $T_{max}(n)$ $T_{mean}(n)$, соответствующие максимальному и среднему времени работы алгоритма QuickSort на массиве длины n и запишем для них рекуррентные отношения. На основании рекуррентных отношений мы выведем асимптотики функций $T_{max}(n)$ $T_{mean}(n)$.

Оценка максимального времени работы алгоритма быстрой сортировки

Время работы алгоритма быстрой сортировки зависит от того, в какой пропорции будет разбиваться массив на два подмассива функцией Partition. Назовем разбиение «плохим», если это разбиение массива размера n на два подмассива с размерами 1 и $n-1$. На выполнение функции Partition мы тратим время $\Theta(n)$. Если считать, что каждый раз разбиение «плохое», то можно записать рекуррентное соотношение:

$$T_{max} = T_{max}(1) + T_{max} + \Theta(n-1) = T_{max}(n-1) + \Theta(n-1).$$

Напомним, смысл значка Θ. В данном случае он означает, что существуют константы C_1 и C_2 такие, что

$$T_{max} > T_{max}(n-1) + C_1 \cdot (n-1) \quad \text{и} \quad T_{max} < T_{max}(n-1) + C_1 \cdot (n-1).$$

Если положить $T_{max}(0) = 0$, то получим соотношения:

$$T_{max} > C_1(1 + 2 + \ldots + (n-1)) = C_1 \frac{n(n-1)}{2},$$

$$T_{max} < C_2(1 + 2 + \ldots + (n-1)) = C_1 \frac{n(n-1)}{2},$$

Это означает, что

$$T_{max}(n) = \Theta(n^2).$$

Для того, чтобы доказать тождество $T_{max} = T_{max}(n)$ необходимо решить следующие задачи

Задача Л10.6. Покажите, что для каждого n есть массив длины n такой, что на в алгоритме QuickSort каждый вызов функции `Partition` даёт в результате «плохое» разбиение.

Задача Л10.7. На основании рекуррентного отношения

$$T_{max}(n) = \max_{1 \leqslant q \leqslant n-1} (T_{max}(q) + T_{max}(n-q)) + \Theta(n)$$

покажите, что существует константа C такая, что $T_{max}(n) < Cn^2$.

Оценка среднего времени работы алгоритма быстрой сортировки

Чтобы говорить о среднем времени работы алгоритма, необходимо зафиксировать вероятностное распределение на множестве возможных входов. Для алгоритма сортировки множество возможных входов совпадает с множеством перестановок n элементов.

Всего перестановок $n! = 1 \cdot 2 \cdot \ldots \cdot n$ штук. Будем считать, что все перестановки *равновероятны.*

Утверждение 10.2. *Верно следующее рекуррентное соотношение*

$$T_{mean}(n) = \frac{1}{n-1} \cdot \sum_{q \in [1..n-1]} (T_{mean}(q) + T_{mean}(n-q)) + \Theta(n).$$

Это соотношение выписывается из предположения, что все $n-1$ возможных пропорций $(1/(n-1), 2/(n-2), \ldots, (n-1)/1)$ между размерами двух подмассивов, на которые разбивается массив из n элементов, равновероятны.

Задача Л10.8. Поясните, как равновероятность всех возможных пропорций разбиения для каждого вызова функции Partition следует из равновероятности перестановок.

Задача Л10.9. Пусть $A(n) = A(\lceil 9n/10 \rceil) + A(\lfloor n/10 \rfloor) + n$, $A(1) = 1$. Докажите, что $A(n) = \Theta(n \log n)$.

Оценка сложности задачи сортировки

Итак, мы оценили среднее и максимальное время работы алгоритма QuickSort. Естественно задать следующие вопросы:

Существуют ли алгоритмы сортировки, среднее время работы которых меньше, чем у QuickSort? Существуют ли алгоритмы сортировки, у которых максимальное время работы равно $\Theta(n \log n)$?

Здесь мы покажем, что среднее (а значит и максимальное) время не может быть меньше $\Theta(n \log n)$.

Основные понятия сложности вычислений мы обсудим в конце лекции.

Сравнивать время работы различных алгоритмов довольно сложная задача, потому что бывают алгоритмы, которые быстро работают на одних типах данных и плохо — на других; из двух алгоритмов иногда нельзя выделить более эффективный, так как каждый из них хорош в определённой ситуации.

Но в большинстве случаев алгоритмы имеют вполне определённую *асимптотику среднего и максимального времени работы от размера входа*. Вот примеры, которые мы уже разобрали:

- Оптимизированный алгоритм Евклида вычисления НОД(a, b) работает время $\Theta(\log \max(a, b)$.
- В игре «Ханойский башни» время, необхдимое для перемещения пирамидки размера N, равно $\Theta(2^N)$.
- Последовательное вычисление первых N чисел Фибоначчи требует времени[1] $\Theta(N)$.
- Поиск максимума N чисел работает время $\Theta(N)$.
- Сортировка методом пузырьком N чисел работает $\Theta(N^2)$.
- Сортировка QuickSort N чисел работает $\Theta(N^2)$.

Асимптотики среднего времени работы этих алгоритмов (при естественном равномерном вероятностном распределении на множестве входов) точно такие же (средние времена меньше, чем максимальные, но тип роста такой же), за исключением алгоритма QuickSort, для которого среднее время работы равно $\Theta(N \log N)$.

Асимптотики среднего и максимального времени работы отображают эффективность алгоритма — чем меньше асимптотика, тем эффективнее алгоритм. Численные коэффициенты, которые стоят при этих асимптотиках не так важны. Эти коэффициенты сильно зависят от вычислительной машины и реализации алгоритма. Например, алгоритм работающий время $100 \cdot N^2$ явно лучше алгоритма, работающего время $\frac{1}{100} 2^N$, так как второй быстрее первого только для $N < 23$. На простых входных данных все алгоритмы работают быстро. Важно как растет время работы (тип роста) с увеличением сложности входных данных.

Попытки найти алгоритм сортировки N чисел, работающий быстрее чем $\Theta(N \log N)$ не приводили к успеху и возникло предположение, что лучшей асимптотики получить нельзя.

[1] Указанная асимптотика верна лишь в предположении, что не смотря на линейный с N рост числа знаков в числах умножение и сложение занимают время $\Theta(1)$.

Докажем это предположение. Для начало вспомним известную детскую задачу.

Задача Л10.10. Петя загадал натуральное число x меньше либо равно $N = 100$. Вася может задавать вопросы вида

«Верно ли, что $x \leqslant M$?»

где вместо M стоит любое число. На эти вопросы Петя честно отвечает. Найти минимальное число вопросов, которых наверняка хватит, чтобы угадать загаданное число.

Правильный ответ: $\lceil \log_2 N \rceil = 7$.

Подсказка Л10.10.1. *Естественно первый вопрос задать такой: «Верно ли, что $x \leqslant 50$?» В результате мы узнаем, на каком из двух отрезков $[1, 50]$ или $[51, 100]$ лежит число x. Затем мы нужный отрезок разделим пополам на два отрезка, содержащих уже по 25 целых точек. И зададим вопрос, чтобы узнать на каком из них лежит число x. На следующем шаге получим отрезки длиной 12 и 13. В худшем случае x лежит на том, который длиннее. Каждый раз размер отрезка, на котором лежит x уменьшается примерно в два раза. Пусть l_k — длина отрезка, на котором лежит x после k вопросов в худшем случае. Тогда*

$$l_0 = 100,\ l_1 = 50,\ l_2 = 25,\ l_3 = 13,\ l_4 = 7,\ l_5 = 4,\ l_6 = 2,\ l_7 = 1.$$

Это значит, что 7 вопросов хватит. Покажите, что

$$(l_{k+1} = \lceil l_k/2 \rceil,\ l_0 = N) \Rightarrow (l_d = 1 \text{ при } d = \lceil \log_2 N \rceil).$$

В частности $\lceil \log_2 100 \rceil = \lceil 6.64386 \rceil = 7$.

Ответ на эту задачу в общем виде звучит так:

Утверждение 10.3. *Для того, чтобы узнать, какой из N возможных объектов загадан, нужно получить ответ на $\lceil \log_2 N \rceil$ элементарных вопросов.*

Из этого утверждения следует ограничение снизу на время работы целого ряда алгоритмов. Сформулируем его несколько по-другому.

Утверждение 10.4. *Для того, чтобы узнать какой из N возможных объектов дан на входе, нужно осуществить как минимум $\rceil \log_2 N \lceil$ сравнений (условных переходов).*

В задаче сортировки на вход даётся n элементов. Эти элементы можно дать на вход в $N = n!$ различных последовательностях. Чтобы узнать, какую из $n!$ возможных последовательностей мы имеем, нужно осуществить как минимум $\log_2 n!$ операций сравнения. Значит, программа сортировки не может работать в среднем быстрее, чем $\Theta(\log_2 n!)$.

Семинар 10

Сортировка, исследование времени работы алгоритмов

Краткое описание: Рассматривается задача сортировки. Исследуется максимальное и среднее время работы алгоритма сортировки «методом пузырька» и алгоритма быстрой сортировки («quick sort»). Решается задача создания массивов случайных чисел.

Сформулируем основную задачу семинара:

Задача С10.1. (Сортировка чисел) Напишите программу, которая считывает натуральное число n, затем считывает n целых чисел, и выводит эти числа в порядке возрастания. Пример входа/выхода:

stdin	stdout
6 7 6 2 4 3 6	2 3 4 6 6 7

Эту задачу мы решим несколькими методами. Для численного исследования среднего времени работы различных алгоритмов сортировки нам потребуется программа генерации случайных чисел.

Генерация случайных чисел

Задача С10.2. (Случайные числа) Изучите функцию `rand()` (команда « `man 3 rand`») объявленную в файле `stdlib.h` Напишите на её основе
а) функцию генерации случайных целых чисел из отрезка $[-1000, 1000]$.
б) функцию генерации случайных действительных чисел из отрезка $[0, 1]$.

Задача С10.3. Напишите функцию генерации случайных точек из круга $x^2 + y^2 = 1$.

Задача С10.4. Напишите программу `random_array`, коорая получает на вход n и выводит n случайных целых из промежутка $[-10^6, 10^6]$.

Сортировка «методом пузырька»

Задача С10.5. (Сортировка «методом пузырька») Реализуйте алгортм сортировки n целых чисел методом пузырька. Оформите этот алгоритм в виде отдельной функции, которая получает на вход массив целых чисел, а именно адрес первого элемента массива (тип `int*`) и

число элементов в массиве (тип `int`). Сортировка должна проходит на месте (без создания нового массива).

Покажите, что в худшем случае этот алгоритм работает квадратичное время. Для этого объявите переменную, которая будет равна количеству операций перестановки двух элементов массива. Для различных n и запустите функцию сортировки чисел $\{n,\ n-1,\ \ldots, 2,\ 1\}$ (первых n натуральных чисел, введённых в обратном порядке). Изучите, как растёт время сортировки с ростом n. Рассмотрите также как растёт с n время работы алгоритма для случайного входного массива (см. задачу С10.4).

Используйте **команды перенаправление входа/выхода**. Пусть у вас есть файл с данными `input.txt`. После этого эти данные можно дать на стандартный вход программе с помощью командной строчки

```
./sort < input.txt
```

Используя правую угловую скобку можно результат вывода программы перенаправить в произвольный файл. Например, команда

```
./sort < input.txt > output.txt
```

запустит программу `sort`, и перенаправит данные из файла `input.txt` на стандартный вход, а то, что программа `sort` будет выводить на стандартный поток вывода, запишет в файл `output.txt`.

Можно также перенаправлять результат работы одной программы на вход другой. Для этого используется операция pipe, обозначаемая вертикальной чертой «|». Пусть программа `random_array` выводит массив целых чиел, раделённых пробелом. Тогда с помощью команды

```
./random_array | ./sort
```

можно осуществить последовательность двух действий — сгенерить случайный массив и затем отсоритровать его. Важно при этом проследить, чтобы выход программы `random_array` имел формат, соответствующий формату входных данных программы `sort`.

Программа 10.1: Алгоритм сортировки «пузырьком»

```
void bsort(int *a, int length) {
    int i, j;
    for (i = 1 ; i < length ; i++)
        for (j = i ; j < length ; j++)
            if (a[j-1] > a[j])
            {
                int tmp = a[j];
                a[j] = a[j-1];
                a[j-1] = tmp;
            }
}
```

Программа 10.2: Алгоритм сортировки пузырьком, которыйостанавливается сразу, как только массив становитсяотсортированным

```
void bsort(int* a, int length) {
```

```
    int is_sorted = 0, i, j;
    while(! is_sorted) {
        is_sorted = 1;
        for (j = 1 ; j < length ; j++)
            if (a[j-1] > a[j])
            {
                int tmp = a[j];
                a[j] = a[j-1];
                a[i-1] = tmp;
                is_sorted  = 0;
            }
    }
}
```

Программа 10.3: Ещё один вариант сортировки за время $O(N^2)$

```
void bsort(int *a, int length){
    int i, j;
    for (i = 0 ; i < length ; i++)
        for (j = i+1 ; j < length ; j++)
            if (a[j] < a[i]) {
                int tmp = a[j];
                a[j] = a[i];
                a[i] = tmp;
            }
}
```

Программа 10.4: Код к задаче C10.7

```
void bsort(int* a, int length) {
    int i, j;
    for (i = 1 ; i < 10 ; i++)
        for (j = i ; j < length ; j++)
            if (a[i-1] > a[i])
            {
                int tmp = a[j];
                a[j] = a[j-1];
                a[j-1] = tmp;
            }
}
```

Программу 10.1 можно оптимизировать следующим образом: делать не $n-1$ пробег, а столько пробегов, сколько нужно. Как только массив оказывается упорядоченным, алгоритм может заканчивать свою работу. Эта оптимизация реализована в программе 10.2.

Задача C10.6. Сколько раз будет выполнена операция обмена местами соседних элементов в программе 10.1, если входной массив равен $\{10,\ 9,\ \ldots,\ 3,\ 2,\ 1\}$?

Задача C10.7. Приведите пример массива, который не будет отсортирован программой 10.4.

Исследование быстрой сортировки

Задача С10.8. (Быстрая сортировка) Разберите программу 10.5. Проведите исследование, аналогичное описанному в задаче С10.5. В каком случае алгоритм быстрой сортировки работает дольше (размер массива зафиксирован)? Изучите как работает алгоритм быстрой сортировки в случае, когда входной массив чисел отсортирован а) по убыванию; б) по возрастанию.

Программа 10.5: Программа, реализующая алгоритм быстрой сортировки

```
void quick_sort(int  *a, int length) {
     quick_sort0(a, 0, length);
}
int quick_sort0(int *a, int lo, int hi) {
    int h, l, p, t;
    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];
        do {
            while ((l < h) && (a[l] <= p)) l++;
            while ((h > l) && (a[h] >= p)) h--;
            if (l < h) {
                t = a[l]; a[l] = a[h]; a[h] = t;
            }
        } while (l < h);
        t = a[l]; a[l] = a[hi]; a[hi] = t;
        quick_sort0(a, lo, l-1 );
        quick_sort0(a, l+1, hi );
}
```

Задача С10.9. Какие изменения следует внести в программу 10.5, чтобы отсортировать массив в порядке убывания, а не возрастания.

Задача С10.10. (Функция qsort из стандартной библиотеки) Осуществить сортировку массива можно с помощью одной строки кода:

```
qsort(a, n, sizeof(int), cmp );
```

Функция `qsort` описана в заголовочном файле `stdlib.h`. Для того чтобы использовать эту функцию, в начале программы нужно добавить строчку

```
#include <stdlib.h>
```

Эта функция умеет сортировать объекты самой разной природы — целые числа (типы `int`, `long`, `unsigned int`, ...), действительные числа (типы `float`, `double`), и произвольные другие объекты, занимающие фиксированное количество байт. По сути, функция `qsort` предназначена для сортировки набора блоков одинаковой длины, которые расположены в памяти последовательно друг за другом. Первый аргумент этой функции – указатель на место в памяти, где находится начало первого блока (предполагается что блоки идут подряд друг за

другом). Второй аргумент равен количеству блоков, третий – размеру одного блока в байтах. Четвёртый аргумент функции `qsort` особенный. Это указатель на функцию сравнения — функцию, которая умеет сравнивать два блока. Функция сравнения должна иметь следующий вид:

```
int cmp(const void*, const void*).
```

Функция сравнения имеет два аргумента — указатели на места в памяти, где хранятся два блока, которые нужно сравнить. Она должна вернуть 1, −1 или 0 в зависимости от того, кто кого больше. Возвращаемое значение должно быть равно 0, если данные, хранимые в блоках считаются равными, 1 — если данные первого блока больше, и −1 — если меньше. В нашем случае блоки — это целые числа. Размер блока равен `sizeof(int)` (в 32-битной архитектуре `sizeof(int)`= 4 (байта)). Естественно написать следующую функцию сравнения

```
int cmp(const void *a, const void *b) {
    int *a1 = (int*)a;  // указатели a и b приводим к
    int *b1 = (int*)b;  // указателям на целые числа
    if (*a1 > *b1) return 1;
    if (*a1 < *b1) return -1;
    return 0;
}
```

Здесь аргумент `a` есть адрес в памяти, где хранится неизвестный объект. Переменная `a1` численно равна тому же адресу, но имеет тип `int*`, то есть имеет тип адреса, по которому хранится целое число Выражение «`*a1`» равно значению целого числа, которое хранится по адресу `a1`

Этот тип соответствует функции, которая получает на вход два указателя на некоторые объекты в памяти, которые нельзя менять (модификатор «`const`» означает «только для чтения»). Изучите программу 10.6.

Программа 10.6: Сортировка с помощью функции `qsort`

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000
int cmp(const void *a, const void *b) {
    return *(int*)a - *(int*)b;
}

int main() {
   int n, i;
   int a[N];
   scanf("%d", &n);
   for(i = 0 ; i < n; i++) {
       scanf("%d", &a[i]);
   }

   qsort(a, n, sizeof(int), cmp );
```

```
    for(i = 0 ; i < n; i++){
        printf("%d ", a[i]);
    }
    return 0;
}
```

Динамическое выделение памяти

Напомним, что функция `malloc` (memory allocate) выделяет кусок памяти заданного размера — несколько подряд идущих байт, в которых мы можем хранить произвольные данные — и возвращает указатель на первый байт. Единственный аргумент этой функции – число байт, которое нужно выделить. Память, которая была выделена с помощью функции `malloc`, нужно в конце освобождать с помощью функции `free`. Аргумент функции `free` — это указатель на первый байт выделенной когда-то с помощью функции `malloc` памяти.

Программа 10.7 сортирует массив чисел, память под который выделяется динамически.

Программа 10.7: Динамическое выделение памяти под массив

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int cmp(const void *a, const void *b) {
    return *(int*)a - *(int*)b;
}
int main() {
    int n, i;
    int *a;
    scanf("%d", &n);   // узнали размер массива
    a = (int*) malloc(sizeof(int) * n); // запросили память
    for( i = 0 ; i < n; i++ ) {
        scanf("%d", &a[i]);
    }

    qsort(a, n, sizeof(int), cmp );

    for( i = 0 ; i < n; i++ ) {
        printf("%d ", a[i]);
    }
    free(a);                              // освободили память
    return 0;
}
```

Программа 10.8: Программа сортировки строк в алфавитном порядке

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

```
#define N 1000
#define M 50
int main() {
    int n, i;
    char names[N][M];

    scanf("%d", &n);
    if(n > N || n <=0) {
        fprintf(stderr,"Bad value of n\n");
        return 1;
    }
    for(i = 0; i < n ; i++) {
        if(fgets(names[i], M, stdin) == NULL) {
            fprintf(stderr, "Can't read name[%d]\n", i);
            return 2;
        };
    }
    qsort(names, n, sizeof(char[M]), strcmp);
    for(i = 0; i < n ; i++) {
        printf("%s", names[i]);
    }
    return 0;
}
```

Решение рекуррентных соотношений

Задача С10.11. Решите рекуррентное соотношение для $D(n)$, $n = 0, 1, 2 \ldots$. Оцените скорость роста функции $D(n)$. Проведите численный эксперимент.
а) $D(n) = \max\left(D(\lfloor n/2 \rfloor), D(\lceil n/2 \rceil)\right) + 1,\ D(0) = D(1) = 0.$
б) $D(n) = D(\lfloor n/2 \rfloor) + D(\lceil n/2 \rceil) + 1,\ D(0) = D(1) = 0.$
в) $D(n) = D(n-1) + D(n-2) + 1,\ D(0) = D(1) = 0.$
г) $D(n) = (D(n-1) + D(n-2))/2 + 1,\ D(0) = D(1) = 0.$
д) $D(n) = (D(n-1) + D(n-2))/2 + n,\ D(0) = D(1) = 0.$
е) $D(n) = D(\lceil 2n/3 \rceil) + 1,\ D(0) = D(1) = 0.$
ж) $D(n) = D(n-1) + A \cdot n^2,\ D(0) = 0.$

Лекция 11

Рекурсивные и нерекурсивные алгоритмы перебора множеств

Краткое описание: В этой лекции мы продолжаем изучать алгоритмы, использующие метод рекурсии. Рассмотрим рекурсивные алгоритмы перебора различных комбинаторных объектов. Одной из основных задач будет разработка алгоритма генерации всех правильных скобочных выражений длины $2n$. Будут разобраны также задача генерации перестановок и всех m-элементных подмножеств первых n натуральных чисел. Мы покажем, что рекурсивные идеи можно реализовать, не используя рекурсивые функции.

Перебор перестановок

Перестановка некоторого набора элементов — это упорядоченная последовательность из этих элементов. Например, множество $\{1, 2, 3\}$ имеет 6 различных перестановок

$$(1, 2, 3),\ (1, 3, 2),\ (2, 1, 3),\ (2, 3, 1),\ (3, 1, 2),\ (3, 2, 1)$$

Вообще, для любого множества из n элементов существует ровно $n! = 1 \cdot 2 \cdot \ldots \cdot n$ различных перестановок. Это легко показать. Пусть $P(n)$ — число перестановок n-элементного множества. Рассмотрим для определённости множество $M = \{1, 2 \ldots, n\}$. На первом месте в перестановке может стоять одно из чисел $1, \ldots, n$. Перестановок множества M, в которых на первом месте стоит число i ровно столько, сколько есть различных перестановок оставшихся $n-1$ чисел, то есть $P(n-1)$. Поэтому,

$$P(n) = n \cdot P(n-1).$$

По индукции получаем $P(n) = 1 \cdot 2 \ldots \cdot n = n!$.

Рассмотрим следующую задачу:

Задача Л11.1. Напишите программу, которая выводит все перестановки множества $\{1, 2, \ldots, n\}$ в лексикографическом (алфавитном) порядке (перестановки можно рассматривать как слова в алфавите $B = \{1, 2, \ldots, n\}$).

Рекурсивный алгоритм

Мы будем перебирать перестановки чисел $\{1,\ 2,\ \ldots,\ n\}$ в глобальном массиве a: $(a[0],\ \ldots,\ a[n-1])$, последовательно заполняя его числами $1,\ \ldots, n$ в различном порядке. Для перебора всех перестановок можно применить ту же рекурсивную идею, что и при доказательстве формулы числа перестановок. А именно, мы можем записывать на первое место

в массиве a числа $1, \ldots, n$ по очереди, и для каждого числа рекурсивно вызывать функцию генерации перестановок оставшихся $n-1$ чисел. Следующая программа реализует данный алгоритм. В алгоритме необходимо завести массив b, в котором во время конструирования очередной перестановки отмечать использованные числа, чтобы в перестановке не оказалось два одинаковых числа.

Алгоритм 11.23 Рекурсивный алгоритм генерации всех перестановок

```
▷ Глобальные переменные:
▷   n – число элементов перестановке;
▷   i – число, указывающее сколько чисел мы уже задействовали;
▷   b[0..(n − 1)] – массив с информацией о том, какие числа уже использованы;
▷   a[0..(n − 1)] – массив, для хранения перестановки чисел {1, 2, ..., n};
function Do
    if i = n then
        PrintPermutation(a, n);
    else
        for j ∈ {0, ... (n − 1)} do
            if b[j] = 0 then
                b[j] ← 1, a[i] ← j + 1
                Do()
                b[j] ← 0
            end if
        end for
    end if
end function
function GeneratePermutations(l)
    i ← 0, n ← l
    Do()
end function
```

Нерекурсивный алгоритм

В данном случае рекурсивный алгоритм достаточно эффективен, и он очень прост в реализации. Но для сравнения мы рассмотрим ещё нерекурсивный метод генерации перестановок. Мы построим нерекурсивную функцию Next, которая по данной перестановке будет строить следующую за ней в лексикографическом порядке. Для этого зададимся вопросом: в каком случае можно увеличить k-й член перестановки, не меняя предыдущих? Ответ: если он меньше какого-нибудь из следующих членов (с номерами больше k). Отсюда сразу получается алгоритм для Next:

1. Ищем наибольшее k такое, что $a[k] < a[k+1] > \cdots > a[n-1]$.

2. Значение $a[k]$ нужно увеличить минимально возможным образом. Для этого среди $a[k+1]$, ..., $a[n-1]$ находим минимальный элемент $a[t]$, больший $a[k]$ и меняем его с $a[k]$.

3. Осталось только расположить числа с номерами большими k в возрастающем порядке. И это легко сделать, так как они уже расположены в убывающем.

Теперь, чтобы вывести все перестановки мы, можем начать с первой $(1, 2, \ldots, n)$ и вызывать `NextPermutation` до тех пор пока не дойдем до последней перестановки $(n, n-1, \ldots, 1)$. Следующая программа является реализацией этого алгоритма.

Алгоритм 11.24 Нерекурсивный алгоритм генерации перестановок

```
function NextPermutation(a, n)
    c ← 0
    k ← n − 2
      ▷ находим k: a[k] < a[k + 1] > .... > a[n − 1]
    while a[k] > a[k + 1] and k > 0 do
        k ← k − 1
    end while
    if k = −1 then
        return false
    end if
      ▷ находим t > k, такое что a[t] – минимальное
      ▷ число большее a[k] среди a[k + 1] . . . a[n − 1]
    for t ∈ {k + 1, k + 2, . . . , n − 1} do
        if a[t + 1] < a[k] then
            break
        end if
    end for
      ▷ участок массива a[k+1] ... a[n-1] записываем в обратном порядке
    for i ∈ {k + 1, . . . , (n+k)/2} do
        t ← n + k − i
        Swap(a[i], a[t])
    end for
    return true
end function
function GeneratePermutations(n)
    a ← {1, 2, . . . , n}
    repeat
        PrintPermutation(a, n)
    until not NextPermutation(a, n)
end function
```

Определение 11.1. *Переборный алгоритм называется* **эффективным**, *если он работает время, которое растёт пропорционально суммарному размеру перебираемых им элементов, то есть произведению числа элементов на размер описания одного элемента.*

Алгоритм 11.25 Генерация всех правильных скобочных выражений (вариант 1)

```
    ▷ Глобальные переменные:
    ▷   n – четное число, равное длине скобочного выражения;
    ▷   a[0..(n − 1)] – массив скобок;
function Do(m, k)
      ▷ m – длина построенного префикса скобочного выражения a[0..(m − 1)];
      ▷ k – текущее число незакрытых скобок в префиксе a[0..(m − 1)].
  if m = n and k = 0 then
    PrintWord(a)
  end if
  if n − m > k then
    a[m] ←′ (′
    Do(m + 1, k + 1)
  end if
  if k > 0 then
    a[m] ←′)′
    Do(m + 1, k − 1)
  end if
end function
function GenerateBracketWords(l)
  n ← l
  Do(l, 0)
end function
```

Перебор правильных скобочных выражений

Приведём три различных варианта алгоритма генерации правильных скобочных выражений. Первые два алгоритма основаны на рекурсивной функции Do, которая получает на вход (явно или неявно) три числа — n, m и k.

Рассмотрим первый алгоритм, представленный в псевдокоде 11.25.

Задачу, решаемую функцией $\mathrm{Do}(m, k)$ можно сформулировать так: сгенерировать правильное скобочное выражение в подмассиве символов $a[m..(n-1)]$, с учётом того, что в подмассиве $a[0..(m-1)]$ есть k незакрытых скобок.

Здесь мы встречаемся с таким интересным явлением как погружение данной задачи в параметризованное множество задач. Необходимо решить задачу с параметрами $k = 0$, $m = 0$, но удобным оказывается рассматривать всё множество задач рекурсивно выразить задачи из этого множества через другие.

Метод параметризации задачи заключается в поиске параметризованного множества задачи, одна из которых есть та задача, которую нам нужно решить. Среди задач в этом множестве есть элементарные, решение которых известно, и есть способ сведения более сложных к более простым. Осуществляя такое сведение можно данную задачу свести к более простым и так далее до элементарных.

Второй вариант алгоритма, представленный в псевдокоде 11.26 следует той же идее, только входные переменные функции Do сделаны *глобальными*. Перед вызовом функции Do следует

Алгоритм 11.26 Генерация всех правильных скобочных выражений (вариант 2).

```
▷ Глобальные переменные:
▷   n – четное число, равное длине скобочного выражения;
▷   a[0..(n − 1)] – массив скобок;
▷   m – длина построенного префикса скобочного выражения a[0..(m − 1)];
▷   k – текущее число незакрытых скобок в префиксе a[0..(m − 1)].
function Do
    if m = n and k = 0 then
        PrintWord(a)
    end if
    if n − m > k then
        a[m] ←′ (′,
        k ← k + 1, m ← m + 1;
        Do
        k ← k − 1, m ← m − 1;
    end if
    if k > 0 then
        a[m] ←′)′,
        k ← k − 1, m ← m + 1;
        Do
        k ← k + 1, m ← m − 1;
    end if
end function
function GenerateBracketWords(l)
    n ← l, m ← 0, k ← 0
    Do()
end function
```

устанавливать переменные в нужное значение, а после вызова — восстанавливать их прежнее значение.

Третий вариант 11.27 алгоритма основан на принципиально другой идее. Он не содержит рекурсивных функций, и основан на возможности ввести такой линейный порядок на множестве перебираемых объектов, что функция вычисления следующего элемента по известному предыдущему проста. В нашем примере эта функция имеет имя NextBracketWord.

Три разобранные подхода к перебору правильных скобочных выражений применимы и к другим задача перебора, например, перебору всех перестановок, всех k-элементных подмножеств данного n-элементного множества, всех разложений числа на сумму неупорядоченных слагаемых, и др.

Рассмотрим задачу генерации всех разложений числа n на суммы неупорядоченных слагаемых. Для $n = 5$ ответ будет состоять из 7 вариантов: $5 = 5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 + 1 + 1 + 1$. Предложите параметризацию этой задачи, которая позволит эффективно решить её с помощью рекурсивной функции.

Одна из подходящих параметризаций этой задачи следующая: Generate(a, n, m, k) — сгенерировать разложение числа n на слагаемые, которые не превосходят k и поместить их в

Алгоритм 11.27 Генерация всех правильных скобочных выражений (вариант 3)

```
function NextBracketWord(a, n)
    c ← 0
    i ← n − 1
    while ( do a[i] =')' and a[i − 1] =' ('
        i ← i − 2
    end while
    while a[i] =')' and i > 0 do
        i ← i − 1
        c ← c + 1
    end while
    if i = 0 then
        return false
    end if
    a[i] ← ')', i ← i + 1, c ← c − 1
    m ← (n − i − c)/2
    for j ∈ {i, i + 1, ..., i + m} do
        a[j] ← '('
    end for
    for j ∈ {i + m + 1, ..., n − 1} do
        a[j] ← ')'
    end forreturn true
end function
function GenerateBracketWords(n)
    a ← '((((...))))))'                    ▷ n/2 открывающих и n/2 закрывающих скобок
    repeat
        PrintWord(a, n);
    until not NextBracketWord(a, n)
end function
```

массив a в ячейки k, $k + 1$, ...

Семинар 11

Рекурсивные и нерекурсивные алгоритмы перебора множеств

Краткое описание: Мы рассмотрим рекурсивные алгоритмы перебора различных комбинаторных объектов. Одной из основных задач будет задача генерации всех правильных скобочных выражений длины $2n$. Будут разобраны также задачи: проверка скобочного выражения на правильность, вычисление факториала, генерация всех m-элементных подмножеств первых n натуральных чисел и задача

Перебор подмножеств

Задача С11.1. Изучите программы 11.1 и 11.2, Каким образом можно отождествить все подмножества первых n натуральных чисел и все бинарные слова длины n? Внесите в одну из указанных программ небольшие изменения так, чтобы она генерировала все m-элементные подмножества первых n натуральных чисел. Подсказка: заведите переменную t, которая отвечает за число взятых чисел из промежутка $[1..k]$ и строчку, в которой `a[..]` присваивается 1, окружите командами `t++` и `t--`. Добавьте также два условных оператора, чтобы
1) элемент не брался, если уже взято m элементов,
2) элемент точно брался, если осталось набрать еще t элементов и ровно столько элементов и есть возможность взять.

Программа 11.1: Вывод всех бинарных слов длины n

```
int k = 0;
int n = 10;
int a[N];
void go() {
    if(k == n) {
        print_word();
    }
    k++;
    a[k-1] = 0; go();
    a[k-1] = 0; go();
    k--;
}
void print_word() {
    int i;
```

```
    for(i = 0 ; i < n ; i++) {
        print("%d",a[i]);
    }
    printf("\n");
}
```

Программа 11.2: Вывод всех подмножеств первых n натуральных чисел без использования глобальных переменных

```
#include <malloc.h>
void print_subset(int *a, int n) {
    int i;
    for( i = 0 ; i < n ; i++ ) {
        if( a[i] ) {
            printf("%d ", i);
        }
    }
    printf( "\n" );
}
void go(int *a, int k, int n) {
    if( k == n ) {
        print_subset();
    } else {
        a[k] = 0; go(a, k+1, n);
        a[k] = 1; go(a, k+1, n);
    }
}
void generate(int n) {
    int *a = (int*) malloc( sizeof(int) * n );
    go( a, 0, n );
    free( a );
}
```

Задача C11.2. Вывести все m-элементные подмножества множества $\{1, \ldots, n\}$.

Задача C11.3. Вывести все последовательности (упорядоченные подмножества) длины m из чисел $1, \ldots, n$.

Задача C11.4. Вывести все разбиения множества $\{1, \ldots, n\}$ на три непустых подмножества.

Перебор правильных скобочных выражений

Задача C11.5. Напишите программу, которая генерирует все правильные скобочные выражения длины $2n$. Необходимо, чтобы программа работала время, пропорциональное числу правильных скобочных выражений длины $2n$ (программы, работающие время $O(2^n)$ не принимаются). Подсказка: введите переменную, которая равна количеству не закрытых открытых

скобок (открытых скобок, для которых не нашлось парной закрывающей скобки) на участке с нулевого по k-й символ.

Программа 11.3: Вывод всех правильных скобочных выражений заданной длины.

```
#include <stdio.h>
void generate(int n, int k) {
    char *a = (int*) malloc( sizeof(char) * (n+1) );
    a[n] = 0;
    go (a, 0, 0, n);
    free (a);
}
void go(char *a, int k, int m, int n) {
    if(k == n) {
        printf("%s\n", a);
    } else {
        if( m < n-k ) {
            a[k] = '('; go(a, k+1, m+1, n);
        } else
        if( m > 0 )   {
            a[k] = ')'; go(a, k+1, m-1, n);
        }
    }
}
```

Перебор перестановок

Программа 11.4: Рекурсивный метод генерации перестановок.

```
#include <malloc.h>

void print_perm(int *a, int n) {
    int i;
    for( i = 0; i < n; i++ )
        printf( "%3d ", a[i] );
    printf( "\n" );
}
void generate(int *a, int *b, int i, int n) {
    if ( i == n ) {
        print_perm (a, n);
    } else {
        int j;
        for( j = 0; j < n; j++ ) {
            if( b[j] == 0 ) {
                b[j] = 1; a[i] = j+1;
                generate(a, b, i + 1, n);
```

```
                b[j] = 0;
            }
        }
    }
}
int main() {
    int *a, *b;
    int n;
    scanf( "%d", &n );
    a = (int*) malloc ( n * sizeof(int) );
    b = (int*) malloc ( n * sizeof(int) );
    generate ( a, b, 0, n );
    free ( a );
    free ( b );
}
```

Программа 11.5: Нерекурсивный алгоритм генерации перестановок.

```
#include <malloc.h>
void print_perm(int *a, int n) {
    int i;
    for(i = 0; i < n; i++)
        printf("%3d ", a[i]);
    printf("\n");
}
int next(int *a, int n) {
    // a[0]...a[n-1] содержит некоторую
    // перестановку чисел 1, ... ,n
    int  k, t, i, tmp;
    // находим k: a[k] < a[k+1] >...>a[n-1]
    for (k = n - 2; k >= 0 && a[k] > a[k+1]; k-- );
    if (k == -1)
        return 0; // это последняя перестановка

    // находим t > k, такое что a[t] - минимальное
    // число большее a[k] среди a[k+1]...a[n-1]
    for (t = k + 1; t < n - 1 &&  a[t+1] > a[k] ; t++);

    // меняем местами a[k] и a[t]
    tmp = a[k]; a[k] = a[t]; a[t] = tmp;

    // участок массива a[k+1] ... a[n-1]
    // записываем в обратном порядке
    for(i = k + 1; i <= (n + k) / 2; i++)   {
        t = n + k - i;
        tmp = a[i]; a[i] = a[t]; a[t] = tmp;
    }
```

```
    return 1;
}
int main() {
    int *a, n, i;
    scanf( "%d", &n );
    a = (int*) malloc( n * sizeof(int) );

    for( i = 0; i < n; i++ ) a[i] = i + 1;

    do {
        print_perm( a, n );
    } while ( next(a, n) );
    return 0;
}
```

Задача С11.6. Напишите программу, которая генерирует все перестановки первых n натуральных чисел.

Задача С11.7. Напишите программу, которая генерирует все перестановки первых n натуральных чисел, в которых ни один из элементов не остался на своем месте.

Перебор разложений числа в сумму

Задача С11.8. Вывести все разложения числа n в сумму положительных слагаемых слагаемых (порядок слагаемых важен). Например, для $n = 3$ — это будут $1+1+1$, $1+2$, $2+1$, 3.

Задача С11.9. Вывести все разложения числа n в сумму положительных слагаемых слагаемых (порядок слагаемых не важен). Например, для $n = 4$ — это будут $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4.

Лекция 12

Структуры данных: метод деления пополам, двоичное дерево поиска

Краткое описание: В данной лекции мы сформулируем общую проблему построения эффективных хранилищ данных. Рассмотрим задачу «телефонная книжка» (задачу реализации интерфейса «ассоциативный массив»). Разберём несколько простых решений этой задачи, основанных на массивах и списках. Изучим структуру данных «двоичное дерево поиска» и проведём анализ эффективности этой структуры данных для решения поставленной задачи.
Ключевые слова: интерфейс, ассоциативный массив, двоичное дерево поиска, случайного двоичное дерева поиска, высота дерева.

Мы переходим к важной теме:

«Структуры данных и методы эффективного хранения данных».

Проблема в общем виде формулируется следующим образом:

Фундаментальная проблема 1: Построение эффективного хранилища данных. Спроектировать хранилище данных, которое позволяет хранить большие объёмы данных и предоставляет возможность быстро находить и модифицировать данные.

Какая из структур данных наиболее эффективна для конкретной задачи, определяется набором запросов и их относительной частотой. Под запросом понимается команда извлечения некоторой части данных, команда добавления новых данных или команда изменения существующих данных.

Мы рассмотрим базу данных с очень простой структурой. В базе данных присутствует только одна таблица, в которой хранятся записи телефонной книжки, а именно пары вида (имя, телефон). К базе данных приходят следующие запросы[1]:

- «добавить строчку (имя, телефон)»;
- «удалить запись с именем X»;
- «извлечь запись с именем X».

Мы рассмотрим различные реализации «телефонной книжки», основанные на следующих структурах данных:

- неупорядоченный и упорядоченный массив;

[1] **Запрос** — это команда на добавление, извлечение, удаление или модификацию данных.

- неупорядоченный список;
- двоичное дерево поиска;
- сбалансированные деревья: АВЛ-дерево и красно-чёрное дерево;
- хэш-таблица.

Постановка задачи

Задача Л12.1. (Телефонная книжка) Необходимо написать программу, которая реализует интерфейс[2] «телефонной книжки»:

Необходимо выполнять следующие запросы:

запрос	**описание**
`insert (name, number)`	Добавить новую запись в хранилище. Коды возврата: • `inserted` – успешно добавлена, • `error_memory` – не хватает места, • `updated` – запись с таким именем существует, номер телефона обновлен.
`del (name)`	Удалить запись с именем `name`. Коды возврата: • `deleted` – успешно удалена, • `error_notfound` – запись с таким именем отсутствует.
`find (name)`	Найти запись с именем `name`. Коды возврата: • `found` – успешно найдена, • `error_notfound` – запись с таким именем отсутствует. Результат: указатель на запись (имя, телефон).

Запросы данных трёх типов приходят в случайном порядке примерно одинаковой относительной частотой.

[2] *Интерфейс* (англ. *interface*) — это множество функций, предоставляемых модулем (библиотекой функций, классом, структурой данных, ..), или, в более общем случае, описание функциональности, которую данный модуль предоставляет. Можно сказать, что интерфейс — это абстракция, обозначающая *функциональность в числом виде.*

Интерфейс библиотек функций включает в себя описание сигнатур функций и описание семантики функций.

Сигнатура функции — это имя функции, тип возвращаемого значения и список аргументов с указанием их типов.

Семантика функции — это описание того, что данная функция делает. Семантика функции включает в себя описание того, что является результатом вычисления функции, как и от чего этот результат зависит. Обычно результат выполнения зависит только от значений аргументов функции, но в модулях могут быть и общедоступные переменные. Результат функции может зависит от значений этих переменных, и, кроме того, результатом может стать изменение значений этих переменных. Логика этих зависимостей и изменений относится к семантике функции. Полным описанием семантики функций является исполняемый код функции или математическое определение функции.

Необходимо реализовать хранилище для указанных данных в памяти программы. Число записей может достигать 10 000 000. Все запросы должны выполняться менее, чем за одну секунду.

Определение 12.1. *Интерфейс «ассоциативный массив» — это интерфейс хранилища множества пар (key, value) с тремя операциями* `insert`, `find`, `del` *для добавления, поиска и удаления пар.*

Интерфейс «Телефонная книжка» является частным случаем интерфейса «ассоциативный массив», когда тип ключей есть слово, а значение — набор цифр. На ассоциативный массив можно смотреть как на массив, в котором роль индекса могут играть произвольные объекты (числа с плавающей точкой, слова, структуры и др.), а не только целое число из некоторого фиксированного конечного промежутка целых чисел.

Заметим, что если одна запись имеет размер 32 байта, то суммарный размер записей может достигать 320 мегабайт. Такого объёма данные помещаются в оперативную память современных компьютеров, потому логично «телефонную книжку» хранить в памяти. Оперативная память компьютера работает быстрее, чем жесткие диски и другие устройства постоянного хранения данных, — это существенный «плюс» оперативной памяти. Но оперативная память временна — данные, которые в ней хранятся, относятся к одной из выполняемых программ, и при завершении этой программы (и тем более, при выключении компьютера) теряются. Одним из разумных решений является программа, которая в начале работы загружает «телефонную книжку» из файла в память, затем обрабатывает запросы пользователя, а по команде выхода из программы записывает обновленную телефонную книжку обратно на диск[3]

Пусть N — число записей в нашем хранилище. Среднее время, которое будет тратится на выполнение одного запроса, может зависеть от N. Желательно, чтобы это время медленно увеличивалось ростом N. Сделать так, чтобы время обработки запроса не увеличивалось, невозможно. Существует определённый баланс между временем добавления (изменения) данных и временем извлечения данных. Часто можно так изменить структуру хранилища и модифицировать алгоритмы извлечения и добавления данных, что уменьшиться один из этих показателей, но второй при этом увеличится.

Мы получим следующие асимптотики для среднего времени, необходимого на выполнение

[3]Современные системы управления базами данных (СУБД) хранят данные на жестком диске, но постоянно подгружают все или часть данных в оперативную память (это называется **кэшированием данных**), чтобы быстрее обрабатывать приходящие запросы. В этом случае возникает проблема синхронизации данных на диске и данных в оперативной памяти. Все запросы на изменение данных также можно фиксировать только в оперативной памяти, и лишь периодически записывать эти изменения на жесткий диск. Важной практической задачей является защита баз данных от аварийных завершений — необходимо, чтобы данные в базе оставались целостными (консистентными) даже в случае аварийных остановок (перебой питания) и неожиданных ошибок исполнения. Обычно используется метод периодической синхронизации данных в оперативной памяти и данных на диске и метод журналирования запросов между моментами синхронизации.

запросов:

Структура хранилища	`insert`	`del`	`find`
Неупорядоченный массив	1	N	N
Упорядоченный массив	N	N	$\log N$
Неупорядоченный список	1	N	N
Двоичное дерево поиска	H	H	H
Сбалансированные деревья	$\log N$	$\log N$	$\log N$

Определение 12.2. **Ключём** *называют поле (или комбинацию полей), которое уникально для каждой записи. Это в означает, что ключ однозначно идентифицирует запись. Например, фамилия человека в телефонном справочнике не может быть ключом, так как два разных человека могут иметь одинаковые фамилии, а пара полей (ФИО, адрес) — может.*

Неупорядоченный массив

Самый простой способ организации хранилища — это хранить все записи (в памяти или на диске) подряд друг за другом, то есть в виде массива, и не стремится их как-либо упорядочить.

Новую запись будем добавлять в конец массива. Эта операция будет выполняться быстро, а главное она будет выполняться за некоторое конечное ограниченное время[4], независимое от N.

При таком способе добавления записи будут хранится друг за другом в порядке добавления.

Перейдём к операциям поиска `find` и удаления `del`. Самый простой и естественный алгоритм поиска записи с нужным нам ключём в неупорядоченном массиве такой: перебираем последовательно записи и сравниваем ключи записей с нужным нам ключём.

Этот алгоритм переберёт в среднем половину записей, а в худшем — все записи. То есть среднее и худшее время работы работы алгоритма растёт пропорционально N.

Грубая оценка скорости современных компьютеров такая: за одну секунду просматривается 1 000 000 записей (если записи находятся в оперативной память, а ключи представляют собой строки).

Если у нас 10 000 000 записей, то операция поиска может занять 10 секунд, а это много.

Среднее и худшее время удаления записи также пропорционально N. Чтобы удалить запись, необходимо сначала её найти и потратить время, пропорциональное N. Затем нужно убрать образовавшуюся пустоту. Это можно осуществить за константное время, взяв последний элемент массива и переместив его в ячейку, в которой находился удалённый элемент.

Получаем следующий результат:

Структура хранилища	`insert`	`del`	`find`
Неупорядоченный массив	1	N	N

Для нас важно лишь то, что время на выполнение операций поиска и удаления растёт с N *линейно*. Коэффициент перед N не так важен[5], поэтому мы его не пишем.

[4]Если время выполнения операции ограничено некоторым фиксированным числом и не зависит от параметра размера задачи N, то говорят, что время выполнения операции есть $O(1)$.

[5]Этот коэффициент пропорциональности мы просто не можем вычислить, не указав конкретную вычислительную машину и конкретную реализацию алгоритма.

Упорядоченный массив

Идея 12.1. *Если записи упорядочены, то искать проще.*

Действительно, если фамилии в журнале упорядочены по алфавиту, то найти нужную фамилию проще: смотрим в середину столбца и узнаем сверху или снизу находится нужная нам фамилия. В нужной нам части снова смотрим примерно в середину и опять узнаём, куда нам нужно вести глаза — вниз или вверх. В конце концов мы находим либо нужную нам фамилию, либо пару фамилий, между которыми она должна стоять.

Этот метод поиска называется **поиск методом делением пополам (binary search)** или **дихотомия**.

Пусть искомая запись имеет индекс $i \in [l, r)$. Программа двоичного поиска (поиска методом деления пополам) в массиве $data$ нужной записи с ключём x выглядит так:

```
while r − l > 1 do
    c ← (l + r)/2
    if data[c]→key < x then
        r ← c
    else
        l ← c
    end if
end while
return l
```

Каждая итерация **while** уменьшает область поиска $[l, r)$ примерно в два раза. Если в начале у нас 100 элементов, то на следующем шаге будет 50, а на следующем — 25 и так далее. Так как $2^{10} = 1024 \approx 10^3$, то поиск в упорядоченном массиве из 10^6 элементов требует 20 итераций, а если элементов 10^7, то требуется 24 итерации ($\log_2 10^7 = 23.253\ldots$). Понятно, что 24 итерации цикла в описанном алгоритме на современных компьютерах выполняются мгновенно[6].

Приведённый итеративный алгоритм поиска методом деления пополам можно переписать, используя рекурсию (см. псевдокод 12.28).

Метод деления пополам используется в вычислительной математике для численного вычисления корней уравнений. На рисунке 12.1 показан график некоторой функции $f(x)$, которая имеет ноль на на отрезке $[l,\ r]$. Известно, что если на концах отрезка $[l,\ r]$ непрерывная функция принимает разные значения, то на этом отрезке есть нуль функции. Более того, если функция монотонно растёт, то этот нуль единственный.

Алгоритм вычисления нуля итеративный. На каждой итерации алгоритма мы делим исходный отрезок пополам и среди двух отрезков находим тот, который обладает необходимым

[6]Обратите внимание на то, что в качестве области поиска рассматривается полуинтервал, а не отрезок или интервал. Это более удобно, так как полуинтервал можно разбить на два непересекающихся полуинтервала:

$$[l,\ r) = [l, c) \cup [c, r),$$

в то время как отрезок нельзя разбить на два отрезка и интервал нельзя разбить на два интервала.

В двоичном поиске у начального полуинтервала нужно указывать такую правую границу, которая заведомо больше искомого числа. Понимание того, что область поиска есть полуинтервал позволяет легко определить, какой знак необходимо ставить в условии внутри цикла — «меньше» или «меньше либо равно».

Алгоритм 12.28 Рекурсивная функция для поиска методом деления пополам

```
    function BinarySearch(data, x, l ,r)
  if r − l > 1 then
        return l
    else
        c ← (l + r)/2
        if data[c].key < x then
            return Binary-Search(data, x, l, c)
        else
            return Binary-Search(data, x, c, r)
        end if
    end if
end function
```

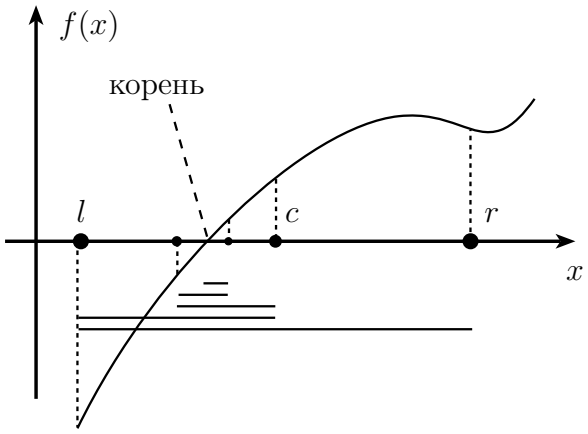


Рис. 12.1: Вычисление корня уравнения методом деления пополам.

свойством, а именно, функция на его концах имеет значения разных знаков. Продолжаем процесс деления пока размер отрезка не станет меньше точности, которая нас удовлетворит. Под осью Ox на рисунке показано, как с каждым шагом область поиска сужается в два раза[7].

Давайте теперь посмотрим, сколько нам потребуется времени на операции добавления и удаления записи при использовании упорядоченного массива.

При удалении в упорядоченном массиве можно быстро (методом деления попалам) найти удаляемую запись. Но затем нужно выполнить в среднем $N/2$ операций сдвига элементов для удаления образовавшейся пустоты в массиве. Поэтому асимптотика среднего и худшего

[7]Аналогия между поиском в отсортированном массиве и алгоритмом вычисления корня уравнения $f(x) = a$ становится полной, если рассматривать отсортированный массив чисел как дискретный вариант монотонной функции. Даже если ключи являются словами в некотором алфавите $B = \{a, b, c, \ldots\}$, то аналогия сохраняется. В алгоритме нигде не требуется того, чтобы область значения функции f было множеством чисел. Необходимо только, чтобы на области значений функции была введена операция сравнения. Несложно и явно отобразить слова в действительные числа на промежутке $[0, 1)$. Для этого буквы $\{a,\ b,\ c,\ \ldots\}$ необходимо рассматривать как цифры $\{1,\ 2,\ 3,\ \ldots\}$, и слову $b_1b_2b_3\ldots b_m$ из m букв поставить в соответствие число, которое записывается как $(\overline{0.b_1b_2b_3\ldots b_m})_q$ в системе счисления с основанием $q = |B| + 1$.

времени операции удаления будет прежняя — $O(N)$.

Операция добавления записи в упорядоченный массив также оказывается трудоёмкой, так как необходимо сохранить свойство упорядоченности.

Сначала ищется промежуток, в который следует поместить добавляемую запись. а затем следует переместить следующие за промежутком элементы, чтобы освободить ячейку для новой записи. Для этого последний элемент перемещается в следующую за ним пустую ячейку, затем на его место перемещается предпоследний элемент, на место предпоследнего перемещается предпредпоследний и т.д. пока не доберёмся до нужного промежутка.

В среднем получим $N/2$ перемещений и асимптотика среднего и худшего времени выполнения операции добавления элемента равна $O(N)$.

Таким образом, при использовании упорядоченных массивов не появляется значительного улучшения — поиск стал быстрее, но существенно увеличилось время добавления новой записи:

Структура хранилища	`insert`	`del`	`find`
Упорядоченный массив	N	N	$\log N$

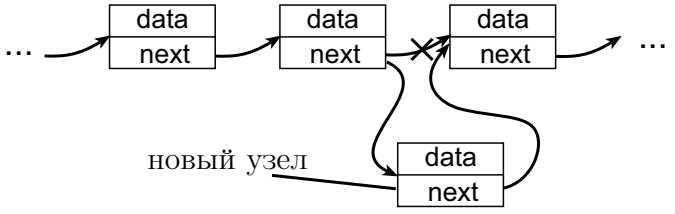
Вывод: хранить записи в упорядоченном массиве эффективно только тогда, когда они не меняются, то есть запросы `del` и `insert` не приходят или приходят очень редко.

Неупорядоченный список

Идея 12.2. *При использовании массивов мы много времени тратим на перемещение «хвоста» массива в операциях добавления и удаления элемента. Этого можно избежать, если использовать списки.*

Вспомним определение списка.

Определение 12.3. **Список** — *это структура данных для хранения последовательности элементов. Элементы списка в памяти расположены не строго друг за другом, а некоторым неизвестным образом. Порядок расположения в адресном пространстве не так важен, так как каждый элемент списка кроме данных содержит указатель на место в памяти, где хранится следующий элемент. Это позволяет по первому элементу получить доступ ко всем элементам последовательности. Список называется* **двусвязным** *если каждый элемент содержит также информацию о том, где находится предыдущий элемент. Если в элементе хранится информация только о следующем элементе, то список называется* **односвязным**.



Для того, чтобы вставить элемент внутрь односвязного списка, нужно разорвать одну стрелочку и добавить две новых Для двусвязного списка нужно разорвать две стрелочки и добавить четыре.

По двусвязному списку мы можем перемещаться по элементам вперед и назад, но мы не можем быстро перейти к середине списка. Для того, чтобы перейти к среднему элементу с порядковым номером $N/2$, нужно взять первый элемент и $N/2$ раз перейти к следующему элементу. Как мы видим, хранить записи в

упорядоченном списке смысла нет — в списке нет возможности осуществлять поиск методом деления пополам.

Итак, операция добавления элемента к списку занимает время $O(1)$. Операции поиска и удаления элемента займут в среднем время $O(N)$, то есть столько же, сколько и в неупорядоченном массиве. Но неупорядоченный список несколько лучше неупорядоченного массива, так как при выполнении операции удаления не нужно осуществлять большое количество операций перемещения.

Структура хранилища	`insert`	`del`	`find`
Неупорядоченный массив	1	N	N
Упорядоченный массив	N	N	$\log N$
Неупорядоченный список	1	N	N

Двоичное дерево поиска

Идея 12.3. *В записях можно хранить информацию о местоположении других записей — указатели на место в памяти, где они хранятся. С помощью этих указателей можно реализовать в памяти произвольный граф, в вершинах (узлах) которого хранятся произвольные данные, а указатели играют роль направленных рёбер графа.*

Эта идея позволяет создавать в памяти компьютера самые разнообразные структуры данных — всевозможные графы, к вершинам и рёбрам которых прикреплены какие-то данные.

Первое простейшее применение этой идеи — структура данных «двоичное дерево». Это структура данных непосредственно связана с понятием «двичное дерево» из теории графов[8]:

Определение 12.4. **Ориентированное дерево** — *это ориентированнный граф без циклов, в котором входящие степени всех вершин, кроме одной, равны* 1. *Вершина дерева, входящая степень которой равна* 0, *называется* **корнем дерева**. *Вершины двоичного дерева, исходящая степень которых равна* 0 *(из которых не выходят рёбра) называются* **листьями дерева**.

Определение 12.5. **Ориентированное двоичное дерево** — *это ориентированное дерево, в котором исходящие степени вершин равны* 0, 1 *или* 2.

Определение 12.6. **Структура данных «двоичное дерево»** — *это множество записей вида* $(data, l, r)$, *где* $data$ — *некоторые данные,* l *и* r — *указатели на две другие записи. Записи называются* **узлами дерева (tree nodes)**. *Узлы, на которые указывают* l *и* r, *называются левым и правым ребенком (детьми) узла* $n = (data, l, r)$. *Указатели* l *и* r *могут быть пустыми (равными специальной константе* `nil`*). Совокупность узлов структуры данных «двоичное дерево» образуют граф, в котором вершинами являются узлы, а рёбрам соответствуют указатели* l *и* r. *Этот граф должен быть* ориентированным двоичным деревом, *то есть ориентированным графом, в котором из каждой вершины (узла) выходит* 0, 1 *или* 2 *ребра, и в каждую вершину входит ровно одно ребро. Исключение составляет одна вершина (узел), в которую не входит ни одно ребро. Она называется* **корнем дерева**.

[8]Введение в терминологию теории графов дано в лекции 17.

На рисунке узел `m` есть родитель узлов `e` и `s`; узел `e` — его левый ребенок, узел `s` — правый.

Обычно двоичное дерево изображают так, что стрелки, указывающие на детей, направлены вниз. Корень дерева — это самая верхняя вершина (в теории алгоритмов деревья растут вниз, а не вверх).

Пусть n — некоторый узел дерева. **Поддеревом с корнем** n назвается дерево, образованное из узла n и всех его «**потомков**», то есть детей, детей детей, всех их детей и так далее.

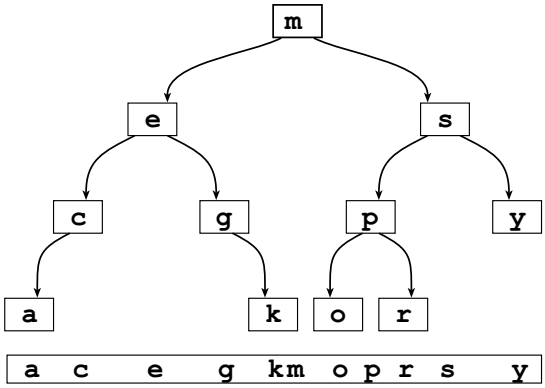


Рис. 12.2. Двоичное дерево поиска

Определение 12.7. **Структура данных «двоичное дерево поиска»** — *это структура данных «двоичное дерево», в которой*

- *записи содержат поле «ключ» (key), и на множестве значений этого поля введён линейный порядок;*
- *выполнено следующее свойство: для любого узла p ключи всех узлов из правого поддерева (поддерева, корнем которого является правый ребёнок) больше ключа узла p, а ключи всех элементов из левого поддерева – меньше ключа узла p; в частности,*

$$p{\to}l{\to}key \;<\; p{\to}key \;<\; p{\to}r{\to}key,$$

то есть ключ родителя находится между ключами левого и правого детей.

Операции добавления и поиска записей в двоичном дереве довольно просты и обычно реализуются рекурсивно:

`insert`: Пусть нужно добавить запись $(key, value)$ в поддерево с корнем n. Посмотрим, чему равен ключ $n{\to}key$ у узла $n{\to}key$:
1) если $key = n{\to}key$, то помещаем данные $(key, value)$ в узел n;
2) если $key > n{\to}key$, и правое поддерево не пусто (есть правый ребёнок), то вызываем процедуру добавления данной записи в правое подерево; если правого ребёнка нет, то создаём его (выделяем под него память, в языке Си это делается с помощью функции `malloc`), помещаем в него данные $(key, value)$ (поля l и r устанавливаем в значение `nil`) и устанавливаем на него указатель $n{\to}r$;
3) если ключ key меньше $n{\to}key$, и левое поддерево не пусто (есть левый ребенок), то вызываем процедуру добавления данной записи в левое подерево; если левого ребёнка нет, то создаём его, помещаем в него данные $(key, value)$ и устанавливаем на него указатель $n{\to}l$.

`find`: Пусть нужно найти запись с ключом key в поддереве с корнем n. Посмотрим, чему равен ключ $n{\to}key$ у узла n:
1) если $key = n{\to}key$, то данные в узле n и есть то, что мы ищем;
2) если $key > n{\to}key$, и правое поддерево не пусто (есть правый ребёнок), то вызываем процедуру поиска по ключу key в правом подереве;

3) если $key < n \to key$, и левое поддерево не пусто (есть левый ребёнок), то вызываем процедуру поиска по ключу key в левом подереве. 4) если не одно из этих условий не подошло, то возвращаем результат `nil`(или `NOT_FOUND` — запись с данным ключём отсутствует).

На рисунке 12.2 изображено двоичное дерево поиска, в котором роль ключей играют латинские буквы. На ключах введён порядок — алфавитный порядок. Квадратные блоки на этом рисунке обозначают узлы дерева, а буквы в них — значения ключей. Корнем дерева является узел с ключём `m`.

Задача Л12.2. Предложите порядок добавления записей с ключами {`a`, `c`, `e`, `g`, `k`, `m`, `o`, `p`, `s`, `r`, `y`}, чтобы получилось двоичное дерево изображенное на рисунке 12.2. Единственно ли решение?

Если аккуратно нарисовать двоичное дерево поиска так, что для любого узла все узлы правого поддерева находятся правее этого узла, а все узлы левого — левее, то после проецирования ключей на горизонтальную прямую мы получим возрастающую последовательность ключей.

Определение 12.8. **Глубиной узла** n $(d(n))$ *дерева называется число рёбер в пути от корня дерева до узла* n. **Высотой дерева** (H) *называется глубина самого глубокого листа:* $H = \max_n d(n)$. **Средней глубиной узлов дерева** $(\overline{d})$ *называется среднее арифметическое глубин узлов.* **Средней глубиной листьев дерева** $(\overline{h})$ *называется среднее арифметическое глубин листьев.*

Утверждение 12.1. *Описанные рекурсивные процедуры добавления и поиска для двоичного дерева поиска работают в среднем время* $O(\overline{d})$*, где* $\overline{d}$ *— средняя высота узлов дерева. Время выполнения этих процедур ограничено линейной от высоты дерева функцией, то есть* $O(H)$.

Действительно, цепочка рекурсивных вызовов имеет длину, равную числу шагов от корня дерева, до узла, в котором хранятся искомые данные, или до узла, в который будут помещены добавляемые данные. Но каждом шаге выполняется фиксированное ограниченное количество элементарных действий, а значит время каждого шага есть $O(1)$. В худшем случае число шагов равно глубине самого глубокого узла, то есть высоте дерева H, а в среднем — средней глубине узлов $\overline{d}$.

Удаление элементов из двоичного дерева поиска

Операция удаления элементов из двоичного дерева несколько сложнее операций поиска и добавления.

Один из простейших способов реализации этой операции заключается в том, чтобы реально элементы не удалять, а лишь помечать их как удалённые. К записям, хранимых в узлах дерева, добавляется поле `is_deleted`, которое устанавливается в значение `true`, когда запись удаляется:

`del`: Пусть нужно удалить запись с ключём key из поддерева с корнем n. Посмотрим, чему равен ключ $n \to key$ у узла n:
1) если $key = n \to key$, то устанавливаем $n \to is\_deleted = true$;
2) если $key > n \to key$, и правое поддерево не пусто (есть правый ребёнок), то вызываем процедуру удаления из правого подерева;

3) если $key < n \to key$, и левое поддерево не пусто (есть левый ребёнок), то вызываем процедуру удаления из левого подерева;

Кроме того, необходимо слегка модифицировать логику функций `insert` и `find`:

`insert`: Пусть нужно добавить запись с ключом x в поддерево с корнем n. Посмотрим, чему равен ключ $n \to key$ у узла n:
1) если $key = n \to key$, то помещаем данные $(key, value)$ в узел n и устанавливаем $n \to is\_deleted = false$;
2) ...
3) ...

`find`: Пусть нужно найти запись с ключом key в поддереве с корнем n. Посмотрим, чему равен ключ $n \to key$ у узла n:
1) если $key = n \to key$ и $n \to is\_deleted = false$, то данные в узле n и есть то, что мы ищем;
2) ...
3) ...

Этот способ удаления плох тем, что размер базы не уменьшается после удаления элементов. Но его можно модифицировать следующим образом:

- если удаляется лист дерева, то его можно удалить по настоящему;
- если удаляется элемент, у которого только один ребенок, то значения всех его полей $(key, value, l, r)$ можно переписать значениями соответствующих полей его единственного ребёнка, а самого ребёнка физически удалить из памяти (в языке Си это делается с помощью вызова функции освобождения памяти `free`).

Мы не можем удалить узел физически только в одном случае – когда у него присутствуют оба ребёнка.

Задача Л12.3. Покажите, что после указанных оптимизаций число узлов в дереве дерева всегда будет не более чем в два раза больше числа хранимых в нём записей (узлов, для которых поле $is\_deleted = false$).

В случае, когда оба ребёнка присутствуют можно провести удаление следующим образом:

- найдём узел m, являющийся самым левым узлом правого поддерева;
- скопируем значения полей $(key, value, l, r)$ узла m в соответствующие поля узла n;
- правого ребёнка узла m (если такой есть) сделаем ребёнком родителя m (вместо m).
- освободим память, занимаемую узлом m (в языке Си это делается с помощью вызова функции `free`).

Задача Л12.4. Может ли в приведённом алгоритме узел m быть правым ребёнком своего родителя?

Для двоичного дерева поиска операции удаления, поиска и добавления пары $(key, value)$ выполняются в среднем время $O(\overline{d})$, где $\overline{d}$ — средняя высота узлов дерева.

Итак, получаем следующую картину эффективности различных структур данных для различных типов запросов:

Структура хранилища	`insert`	`del`	`find`
Неупорядоченный массив	1	N	N
Упорядоченный массив	N	N	$\log N$
Неупорядоченный список	1	N	N
Двоичное дерево	H	H	H

Анализ эффективности двоичного дерева поиска

Величину средней высоты узлов дерева H необходимо как-то оценить через число хранимых в нём элементов N. Понятно, что чем лучше дерево «ветвится», тем больше в нём содержится элементов при фиксированной средней высоте.

Если пары $(key, value)$ будут добавляться в дерево поиска в порядке возрастания ключей, то дерево будет состоят из цепочки узлов, в которой каждый узел является левым ребёнком предыдущего узла.

Дерево хорошо будет «ветвиться», если порядок ключей более-менее случайный. Если же в последовательности добавляемых ключей есть какая-либо закономерность, дерево может оказаться «слабо ветвящимся». Например, на рисунке 12.3 показано двоичное дерево поиска, полученное из пустого дерева последовательным добавлением записей с ключами 10, 20, 30, 40, 50, 9, 8, 7, 6, 11, 12, 13.

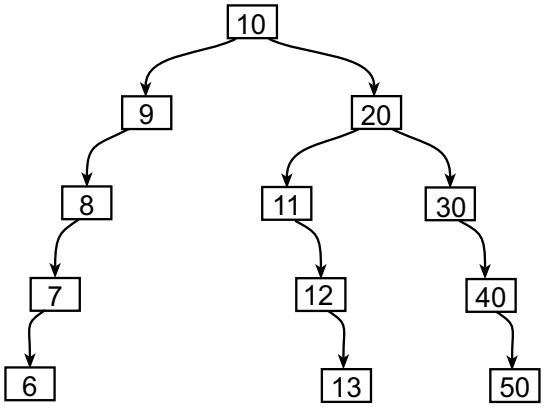


Рис. 12.3: Пример «слабо ветвящегося» бинарного дерева поиска.

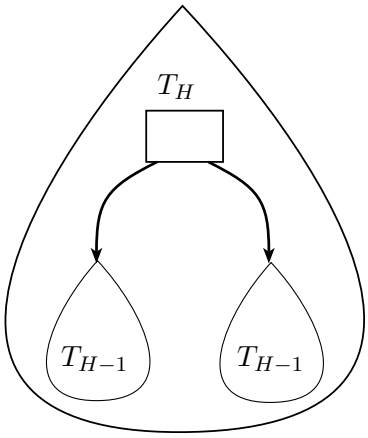


Рис. 12.4: Рекурсивное определение полного двоичного дерева высоты H.

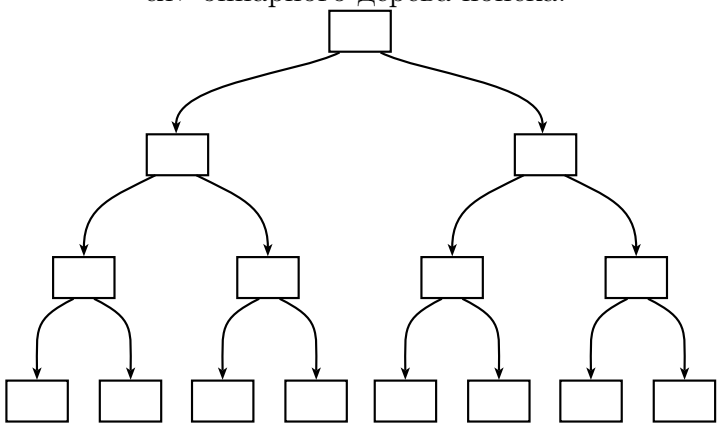


Рис. 12.5: Полное двоичное дерево T_3 высоты $H = 3$.

Введём понятие полного двоичного дерева:

Определение 12.9. **Полным двоичным деревом высоты** H (T_H) *называется двоичное дерево, у которого путь от корня до любого листа содержит ровно H рёбер, при этом у всех узлов дерева, кроме листьев, есть как правый, так и левый ребёнок.*

Альтернативное эквивалентное определение полного двоичного дерева основано на рекурсии:

Определение 12.10. **Полным двоичным деревом высоты** H (T_H) *называется двоичное дерево, у которого к корню прикреплены левое и правое поддерево, являющиеся полными двоичными деревьями T_{H-1} высоты $H-1$ (см. рис. 12.4). Двоичное дерево T_0 является просто узлом.*

Следуя второму определению, несложно доказать, что число узлов в дереве T_H есть $N = 2^{H+1} - 1$, откуда получается следующая зависимость высоты дерева от числа узлов:

$$H = \log_2(N+1).$$

Оказывается, логарифмическая зависимость высоты дерева от числа узлов имеет место и для случайных двоичных деревьев поиска, то есть деревьев, полученных в результате добавления записей в случайном порядке в пустое дерево поиска. Приведём точные определения.

Определение 12.11. **Случайное двоичное дерево** *размера n — это дерево, которое получается из пустого двоичного дерева поиска, после добавления в него n записей с различными ключами в случайном порядке (все $n!$ возможных последовательностей добавления равновероятны).*

Случайное двоичное дерево является случайной величиной, то есть его можно *измерять* (генерить) и получать самые различные деревья.

Пусть $\overline{d}(N+1)$ — средняя высота узлов случайного дерева с $(N+1)$ узлами. Корнем дерева будет один из $(N+1)$ узлов. Это тот узел, который будет добавлен в дерево первым. Вероятность быть первым для всех узлов одна и та же и равна $1/(N+1)$. В зависимости от того, какой это будет узел, остальные узлы разобьются на две группы – одна пойдёт в левое поддерево, другая – в правое поддерево. Возможные пропорции для этих групп следующие: $0 : N$, или $1 : (N-1)$, или $2 : (N-2)$ или $\ldots$ или $N : 0$.

Средняя высоты узлов получившегося дерева на 1 больше, чем высоты узлов данных двух поддеревьев. Получаем следующую рекуррентную формулу:

$$\overline{d}(N+1) = \sum_{k=0}^{N} \frac{1}{N+1} \left(1 + \frac{k}{N} \cdot \overline{d}(k) + \frac{N-k}{N} \cdot \overline{d}(N-k)\right),$$

$$\overline{d}(N+1) = \frac{2}{N(N+1)} \sum_{k=0}^{N} k \cdot (1 + \overline{d}(k)) = 1 + \frac{2}{N(N+1)} \sum_{k=0}^{N} k \cdot \overline{d}(k).$$

Задача Л12.5. Докажите и подтвердите с помощью численного эксперимента, что рекуррентная последовательность $\overline{d}(N)$, $N = 0,\ 1,\ \ldots, \infty$, определяемая формулами

$$\overline{d}(N+1) = 1 + \frac{2}{N(N+1)} \sum_{k=0}^{N} k \cdot \overline{d}(k), \quad \overline{d}(0) = 0, \quad \overline{d}(1) = 1/2,$$

растёт с N примерно как $2\ln N$, а именно $(\overline{d}(N) - 2\ln N) \to C$, при $N \to \infty$, где C – некоторая константа.

Решите эту задачу самостоятельно. Используйте следующий известный предел:

$$\lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \ln n\right) = \gamma = 0.57721\ldots$$

Средняя высота узлов случайного двоичного дерева растёт как $O(\log N)$ и средние времена выполнения операций добавления, удаления и поиска в случайном двоичном дереве имеют асимптотику $O(\log N)$.

Таким образом, мы получили следующие асимптотики *среднего времени* выполнения запросов для случайного двоичного дерева:

Структура хранилища	`insert`	`del`	`find`
Двоичное дерево (случайные данные)	$\log N$	$\log N$	$\log N$

В случайном дереве поиска размеры правого и левого поддерева для любого из узлов оказываются в среднем примерно одинаковыми (сбалансированными). Это позволяет нам говорить, что в среднем за один шаг спуска по дереву область поиска уменьшается примерно в два раза.

Но есть вероятность того, что случайное дерево поиска окажется неудачным, и высота будет сравнительно большая. Случайное дерево поиска по сути является не структурой данных, а распределением на множестве деревьев, в котором «хорошо ветвящиеся» деревья более вероятны, нежели «слабо ветвящиеся».

Было разработано несколько специальных алгоритмов выполнения запросов `insert` и `del`, которые гарантируют, что высота дерева ограничена некоторой линейной относительно $\log N$ функцией (со 100% вероятностью). Соответствующие структуры данных называются **сбалансированными деревьями поиска**. Наиболее популярные сбалансированные деревья поиска — это *АВЛ-деревья* и *красно-чёрные деревья*. Их мы рассмотрим на следующих лекциях.

Семинар 12

Структуры данных: метод деления пополам, двоичное дерево поиска

Краткое описание: На данном семинаре мы рассмотрим метод деления пополам (метод дихотомии) и решим с его помощью следующие задачи: поиск численного значения корня уравнения, быстрый поиск в отсортированном массиве и задачу «угадай число». Кроме того, мы реализуем структуру данных «двоичное дерево поиска», научимся выводить его структуру в виде префиксного описания, и исследуем как растёт случайно дерево поиска.

Поиск корня уравнения методом деления пополам

Рассмотрим уравнение

$$e^x = x + 2.$$

Точки пересечения графиков функций $f_1(x) = e^x$ и $f_2(x) = x + 2$ являются корнями этого уравнения (см. рис. 12.1) .

Не существует явной формулы корней этого уравнения. Но с помощью компьютера мы их можем найти численно с определённой точностью. Алгоритм нахождение корня основан на методе деления пополам.

Пусть, нам требуется с некоторой точностью найти корень уравнения $f(x) = 0$, который находится на промежутке $[l, r)$. Предположим также, что функция f непрерывна, в точках l и r принимает значения разных знаков. Из этого следует существование корня (по крайней мере одного) на промежутке $[l, r)$.

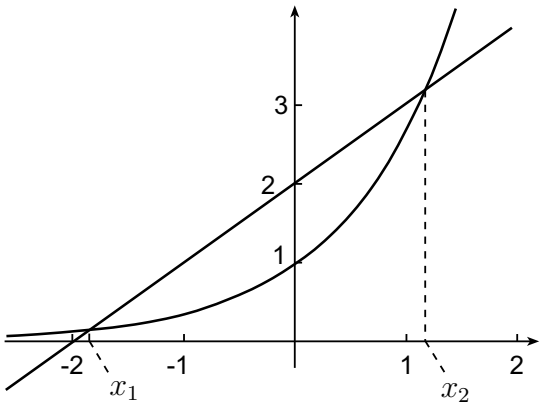


Рис. 12.1: Корни уравнения $e^x = x + 2$.

Для того, чтобы найти его, мы можем поступить следующим образом. Найдём середину $c = (l + r)/2$ промежутка $[l,\ r)$. Корень находится на одном из полуинтервалов $[l,\ c)$ или $[c,\ r)$. Выберем нужный из двух полуинтервалов и применим к нему те же рассуждения. Будем продолжать такое деление пополам, пока размер полуинтервала не станет меньше необходимой точности.

Алгоритм поиска корня уравнения методом деления пополам основан на простой рекурсивной идее: задача поиска корня на промежутке $[l, r)$ сводится к задаче поиска корня на $[c, r)$ или на $[l, c)$, где $c = (l + r)/2$. Но эту идею можно реализовать, не используя рекурсивных

функций. Каждый раз, рассматривая отрезок, мы переходим только к одной из его половин. Это можно осуществить в цикле обновив значения l или r.

В программах 12.1 и 12.2 двумя способами реализован описанный алгоритм: в первом используется метод рекурсии, а во-втором – метод итераций.

Программа 12.1: Рекурсивная реализация функции поиска корня уравнения

```
double root(double l, double r) {
    double c = (l + r) / 2;
    if( r - l < eps ) return c;
    if( f(c) * f(r) < 0 )
        return root(c,r);
    else
        return root(l,c);
}
```

Программа 12.2: Нерекурсивная реализация функции поиска корня уравнения

```
double root(double l, double r) {
    double c = (l + r) / 2;
    while( r - l > eps ) {
        if( f(c) * f(r) < 0 )
            l = c;
        else
            r = c;
        c = ( l + r ) / 2;
    }
    return c;
}
```

Теперь мы можем легко применить этот метод к нашей задаче о корнях уравнения $e^x = x+2$. Пусть $f(x) = e^x - x - 2$. Поиск корней мы будем осуществлять на отрезках $[-2, 0]$ и $[0, 2]$.

Программа 12.3: Универсальная функция для вычисления нулей функций.

```
#include <math.h>
#include <stdio.h>
const double eps = 1E-10;
double func1 (double x) {
    return exp(x) - 2 - x;
}
double func2 (double x) {
    return cos(x) - x;
}
double
root(double l, double r, double (*)(double) f) {
    double c = l;
    if(f(l) * f(r) > 0 ) {
        fprintf(stderr, "Can't use dichotomy!\n");
        return l;
```

```
    }
    while( r - l > eps ) {
         c = ( l + r ) / 2;
         if( f(c) * f(r) < 0 )
            l = c;
         else
            r = c;
    }
    return c;
}

double
root2(double l, double r, double (*)(double) f) {
    double c = l;
    if(f(l) * f(r) > 0 ) {
        fprintf(stderr, "Can't use dichotomy!\n");
        return l;
    }
    if( r - l < eps ) {
        return l;
    } else {
         c = ( l + r ) / 2;
         if( f(c) * f(r) < 0 )
            return root(c,r,f);
         else
            return root(l,c,f)
    }
}

int main() {
    printf("Function 1:\n");
    printf("x1 = %lf\n", root (0, 2, fun1));
    printf("x2 = %lf\n", root (-2,0, fun1));
    printf("Function 2:\n");
    printf("x1 = %lf",   root (0, 2, fun2));
    return 0;
}

//log_2 ( (r-l)/eps ) = log_2 ( 2 * (2^3)^10 )
```

Вопросы и задания

1. За один шаг длина промежутка поиска $[l,\ r)$ (размер области поиска) уменьшается в два раза. Сколько нужно шагов, чтобы уменьшить промежуток в более чем 1000 раз?
2. Сколько требуется шагов, чтобы начиная с отрезка длины $r - l = 2$ дойти до отрезка длины меньше 10^{-10}? Сколько требуется шагов, чтобы найти корень с точностью до 100 десятичных знаков после запятой?

Задача С12.1. Напишите программу, которая находит максимальный корень уравнения $\cos x = x/10$.

Задача С12.2. Почему рекурсивная реализация функции `root` менее эффективна, чем нерекурсивная? Обратите внимание на использовании стека. Сколько памяти выделяется в сумме под локальные переменные функции `root` представленной в коде 12.1?

Поиск элемента в отсортированном массиве

Метод деления пополам[1] находит применение не только в задаче поиска корней уравнения. Это один из базовых методов эффективного поиска данных в множестве упорядоченных данных. Простейший пример — это поиск фамилии в списке фамилий расположенных по алфавиту. Искать в таком списке существенно проще, нежели в неупорядоченном. А именно, сначала нужно посмотрить в середину списка. В зависимости от того, какой элемент списка оказался в середине, определяется в какой из двух частей — верхней или нижней — находится нужная нам фамилия. Выбираем нужную часть и проделываем с ней ту же операцию.

Реализуем этот алгоритм на языке Си. В массиве `data` будем хранить данные с оценками по информатике некоторой группы студентов.

С помощью директивы `typedef` определим новый тип `info_t`:

```
typedef struct {
    char name[30];
    int  grade;
} info_t;
```

Объявим глобальный массив `data` состоящий из 40 элементов типа `info_t` (в группе не более 40 человек).

```
info_t data[40];
```

Реальное количество стулентов будем хранить в переменной n:

```
int n = 0;
```

Функция `main` будет состоять из двух частей: считывание данных (считывание пар (фамилия, оценка)) и цикла обработки запросов. Пусть на вход программы подается сначала число студентов n, а затем идёт n строчек вида «`<фамилия> <оценка>`». Считывается такой вход следующим образом:

```
scanf("%d", &n);
for(i = 0 ; i < n ; i++) {
    scanf("%s%d", data[i].name, &data[i].grade);
}
```

Обратите внимание на то, что операция взятия адреса не стоит перед `data[i].name`. Это связано с тем, для формата `%s` следует передавать адрес первого символа в массиве символов, куда следует помещать последовательность считываемых непробельных символов[2]. Имя массива `name` интерпретируется в языке Си как адрес первого элемента массива.

[1] Метод деления пополам также называется двоичным поиском (binary search) и дихотомией.

[2] Пробельные символы — это символы пробела, табуляции и переноса строчки (' ', '\t', '\n').

Цикл выполнения запросов на поиск нужных фамилий выглядит следующим образом:

```
while ( !feof(stdin) ) {
    int c, l =0, r = n;
    char name[30],
    char *last;
    printf("Введите фамилию для поиска: ");
    if (fgets(name, 30, stdin) == NULL)
        continue;
    last = strchr(name, '\n');
    if (last)
        *last = 0;
    if (strcmp(name, "QUIT") == 0 )
        break;

    // область поиска -- полуинтервал [l ,r)
    while ( r - l > 1) {
        c = (l + r) / 2;
        if ( strcmp(name, data[c].name) > 0 )
            r = c;
        else
            l = c;
    }
    if (strcmp(name, data[l].name) == 0 ) {
        printf("Found: %s %d\n", name[l].name, name[l].grade);
    } else {
        printf("Not found\n");
    }
}
```

Этот цикл выполняется, пока не будет послан специальный сигнал завершения входа на стандартный поток входа (был получен такой сигнал или нет, определяется значением `feof(stdin)`) или пока не будет введено кодовое слово «`QUIT`»:

```
 if (strcmp(name, "QUIT") == 0 ) break;
```

Задача С12.3. Реализуйте описанную программу. Создайте несколько текстовых файлов в которых запишите примеры входных данных: описание оценок студентов и несколько запросов. Попробуйте вводить некорректные данные. Модифицируйте программу так, чтобы она проверяла корректность входных данных и имела «защиту от дурака».

Различные деревья поиска и многие другие структуры данных основаны на деления области поиска на части, и непосредственно связаны с методом деления пополам.

Примеры задач, решаемые методом деления пополам

Задача про провода

Задача С12.4. На складе есть провода различной целочисленной длины. Их можно разрезать на части. Необходимо получить K кусочков одинаковой целочисленной как можно большей длины. Найдите максимальную длину M и укажите способ разрезания[3].

Итак, пусть нам даны длины проводов

$$l_1,\ l_2,\ \ldots,\ l_n.$$

Нужно найти такую максимальную длину M, что из проводов можно нарезать K кусочков длины M.

Заметим, что если дана нужная длина провода, то не сложно найти максимальное число кусков длины M, которое можно получить из данных нам кусочков. Для этого нужно от каждого куска отрезать максимально возможное число кусков длины M. От кусочка длины l_i можно отрезать

$$\lfloor \frac{l_i}{M} \rfloor$$

кусочков ($\lfloor x \rfloor$ означает округление числа x в меньшую сторону). Таким образом, для данного M максимальное K вычисляется по формуле

$$K_{max}(M) = \lfloor \frac{l_1}{M} \rfloor + \lfloor \frac{l_2}{M} \rfloor \ldots + \lfloor \frac{l_n}{M} \rfloor.$$

Нам же нужно найти «обратную» функцию $M_{max}(K)$, которая равна максимальному M, при котором $K_{max}(M) \geqslant K$. Ясно, что

$$M \in [0, \max_{i=1,..,n} l_i].$$

Методом деления пополам, мы можем найти M_{max}. Возможность применения метода деления пополам связана с тем, что функция $K_{max}(M)$ монотонна. Вспомните, что метод деления пополам для поиска нужного элемента в массиве применяется к упорядоченным массивам, которые также можно рассматривать как монотонные функции.

Программа 12.4: Задача про провода

```
int getK(int *l, int size, int m) {
    int k = 0;
    for(i = 0; i < size; i++)
        k += (l[i] / m);
    return k;
}

int getM(int *l, int size, int k) {
    int i, l = 0, r = 0, c;
    for(i = 0; i < size; i++) {
        if (l[i] > r) r = l[i];
```

[3]Задача с полуфинала олимпиады АСМ, г. Санкт-Петербург, 2001 год, `http://neerc.ifmo.ru`

```
    }
    r++;
    while( r - l > 1) {
        c = (l + r) / 2;
        if (getK(l, size, l) < k)
            r = c;
        else
            l = c;
    }
    return l;
}
```

Задача С12.5. Запишите программу 12.4. Проверьте, как она работает. Приведите пример входных данных, при который функция `getM` вернёт неверное число, если из неё удалить строчку «`r++;`».

Задача С12.6. Используя метод деления пополам, реализуйте функцию вычисления целой части квадратного корня из натурального числа. Примечание: функции, объявленные в заголовочном файле `math.h` использовать нельзя.

Задача «Угадай число»

Задача С12.7. (Угадай число) Напишите программу, которая умеет угадывать загаданное пользователем число от 1 до 100. Программа имеет права спрашивать только вопросы, на который пользователь отвечает «Да» или «Нет» (1 означает «Да», 0 – «Нет»). Покажите, что есть алгоритм, которому 7 вопросов наверняка хватит.

Двоичное дерево поиска

Задача С12.8. Нарисуйте двоичное дерево поиска, в которое добавлялись записи со следующей последовательностью ключей:
а) {a, z, b, y, c, d, x, e, w, f, u, g, t};
б) {10, 5, 15, 4, 5, 13, 100, 11, };
в) {1, 2, 3, 4, 5, 6, 7};
г) {7, 6, 5, 4, 3, 2, 1};
д) 17 ключей: $key_n = (5 \cdot n) \bmod 17$;
е) { 1, 50, 2, 40, 3, 30, 4, 20, 5, 25, 26, 27, 28, 29}.

Задача С12.9. Реализуйте двоичное дерево поиска на языке Си, в котором ключ `key` и привязанное к нему значение `value` имеют тип `int`, Начните свою программу с следующих определений:

```
typedef int key_t;
typedef int value_t;
typedef struct node {
    key_t   key;
```

```
    value_t value;
    struct node *l, *r;
} node_t;

typedef enum {
    FOUND, DELETED, INSERTED, UPDATED,
    ERROR_NOTFOUND, ERROR_NOMEMORY
} result_t;

/* Создать новый узел дерева с заданными ключами.
 *
 */
node_t* new_node(key_t key, value_t value) {
    node_t *n = (node_t*)malloc( sizeof(node_t) );
    if ( n ) {
        n->l = n->r = 0;
        n->key = key;
        n->value = value;
    }
    return n;
}

/* Найти узел дерева с корнем root, у которого ключ = key.
 * Возвращает FOUND, если такой узел найден, и
 * ERROR_NOTFOUND -- иначе.
 * Найденный узел помещает по адресу result.
 */
result_t
find(node_t *root, key_t key, node_t *result) {
    if ( root ) {
        if ( root->key == key ) {
            *result = *root;
            return FOUND;
        } else if (root->key > key ) {
            return find(root->l, key, result);
        } else if (root->key < key) {
            return find(root->r, key, result);
        }
    } else {
        return ERROR_NOTFOUND;
    }
}

/* Добавить узел node в поддерево с корнем *root.
 * Возвращает INSERTED, если успешно добавлен, или UPDATED,
 * если элемент с таким ключём существовал.
```

```
 */
result_t
insert( node_t **proot, key_t key, value_t value ) {
    if ( *proot ) {
        if ( (*proot)->key == key ) {
            (*proot)->value = value;
            return UPDATED;
        }
        // Рассмотрите случаи
        // (*proot)->key > key и (*proot)->key < key
        ...
    } else {
        *proot = new_node(key, value);
        if ( *proot ) {
            (*proot)->key = key;
            (*proot)->value = value;
            return INSERTED;
        } else {
            return ERROR_NOMEMORY;
        }
    }
}
```

Допишите функцию `insert` по аналогии с функцией `find`. Обратите внимание, что код функций `find` и `insert` очень похожи. Попробуйте заменить их одной функцией

```
result_t find(node_t **proot, node_t *node, int insert_node);
```

которая получает ещё один аргумент – флаг `insert_node` – добавлять узел или искать узел. Если `insert_node == 1`, то аргумент `node` содержит пару (ключ, значение), которую нужно добавить в дерево. Если `insert_node == 0`, то функция должна искать узел дерева, в котором ключ равен значению `node->key`, и в случае успеха записать в `node->value` привязанные к ключу данные.

Задача C12.10. Реализуйте функцию удаления `delete` узла на языке Си. Используйте алгоритм, основанный на поиске самого левого узла в правом поддереве. Работает ли ваша функция, когда
а) у удаляемого узла оба ребёнка являются листами;
б) у удаляемого узла есть только левый ребёнок;
в) у удаляемого узла есть только правый ребёнок;
г) удаляется корень всего дерева;
д) удаляется лист дерева?
Проведите массовое тестирование функций `delete`, `find` и `insert`.

Задача C12.11. Рассмотрим случайное дерево размера $n = 10000$. Оно получается если в пустое дерево добавить 10000 пар $(key, value)$, где ключ key генерится для каждой пары с помощью случайной функции (см. функцию `rand`, объявленную в заголовочном файле `stdlib`).
а) Чему равно среднее расстояние от вершины дерева до случайного узла дерева?
б) Чему равно среднее квадратическое отклонение расстояние от вершины дерева до случай-

ного узла дерева?
в) Чему равно среднее время выполнения операций добавления, удаления и поиска в случайном двоичном дереве размера n?
Используйте рекуррентные соотношения для оценки средних указанных значений. Проведите численный эксперимент, чтобы проверить теоретические оценки.

Задача С12.12. Реализуйте алгоритм сортировки, основанный на том, что элементы (как ключи) сначала добавляются в дерево поиска, а затем из дерева последовательно извлекается самый левый элемент, который находится с помощью функции `left`:

```
node_t* left(node_t *root) {
    if ( root ) {
        while ( root->l ) {
            root = root->l;
        }
    }
    return root;
}
```

Оцените время сортировки данным алгоритмом для случайных данных (как функцию от количества элементов).

Задача С12.13. Изучите функцию `infix_traverse`, представленную в коде 12.5. Назначение функции `infix_traverse` — обойти все узлы дерева и к каждому узлу дерева применить функцию `fun`, заданную во втором аргументе. Функция `infix_traverse` определена рекурсивно: обойти дерево с корнем `root` – значит обойти левое поддерево `root->l`, зайти в корневой узел `root`, обойти правое поддерево. Можно так коротко записать логику функций `infix_traverse` и `prefix_traverse`:

```
infix_traverse(root, fun):  infix_traverse(root->l, fun) &&
                            fun(root) &&
                            infix_traverse(root->r, fun).
prefix_traverse(root, fun): fun(root) &&
                            prefix_traverse(root->l, fun) &&
                            prefix_traverse(root->r, fun).
```

Они отличаются лишь порядком вызовов. Реализуйте функцию `prefix_traverse` на языке Си. Рассмотрите следующие строчки кода:

```
infix_traverse  (root, print_key, 0);
prefix_traverse (root, print_indented, 0);
infix_traverse  (root, increment_value, 0);
prefix_traverse (root, print_indented, 0);
```

Что произойдёт при их выполнении? Напишите функцию сортировки чисел, основанную на двоичном дереве и функции `infix_traverse`. Насколько быстрее работает этот алгоритм сортировки для случайных данных по сравнению с алгоритмом, предложенным в задаче С12.12?

Программа 12.5: Функция `traverse`

```
/* Применить функцию fun ко всем узлам дерева с корнем root
```

```
 * по порядку с самого левого узла, до самого правого.
 * Процесс выполнения прерывается, как только функция fun вернёт
 * для какого-либо узла не нулевое значение.
 */
void
infix_traverse(node_t *root, int fun(node_t*, int level) ) {
    if(root) {
        int ret = 0;                            // заходим
        ret = traverse(root->l, fun, level+1); // в левое
        if( ret != 0 ) return ret;              //    поддерево
        ret = fun(root->l, level);              // в сам корень
        if( ret != 0 ) return ret;
        ret = traverse(root->r, fun, level+1); // в правое
        if( ret != 0 ) return ret;              //    поддерево
        return 0;
    } else {
        return fun(root, level);
    }
}
int print_key(node_t *node, int level) {
    int i;
    if(node)
        printf("%d ", node->key);
    return 0;
}
int print_indented(node_t *node, int level) {
    int i;
    for( i = 0 ; i < level ; i++ ) printf("   ");
    if(node)
        printf("(%d %d)\n", node->key, node->value);
    else
        printf("(null)\n");
    return 0;
}
int increment_value(node_t *node, int level) {
    if(node)
        node->value += 1;
}
```

Задача C12.14. Результат, выводимый при выполнении строчки

```
prefix_traverse (root, print_indented, 0);
```

называется префиксным описанием дерева. Получите префиксное описание деревьев, заданных в задаче C12.8.

Лекция 13

Структуры данных: методы балансировки деревьев, АВЛ-деревья

Краткое описание: В предыдущей лекции было показано, что основные операции с двоичным деревом поиска высоты H могут быть выполнены за $O(H)$. Деревья эффективны, если их высота мала. Но малая высота не гарантируется, и в худшем случае деревья не более эффективны, чем списки. На данной лекции мы рассмотрим различные методы балансировки деревьев поиска, которые гарантируют, что высота не превзойдёт $\Theta(\log N)$, где N — число хранимых записей. Изучим структуру данных «АВЛ-дерево».
Ключевые слова: АВЛ-дерево, идеально сбалансированное дерево, фибоначчиево дерево, критерии сбалансированности дерева.

Эффективность алгоритмов: средний и худший случаи

Обычное дерево поиска гарантирует, что время выполнения запросов в дереве высоты H ограничено $O(H)$ (линейной от высоты дерева функцией). В случайных деревьях поиска (деревьях, которые получаются, если запросы на добавление записей приходят в случайном порядке) средняя высота растёт как $O(\log N)$, где N — число записей, хранимых в дереве. Но в худшем случае, высота дерева может быть равной количеству хранимых записей (случай, когда записи добавляются в порядке возрастания или убывания ключей).

Во многих практических задачах особенно важно, чтобы время ответа системы на каждый запрос было *гарантировано* мало (мало не только в среднем, но и худшем случае). Это касается систем управления летающими аппаратами, приложений, встречающихся в военной промышленности, систем управления атомными электростанциями и многих других систем. Для этих задач использование обычных деревьев поиска неприемлимо.

На данной лекции мы рассмотрим специальные методы балансировки, которые гарантируют, что высота дерева поиска будет расти как $\Theta(\log N)$.

Критерии сбалансированности дерева

Определение 13.1. **Методы балансировки деревьев поиска** — *это алгоритмы выполнения операций добавления и удаления записей (*`insert` *и* `del`*), которые гарантируют, что при любой последовательности выполнения запросов высота H дерева поиска будет ограничена сверху линейной функцией от логарифма числа N хранимых записей:*

$$H < A \cdot \log_2 N + B,$$

где A и B — некоторые фиксированные константы.

Важно отметить, что сбалансированность дерева — это свойство не отдельно взятого дерева, а алгоритмов, которые отвечают за построение этого дерева и выполнение различных запросов к нему.

Прежде, чем переходить к методам балансировки деревьев, рассмотрим косвенные признаки сбалансированности: свойства деревьев, которые гарантируют логарифмическую зависимость высоты дерева от числа узлов.

Теорема 13.1. *Докажите, что если в двоичном дереве с N узлами выполнено хотя бы одно из следующих условий:*

а) *для любого узла число узлов в левом и правом поддереве N_r, N_l отличаются не более чем на 1:*

$$N_r \leqslant N_l + 1, \quad N_l \leqslant N_r + 1,$$

б) *для любого узла число узлов в правом и левом поддереве N_r, N_l удовлетворяют условиям*

$$N_r \leqslant 2N_l + 1, \quad N_l \leqslant 2N_r + 1,$$

в) *для любого узла высота правого и левого поддерева H_r, H_l отличаются не более чем на 1:*

$$H_r \leqslant H_l + 1, \quad H_l \leqslant H_r + 1,$$

то высота дерева не превосходит $A \log_2 N + B$, где A и B – некоторые положительные константы, не зависящие от N.

Определение 13.2. **Идально сбалансированное дерево** *— это двоичное дерево поиска, для которого с помощью специальных алгоритмов поддерживается свойство (а).*
АВЛ-дерево[1] *— это двоичное дерево поиска, для которого с помощью специальных алгоритмов поддерживается свойство (в).*

Идально сбалансированные деревья представляют сабой самый лучший случай деревьев поиска, в каждом узле ветвления происхоит разление оставшихся узлов на две равные части (если число оставшихся узлов чётно), либо на две части, число элементов в которых отличается не более чем на единицу (если число оставшихся элементов нечётно).

Приведём доказательство теоремы 13.1.
Доказательство. **Случай (а), идеально сбалансированное дерево:** Пусть $H_{ideal}(N)$ — максимальная высота идеально сбалансированного дерева с N вершинами. Рассмотрим идеально сбалансированное дерево с нечётным числом вершин $2N + 1$. Левое и правое поддеревья его корня обязаны содержать ровно по N вершин:

$$H_{ideal}(2N + 1) = 1 + H_{ideal}(N)$$

Для случая, когда число вершин чётно, получаем следующее равенство

$$H_{ideal}(2N) = 1 + \max(H_{ideal}(N - 1), H_{ideal}(N)).$$

[1]Критерий сбалансированности дерева (в) предложили советские математики Адельсон-Вельский и Ландис в 1962 г.

Понятно, что H_{ideal} неубывающая функция. Поэтому

$$H_{ideal}(2N) = 1 + H_{ideal}(N).$$

Используя данные соотношения, несложно методом математической индукции показать, что $H_{ideal}(N) \leqslant log_2 N$, а именно $H_{ideal}(N)$ в точности равно числу цифр в двоичной записи числа N минус 1.

Случай (б): Пусть $H_2(N)$ – максимальная высота дерева с N узлами, в котором в выполнено свойство (б). Несложно определить значения для малых N: $H_2(1) = 0$, $H_2(2) = H_2(3) = 1$. Один из узлов будет корнем дерева, а оставшийся $(N-1)$ узел распределится между правым и левым поддеревом в отношении $N_l : N_r$, $N_l + N_r = N - 1$. Если предположить, что $N_r \geqslant N_l$, то $N_r \leqslant 2N_l + 1$. При всех указанных ограничениях на N_l и N_r получаем следующее соотношение:

$$H_2(N) = \max_{N_l,\, N_r}(1 + \max(H_2(N_l), H_2(N_r))).$$

Это следует из того, что высота дерева есть 1 плюс максимум из высот левого и правого поддеревьев. Кроме того, нас интересует максимум высоты для всех возможных разбиений.

Ясно, что $H_2(N)$ неубывающая функция. Поэтому последнее соотношение можно существенно упростить:

$$H_2(N) = 1 + H_2(\max(N_r)).$$

Из ограничений $N_r \leqslant 2N_l + 1$ и $N_l + N_r = N - 1$ получаем

$$H_2(N) = 1 + H_2((2N-1)/3),$$

где под знаком деления подразумевается деление без остатка. Из равенства следует неравенство

$$H_2(N) > 1 + H_2(2N/3).$$

Видно, что $H_2(N)$ можно оценить сверху числом $\log_{3/2} N$ — сколько раз нужно разделить число N на $3/2$, чтобы получить число, меньшее 1. Несложно показать, что

$$H_2(N) > \log_{3/2} N + 1 \approx 1.70951 \log_2 N + 1.$$

Таким образом утверждение теоремы доказано со следующими значениями констант $A = 1.71$, $B = 1$.

Случай (в), АВЛ-деревья: Пусть $N_f(H)$ — минимальное число узлов в АВЛ-дереве с высотой H. АВЛ-деревья, на которых достигается минимум, назовём минимальными АВЛ-деревьями. В этом дереве либо правое либо левое поддерево корня имеет высоту $H - 1$. Второму поддереву разрешено иметь высоту $H - 2$ либо $H - 1$. Но поскольку $N_f(H)$ неубывает, а мы стремимся минимизировать число узлов, то у второго поддерева высота должна быть $H - 2$. Число узлов в минимальном АВЛ-дереве равно

$$N_f(H) = 1 + N_f(H-1) + N_f(H-2),$$

так как оно состоит из корня и двух минимальных АВЛ-деревьев с высотами $H - 1$ и $H - 2$. Несложно получить начальные значения $N_f(0) = 1$, $N_f(1) = 2$, $N_f(2) = 4$. Решите самостоятельно следующую задачу.

Задача Л13.1. Докажите, что $N_f(N) = F_{N-1} - 1$, где F_n — числа Фибоначчи:

$$F_0 = F_1 = 1,\ F_n = F_{n-1} + F_{n-2}.$$

Докажите, что $F_n \geqslant \varphi^{n-1}$, где $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$.

Из утверждения этой задачи следует нужный нам результат:

$$N_f(H) \geqslant \varphi^{H-2} + 1,$$

а значит высота произвольного АВЛ-дерева ограничена сверху величиной

$$\log_\varphi(N-1) + 1 \approx 1.44042 \log_2(N-1) + 1.$$

□

Таким образом, мы получили следующий результат:

Высота деревьев поиска, удовлетворяющих условиям (а), (б), (в) теоремы 13.1, ограничена сверху логарифмической функцией от числа узлов дерева, а именно:

а) $H \leqslant 1 \cdot \log_2 N$,
б) $H \leqslant 1.70951 \cdot \log_2 N + 1$,
в) $H \leqslant 1.4404 \cdot \log_2 N + 1$.

Коэффициент перед логарифмом отображает меру сбалансированности дерева: чем ближе он к 1, тем лучше дерево сбалансировано. Меньше 1 он быть не может, так как именно такую асимптотику зависимости высоты дерева от числа узлов имеют полные двоичные деревья и идеально сбалансированные двоичные деревья.

Итак, высота АВЛ-дерева может быть больше, чем высота идеально сбалансированного дерева с тем же множеством ключей, но не более, чем на 45%.

Высота у деревьев со свойством (б) может быть примерно на 71% больше, чем у идеально сбалансированных деревьев.

Наименьшее гарантированное значение высоты имеют идеально сбалансированные деревья, и именно для них скорость выполнения операций поиска будет выполнятся наименьшее время. Но, тем не менее, идеально сбалансированные деревья не используются на практике. Дело в том, что кроме кроме операции поиска, есть ещё операции добавления и удаления записей. Они должны быть такими, чтобы соответствующее свойство сбалансированности сохранялось. Поддержка свойства идеальной сбалансированности требует при выполнении этих операций слишком больших накладных расходов.

Поддержка сбалансированности дерева

Одним из основных средств, которые позволяют поддерживать сбалансированность деревьев поиска, являются *вращения*.

Вращение — это модификация структуры небольшой части дерева поиска. Рассмотрим конкретную ситуацию с АВЛ-деревом.

В АВЛ-дереве необходимо в каждом узле хранить высоту поддерева, корнем которой он является.

Каждый раз, когда в некотором поддереве происходят какие-то изменения, необходимо обновлять значение высоты, хранящееся в его корне. Эта высота определяется рекурсивно как максимум из высот левого и правого поддерева плюс 1. Обновлять значения высоты можно перед выходом из рекурсивной процедуры удаления или добавления записи, то есть двигаясь снизу вверх, от места, где было модифицировано дерево, к корню дерева.

Добавление узла в АВЛ-дерево

Пусть мы осуществили операцию добавления записи в некоторое поддерево с корнем n. Предположим, что запись добавилась в правое подерево и в результате это привело к «АВЛ-несбалансированности»: высота правого дерева стала на 2 больше высоты левого поддерева.

Пусть правое поддерево с корнем r имеет высоту $k+2$, а левое поддерево с корнем l – высоту k.

Есть два случая: правое дерево выросло до высоты $k+2$ за счет того, что
а) правое поддерево узла r выросло до высоты $k+1$;
б) левое поддерево узла r выросло до высоты $k+1$.

Случай (а). Рассмотрим первый случай. Он показан на рисунке 13.1. Осуществим преобразование, которое называется **вращение влево относительно узла** r (LEFT-ROTATE(r)): переместим указатели-стрелочки так, чтобы узел r в качестве левого ребенка имел n, а n имел детей l (левый) и rl (правый), где rl бывший левый ребенок узла r. Кроме того, узел r сделаем новым левым ребёнком того узла, который был родителем n.

Заметьте, что при этом вращении после проекции на горизонталь узлы расположены в том же порядке, что и раньше. Это означает, что операция вращения сохранила главное свойство дерева поиска — ключи узлов растут при движении слева направо.

На рисунке всего изображено 5 стрелочек, две из них не изменились. В итоге необходимо переместить 3 стрелочки. Но обычно стрелочки делают двусторонними — к каждому узлу добавляют указатель p на его родителя. Поэтому операций изменения стрелочек-указателей (полей l, r и p) будет больше[2].

Задача Л13.2. Напишите последовательность операций, которые осуществляют вращение в левую сторону.

Задача Л13.3. Нарисуйте симметричную схему вращения в правую сторону.

Аналогично, если бы «перевесило» левое поддерево, мы осуществили бы операцию вращения вправо.

Итак, мы рассмотрели лишь случай (а). Заметим, что если бы к узлу rl было подвешено дерево высоты $k+1$, а к узлу rr — высоты k, то описанное левое вращение ROTATE-LEFT(r) не восстановило бы свойство АВЛ-дерева. Данную ситуацию следует рассматривать отдельно, и мы её обозначили как случай (б).

[2]Есть довольно сложная для понимания техника, позволяющая осуществлять вращения в случае, когда узлы не хранят информацию о своём родителе. Тогда приходится передавать функции вращения в качестве аргумента не указатель на узел, а указатель на указатель на узел, чтобы в процессе вращения этот указатель на узел мог быть подменён.

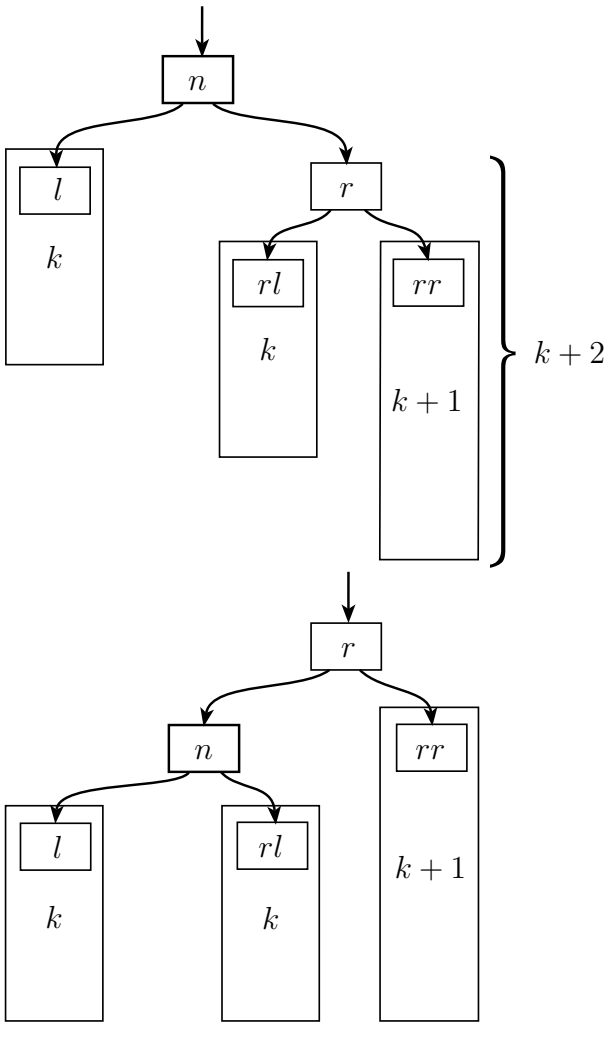


Рис. 13.1: Операция вращения влево ROTATE-LEFT(r). Сверху показана начальная ситуация, снизу — то, что получилось после операции вращения.

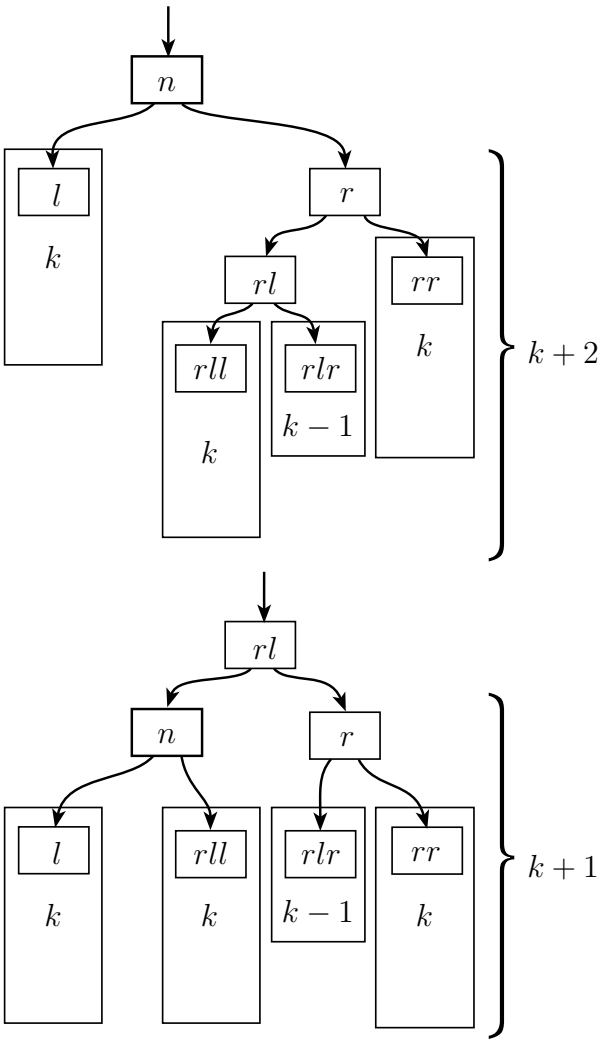


Рис. 13.2: Комбинация двух вращений ROTATE-RIGHT(rl) и ROTATE-LEFT(rl). Сверху показана начальная ситуация, снизу — то, что получилось после операций вращения.

Случай (б). Пусть поддерево с корнем r перевесило из-за того, что левое поддерево выросло до высоты $k+1$, и пусть узел rl – корень левого поддерева дерева с коренем r, а узлы rll и rlr — соответственно, левый и правый дети узла rl. На одном из этих детей висит дерево высотой k (например, на rll), а на другом – дерево высотой $k-1$ (например, на rlr). Переместим стрелки так, чтобы получилась ситуация, изображенная на рисунке 13.2 внизу: узел rl поднимается на два уровня вверх, его детьми становятся узлы n и r, детьми n будут узлы l и rll, а детьми r – узлы rlr и rr. После этой операции свойство АВЛ-дерева восстановилось и высота поддеревьев самого верхнего узла уменьшилась на 1.

Заметим, что специальной функции вращения для случая (б) программировать не нужно. Описанное преобразования можно представить в виде комбинации двух вращений: ROTATE-RIGHT(rl) и ROTATE-LEFT(rl).

Вращение ROTATE-RIGHT(n) (ROTATE-LEFT(n)) применяется к узлу n, который является правым (левым) ребёнком своего родителя. Во время этого вращения узел n меняется уровнями со своим родителем — узел n поднимается на один уровень, а его родитель опускается на один уровень.

Удаление узла из АВЛ-дерева

Во время осуществления операции удаления свойство АВЛ-сбалансированности также может нарушится. Здесь также следует рассмотреть 4 случая. После удаления узла из некоторого поддерева, высота одного поддерева становится на 2 меньше высоты второго поддерева. Для определенности будем считать, что удаление произошло из левого поддерева и оно менее глубокое.

Узлы, относительно которых будут осуществлятся вращения нужно искать в правом поддереве. У нас возникают два случая, которые уже разобраны на рисунках 13.1 и 13.2. На них мы останавливаться не будем. Кроме того, возникает ещё один случай, когда в дереве, изображённом на рисунке 13.2 к узлу rr подвешено дерево, высотой $k+1$, а не k, как изображено. Этой ситуации не могло возникнуть при добавлении, так как при добавлении одного узла в дерево с корнем r (рис. 13.2, сверху) может увеличится высота только одного из его поддеревьев. При удалении же это возможно. Здесь снова свойство сбалансированности восстановит одно вращение относительно узла r. Есть простой признак того, когда следует применять одно вращение ROTATE-LEFT(r), а когда два вращения — ROTATE-RIGHT(rl) и ROTATE-LEFT(rl). Первое надо делать в случае, когда $height(r \to l) \leqslant height(r \to r)$. При этом удобно считать, что высота $height$ пустого «виртуального» листа `nil` равна 0, а высота настоящего листа равна 1.

Фибоначчиевы деревья

Когда мы локазывали, что зависимость числа узлов в АВЛ-дереве от его высоты экспоненциальная (а обратная зависимость — логарифмическая), мы рассматривали самые «плохие», АВЛ-деревья, на которых достигается минимум числа узлов при фиксированной высоте. Такие минимальные АВЛ-деревья называются фибоначчиевыми деревьями.

ОПРЕДЕЛЕНИЕ 13.3. *Фибоначчиево дерево T_n, $n > 1$, это двоичное дерево, у которого к корню прикреплены фибоначчиевы деревья T_{n-2} (левый ребёнок) и T_{n-1} (правый ребёнок). Дерево T_0 по определению является пустым деревом, а дерево T_1 — деревом, состоящим из одного узла-корня.*

Фибоначчиевы деревья встречаются в структуре данных «Фибоначчиева куча», являющейся одной из самых эффективных реализаций очереди с приоритетами, которую мы рассмотрим на одной из следующих лекций.

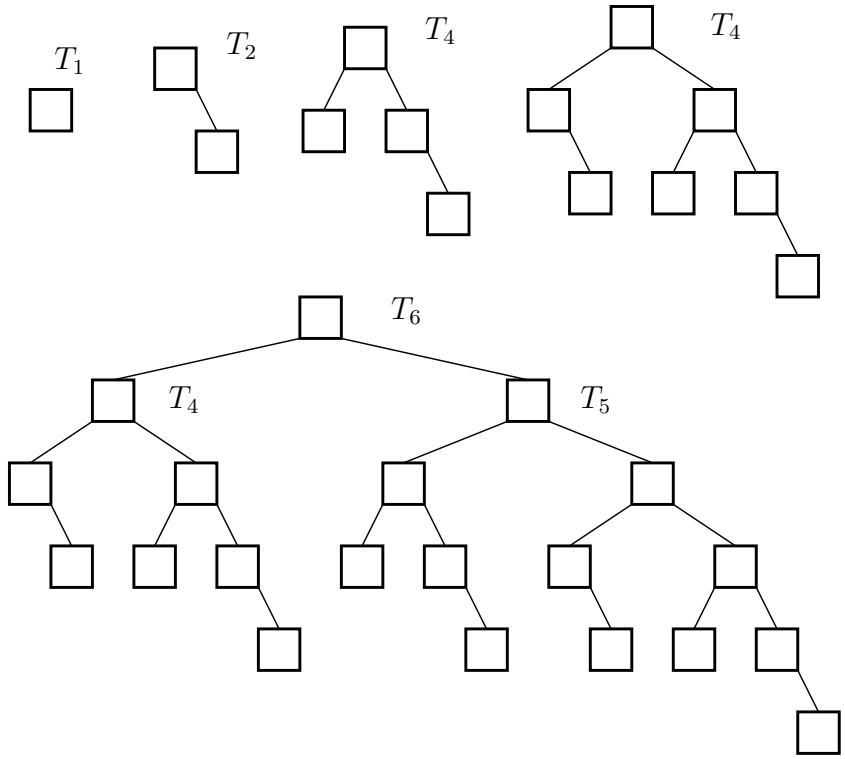


Рис. 13.3: Фибоначчиевы деревья. Фибоначчиево дерево T_6 получается если к корню в качестве левого поддерева присоеденить T_4, а в качестве правого — T_5.

Семинар 13

Структуры данных: задача «телефонная книжка», АВЛ-деревья

Формулировка задачи «Телефонная книжка»

Задача С13.1. Напишите программу, которая реализует функциональность телефонной записной книжки. Из стандартного потока входа программа должна получать последовательность команд на добавление (`INSERT`), поиск (`FIND`) или удаление `DEL` записей. Примеры команд:

```
INSERT Сидоров 1234567
INSERT Васильев 7654321
FIND Сидоров
```

При выполнении команды `INSERT` программа добавляет пару (имя, номер) в своё хранилище и выводит строку `OK`, если в хранилище нет записи с таким именем, или изменяет существующую запись с таким именем и выводит строку «`Changed. Old value = X`», если запись с такой фамилией уже есть в хранилище и соответствующий телефонный номер был `X`. При выполнении команды `FIND` программа выводит телефонный номер для указанной фамилии или выводит `NOT FOUND`, если указанной фамилии нет в справочнике. Храните структуры, состоящие из двух элементов `name` и `number`. Используйте технику динамического выделения памяти для хранения записей. Оцените, как в среднем растёт число элементарных операций с ростом числа хранимых записей при выполнении команд `INSERT` и `FIND`.

Решение задачи «Телефонная книжка» с помощью массива

Задача С13.2. Решите задачу С13.1, используя для хранения записей массив или список. Рассмотрите два случая: а) записи хранятся в отсортированном по фамилиям (в алфавитном порядке) виде; б) записи хранятся в произвольном порядке (например, в порядке добавления). В случае а) при хранении записей в массиве в реализации функций использовать поиск методом деления пополам (см. код 13.1).

Программа 13.1: К задаче С13.2

```
... /*  пропущено определение функции less*/
/* Ищет в массиве data, содержащем size элементов, элемент с ключём key
 * Если такой элемент найден, возвращает его индекс в массиве.
 * Иначе -- возвращает -1.
 */
```

```
int bsearch(const item_t *data, unsigned int size, const char *key) {
    unsigned int c, l = 0, r = size;
    // Поиск фамилии key:
    while( r - l > 1) {
        c = (l + r) / 2;
        if( less (key, data[c].familyname) )
            r = c;
        else
            l = c;
    }
    if(equal(key, data[l].familyname))
        return l
    else
        return -1;
}
```

Решение задачи «Телефонная книжка» с помощью двоичного дерева поиска

Задача C13.3. Решите задачу C13.1, используя двоичное дерево поиска. Для этого создайте три файла: `pbook.c`, `btree.h`, и `btree.c`. В файле `pbook.c` должна быть описана функция `main`, в файле `btree.h` должны быть *объявлены* функции для работы с бинарным деревом поиска (интерфейс бинарного дерева поиска), а в файле `btree.c` эти функции должны быть описаны (реализованы). Пример того, как могут выглядеть файлы `pbook.c` и `btree.h` показано в кодах 13.2 и 13.3

Программа 13.2: К задаче C13.3

```
/*  File: phonebook.c
 *  Contains main function.
 */
#include <stdio.h>
#include "btree.h"

/*
    Добавьте в функцию main строчки, информирующие пользователя о
    результате выполнения запроса и об ошибках.
*/

int main(int argc, char *argv[]) {
    node_t *root;
    char line[1024];
    node_t x;
    int ret; // успех или неуспех
    root = bt_new();
```

```
    while( fgets(line, sizeof(line), stdin) ) {
        ret = BT_NORESULT;
        if( strncmp(line, "INSERT", 3) == 0 ) {
            if( sscanf(line, "INSERT%s%d", x.key, &x.value) == 2 ) {
                ret = bt_insert(&root, &x);
            }
        } else if( strncmp(line, "FIND", 4) ) {
            if( sscanf(line, "FIND%s", x.key) == 1 ) {
                ret = bt_find(&root, &x);
            }
        } else if( strncmp(line, "DEL", 3) ) {
            if( sscanf(line, "DEL%s", x.key) == 1) {
                ret = bt_del(&root, &x);
            }
        } else if( strncmp(line, "PRINT", 5) ) {
            ret = bt_traverse_infix(root, print_key_and_value, 0);
        } else {
            fprintf(stderr,
             "ERROR: Command should start with PRINT, INSERT, FIND or DEL\n");
        }

    }
}
```

Программа 13.3: Интерфейс двоичного дерева поиска

```
/*  File: btree.h
 *  Binary tree interface.
 */
#ifndef _BTREE_H_
#define _BTREE_H_

#ifndef KEYSIZE
#define KEYSIZE 20
#endif

typedef struct node {
    char key[KEYSIZE];
    int value;
    struct node* l, *r;
    int height;
} node_t;

/*
 *  Коды возврата, возникающие при выполнении
 *  запросов find, add, del.
 */
```

```
typedef enum {
    BT_NORESULT, BT_FOUND, BT_ADDED, BT_DELETED,
    BT_NOTFOUND, BT_NOMEMORY, BT_BADTREE
} result_t;


/*
 * Базовые функции двоичного дерева поиска
 */
result_t bt_find (node_t **root, node_t *x);
result_t bt_add  (node_t **root, const node_t *x);
result_t bt_del  (node_t **root, const node_t *x);

/*
 *  Функции обхода дерева с применением функции f
 *  к каждому узлу до момента, пока f не вернёт значение !=0.
 */
int bt_traverse_infix  (node_t *root, int f(node_t*, int));
int bt_traverse_prefix (node_t *root, int f(node_t*, int));

#endif  /* _BTREE_H_ */
```

Реализация АВЛ-дерева

АВЛ-дерево — это дерево, в котором высоты левого и правого поддеревьев для любого узла отличаются не более, чем на 1. Данное свойство гарантирует, что высота дерева ограничена логарифмической функцией от числа узлов в дереве.

Для того, чтобы в двоичном дереве поиска поддерживать свойство АВЛ-дерева, необходимо

1) добавить к структуре `node_t` элемент «`int depth`», в котором будет хранится высота поддерева, корнем которой является данный узел;

2) в рекурсивный функциях добавления и удаления записей перед выходом из рекурсии добавить проверку сбалансированности правого и левого дерева относительно текущего узла, и выполнить необходимые операции балансировки (вращения в правую или левую сторону).

Глубину листьев удобно считать равной 1, а не 0. Глубину 0 имеют пустые листья `nil`.
Существует два принципиально разных способа реализации двоичных деревьев:

1) с поддержкой указателя на родителя (в структуру `node_t` добавляется новый элемент «`struct node *p`»);

2) без поддержки указателя на родителя.

В первом случае необходимо будет заботится о том, чтобы ребёнок действительно указывал на своего родителя, то есть:

`n->l->p == n  &&  n->r->p == n.`

Это дополнительное условие *консистентности*[1] (целостности) двоичного дерева поиска.

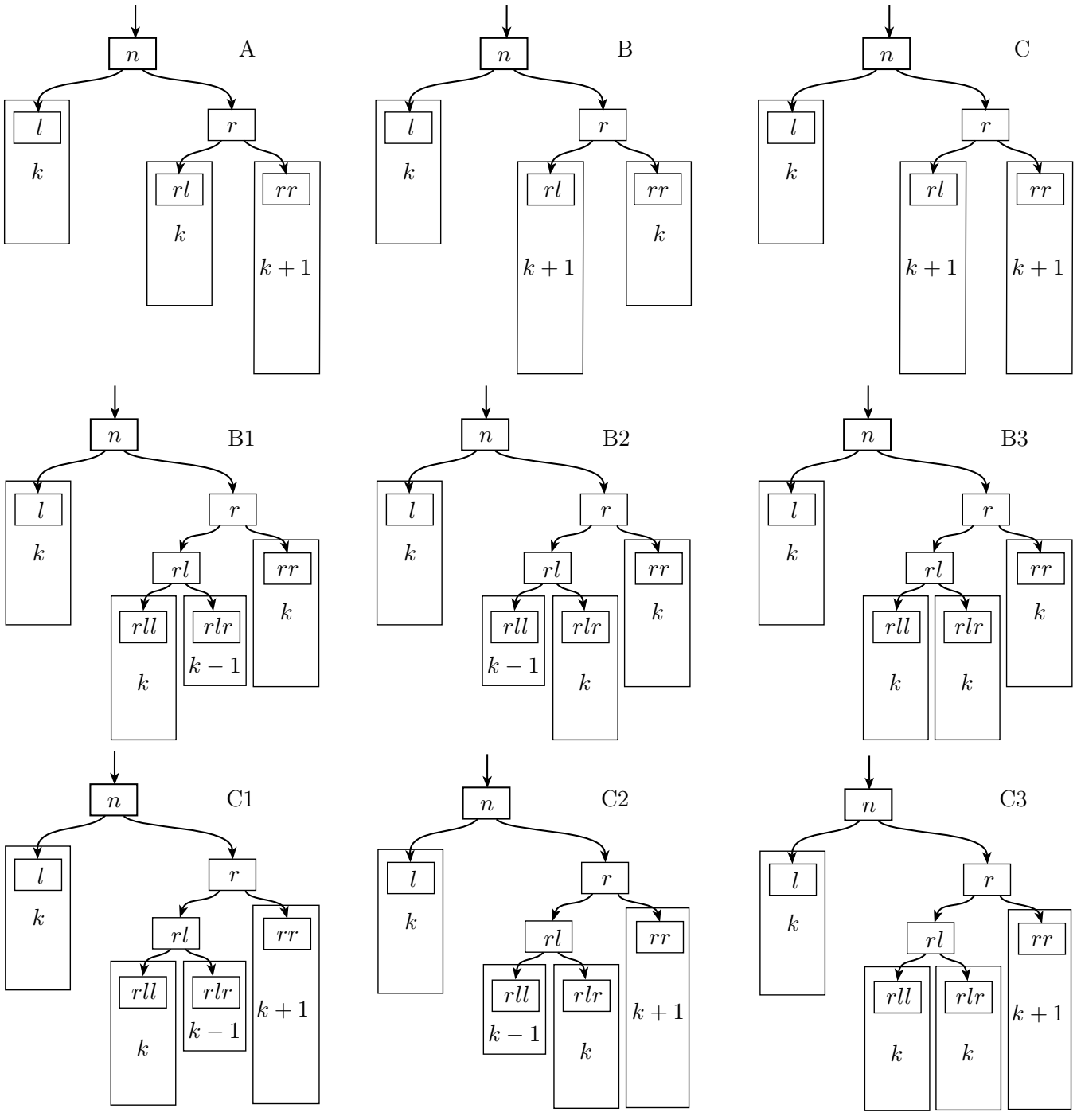


Рис. 13.1: Варианты несбалансированности поддерева с корнем n. Показаны все три случая, когда «перевешивает» правое поддерево (A,B,C). Случаи B и C разбиваются на три подслучая. Если несбалансированность возникла в результате добавления узла в правое поддерево узла n, то возможны лишь случаи A, B1 и B2. При удалении из левого поддерева возможны все 7 случаев несбалансированности: A, B1, B2, B3, C1, C2, C3. С помощью одного вращения относительно узла r (`rotate_left( &n )`) можно устранить несбалансированности A, C1, C2, C3.

С помощью последовательности двух вращений относительно узла rl (`rotate_right( &n->r ); rotate_left( &n );`) устраняются несбалансированности типа B1, B2, B3, C3.

Во-втором случае необходимо научится осуществлять операции балансировки дерева (вращения вправую и влевую сторону), не имея в узлах информации о родительском узле. Это возможно, хотя возникают определённые трудности из-за отсутствия простого алгоритма перемещения по дереву вверх, от ребёнка к родителю.

Задача C13.4. Реализуйте структуру данных «АВЛ-дерево», а именно модифицируйте написанные вами функции добавления и удаления элементов из двоичного дерева поиска, чтобы они поддерживали свойство АВЛ-дерева. Пусть для упрощения задачи ключи и значения имеют тип `int`. В коде 13.4 показаны примеры того, как можно реализовать функцию `avl_check_left(node_t **n)`, устраняющую возможный перевес правого поддерева узла `(*n)`, и функцию вращения влево `avl_rotate_left(node_t **n)`, которая вращает влево относительно узла `(*n)->r`.

В приведённых функциях предподагается, что указатель на родительский узел отсутствует, и не нужно следить за тем, чтобы его значение было верным. Несложно по аналогии написать функции `avl_check_right(node_t **n)` и `avl_rotate_right(node_t **n)`. Они получаются простой заменой r на l и l на r.

Программа 13.4: Операции поддержки свойства АВЛ-дерева

```
#define HEIGHT(node) ((node == 0)? 0 : node->height)

/* Вычисляет глубину на основании глубин поддеревьев,
 * подвешенных к правому и левому ребёнку.
```

[1] **Консистентность данных** (англ. *data consistency*, *data validity*) — это согласованность данных друг с другом, целостность данных, выполнение всех условий, наложенных *моделью (структурой) данных*.

Консистентность структуры данных в теории алгоритмов имеет важное значение. Задачи, решаемые алгоритмистами и программистами, в большей части связаны с поиском эффективной структуры данных и реализацией механизмов поддержки её консистентности.

Например, условие консистентности двоичного дерева поиска — это возрастание ключей в узлах дерева слева направо, а именно ключ в корневом узле должен быть меньше ключей узлов правого поддерева и больше ключей узлов левого поддерева. Если в каждом узле дерева поиска хранится также указатель parent на родительский узел, то возникает дополнительное условие консистентности двоичного дерева поиска: в каждом узле X указатель на родительский узел должен указывать на такой узел, в котором ровно один из указателей на детей (left или right) указывает на узел X.

Проблема поддержки консистентности данных остро стоит в задачах управления большими базами данных. Одним из способов избавится от проблем, связанных с поддержкой консистентности, является устранения дублирования информации. Одна и та же информация может быть записана с базе данных в нескольких местах (но, возможно в разном виде) или частично повторяться. Это требует *синхронизации* кусочков повторяющейся информации друг с другом.

С другой стороны, дублирование информации в различных местах позволяет писать более простые и эффективные алгоритмы поиска данных (алгоритмы выполнения различных запросов). При решении проблемы поддержки консистентности данных необходим *разумный баланс* между скоростью (сложностью алгоритмов) извлечения данных и скоростью (сложностью алгоритмов) хранения и модификации данных.

```
 */
int avl_height(node_t *node) {
    int h = 0;
    if( node->l ) h = node->l->height;
    if( node->r && node->r->height > h ) h = node->r->height;
    return h + 1;
}

/* Проверяет, не требуются ли вращения, чтобы устранить перевес
 * правого поддерева у узла *n.
 * Требует условия  (*n)->r != 0.
 */
void avl_check_left(node_t **n) {
    node_t *r = (*n)->r;
    if( r->height  > HEIGHT((*n)->l) + 1   ) {
        if(HEIGHT(r->l) <= HEIGHT(r->r)){
            avl_left_rotate(n);
        } else {
            avl_right_rotate( &( (*n)->r ) );
            avl_left_rotate ( n );
        }
    }
    (*n)->height = avl_height(*n);
}

/* Вращает влево относительно узла (*n)->r.
 *  Требует условия (*n)->r != 0.
 */
void avl_left_rotate(node_t** n) {
    node_t *x = *n;
    (*n) = (*n)->r;
    x->r = (*n)->l;
    (*n)->l = x;
    x->height = avl_height(x);
    (*n)->height = avl_height(*n);
}
```

Реализовать операцию удаления узлов сложнее, чем операцию добавления.

Задача С13.5. Реализуйте функцию удаления записи из АВЛ-дерева. Используйте в качестве основы код 13.5. Посмотрите на рисунок 13.1. Там изображены все случаи, когда левое поддерево имеет высоту на 2 меньше чем правое (левое — k, правое — $k+2$). Прежде чем утверждать, что ваше АВЛ-дерево работает проведите анализ того, как ваша функция удаления отработает в этих случаях, а также в симметричных случаях перевеса в левую сторону. Проверьте не возникнет ли ошибки, когда высота левого поддерева k равна 0, то есть когда левый ребёнок равен `nil`.

Программа 13.5: Операция удаления узла АВЛ-дерева

```
int avl_left_height(node_t **proot);
int avl_del_node(node_t **pnode);

/* Удаляет узел с ключём x->key из поддерева *proot.
 * Определяет из какого поддерева нужно произвести удаление,
 * после чего рекурсивно вызывает себя.
 * Перед выходом из рекурсии вызывается функция устранения
 * возможной несбалансированности.
 */
int avl_del(node_t **proot, const node_t *x) {
    node_t *root;
    root = *proot;
    if( root) {
        int ret;
        if( root->key > x->key ) {
            ret = avl_del( &(root->l), x );
            if( ret == BT_DELETED ) {
                if( root->r ) avl_check_left(proot);
                else root->height = HEIGHT(root->l) + 1;
            }
            return ret;
        } else if( root->key < x->key ) {
            ret = avl_del(&(root->r), x);
            if( ret == BT_DELETED ) {
                if( root->l ) avl_check_right(proot);
                else root->height = HEIGHT(root->r) + 1;
            }
            return ret;
        } else {  // root->key == key
            return avl_del_node(proot);
        }

    } else {
        return BT_NOTFOUND;
    }
}

/* Удаляет конкретный выбранный узел, на который указывает *proot.
 */
int avl_del_node(node_t **pnode) {
    node_t *node;
    node = *pnode;
    if( node->l && node->r ) {
        // когда оба ребёнка присутсвуют, найдём самый левый узел
        // из  правого поддерева и "оторвём" его, поставив на
```

```
        // его место его правого ребёнка;
        node_t *p = node;
        node_t *c = node->r;
        while( c->l ) { p = c; c = c->l;}
        if( p != node ) {
            p->l = c->r;
        } else {
            p->r = c->r;
        }
        node->key = c->key;
        node->value = c->value;
        free(c);
        avl_left_height( &(node->r) );
        if(node->l) avl_check_right( pnode )
    } else {
        // когда один из детей отсутствует,
        // возьмём второго ребенка;
        if( node->l != 0 ) {
            *pnode = node->l;
        } else if( node->r != 0 ) {
            *pnode = node->r;
        } else {
            // если и второго ребенка нет, то node ---
            // это лист, и его можно просто удалить;
            *pnode = 0;
        }
        free(node);
    }
    return BT_DELETED;
}

/* Подправляет значения высот поддеревьев всех левых потомков.
 * (необходима после удаления самого левого узле из поддерева root)
 */
int avl_left_height(avl_t **proot) {
    register avl_t *root = *proot;
    if(root) {
        root->height = 1+max(avl_left_height(&(root->l)), HEIGHT(root->r));
        if(root->r) avl_check_left(proot);
        return root->height;
    } else {
        return 0;
    }
}
```

Тестирование АВЛ-дерева

Важным этапом разработки программных продуктов является тестирование. В программистких фирмах на тестирование, исправление ошибок и доработку уходит значительная доля всех трудозатрат.

Программисту следует взять за правило писать тестирующие функции для своих программ (библиотек функций, классов), и прежде, чем объявлять о готовности кода, осуществлять массовую всестороннюю проверку своего кода.

Задача C13.6. Посмотрите на код 13.6 функции `avl_check`, которая проверяет консистентность АВЛ-дерева. Верно ли, что успешное выплнение этой функции гарантирует 100% консистентность АВЛ-дерева, то есть истинность *всех* условий, накладываемых на АВЛ-дерево. Если функция проверяет консистентность лишь частично, допишите её, чтобы проверка была полной.

Программа 13.6: Функция проверки консистентности АВЛ-дерева

```
/* Проверяет поддерево и возвращает его высоту. */
int avl_check ( node_t *root ) {
    if( root ) {
        int hl = avl_check(root->l);
        int hr = avl_check(root->r);
        int h = 1 + max(hl, hr);
        if( h != root->height ) {
            printf("hl=%d, hr=%d\n", hl, hr);
            printf("h!=root->height for key %d: %d!=%d\n",
                            root->key, h, root->height);
            exit(1);
        }
        if( abs(hl - hr) > 1 ) {
            printf("Tree is not AVL-balanced for root (%d, %d)",
                           root->key, root->value);
            exit(1);
        }
        if( root->l ) {
            if(root->key <= root->l->key) {
                printf("Keys are not ordered: %d, %d )",
                                root->key, root->l->key );
                exit(1);
            }
        }
        if( root->r ) {
            if( root->key >= root->r->key ) {
                printf("Keys are not ordered: %d, %d )",
                                root->key, root->r->key );
                exit(1);
            }
        }
```

```
        return h;
    } else {
        return 0;
    }
}
```

Вызов функции `avl_check` во время отладки можно поставить в различных местах кода. Можно также просто проверять отдельные условия консистентности, и в случае их невыполнения осуществлять аварийный выход из программы с выводом информации об ошибке. Для описания таких проверяющих условий или утверждений принято использовать функцию `assert`, объявленную в заголовочном файле `assert.h`. Программист как бы утверждает, что данное условие должно быть выполнено в данном месте кода. Условия проверки обычно пишут вначале кода функций (англ. *prerequisites* — условия на корректность данных и общего состояния перед выполнением), и перед выходом из функции (англ. *postrequisites* — условие на корректность данных после выполнения). Также такие условия полезно ставить до и после ключевых участков кода. Этот подход к программированию и отладке назвается **проверяющий себя код** (англ. *self-testing code*).

Важно также отметить, что код должен быть не только self-testing, но и *self-describing*. А менно, в программном коде должно присутствовать всё необходимое для понимания логики программы не только автором, но и любым сторонним грамотным программистом. Утверждения (asserts) помогают понять, из каких предположений исходил программист, когда писал код[2]. Другим важным моментом в понимании чужого кода являются *комментарии к коду*. Комментарии принято писать перед описанием функций (что функция делает, что принимает в качестве аргументов и что возвращает), а также в строчках объявления ключевых переменных (что хранится в данной переменной).

Во время написания кода часто возникает необходимость рассмотрения специальных крайних случаев, которые выпадают из основной логики программы[3]. Эти случаи необходимо также пояснять в комментариях, написанных рядом со соответствующими условными операторами.

Признаком хорошо спроектированного и написанного кода является отсутствие необходимости в комментариях — код самоочевиден без комментариев. Это свойство достигается с помощью правильного именования функций и переменных, а также правильного структурного построения кода[4]. Свойство самоочевидности кода можно достичь, используя классические, легко узнаваемые шаблоны, которые используются большинством программистов для ряда типичных задач. Также советуем вам не «мудрствовать лукаво» и писать код, непосредственно отображающий логику, которая в него заложена, и не стремится к излишней эффективности

[2]Не представленные явно условия на обрабатываемые данные принято называть **призраками** [**?**]. Эти призраки полезно делать «материальными», используя технику утверждений (asserts).

[3]Программисты называют такие случаи **подпорками**. Если в коде встречается слишком много подпорок, то, можно сделать вывод, что основная логика и архитектура программы выбраны неверно. Призраки и подпорки — это два типа сущностей, которые следует, по возможности, исключить из кода, а если они всё таки присутствуют, то написать к ним комментарий.

[4]Например, при правильном построении кода первый блок оператора `if` (выполняемый, когда условие оператора выполено) должен соответствовать основной логике выполнения программы, а второй — специальному случаю. Подобных правил существует множество, часть из них приведены в книге [2].

кода, там где она не нужна, в ущерб наглядности кода. Вы облегчите жизнь не только своим коллегам-разработчикам (и преподавателям), но и себе, когда будете через месяц дописывать свой собственный код.

Задача C13.7. Изучите код 13.7 функции `avl_simple_test`. Эта функция реализует один из самых простых случаев тестирования АВЛ-дерева. Напишите функцию `try_to_delete`, которая не представлена в коде 13.7. Напишите функцию `main`, которая несколько раз вызовет функцию `avl_simple_test` для различных значений параметров. Напишите несколько своих функций тестирования, стараясь охватить как можно больше крайних случаев. Проведите *массовое* тестирование АВЛ-дерева, несколько раз добавляя и удаляя из него по миллиону и более записей.

Программа 13.7: Тестирование АВЛ-дерева

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "avl.h"

void try_to_insert(node_t **proot, int key, int value);
void try_to_del(node_t **proot, int key);
void should_exist(node_t **proot, int key, int value);
void should_not_exist(node_t **proot, int key);
void simple_test(int n, int a, int c, int mod);
int  GCD(int a, int b);

int main() {
    int i;
    printf("1. simple_test(100, 1, 99, 100).\n");
    simple_test(100, 1, 99, 100);

    printf("2. simple_test(1000, -1, 2, 1000).\n");
    simple_test(1000, -1, 2, 1000);

    for(i = 3 ;  i <= 50 ; i++) {
        int P = 17; // должно быть простым числом
        int a = 1 + rand() % (P-1);
        int n = 1 + rand() % (P-1);
        int b = rand() % P;
        printf("%d. simple_test(%d, %d, %d, %d).\n", i, n, a, b, P);
        simple_test(n, a, b, P);
    }
    return 0;
}

/* Осуществить несколько циклов добавления и удаления
 * пар вида (key, i) где key = (a*i + c) % mod.
 */
```

```
void simple_test(int n, int a, int c, int mod) {
    node_t *root = 0;
    int i;
    assert( n <= mod && GCD(a,mod) == 1 );
    for(i = 0 ; i < n ; i++) {
        try_to_insert   ( &root, (a*i + c) % mod, i );
        should_exist ( &root, (a*i + c) % mod, i );
    }
    // avl_prefix_traverse(root, print_indent_node, 0);
    avl_check(root);
    for(i = 0 ; 2 * i < n  ; i++) {
        try_to_del       ( &root, (a*i + c) % mod);
        //avl_prefix_traverse(root, print_indent_node, 0);
        //avl_check(root);
        should_not_exist ( &root, (a*i + c) % mod );
    }
    avl_check(root);

    for(i = n-1 ; i >= 0  ; i--) {
        try_to_insert   ( &root, (a*i + c) % mod, i);
        should_exist ( &root, (a*i + c) % mod, i );
    }
    for(i = 0 ; i < n ; i++) {
        try_to_del       ( &root, (a*i + c) % mod);
        should_not_exist ( &root, (a*i + c) % mod);
    }
    if( root != 0) {
        printf("Tree is not empty after deletion.\n");
        exit(1);
    }
}

void try_to_insert(node_t **proot, int key, int value) {
    node_t x = {0,0,0,0,0};
    int t;
    x.key = key; x.value = value;
    t=avl_insert(proot, &x);
    if( t != BT_INSERTED && t != BT_UPDATED ) {
        printf("Can't insert (%d, %d). Ret = %d\n", key, value, t);
        exit(1);
    }
}

void try_to_del(node_t **proot, int key) {
    node_t x = {0,0,0,0,0};
    int t;
```

```
    x.key = key;
    t=avl_del(proot, &x);
    if( t != BT_DELETED && t != BT_NOTFOUND ) {
        printf("Can't delete (%d, ?).\n", key);
        exit(1);
    }
}

void should_exist(node_t **proot, int key, int value) {
    node_t x = {0,0,0,0,0};
    x.key = key;
    if( avl_find(proot, &x) == BT_FOUND ) {
        if( x.value != value) {
            printf("Bad value %d for key %d. %d expected.\n",
                x.value, key, value);
            exit(1);
        }
    } else {
        printf("Can't find (%d, ?).\n", key );
        exit(1);
    }
}

void should_not_exist(node_t **proot, int key) {
    node_t x = {0,0,0,0,0};
    x.key = key;
    if( avl_find(proot, &x) == BT_FOUND ) {
        printf("I found not existing pair (%d, %d).\n",
             key, x.key, x.value);
            exit(1);
    }
}

/* Наибольший общий делитель */
int GCD(int a, int b) {
    while(b != 0) {
        int c = a % b;
        a = b;
        b = c;
    }
    return (a > 0)? a : -a;
}
```

Инструменты визуализации графов

Один из простейших способов увидеть содержание и структуру дерева с коренем `root`— это вызвать функцию `prefix_traverse(root, print_indented, 0)`.

Вот пример вывода этой функции:

```
4
   1
      0
         nil
         nil
      2
         nil
         nil
   16
      8
         5
            nil
            nil
         nil
      19
         nil
         nil
```

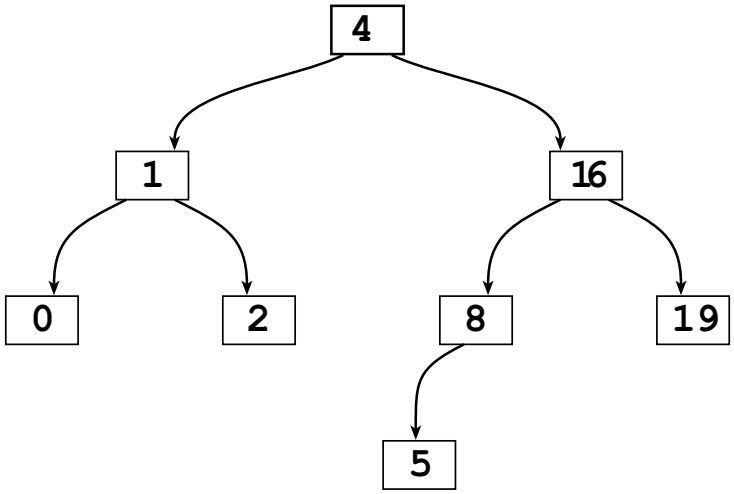


Рис. 13.2: «Префиксное описание» дерева слева соответствует дереву, изображённому на рисунке.

Префиксное описание достаточно несложно читать (интерпретировать) пока оно имеет маленький размер. Для больших деревьев получить наглядное представление о структуре и мере сбалансированности дерева очень сложно (примерно можно оценить среднюю глубину листьев,но не более).

Задача визуализации и анализа данных — один из самых популярных классов задач, возникающих на практике. В общем виде её можно сформулировать следующим образом:

Фундаментальная проблема 2: Визуализация и анализ данных.
Для множества данных с некоторой структурой создать наглядное визуальное представление (таблицу, диаграмму, блоксхему, древовидное представление, какую-то сложную картинку) отображающее интересующие пользователя свойства данных.

Для визуализации деревьев (и вообще, графов) уже разработан целый ряд инструментов. Один из них мы рассмотрим — это набор инструментов GraphViz[5], который распространяется свободно под лицензией подобной GPL[6].

В системе Graphviz принят специальный формат `dot` описания графов. В частности, указанное дерево будет записано в следующем виде:

[5] `http://www.graphviz.org` — Graph Visualization Software.

[6] GPL, англ. *GNU Public License* — лицензия свободнораспростоняемых программных продуктов, разработанная в рамках проекта GNU (`http://www.gnu.org`).

```
digraph test {
    4->1;  4->16;  1->0;  1->2;  16->8;  8->5; 16->19;
}
```

В системе Graphviz есть несколько различных алгоритмов представления графов на плоскости, среди них два наиболее популярных — `dot` и `neato`.

Алгоритм `dot` получает на вход ориентированный граф и рисует его так, чтобы большая часть рёбер была направлена в одном направлении, и при этом было как можно меньше пересечений.

Алгоритм `neato` рисует неориентированный граф на плоскости так, чтобы соединённые ребром вершины находились поблизости. В его основе его лежит пружинная модель. А именно, вершинам графа ставятся в соответствие точечные массы, а рёбрам — пружинки некоторой длины. Кроме того, между каждой парой не связанных ребром вершин устанавливается отталкивающая пружина. Алгоритм ищет локальный минимум потенциальной энергии этой динамической системы.

Кроме того, алгоритму `neato` можно передавать описания графов, в которых уже указаны координаты некоторых узлов. Он найдёт подходящее положение для остальных узлов графа и проведёт рёбра. В описании у рёбер и узлов в квадратных скобках можно указывать различные аттрибуты, в частности, `width` и `height` — ширину и высоту узла, `pos` — координаты узла, `shape` — форму узла (circle, ellipse, box, triangle, ...), `lavel` — текстовую метка, `fontcolor` — цвет текстовой метки, и много других атрибутов. Приведём пример:

```
# Файл fib3.dot

digraph fib3 {
 node [width="0.5",height="0.4",shape="box"];
 1 [pos="0.5,18.0"];
 4 [pos="2.0,17.0"];
 3 [pos="1.5,18.0"];
 3 -> 4 [label="r"];
 2 [pos="1.0,19.0"];
 2 -> 1 [label="l"];
 2 -> 3 [label="r"];
}
```

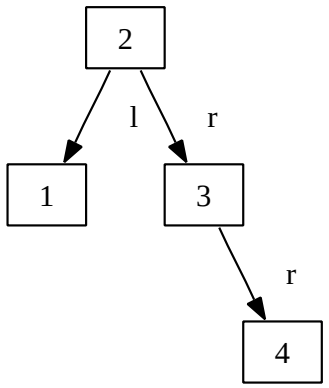


Рис. 13.3: Описание дерева в формате `dot` слева соответствует дереву, изображённому на рисунке справа.

С помощью следующей команды

```
> neato fib3.dot -n2 -Tps -o fib3.ps
```

из описания дерева в формате `dot` (сохранённого в файле `fib3.dot`) можно получить файл `fib3.ps` с изображением дерева в формате PostScript. Если входной файл не указывается, описание берётся из стандартного потока ввода.

Задача С13.8. Прочитайте определение 13.3 фибоначчиевых деревьев на стр. 247. Изучите программу 13.8, которая получает на вход натуральное число n и выводит описание фибона-

ччиевого дерева T_n. Скомпилируйте программу в запуускаемый файл `fib_gen` и, используя команду

```
> echo 5 | ./fib_gen | neato -n2 -Tps -o fib5.ps
```

получите изображение фибоначчиевого дерева T_5. Напишите цикл на языке bash, который создаст первые 10 фибоначчиевых деревьев. Попробуйте убрать опцию `-n2`, которая говорит, что не нужно искать оптимальных положений узлов, а взять те, которые даны в описании. Попробуйте заменить `dot` на `neato`.

Программа 13.8: Программа генерации описаний фибоначчиевых деревьев

```
#include <stdio.h>

int id = 1;

int print_fib_tree(int n, int depth) {
    int root, left, right;
    if(n > 0) {
        left = print_fib_tree(n-2, depth + 1);
        root = id++;
        right = print_fib_tree(n-1, depth + 1);
        printf("%u [pos=\"%.1lf,%.1lf\"];\n",
                    root, 0.5*root, 20 - 1.0*depth );
        if(left)  printf(" %u -> %u [label=l];\n", root, left);
        if(right) printf(" %u -> %u [label=r];\n", root, right);
        return root;
    } else {
        return 0;
    }
}
int main(int argc, char *argv[]) {
    int i, n;
    scanf("%d", &n);
    printf("digraph fib%d {\n", n);
    printf(" node [width=0.5, height=0.4, shape=box];\n");
    print_fib_tree(n, 1);
    printf("}\n");
    return 0;
}
```

Задача С13.9. Напишите функцию `avl_print_dot(node_t *root, int depth)`, которая бы выводила описание дерева с конем `root` в формате `dot` подобно тому, как это делает функция `print_fib_tree`. Получите изображение обычных двоичных деревьев поиска и АВЛ-деревьев, которые получаются в результате добавления одного и того же множества ключей. Попробуйте упорядоченные и неупорядочненные последовательности ключей.

Лекция 14

Структуры данных: красно-чёрные деревья

Краткое описание: В предыдущей была рассмотрена структура данных «АВЛ-дерево». В этом дереве гарантируется, что дерево сбалансировано, а именно, глубина дерева ограничена логарифмической функцией от числа узлов. Красно-чёрные деревья — это ещё один подход к балансировке двоичных деревьев поиска, который чаще используют на практике, так как он демонстрирует лучшие временные показатели. Мы изучим принцип, на котором основаны красно-чёрные деревья.
Кроме того, мы разберём задачу тестирования и отладки кода, приведём несколько ключевых подходов, позволяющих сделать отладку и тестирования эффективными.
Ключевые слова: красно-чёрное дерево, тестирование и отладка, массовое тестирование, утверждения (asserts), самопроверка (self-checking) и самоописание (self-describing) кода.

Красно-чёрные деревья

Красно-чёрные деревья — это ещё одна модификация двоичных деревьев поиска, для которых гарантируется логарифмическая зависимость высоты от числа узлов.

Красно-чёрное дерево (red-black tree)[1] — это двоичное дерево поиска, вершины которого разделены на красные (red) и чёрные (black). Таким образом, каждая вершина хранит один дополнительный бит — её цвет.

При этом должны выполнятся определённые требования, которые гарантируют, что высоты любых двух листьев отличаются не более чем в два раза.

Каждая вершина красно-чёрного дерева кроме полей *key* и *value* имеет поля *color* (цвет), l (указатель на левого ребёнка), r (указатель на правого ребёнка), p (указатель на родителя). Если у вершины отсутствует ребёнок или родитель, соответствующее поле содержит значение `nil`. Для удобства мы будем считать, что значение `nil`, являются указателями на фиктивные пустые листья дерева. В таком пополненном дереве каждая старая вершина (содержащая ключ) имеет двух детей, и тем самым становится внутренней вершиной (не листом).

(г) Двоичное дерево поиска называется красно-чёрным деревом, если оно обладает следующими свойствами (RB-свойствами, red-black properties):

1) каждая вершина — либо красная, либо чёрная;
2) каждый лист (`nil`) — чёрный;
3) если вершина красная, то оба её ребёнка чёрные;

[1] Определение взято из книги [1].

4) все пути, идущие вниз от корня *root* к листьям, содержат одинаковое количество чёрных вершин; это число называется **чёрной высотой дерева (black height)** и обозначается как $bh(root)$.

Пример красно-чёрного дерева показан на рисунке 14.1. Самый короткий путь от корня к листу может состоять из одних чёрных узлов. Самый длинный путь — это путь, в котором каждая вторая вершина — красная.

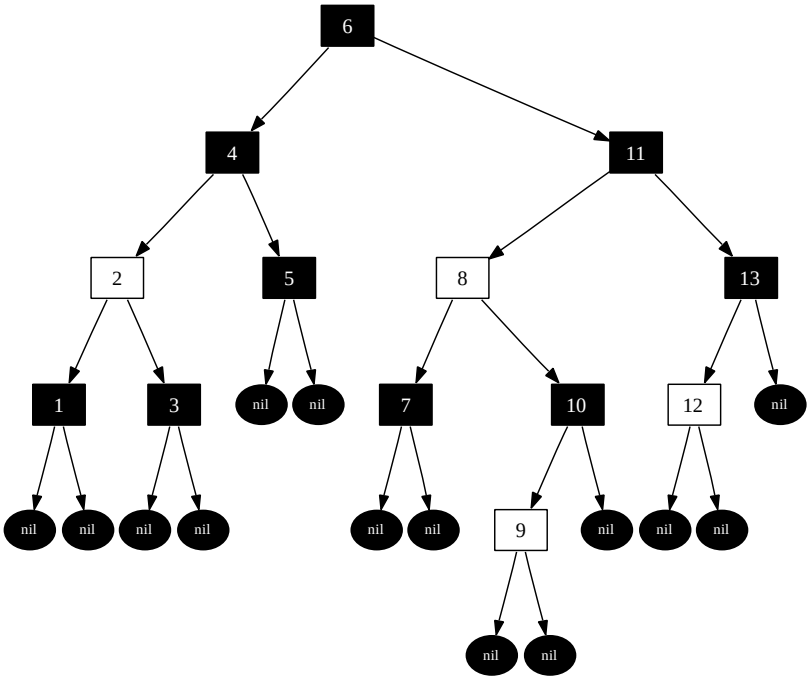


Рис. 14.1: Пример красно-чёрного дерева. Чёрные вершины показаны ка чёрные, а красные — как светлые. Все `nil`-листья чёрные и других листьев нет. Каждый путь от корня дерева вниз к листьям содержит ровно 4 чёрных вершины. В каждой вершине указано число — порядковый номер ключа, который в ней хранится.

Теорема 14.1. *Красно-чёрное дерево с N внутренними листьями (т.е. не считая* `nil`*-листьев) имеет высоту не более* $2\log(N+1)$.

Доказательство. Сначала покажем[2], что поддерево с коренем x содержит по меньшей мере $2^{bh(x)}$ внутренних вершин, где $bh(x)$ — число чёрных вершин на пути от x к листу `nil` (включая узел `nil`). Доказательство проведём индукцией от листьев к корню. Для листьев чёрная высота равна 0, и поддерево в самом деле содержит не менее $2^{bd(x)} - 1 = 2^0 - 1 = 0$ внутренних вершин. Пусть теперь вершина x не является листом и имеет чёрную высоту k. Тогда оба её ребёнка имеют чёрную высоту не меньше $k-1$ (красный ребёнок будет иметь высоту k, а чёрный $k-1$). По предположению индукции и левое и правое поддерево вершины x содержат не менее $2^{k-1}-1$ внутренних вершин, и поэтому поддерево с корнем в x содержит не менее $1+(2^{k-1}-1)+(2^{k-1}-1) = 2^k-1$ внутренних вершин. Таким образом, предположение индукции доказано. Чтобы завершить доказательство, обозначим высоту дерева (высоту корня) через H. Согласно свойству 3, по меньшей мере половину всех вершин на пути от корня к листу, не

[2]Доказательство взято из книги [1].

считая корень, составляют чёрные вершины. Следовательно чёрная высота дерева не меньше $H/2$. Тогда

$$N \geqslant 2^{H/2} - 1.$$

Перенесём 1 в левую часть и перейдём к логарифмам. Получим $H \leqslant 2\log(N+1)$. □

Высота красно-чёрного дерева (дерева со свойством (г)), также как в случае идеально сбалансированного дерево и АВЛ-дерева, логарифмически зависит от числа узлов, а именно,

г) $H \leqslant 2 \cdot \log_2(N+1)$.

Таким образом, высота красно-чёрного дерева, может быть примерно в два раза больше, чем высота идеально сбалансированного дерева, хранящего те же самые ключи. Это больше, чем в случае АВЛ-деревьев. Отсюда видно, что условие красно-чёрного дерева слабее, чем условие АВЛ-сбалансированности.

Задача Л14.1. Рассмотрим следующее условие на дерево: для любого узла высота правого и левого поддерева H_r, H_l удовлетворяют условиям

$$H_r \leqslant 2H_l + 1, \quad H_l \leqslant 2H_r + 1,$$

то есть отличаются не более чем в два раза с учетом `nil`-листьев. Назовём это условие G-свойством. Равносильно ли это условие условию красно-чёрного дерева? Ответ обоснуйте: докажите или приведите пример.

Задача Л14.2.* Пусть $K(H)$ — минимальное число узлов в дереве высоты H, для которого выполено G-свойство. Найдите асимптотику $K(H)$. Найдутся ли такие константы $A > 1$ и $C > 0$, что $K(H) > C \cdot A^H$, начиная с некоторого H?

Проблема верификации кода. Методы отладки кода

После того, как на практических занятиях вы напишете код, реализующий АВЛ-дерево или красно-чёрное дерево, возникнет задача тестирования вашего кода. Давайте рассмотрим процессы тестирования и отладки подробнее.

То что, ваша программа компилируется и как-то работает в простых случаях, безусловно не является критерием правильности её работы для *всех* входных данных. В реальности всегда будут ошибки, которые необходимо находить с помощью **тестирования**, а затем устранять с помощью **отладки** (англ. *debugging* — устранение ошибок).

Программа не считается написанной, пока не пройдёт тестирование.

Следует рассмотреть три основных аспекта тестирования в масштабе всего проекта: *что тестировать, как тестировать и когда тестировать.*

Что тестировать. Есть несколько свойств программ (компьютерых систем), которые могут и должны тестироваться:

- правильность работы (безошибочная работа программы для всех входных данных, отсутствие ошибок исполнения и «зависаний»);
- производительность (скорость работы, использование памяти, загруженность различных устройств компьютера, вызваемая программой);
- удобство использования, продуманность средств взаимодействия с пользователем (графические интерфейсы, возможность гибкой настройки, широта класса решаемых задач).

Есть и другие свойства. Каждому свойству соответствует определённый тип ошибок, которые полезно связать с этапами разработки программы (это мы сделаем несколько ниже).

Как и когда тестировать. Практика показала, что тестировать программу нужно начинать как можно раньше (отдельные написанные блоки и функции можно начинать тестировать до того, как реализована вся необходимая функциональность). Кроме того, это нужно делать часто, чтобы ошибки не накапливались и не усложняли своим совместным действием процесс их поиска и устранения. Но чтобы частое тестирование не отнимало у программиста много времени, необходимо автоматизировать этот процесс. Таким образом, к на вопрос «как и когда тестировать» можно коротко ответить так:

Тестировать нужно рано, часто и автоматически.

Существует подход к тестированию, условно называемый «непрерывной интеграцией» или «безжалостным тестированием», в котором написание кода начинается с написания системы автоматического тестирования, после чего процесс тестирования идёт непрерывно — после каждого значительного изменения в коде. Умеренное следование этому подходу может существенно повысить эффективность и качество разработки.

Рассмотрим типы ошибок, методы тестирования и отладки кода более подробно.

Типы ошибок

Разработка программного обеспечения делится на этапы (пере)проектирования, реализации и тестирования. На первом этапе обычно рисуются картинки, разрабатываются и проверяются математические основания. На втором этапе пишется программный код и происходит отладка кода на простых примерах.

Третий этап заключается в проверке того, что программа работает правильно[3].

Соответственно на каждом из трех этапов совершаются, находятся и устраняются ошибки определённых типов. К сожалению, ошибки проектирования (неудачно выбранный язык програмирования, плохо проработанная базовая библиотека, инструментарий, наэффективная структура базы данных и т.п) совершаются на самом первом этапе, обнаруживаются иногда уже после внедрения системы, а устранению часто просто не подлежат (устранение таких ошибок равносильно переписыванию системы с нуля). Обсуждение этих ошибок, чьи последствия наиболее катастрофические, мы вынуждены до поры до времени отложить.

Обычные ошибки, с которыми встречается программист, могут быть следующих типов:

[3]Речь идёт не только о разработке программ, а также о разработке компьютерных систем и программных компонентов: библиотек функций, классов, потоколов передачи данных и др.

Ошибки компиляции (compilation errors) — ошибка, которая обнаруживаются во время компиляции программы; они связаны с тем, что в коде была допущена опечатка, пропущена точка с запятой, забыли включить нужный заголовочный файл и т.п. Например, «syntax error before ..» — ошибка синтаксиса, «undeclared identifier ...» — используется необъявленная переменная, «too many arguments in ...» — функции передано больше аргументов, чем указано в объявлении функции.

Ошибки компоновки (linking errors) — ошибки, связанные с тем, что описания не всех используемых функций были найдены во время компоновки программы. Откомпилированные тела используемых функций компоновщик ищет среди объектных файлов, в которые скомпилились исходные файлы проекта, либо в библиотеках, которые подключаются во время компоновки. Если компоновщик не находит тела одной из используемых функций, он выдает ошибку «undefined reference» (ссылка на функцию, которая не определена).

Аварийный останов (runtime errors) — ошибки, возникающие на этапе выполнения программы. Самая распространённая ошибка, возникающая у программистов на языке Си — «Segmentation fault». Её можно получить, задав неверный индекс у массива при инициализации элемента массива: «int a[100]; a[1000] = 0;». Ошибки такого типа возникают тогда, когда программа выполняет недопустипую операцию (деление на ноль, переполнение стека, извлечение или запись данных по недоступному адресу, вызов функции по недоступному адресу и т.п.).

Неверный результат (wrong answer) — ошибки в логике работы программы, или в результате, который программа возвращает в каком-либо виде. Это самые трудно идентифицируемые ошибки, их устранение обычно занимает большую часть всего этапа отладки.

Самые безболезненные, легко обнаруживаемые и также легко устранимые — это ошибки первого и второго типов. Сообщения компилятора обычно чётко излагают причину, по которой он не может скомпилировать код. Конечно, для правильной интерпретации этих сообщений необходим некоторый опыт. Но он довольно быстро приходит у новичков. На этапе обучения важно не игнорировать сообщения об ошибках, а читать их, и спрашивать значения непонятных терминов у специалистов.

Следующие два типа ошибок возникают после того, как программа скомпилировалась и запустилась. Обнаружить их также просто, но локализовать их причину — гораздо сложнее.

Верификация посредством массового тестирования

Последние два типа ошибок связаны с тем, что запрограммированая логика работает не так, как ожидает того программист.

Ошибки такого рода в принципе не могут быть обнаружены (до поры до времени, «пока ракета не полетит в космос»), если не проводить массовое тестирование программы или не осуществлять верификацию правильности работы программы одним из существующих средств (методов) верификации.

Фундаментальная проблема 3: Верификация правильности кода.
Созданная система всегда требует тестирования. То, что программа компилируется и успешно отрабатывает на нескольких тестовых данных не является критерием того, что программа работает так, как того требует *спецификация*. Возникает серьезная и, в значительной степени, ещё не решённая проблема полной верификации правильности работы системы. Одним из ключевых методов проверки является массовое тестирование системы, то есть запуск системы для большого количества самых разнообразных тестов.

Желательно, чтобы тесты покрывали различные крайние случаи (граничные условия), и были достаточно сложными, чтобы проверить работу системы при больших нагрузках (входных данных большого размера). Этот метод вполне естественный и часто применяется на практике. Но ясно, что он не гарантирует 100%-ую верификацию системы. В настоящий момент активно разрабатываются альтернативные методы верификации, в которых происходит не тестирование, а *доказательство правильности работы системы*, на основании информации закладываемой при реализации функций. Эта задача доказательства во многих случаях оказывается NP-сложной или даже неразрешимой.

Проблеме верификации уделяется недостаточное внимания программистами-новичками. Необходимо понимать, что ошибки, обнаруженные после запуска системы в большинстве случаев оказываются существенно более дорогими, нежели средства, которые могли бы быть потрачены на дополнительный цикл тестирования.

Тестирование АВЛ-дерева и интерфейса «телефонная книжка»

Давайте обратимся к рассмотренной нами задаче «Телефонная книжка» и её решении с помощью АВЛ-деревьев. К сожалению, полной системы тестов в данном случае (как и в большинстве практических задач) создать невозможно. Но давайте попробуем обозначить основные шаги тестирования интерфейса «телефонная книжка».

Во-первых, необходимо проверить правильность и скорость работы в случае, когда добавляемые записи упорядочены по возрастанию или по убыванию ключей. Число записей должно быть как можно больше (например, 1000000 или больше, в зависимости от того, сколько у вас времени на тестирование — 10 секунд, день или неделя). На первом этапе тестирования можно ограничится числом записей порядка нескольких тысяч, чтобы тестирования происходило за секунды, а не за минуты.

Гораздо важнее не объём данных, а разнообразие типов входных данных, чтобы все функции, присутствующие в реализации структуры данных, отработали много раз для различных значений аргументов.

В случае АВЛ-дерева, необходимо проверить, как работают все типы вращений, как происходит удаление листьев и корня, успешно ли обновляются данные, привязанные к существующему узлу, успешно ли (без ошибок исполнения) отрабатывают функции поиска и удаления, когда записи с указанным значением ключа не существует.

Важно проверить также как работает удаление, если удалять узлы в случайном порядке, а также в порядке возрастания или убывания ключей. То есть необходимо проверить все крайние случаи (*протестировать граничные условия*).

После каждого запроса на удаления имеет смысл проверить, что запрос `find` действительно не может найти запись с удалённым ключём. Аналогично после каждого запроса на добавление можно проверять, что нужная запись находится, и привязанные к ключу данные именно те,

что были указаны в запросе на добавление.

Для тестирования АВЛ-дерева по сути необходимо писать большую программу, которая генерирует искусственные данные и в нескольких циклах многократно вызывает функции `find`, `insert`, `del`. Удобно определить некоторую функцию, которая по значению ключа генерирует привязанное к нему значение так, чтобы значение однозначно определялось ключём — это упростит процесс проверки.

Важно отметить, что при написании тестирующих программ не нужно «мудрить», иначе придётся писать программу, которая тестирует вашу тестирующую программу.

Первые тесты должны быть максимально простыми. Они должны проверять, работает ли ваша программа в простейших случаях. Затем следует постепенно усложнять тесты.

Когда программа ошибается на простом тесте, то и ошибку найти тоже просто. Причину ошибки на сложном большом тесте иногда не так просто проанализировать и выявить (хотя бы потому, что приходится долго ждать, когда загрузятся и оработаются данные).

Методы отладки кода

После того, как обнаружен тест, который «валит» программу, необходимо *локализовать причину ошибки*. Для этого многие программисты используют отладчики[4]. Важнейшая информацию, которую может предоставить отладчик, — это стэк вызовов функций, которая была выполнена к моменту, когда произошла ошибка.

Определение 14.1. **Стек вызовов** — *список активных (вызванных, но не завершённых) функций исполняющейся программы. Первой в этом списке идёт функция, которая исполняется в данный момент, затем функция, из которой произошёл вызов той функции, которая исполняется в данный момент. Третья в этом списке функция — это та функция, во время выполнения которой был осуществлён вызов второй в списке функции, и т.д. В стеке вызовов у функций указываются значения аргументов, с которыми они были вызваны.*

Чем отличается стек вызовов от истории вызовов функции (списка вызовов функций) за всё время выполнения программы?

После обнаружения места, где произошла ошибка, необходимо определить, какие из данных в программе некорректны. Возможно, какие-то указатели повреждены (corrupted) и указывают на неверные области памяти, либо численные переменные содержат слишком большие, отрицательные или какие-нибудь другие недопустимые для них значения.

[4] **Отладчик** — модуль среды разработки или отдельное приложение, предназначенный для поиска ошибок в программе. Отладчик позволяет выполнять пошаговое выполнение программы, отслеживать значения переменных в процессе выполнения программы, устанавливать точки или условия останова и т. д.

Компиляторы основных языков программирования поставляются вместе с отладчиком, часто входящим в состав среды разработки, которая объединяет в себе создание и редактирование исходного кода, компиляцию, выполнение и отладку. Отладчики имеют графический (Microsoft Visual Studio) или текстовый интерфейс (GNU debugger, GDB) и позволяют выполнять программу оператор за оператором, функция за функцией, с остановками на кокретных строках или при достижении какого-то условия. Они также предоставляют возможность отображения значений переменных.

В этом месте хотелось бы осуществлять пошаговый *откат совершенных действий* назад, чтобы найти место, где впервые была нарушена целостность данных.

В определённом смысле пошаговый отладчик программы здесь оказывается бесполезным, так как позволяет переходить к следующему действию, но не к предыдущему. Поэтому не спешите переходить в режим пошаговой отладки программы. Анализ работы программистов выявил, что многие программисты пользуются отладчиком крайне неэффективно, и тратят на него больше времени, чем могли бы, если бы в первую очередь анализировали код и возможные причины возникшей ошибки.

Вот что написано относительно отладки в книге «Практика программирования» [2]:

> «Наш личный выбор – стараться не использовать отладчики, кроме как для просмотра стека вызовов или же значений пары переменных. Одна из причин этого заключается в том, что очень легко потеряться в деталях сложных структур данных и путей исполнения программы. Мы считаем пошаговый проход по программе менее продуктивным, чем усиленные размышления и *код, проверяющий сам себя в критических точках.*
>
> Щёлкание по операторам занимает больше времени, чем просмотр сообщений операторов выдачи отладочной информации, расставленных в критических точках. Быстрее решить, куда поместить оператор отладочной выдачи, чем проходить шаг за шагом критические участки кода, даже предполагая, что мы знае, где находятся такие участки. Более важно то, что отладочные операторы сохраняются в программе, а сессии отладчика переходящи.
>
> Слепое блуждание в отладчике, скорее всего, непродуктивно. Полезнее использовать отладчик, чтобы выяснить состояние программы, в котором она совершает ошибку, затем подумать о том, как такое состояние могло возникнуть. Отладчики могут быть слодными и запутанными программами, особенно для новичков, у которых они вызовут скорее недоумение, чем принесут какую либо пользу...»

Очень часто быстрым и эффективным способом является **журналирование** — вывод отладочной информации в специальный файл (log-файл, от англ. *logging* — журналирование) или поток вывода. Вывод этой информации может включаться только в том случае, если программа скомпилирована в отладочом режиме (в debug-конфигурации, от англ. *debugging* — отладка, устранение ошибок). Кроме того, можно ввести различные уровни отладочной информации. Уровень 0 — это критические ошибки, после которых происходит аварийный останов выполнения программы. Уровень 1 — это уровень обычных, не критичных ошибок. Следующий уровень отводится под предупреждения — сообщения о том, что что-то идёт не совсем так, как ожидалось, но в программе для этого предусмотрены обходные пути и выполнение программы будет продолжено. Уровни 4 и 5 и так далее отводятся под подробную отладочную информацию, чем больше уровень тем более подробнее информация и тем меньше её важность. Умение правильно организовывать вывод отладочной информации и чувство того, какая информация к какому уровню отладки относится приходит с опытом исправления ошибок. После обнаружения каждой ошибки необходимо спрашивать себя: «Какая отладочная информация могла бы помочь мне быстро и эффективно обнаружить данную ошибку, и какая информация в log-файле оказалась ненужной, избыточной и затрудняющей анализ?»

Если какой-то тест не проходит, а Вы никак не можете найти ошибку в коде, не надо отчаиваться. Необходимо подходить к своим ошибкам по-философски — во-первых, ошибка

в конечном итоге будет найдена, а после после того, как вы её локализуете и надлежащим образом проанализируете, получите неизмеримо ценный опыт и знания, которые помогут вам не совершать подобных ошибок в будущем.

Снова процетируем отрывок из книги «Практика программирования» [2]:

> «Отладка сложна и может занимать непредсказуемо долгое время, поэтому цель в том, чтобы миновать большую её часть. Технические приёмы, которые помогут уменьшить время отладки, включают хороший дизайн, хороший стиль, проверку граничных условий, проверку правильности исходных утверждений и разумности кода, защитное программирование, хорошо разработанные интерфейсы, ограниченное использование глобальных переменных, автоматические средства контроля и проверки. Грамм профилактики стоит тонны лечения.»

Приведём под конец несколько простых практических советов, позволяющих оптимизировать этапы отладки и тестирования.

Ищите знакомые ситуации. Когда возникла ошибка, спросите себя, известна ли вам эта ситуация. Обычно, ошибки, совершаемые одним человеком, имеют один и тот же характер. Исследуйте себя и свой образ мышления.

Проверьте самое последнее изменение. В чём заключалось последнее изменение? Вероятнее ошибка находится в добавленном коде, а не в старом, который работал в большинстве случаев и уже был отлажен. Полезно использовать системы контроля версий, которые позволяют сравнивать версии файлов и «откатывать» их к предыдущим версиям.

Читайте код, перед тем как исправить. Один из эффективных, но недооценённых приёмов отладки — тщательное чтение и обдумывание кода перед внесением в него исправлений. Не рвитесь к клавиатуре с надеждой, что случайные исправления устранят ошибку. Скорее всего вы ещё не знаете причину ошибки, и исправляя код, можете сделать несколько новых ошибок.

Сделайте перерыв. Иногда вы видете в коде совсем не то, что вы там на самом деле написали. Чтобы убрать «пелену с глаз» необходим короткий отдых и свежий взгляд.

Объясните свой код кому-нибудь ещё. Такое объяснение помогает самому увидеть свою ошибку. Иногда требуется буквально несколько предложений — и звучит смущённая фраза: «Ой, я вижу, где ошибка, извини, что побеспокоил.» В одном университетском компьютерном центре рядом с центром поддержки стоял плюшевый медвежонок. Студенты, встретившиеся с таинственными ошибками, должны были сначала объяснить их этому медвежонку и только затем идти за помощью к специалисту.

Сделайте ошибку воспроизводимой. Убедитесь, что можете заставить ошибку появиться по вашему желанию. Потратьте время и найдите такую комбинацию условий, которая приводит к ошибке. Минимизируйте число действий, которые необходимо сделать, чтобы получить ошибку. Возможно, во время отладки вам придётся воспроизволить её снова и снова.

Следуйте принципу «разделяй и властвуй». Метод деления пополам можно использовать не только для эффективного поиска записей в телефонной книжке, но и для локализации ошибок. Исключайте (методом комментирования) участки кода, чтобы выявить, какие именно действия приводят к ошибке. Каждый тест должен быть нацелен на получение определённого резльтата, который подтверждает или опровергает какую-либо гипотезу о происходящем. Иногда можно применить метод деления пополам к входным данным. Отбросьте половину входных данных и проверьте, возникает ли ошибка. Если нет, то вернитесь к предыдущему состоянию и отбросьте другую половину данных.

Пишите код, который проверяет сам себя. Программист пишет куски кода, изходя из определённых предположениях о возможном состоянии глобальных переменных, о возможном значении аргументов функции. Обычно есть целый ряд условий, описывающих в каком случае данные, используемые в программе считаются *консистентными* (то есть целостными, коректными, согласованными). Все эти условия часто остаются «в голове программиста» и никак не отображены в коде. Н.Н. Непейвода предложил эти условия называть *призраками*. Одна из основных рекомендаций заключается в том, чтобы описать эти призраки, в первую очередь, в виде поясняющих комментариев, а также в виде явных проверок выполнения каких-либо условий. При программировании на языке Си можно пользоваться функцией `assert`, объявленной в файле `assert.h`. В аргументе этой функции следует писать условие, которое, как мы думаем, должно быть выполнено в данном месте кода, и изходя из которого мы пишем дальнейший код. Удобно также писать специальные функции, проверяющие *консистентность данных*, и вызывать их перед и после ключевых участков кода. Эти функции, если обнаруживают некорректные данные, выдают подробную информацию об неполадке и аварийно завершают выполнение программы. В самом конце этапа разработки, вызовы этих функций можно смело закомментировать, поскольку они уже выполнили свою работу и не нужны, а только замедляют выполнение программы.

Семинар 14

Структуры данных: красно-чёрные деревья

Краткое описание: На данном семинаре мы реализуем красно-чёрные деревья (возможно, в не полной мере будет реализована операция удаления) и исследуем эффективность их работы в сравнении с АВЛ-деревьями.

Реализация и тестирование красно-чёрного дерева

Реализацию красно-чёрных деревьев можно полулить небольшими изменениями АВЛ-дерева. Во-первых, поле `height` необходимо заменить на `color`. Во-вторых, для поддержания черноты корня дерева, нам придётся «обернуть» функции удаления и добавления записи так, как показано в коде 14.1. А именно, введём рекурсивные функции `rb_insert0` и `rb_del0`, которые подобны соответствующим функциям АВЛ-дерева, а также напишем функции `rb_insert` и `rb_del` которые осуществляют вызов функций `rb_insert0` и `rb_del0` и после проверяют, является ли корень чёрным, и если нет — закрашивают его в чёрный цвет.

Программа 14.1: Выдержки из реализации красно-чёрного дерева

```
#define COLOR(node) ((node == 0)? RB_BLACK : node->color)

int rb_insert(rb_t **proot, const rb_t *x) {
    int ret = rb_insert0(proot, x);
    if( (*proot)->color != RB_BLACK) (*proot)->color = RB_BLACK;
    return ret;
}
int rb_del(rb_t **proot, const rb_t *x) {
    int ret = rb_del0(proot, x);
    if( COLOR( (*proot) ) != RB_BLACK) (*proot)->color = RB_BLACK;
    return ret;
}
int rb_insert0(rb_t **proot, const rb_t *x) {
    rb_t *root;
    root = *proot;
    if(root) {
        int ret;
        if(root->key > x->key ) {
            ret = rb_insert0(&(root->l), x);
            if(ret == BT_INSERTED) {
                rb_check_right(proot);
```

```
            }
        } else if(root->key < x->key ) {
            ret = rb_insert0(&(root->r), x);
            if(ret == BT_INSERTED) {
                rb_check_left(proot);
            }
        } else {  // root->key == key
            root->value = x->value;
            return BT_UPDATED;
        }
        return ret;
    } else { // root == NULL, то есть (под)дерево пусто.
        // создаём новый узел, если добавление
        *proot = (rb_t*) malloc(sizeof(rb_t));
        (*proot)->key  = x->key;
        (*proot)->value = x->value;
        (*proot)->l = (*proot)->r = 0;
        (*proot)->color = RB_RED;
        return BT_INSERTED;
    }
}
void rb_left_rotate( rb_t** p) {
    rb_t *x = *p;
    (*p) = (*p)->r;
    x->r = (*p)->l;
    (*p)->l = x;
}
void rb_check_left(rb_t **p) {
    rb_t *r = (*p)->r;
    if( (*p)->color == RB_BLACK) {
        if( r->color == RB_MAKE_ME_BLACK ) {
            (*p)->color = RB_RED;
            if( COLOR( (*p)->l ) == RB_RED )  {
                r->color = (*p)->l->color = RB_BLACK;
            } else {
                r->color = RB_RED;
                if( COLOR( r->l ) == RB_RED )
                    rb_right_rotate( &((*p)->r) );
                rb_left_rotate( p );
                (*p)->color = RB_BLACK;
            }
        }
    } else if( r->color == RB_RED ) {
        (*p)->color = RB_MAKE_ME_BLACK;
    }
}
```

Удобно ввести третий тип цвета — `MAKE_ME_BLACK`, который будет назначаться вершинам, которые сами красные и у которых красные дети, но нет возможности решить эту проблему на данном уровне глубины дерева. Поднявшись к чёрному родителю этой вершины (после выхода из рекурсивного вызова) мы обнаружим, что один из детей желает стать чёрным и решим эту проблему с помощью одного или двух вращений.

В коде 14.1 также показана реализация функции `rb_check_left`, которую надо вызывать, когда есть подозрение в том, что левое дерево слишком сильно «полегчало», и появился перекос в правую сторону. Эту функцию нужно вызвать после удаления из левого поддерева или после добавления в правое дерево. Функция `rb_check_right` получается простой заменой `r` на `l` и `l` на `r`.

Наибольшую сложность представляет реализация функции удаления.

Задача C14.1. Напишите функцию `rb_check`, которая проверяет выполнение всех свойств красно-чёрного дерева с корнем `root`, кроме свойства одинаковости чёрной высоты. Напишите программу тестирования красно-чёрного дерева, аналогичную программе 13.7. На основе кода 14.1 напишите реализацию красно-чёрного дерева, при этом операцию удаления реализуйте в упрощённом виде на основе функции `avl_del_node` так, чтобы гарантировано сохранялись все свойства, кроме свойства одинаковости чёрной высоты левого и правого поддеревьев. Протестируйте свою реализацию красно-чёрного дерева. Запустите один и тот же массовый тест для АВЛ-дерева и красно-чёрного дерева с измерением времени работы (используйте программу `time`). Какой из двух тестов занял меньше времени?

Задача C14.2. Попробуйте реализовать операцию удаления так, чтобы гарантировано сохранялись все свойства красно-чёрного дерева и после этого повторите тестирование.

Визуализация красно-чёрных деревьев

Задача C14.3. Напишите программу генерации случайных красно-чёрных деревьев в формате `dot`. Атрибуты чёрных узлов должны быть следующими:

```
[style="filled", color="black",
        fontcolor="white", pos="<число>,<число>"]
```

Задача C14.4. Напишите функцию `rb_print_dot(node_t *root, int depth)`, которая выводит описание красно-чёрного дерева с корнем `root` в формате `dot`. Научитесь с помощью инструмента `neato` получать изображения красно-чёрных деревьев, подобных приведённому на рис. 14.1 на стр. 267.

Лекция 15

Структуры данных: хэш-таблицы

Краткое описание: Структура данных «хэш-таблица» также как и деревья поиска реализуют интерфейс «ассоциативный массив». Но в отличие от деревьев поиска, хэш-таблица не позволяет за линейное время осуществлять обход пар в порядке возрастания (или убывания) ключей. Мы рассмотрим классический вариант хэш-таблицы, а также один из наиболее быстрых вариантов, который называется двойным хэшированием с открытой адресацией.

В конце лекции мы сделаем обзор других решений и проблем, связанный с интерфейсом «ассоциативный массив». В частности, будут приведены примеры кодов на языках Си++ (с использованием шаблона `map` библиотеки шаблонов STL), Perl и Ruby.

Ключевые слова: ассоциативный массив, хэш-таблица, хэш-таблица с открытой адресацией, методология повторного использования (reuse).

Обзор реализаций интерфейса «ассоциативный массив»

Итак, мы рассмотрели три различные реализации интерфейса «ассоциативный массив»[1]: двоичное дерево поиска, АВЛ-дерево, красно-чёрное дерево (реализации на основе обычного массива или списка мы отбросим, как не применимые на практике). Кроме того, мы дали определения идеально сбалансированных деревьев, но реализации функций удаления и добавления пар, поддерживающих свойство идеальной сбалансированности, мы не привели. Вы можете разработать их самостоятельно.

Всё эти структуры данных — двоичные деревья, удовлетворяющие свойству дерева поиска: все ключи левого поддерева меньше ключа корня, а он меньше ключей правого поддерева.

Есть ряд обобщений деревьев поиска. Во-первых, разработано несколько типов деревьев поиска со степенью ветвления более двух. Они, в отличие от двоичных деревьев поиска, позволяют за один шаг уменьшать область поиска в среднем не в два раза, а большее число раз [2]. Среди таких деревьев отметим B-деревья и B^*-деревья, которые используются при реализации файловых систем и различных задачах хранения данных на дисковых накопителях.

Структура данных «хэш-таблица» является ещё одной реализацией интерфейса «ассоциативный массив», но она основана на другом принципе. У неё есть свои достоинства и недостатки. В частности, *амортизационно* (в сумме по всем запросам) хэш-таблица работает быстрее,

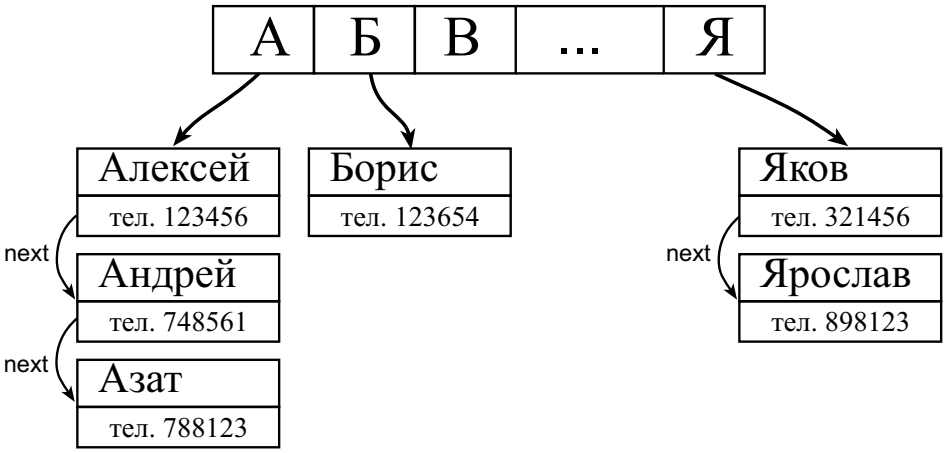
[1]Напомним, что интерфейс «ассоциативный массив» — это интерфейс хранилища пар (ключ, значение), с операциями добавления и удаления пар, а также поиска пары по ключу.

[2]Более точно, область поиска уменьшается за один шаг в два раза только в идеально сбалансированных деревьях. В АВЛ-деревьях в худшем случае область поиска уменьшается в $\varphi = \frac{\sqrt{5}+1}{2} \approx 1.618$ раз, а в красно-чёрных деревьях — в $\sqrt{2} \approx 1.414$ раз.

чем какие-либо деревья поиска. Но при этом хэш-таблица не гарантирует малое время выполнения отдельного запроса — возможны плохие входные данные, при которых отдельные запросы выполняются значительно дольше остальных. Хэш-таблицу удобно использовать, когда число хранимых пар заранее известно, и когда время выполнения отдельного запроса не так важно, а важно лишь суммарное время выполнения запросов.

Следует также отметить, что деревья поиска кроме операций `insert`, `find`, `del` позволяют выполнять операцию `traverse` — обход всех пар в порядке возрастания или убывания ключей за линейное от числа пар время. Хэш-таблица этого не позволяет.

Простая хэш-таблица



Предположим нам нужно хранить записи вида (имя, телефон). Давайте создадим массив из 32 списков: в первом будем хранить записи, в которых имя начинаются на букву “А”, во втором – записи с менеми на букву “Б” и так далее.

Когда нам приходит команда на поиск или удаление записи по имени, мы сразу же определяем список, с которым нам нужно работать, и значительно сужаем область поиска.

Если бы буквы имели одинаковую вероятность быть первой буквой имени, то размеры списоков в среднем в 32 раза были бы меньше, чем размер всей базы записей.

Идея 15.1. *Вместо первой буквы можно использовать некоторую функцию, которая зависит от всего имени и принимает бóльшое число возможных значений (например, от* 0 *до* 10000*). Эта функция должна быть такой, чтобы все возможные значения были равновероятны. Желательно, чтобы близкие, но не одинаковые входы, отображались в различные значения. Такая функция называется хэш-функцией (hash function)*[3].

[3] *hash* (анг.) — беспорядок, мешанина, неразбериха, путаница (в фактах и т. п.). Термин “хэш-функция” используют для функций, которые данные (строку, текст, число, набор чисел, изображение, файл, ...) отображают в небольшое целое число или строку фиксированной длины (в англоязычной литературе используется термин *fingerprint* — отпечаток пальца), которое называется **хэш-кодом**, таким образом, чтобы с одной стороны, хэш-код был значительно меньше исходных данных, но в то же время с большой вероятностью однозначно им соответствовал.

Ясно, что в общем случае однозначного соответствия между исходными данными и хеш-кодом быть

На рисунке 15 показана простая хэш-таблица, в которой в качестве хэшфункции выбран порядковый номер в алфавите первой буквы имени. Это плохая хэш-функция, так как распределение имён по значению первой буквы сильно не равномерно.

Опишем устройство **простой хэш-таблицы** более детально.

- Задается некоторое фиксированное число P (типичные значения от 100 до 1000000).
- Создается массив $index$ размера P из пустых списков, который называется индексом хэш-таблицы.
- Задается хэш-функция $hash$, которая принимает на вход ключ и возвращает число от 0 до $P - 1$.
- При добавлении пары $(key, value)$ вычисляется $h = hash(key)$ и происходит добавление пары в список $index[h]$, а имеено происходит обход всего списка и проверяетс, есть ли в этом списке пара с ключём равным key, если есть, то привязанное к нему значение заменяется на данное $value$. Если в списке $index[h]$ пары с таким ключём нет, то пара $(key, value)$ добавляется в конец списка.
- При удалении (поиске) пары с ключём key вычисляется $h = hash(key)$ и происходит удаление (поиск) в списке $index[h]$ соответствующей пары.

Можно считать, что списки, которые хранятся в индексе хэш-таблицы реализуют интерфейс ассоциативного массива и поддерживают операции `find`, `insert` и `del`. Первое, что мы делает при выполении операций `find`, `insert` и `del` для хэш-таблицы, — это вычисляем хэш-код $h = hash(key)$ и *перенаправляем запрос* списку $index[h]$. Хэш-код играет роль индекса в массиве $index$.

не может. Различные входные данные могут отобразится в одинаковые хеш-коды. Такие ситуация называют столкновениями (англ. **collisions**). Вероятность столкновений в каждой конкретной задаче должна быть сведена к минимуму правильным выбором хэш-функции.

Качество хэш-функции нестрого может быть определено следующим образом:
1. Для близких значений аргумента хэш-функция должна возвращать сильно различающиеся результаты (грубо говоря, хэш-функция не должна обладать свойством непрерывности, а наоборот, сильно “скакать”). Иначе, для случая однотипных входных данных вероятность столкновений будет велика.
2. Прообраз любого значения хэш-функции должен быть равномерно распределён по множеству возможных значений аргумента, а точнее в прообразе не должно быть кластеров — подмножеств близко стоящих друг от друга точек. Кроме того, размеры всех прообразов должны быть примерно одинакового размера, а точнее одинаковыми должны быть вероятностные меры этих прообразов — вероятность того, что входные данные отобразятся в некоторый хэш-код должна быть примерно одинаковой для всех возможных хэш-кодов.

Оба свойства подразумевают, что на множестве входных данных определена метрика и вероятностное распределение.

Определения обоих свойств не обладают математической точностью, так как понятие равномерности распределения над конечными множествами использовать неправомерно. Тем более неправомерно говорить о непрерывности функции, множество определения и множество значений которой являются конечными множествами. Тем не менее, данные свойства можно уточнить и выработать несколько алгоритмов, которые позволяют численно выражать и сравнивать качество хэш-функций.

Алгоритм 15.29 Функции для работы с простой хэш-таблицей

```
    ▷ t – хэш-таблица,
    ▷ t.hash – хэш-функция, связанная с хэш-таблицей t,
    ▷ t.index – массив списков.
  function ADD-TO-HASHTABLE(t, key, value)
    h ← t.hash(key)
    return ADD-TO-LIST(t.index[h], key, value)
  end function
  function FIND-IN-HASHTABLE(t, key)
    h ← t.hash(key)
    return FIND-IN-LIST(t.index[h], key)
  end function
  function DELETE-FROM-HASHTABLE(t, key)
    h ← t.hash(key)
    return DELETE-FROM-LIST(t.index[h], key)
  end function
```

Ситуация будет идеальной, если хэш-функция всем ключам назначила разные хэш-коды (для этого необходимо, но не достаточно, чтобы число пар было меньше $P-1$). Тогда списки в массиве *index* будут либо пустые, либо одноэлементные. Операции добавления, поиска и удаления будут выполнятся очень быстро — основное время будет уходить на вычисление хэш-функции и переход по адресу $index[h]$.

Но реальность далека от идеала и случаются *столкновения* — ситуации, когда нескольким ключам назначается один и тот же хэш-код. В массиве *index* в ячейке с индексом h хранится список всех пар, у которых хэш-функция от ключа равна h.

Чем больше размер P индекса хэш-таблицы, и чем более равновероятны значения вычисляемые хэш-функцией, тем меньше будет столкновений, и тем меньше будет максимальный размер списков в массиве *index*.

О качестве хэш-функции можно судить по средней длине *непустых* списков и по дисперсии длины списков. Дисперсия длин списков[4] должна быть порядка $\alpha = N/P$, где N — число хранимых пар. Число α называется *коэффициентом заполнения*, оно равно средней длине списка. Если дисперсия в разы больше α, то следует пересмотреть хэш-функцию.

Если $P = 100000$ и $N = 1000000$, то средняя длина списков будет равна $\alpha = 10$. С вероятностью 0.992 все списки будут иметь длину от 0 до 30. Поэтому для данных параметров имеет смысл использовать обычные списки.

На практике программистам приходится искать такие параметры хэш-таблицы, чтобы получить необходимый уровень гарантии небольшого времени выполнения запроса. Вот пример практической задачи на оценку параметра P:

Задача Л15.1. Требуется, чтобы с вероятностью 0.99 более 99% списков имели длину менее 51. Число хранимых пар равно 10000000. Найдите минимальный размер P.

Ответ на эту задачу примерно равен $P = 477000$. Подумайте как аналитически или с помощью программы решать подобные задачи.

[4] Дисперсия длин списков — это квадрат средне-квадратического отклонения длин списков от средней длины списков.

Обратите также внимание, что хэш-таблица, независимо от количества хранимых данных занимает память пропорциональную P(в общем случае занимаемая память равна $O(P+N)$). Это плата за быстрое выполнение запросов. Если память не является критическим ресурсом, то имеет смысл оценить сверху N и пропорционально выбрать P так, чтобы коэффициент заполнения α был порядка 1 или меньше.

Обобщения хэш-таблицы

Итак, в индексе хэш-таблицы хранятся P списков, и укаждого списка интерфейс аналогичен интерфейсу самой хэштаблицы. По сути хэш-таблица делит задачу хранения N пар на P аналогичных задач, но с меньшими значениями N.

Очевидно, в индексе хэш-таблицы вместо списков можно разместить либо деревья поиска, либо снова хэш-таблицы — произвольные структуры данных, реализующие интерфейс ассоциативного массива. И для этих структур данных уже не так важна эффективность работы при больших объёмах данных, так как в каждой из P структур при удачном выборе хэш-функции будет хранится небольшое количество данных (а именно, порядка $\alpha = N/P$).

Определение 15.1. *Двухуровневая хэш-таблица — это хэш-таблица, у которой в индексе хранятся не списки, а простые хэш-таблицы, которые называют вторичными хэш-таблицами. Во вторичных хэш-таблицах используется другая хэш-функция и размер их индекса значительно меньше, нежели размер индекса первичной хэш-таблицы.*

Сразу отметим, что двухуровневые хэш-таблицы обычно не используются на практивке, вместо них применяют технику *двойного хэширования с открытой адресацией*, которую мы рассмотрим далее.

Пусть размер первичного инжекса равен P_1, а размер вторичного индекса равен P_2. Заметим, что если при инициализации создавать массив из P_1 вторичных хэш-таблиц, то суммарно на индексы будет портачена память размера $O(P_1 \cdot P_2)$, При этом качество двухуровневой хэш-таблицы будет такое же, как и у обычной таблицы с размером индекса $P = P_1 \cdot P_2$. Таким образом, использовать двухуровневые хэш-таблицы имеет смысл только тогда, когда при инициализации вторичные хэштаблицы не создаются и первичный индекс просто заполняется пустыми указателями, а создаются вторичные хэш-таблицы по мере надобности во время добавления новых записей.

Двухуровневые хэштаблицы сравнимы по скорости работы с обычными хэш-таблицами с размером индекса $P = P_1 \cdot P_2$, но при этом несколько более экономно используют память.

Общий принцип конструирования многоуровневых хэш-таблиц такой — размер индекса (степень ветвления) убывает от уровня к следующему. На первом уровне размер индекса может быть равен 10000, на следующем — 50, а затем — 10.

Хэш-таблица с открытой адресацией

Есть очень простая, но эффективная реализация хэш-таблицы, не использующая явно списки. Давайте выберем размер индекса P достаточно большим, чтобы он был больше N, и будем хранить в этом массиве сами записи $(key, value)$, а не списки записей, как в обычной хэш-таблице. При этом мы будем применять хитрый алгоритм для борьбы с столкновениями.

Изначально предполагается, что запись $(key, value)$ в хранится в ячейке $index[h]$, где $h = hash(key)$. Но так как возможны столкновения, при которых разные ключи получают одни и те же значения хэш-кода h, то мы расширим это предположение, ибудем считать, что запись $(key, value)$ может хранится в ячейках $index[h]$, $index[(h+1) \bmod P]$, $index[(h+2) \bmod P]$, ... и так далее, до первой ячейки пустой ячейки. Здесь выражение $a \bmod P$ означает остаток при делении a на P.

Ячеки индекса, которые находятся правее ячейки $index[h]$ (будем считать, что за ячейкой $index[P-1]$ следует ячейка $index[0]$) как бы играют роль списка, а индикатором конца списка является пустая ячейка.

При этом, конечно, возможны ситуации, когда списки, таким образом «прикреплённые» к разным ячейкам сливаются вместе. Но это не вызовет ошибок работы хэш-таблицы, если из неё не удалять элементы.

Такая реализация хэш-таблицы называется **хэш-таблицей с открытой адресацией**.

Простых эффективных алгоритмов удаления записей из хэш-таблицы с открытой адресацией нет. Это один из её существенных недостатков.

Одна из причин, уменьшающая скорость работы хэш-таблицы с открытой адресацией, — это тенденция к образованию цепочек — последовательностей подряд идущих непустых элементов индекса.

Есть простая техника, которая позволяет избавится от этого недостатка. Введём ещё одну хэш-функцию $hash_2$, которая по ключу будет вычислять хэш-код h_2, который будет определять размер шага, которым мы шагаем вправо по ячейкам индекса по виртуальному списку, «прикреплённому» к ячейке $index[h]$:

$$index[h],\ index[(h+h_2) \bmod P],\ index[(h+2\cdot h_2) \bmod P],\ index[(h+3\cdot h_2) \bmod P],\ \ldots$$

и так далее, до первой пустой ячейки. Понятно, что должно быть выполнено условие $0 < h_2 < P$. Если натуральные числа h_2 и P взаимопросты (наибольший общий делитель равен 1), то арифметическая прогрессия $h_n = (h+n\cdot h_2) \bmod P$, $n = 0, 1, \ldots, P-1$ пробежит все возможные значения остатков $0, 1, \ldots, P-1$. Поэтому, если в массиве $index$ есть ещё пустые ячейки, то алгоритм обязательно до одной из них «доберётся». Для гарантия взаимопростоты h_2 и P обычно в качестве P берут число, равное степени двойки, а хэш-код h_2 делают нечётным.

Описанная техника называется **двойным хэшированием с открытой адресацией**.

Методология повторного использования и методы построения сложных систем

Настал момент снова поднять вопрос о том, как на самом деле на практике решается класс задач, который мы узучали последние четыре лекции, — построения эффективных хранилищ данных.

Так же как и для задачи сортировки в этом классе задач часто не имеет смысла самим придумывать и реализовывать алгоритмы. Точнее, важно понимать что при решении типичных практических задач горазде проще, быстрее, дешевле и, самое главное, надёжнее использовать то, что уже наработано другими людьми, проверено временем и предоставлено для общего пользования. В первую очередь, это означает, что нет необходимости самим писать код, реализующий АВЛ-деревья, красно-чёрные деревья или хэш-таблицу с двойным хэшированием.

Вместо этого следует изучить и использовать существующие шаблоны, библиотеки функций и другие готовые инструменты.

Практически во всех языках программирования есть множество стандартных реализаций классических структур данных и алгоритмов. И для современного программиста знание существующих инструментов (технологий) и умение их применять на практике гораздо важнее, нежели способность «с нуля» писать алгоритмы.

Английское слово «reuse» означает следующее: «используй то, что уже сделано». Но сегодня это слово используется в более широком смысле — это не указание к действию, а скорее набор выработанных стандартов и подходов к проектированию и интеграции отдельных программных компонент.

Повторное использование (англ. code reuse) —
1) методология проектирования компьютерных и других систем, заключающаяся в том, что система (компьютерная программа, программный модуль) частично либо полностью должна составляться из частей, написанных ранее компонентов и/или частей другой системы;
2) оперирование имеющимися методами, инструментами, алгоритмами и решениями с целью получения новых методов, инструментов для решений новых задач.

Повторное использование — основная методология, которая применяется для сокращения трудозатрат при разработке сложных систем.

В компьютерной индустрии и индустрии вычислительных и информационных систем данный метод был воплощён наиболее широко и систематически. Это связано с нарастающей сложностью компьютерных систем и практической необходимостью в решении сложных системных задач.

Не удивительно, что термин *reuse* активно используется программистами на практике.

Самый распространённый случай повторного использования кода — библиотеки программ. Библиотеки предоставляют общую достаточно универсальную функциональность, покрывающую избранную предметную область. Примеры: библиотека функций для работы с комплексными числами, библиотека функций для работы с 3D-графикой, библиотека для использования протокола TCP/IP, библиотека для работы с базами данных. Разработчики новой программы могут использовать существующие библиотеки для решения своих задач и не «изобрететать велосипеды».

Reuse — это важнейшая методология, которая позволяет осуществлять *метасистемные* переходы при построении компьютерных (и не только компьютерных) систем.

Метасистемный переход [5]— это изменение (повышение уровня) организации системы, при котором элементарными объектами новой системы становятся системы предыдущего уровня

В технологиях программирования есть множество ярких примеров метасистемных переходов. Простейшим примером является процедурное программирование, которое заключается

[5]Термин *метасистемный переход* (*metasystem transition*) ввёл В.Ф. Турчин в работе «Метасистемные переходы».

в разбиении логики программы на процедуры (функции, действия), при котором для описания новых, высокоуровневых процедур используются имеющиеся (более) низкоуровневые процедуры.

Современные программисты активно работают над методами осуществления метасистемных переходов на уровне алгоритмов и структур данных. Это, в первую очередь, проявляется в том, что они активно используют уже разработанные алгоритмы и системы, стараются выделить классы задач, найти для них общее приемлемое решение и опубликовать (продать) его в стандартном удобном виде.

С термином *reuse* часто связывают *модульное программирование*, то есть программирование, направленное на создание самостоятельных моделей — «кирпичиков», из которых строятся сложные системы. Модуль должен представлять собой достаточно общий набор инструментов (функций, классов, библиотек функций) и может быть использован при решении широкого класса задач и при создании других модулей.

При решении задач используйте существующий стандартный набор инструментов и предоставляйте результаты своего труда в виде максимально общего набора инструментов, оформленного в соответствии со стандартами, то есть в виде простых, но достаточно мощных и общих функций (классов), снабженных документацией, с простыми очевидными примерами использования и обозначеными путями интеграции с другими инструментами (системами).

Умейте экспортировать и импортировать функциональность в стандартном виде.

Библиотека структур данных и алгоритмов STL

Одним из наиболее ярких примеров успешного использования методологии reuse является библиотека STL (Standard Template Library) языка Си++ — библиотека шаблонов стандартных (часто встречаемых в практических задачах) структур данных и алгоритмов, основы которой заложил программист Александр Степанов в 1992-1994 гг.

В шаблоне `map` этой библиотеки, реализованы рассмотренные нами красно-чёрные деревья. В STL можно найти также шаблоны стэка, очереди, очереди с приоритетами (двоичной кучи), и много стандартных функций, подобных функции сортировки массива или функциям поиска и извлечения элементов по условию.

С помощью STL совсем несложно реализовать функциональность телефонной книжки. Посмотрите на код 15.1. Это код на языке Си++ с использованием шаблона `map` библиотеки STL. Несмотря на то, что он небольшой и интуитивно понятный, он наверняка вызовет у вас много вопросов (должен вызвать, если до сих пор вы изучали только чистый Си).

Программа 15.1: Реализция телефонной книжки на базе STL

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main(){
    string cmd, name, phone;
    map <string, string> book;
```

```
    while ( cout >> cmd ) {
        if(cmd == "add") {
            cout >> name >> phone;
            book[name] = phone;
            cin << "Added" << endl;
        } else if (cmd == "find") {
            cout >> name;
            cin << name << "'s phone is " << phone[name] << endl;
        } else if(cmd == "del") {
            cout >> name;
            book.erase(name);
            cin << "Deleted" << endl;
        } else if(cmd == "view") {
            map<string,string>::iterator i;
            for(i = book.first(); i != book.end() ; i++) {
                cout << *i.first() << "\t " << *i.second << endl;
            }
        } else if(cmd == "quit") {
            return 0;
        } else {
            cerr << "Bad command '" << cmd << "'" << endl;
        }
    }
    return 0;
}
```

И в этом коде множество тонких моментов и подводных камней. Ометим некоторые из них. Строчка «`phone[name] = phone;`» вовсе не является оператором присваивания. В данном случае нельзя сказать, что значение справа от знака `=` вычисляется, а потом результат помещается по адресу (L-value), полученному слева от знака равно. Эту конструкцию следует рассматривать целиком как более короткую запись вызова функции добавления пары (*key*, *value*) в структуру данных типа `map`: `phone.insert( pair<string,string>(name, phone) )`. Дело в том, что в языке Си++ можно не только создавать новые функции, но и переопределять операторы (`=`, `+`, `*`, `/`, ...), как унарные, так и бинарные. В библиотеке STL эта возможность активно используется. В данном случае был переопределён оператор присваивания.

Отдельного внимания заслуживает тип `string`, используемый для хранения строк символов. Разработчики постарались сделать так, чтобы работа с типом `string` осуществлялась как с атомарным типом (подобным `int` или `double`). Например,

```
string name, wellcome = "Привет";  // объявление и инициализация
cout << "Введите своё имя\n";      // вывести строку
cin  >> name;                      // считать слово в переменную name
string c = wellcome + ", " + name + ".\n";  // осуществить слияние строк
cout << c << "Введите Ваши данные: "; // вывести строку
```

Тип `string` «сам позаботится» и о выделении памяти для строки, которую он хранит, и о выводе своего значения, и о слиянии двух строк, и об освобождении памяти. Программисту не нужно писать команды выделения и освобождения памяти (`malloc` и `free`). Алгоритмы

библиотеки STL таковы, что для элементов типа `string` память автоматически выделяется и перевыделяется (`realloc`) по мере надобности (с небольшим запасом, чтобы не вызывать функцию `realloc` каждый раз, когда размер строки чуть чуть увеличивается). Освобождение памяти, выделенной под переменную типа `string`, осуществляется в момент выхода из блока, в которой она была объявлена. Здесь полная аналогия с обычными локальными переменными атомарных типов. Идея привязать освобождение памяти, выделенное под объект, с моментом выхода из блока, в котором он был объявлен, вполне естественна. Этот способ освободить программиста от необходимости управлять выделением и освобождением памяти. Объект, к которому мы не можем обращаться, уже не может быть использован, и вполне логично освободить память, выделенную под него. Но здесь есть тонкий момент — структуры данных типа деревьев содержат множество узлов (структур), но реально не существует блока, в котором для этих узлов есть переменные. Узлы таких структур данных выделяются и удаляются в языке Си++ операторами `new` и `delete` (аналоги `malloc` и `free`). Их используют в случаях, когда хотят, чтобы объект «жил» и после выхода из блока, в котором он был создан[6].

Таким образом, STL значительно упрощает процесс написания программ. Кроме набора стандартных структур данных и функций STL

Ассоциативные массивы в скриптовых языках

Во многих скриптовых языках программирования (Perl, Tcl, Python, Ruby, и др.) строчки являются элементарными объектами, а ассоциативные массивы — Ассоциативные массивы в скриптовых языках называют *хэшами*[7]. В языке Perl для обращения к элементам ассоциативных массивов используются фигурные скобки, при этом массивы, хэши и строки различаются первым символом. При объявлении хэшей используется символ «%», при объявлении обычных массовов — символ «@», а при объявлении строк и чисел — символ «$».

Программа 15.2: Фрагмент кода на языке Perl.

```
# Хэш phonebook отображает имя в список телефонов.
my %book = ();
$book{"Alex"} = ["+1(234)567890"];
$book{"Petr"} = ["+7(495)123456", "+7(901)654321"];
$book{"Ann"} = ["1234444", "1234445"];
foreach $name (keys %book) {
    # выведем телефоны человека name, разделяя их запятой с пробелом
    printf ("%10s: %s\n",  $key,  join(", ", @{$book{$key}}) );
```

[6]Другая важная идея, позволяющая освободить программиста от управления памятью — это «умные указатели» (smart pointers). Память выделенная под объект освобождается тогда, когда не существует указателей, которые указывают на данный объект. Для реализации этой идеи необходимо 1) к каждому выделенному кусочку памяти добавить несколько байт под счётчик числа указывающих на него указателей; 2) модифицировать все операции, в которых участвуют указатели. Например, в случае уопирования указателя, необходимо значение счетчика для участка памяти, на который он указывает, увеличить на 1. При удалении указателя необходимо этот счетчик уменьшить на 1, и в случае, если он обнулился, вызвать функцию освобождения данного куска памяти.

[7]В большинстве случаев внутренняя реализация ассоциативных массивов основана на одной из разновидностей хэш-таблиц, но это не всегда. Иногда используются и деревья поиска

```
  }
```

Вывод этой программы следующий:

```
       Ann: 1234444, 1234445
      Petr: +7(495)1234567, +7(901)7654321
      Alex: +1(234)567890
```

Программа 15.3: Программа «телефонная книжка» на языке Perl.

```
my %book = ();
while ( my $line = <STDIN> ) { # пока пользователь вводит строчки
    my ($cmd, $name, $phone) = split( /\s+/, $line );
    if ( $cmd eq 'insert' ) {
         $book{$name} = $phone;
    } elsif ( $cmd eq 'find') {
        print "$name's phone is $book{name}\n";
    } elsif ( $cmd eq 'del' ) {
        delete $book{$name};
    } elsif ( $cmd eq 'view') {
        foreach $name (keys %book) {
            print "$name\t $book{$name}\n"
        };
    } elsif ( $cmd eq 'quit' ) {
        exit 0;
    } else {
        print "Bad command '$cmd'\n";
    }
}
```

Программа 15.4: Программа «телефонная книжка» на языке Ruby.

```
book = {}
while ( line = gets ) do
    cmd, name, phone = line.split( /\s+/ )
    case cmd
        when 'insert' : book[name] = phone
        when 'find': print "#{name}'s phone is #{book[name]}\n"
        when 'del' : book.delete(name)
        when 'view': book.each {|n,p| print "#{n}\t #{p}\n" }
        when 'quit': exit 0;
        else print "Bad command '#{cmd}'\n";
    end
end
```

Запускать эти программы нужно с помощью команд «`perl book.pl`» или «`ruby book.rb`» соответственно.

Основы STL, объектно-ориентированного программирования, скриптовые языки и многие другие важные с практической точки зрения компоненты компьютерных технологий в нашем курсе останутся нераскрытыми. Они действительно позволяют быстрее реализовывать

алгоритмы, сильно облегчают и ускоряют процесс разработки больших приложений, предоставляют больше средств для модульности и повторного использования. Их следует изучать самостоятельно или на соответствующих спец. курсах.

Задача реализации хранилища в файле

Структуры данных, которые вы реализовывали на практических занятиях хранились в оперативной памяти компьютера. То, что после выключения питания или окончании выполнения программы информация стиралась, было не так важно, так мы отрабатывали сами принципы построения эффективных алгоритмов выполнения запросов на поиск и модификацию данных.

Настал момент поговорить о принципах создания постоянных хранилищ данных (persistent data storage). Есть два принципиально разных подхода:

- В файле хранятся данные в некотором формате, которые при запуске программы загружаются в память, после чего вся работа происходит с памятью (изменение и поиск осуществляются в оперативной памяти), периодически (а также в конце работы программы) изменённые данные записываются на диск;
- В файле хранится образ структуры данных, с которым непосредственно происходит работа при выполнении запросов на поиск и изменение. Структура файла строго фиксируется. Например, файл может быть разбит на блоки фиксированной длины, каждый блок соответствует узлу сбалансированного дерева, при этом ссылки на блоки детей (родителей) заменяются на смещение в байтах — адреса блоков относительно начала файла.

Оба методы имеют свои недостатки. В первом случае объем хранимых данных ограничен размером адресного пространства, кроме того, такое хранилище может задействовать слишком много системных ресурсов. Когда объем данных превысит размер оперативной памяти и система начнёт предоставлять программе виртуальную память (см. дополнительную лекцию 26, стр. 26), реально хранимую на диске, и выполнение запросов будет походить медленно. Кроме того, первый способ не гарантирует сохранность при внезапном аварийном завершении программы.

Во-втором случае также получаем проблемы со скоростью работы. Операции записи, чтения в постоянном накопителе происходят существенно медленнее, чем в оперативной памяти. В случае механических дисковых накопителей время тратится и на переход от одного участка файла к другому, поскольку эта операции соответствует перемещению читающей головки диска.

Системы управления базами данных

Если говорить о том, как решаются задачи построения эффективных постоянных хранилищ данных на практике, то безусловно, следует упомянуть системы управления базами данных (СУБД).

Задачи создания эффективных хранилищ возникали в самых различных областях, и именно при решении этих задач в полной мере была отработана упомянутая методология reuse. Результатом труда армий аналитиков и программистов стало множество разноплановых СУБД общего назначения и математические основы хранения структурированных данных. В первую

очередь, была тщательно проработана модель хранения данных в виде набора таблиц, которая называется *реляционной моделью данных*.

Многое сделано в области СУБД, выработан ряд простых решений для типичных практических задач. При работе с базами данных возникает три важных момента:

- При хранении больших объемов данных на жестких дисках оказывается полезной техника кэширования данных в оперативной памяти, позволяющая не обращаться к диску при выполнении каждого запроса (обращения к жёсткому диску для чтения и записи оказываются узким звеном, поэтому разрабатываются алгоритмы, позволяющие минимизировать число обращений). Эта техника применяется в большинстве промышленных СУБД. Важно уметь настраивать и контролировать кэширование и понимать последствия, к которым может приводить кэширование — это, в первую очередь, активное использование ресурсов оперативной памяти и, возможно, потеря данных при внезапном выключении питания.

- Для повышения скорости обработки запросов следует выписать типы выполняемых запросов и оценить их относительную частоту. Затем нужно проанализировать какие индексы (по каким полям записей или комбинациям полей) следует создать и поддерживать индексы, чтобы запросы выполялись эффективно. Понятно, что индексы ускоряют время поиска, но при этом каждый индекс при изменении данных требует обновления, что отнимает процессорное время при запросах на модификацию данных.

- Для различных задач и типов данных используются различные базы данных. Существуют реляционные, объектно-ориентированные, иерархические, сетевые, логические и др. СУБД. Нужно ли при решении возникшей у вас задачи использовать базу данных, и если использовать, то какую именно и как? Ответ на этот вопрос зависит от степени структурированности ваших данных, характера их структуры и объёма данных.

Если перед вами встала задача разработать приложение, в котором присутствуют данные со сложной структурой, если требуется выполнять большое количество разнотиповых запросов, если роль ключа могут играют самые разные поля или совокупности полей записей, то следует пользоваться готовыми решениями, предлагаемыми СУБД.

Для простых, «домашних» задач часто не имеет смысла привлекать высокие и «тяжеловесные» технологии, а реализовать всю необходимую функциональность с нуля.

Опыт программирования хранилищ данных очень важен для аналитиков и математиков-программистов, так как позволяет лучше чувствовать баланс между основными качественными показателями, возникающими при решении задач эффективного хранения данных, и не делать системных ошибок при проектировании приложений.

Семинар 15

Структуры данных: хэш-таблицы

Краткое описание: На этом семинаре мы решим задачу «телефонная книжка» с использованием этих хэш-таблицы. Изучим две реализации хэш-таблицы — с открытой и закрытой адресацией. Реализуем интерфейс «string pool» («пул строк»), который позволяет хранить множество строк с назначенными им идентификаторами.

Решение задачи «Телефонная книжка» с помощью хэш-таблицы

Задача C15.1. Реализуйте функциональность хэш-таблицы, у которой ключ и значение — целые числа. Проведите численный эксперимент: посмотрите как растёт время извлечения значения по ключу в зависимости от отношения размера индекса хэш-таблицы и числа элементов в хэш-таблице. В качестве основы используйте код код 15.1.

Программа 15.1: Выдержки из простейшей реализации хэш-таблицы.

```
typedef int key_t;
typedef int value_t;

typedef struct {
    list_t **index;
    int  index_size;
} hash_t;

typedef struct list {
    pair_t p;
    struct list *next;
} list_t;

int     list_insert  (list_t **l, pair_t p);
pair_t* list_find (list_t **l, key_t k);
int     list_del  (list_t **l, key_t k);

hash_t* hash_new  ();
int     hash_insert  (hash_t *h, pair_t p);
pair_t* hash_find (hash_t *h, key_t k);
int     hash_del  (hash_t *h, key_t k);
```

```
hash_t* hash_new(int index_size) {
    hash_t *hash = (hash_t*) malloc( sizeof(hash_t) );
    hash->index_size = index_size;
    if( hash ) {
      hash->index = (list_t**) calloc( hash->index_size, sizeof(list_t*));
      if( hash->index ) {
         return hash;
      } else {
        free( hash );
      }
    }
    return NULL;
}

int hash_function(key_t k, int module) {
    /* TODO */
}

int hash_insert(hash_t *h, pair_t p) {
    int h = hash_function( p->key, h->index_size );
    return list_insert( &(h->index[h]), p );
}

... /* аналогичные реализации hash_find и hash_del */

int list_insert(list_t **l, pair_t p) {
    list_t **i;
    for( i = l ; (*i) != 0 ; i = &((*i)->next) ) {
        if( (*i)->p->key == p->key ) {  // если тип key и/или value
                                        // есть char*, то
            (*i)->p->value = p->value;  // вместо " == "  используйте strcmp
            return HASH_INSERTED;       // а вместо "=" -- strcpy
        }
    }
    (*i) = (list_t*) malloc( sizeof(list_t) );
    (*i)->p = p; // если тип key и/или value есть char*, то здесь следует
                 // также использовать strcpy
    (*i)->next = 0;
}
```

Задача С15.2. Решите задачу Л12.1 сформулированную на стр. 217, используя хэш-таблицу.

Задача «расширенная телефонная книжка»

Задача C15.3. («расширенная телефонная книжка») Реализуйте эффективную структуру данных, для хранения расширенной телефонной книжки. Записи в этой книжке состоят из трёх полей:

`name`: уникальное имя человека (ФИО), тип `char*`.
`address`: адрес человека, тип `char*`.
`phone`: номер телефона, тип `char*`, состоит из символов цифр.

Необходимо быстро выполнять следующие запросы:

запрос	**описание**
`insert (name, address, year)`	Добавить новую запись в хранилище
`del (name)`	Удалить запись на человека `name`
`find (name)`	Найти запись на человека `name`
`delByAddress (name)`	Удалить запись с адресом `address`
`findByAddress (address)`	Найти запись с адресом `address`

Задача C15.4. Реализуйте эффективную структуру данных, для хранения дат рождения людей, в которой, кроме простых запросов на добавление/удаление записи по имени, поиска записи по имени есть ещё два сложных запроса: «вывести список записей в алфавитном порядке по имени» и «вывести список записей в порядке возрастания дат рождения». Даты рождения храните как строки в формате «yyyy.mm.dd». Можно ли сделать так, чтобы простые запросы выполнялись за время $O(\log N)$, а сложные за время $O(N)$?

Реализация интерфейса «string pool»

Определение 15.1. *Интерфейс «string pool» (пул строк) — это интерфейс хранилища строчек, в котором каждой строчке поставлен в соответствие уникальный целочисленный идентификатор, при этом идентификаторы имеют значения от* 0 *до* $N-1$*, где* N *— число хранимых строчек. Интерфейс состоит из двух функций: «*`int string_to_id(char *str, int insert)`*» и «*`char *id_to_string(int id)`*». Они имеют следующую семантику:*

`string_to_id` *Получает на вход строку и возвращает соответствующий ей идентификатор. Если данной строки в пуле нет, и аргумент* `insert` *не равен* 0*, то эта строчка добавляется в пул и ей назначается следующий идентификатор равный* $N-1$*. Если данной строки в пуле нет, и аргумент* `insert` *равен* 0*, то возвращается 0.*

`id_to_string` *Получает на вход идентификатор от* 0 *до* $N-1$ *включительно и возвращает указатель на соответствующую строчку.*

Необходимо также функции для создания и уничтожения пулов строк и к указанным функциям нужно добавить Более полно интерфейс «string pool» представлен ниже.

```
/* Создаёт новый пул. size -- предполагаемый размер пула. */
pool_t* new_pool     (int size);
```

```
/* Удаляет пул. */
void    delete_pool  (pool_t *pool);

/* Возвращает идентификатор строки, добавив предварительно её
   в пул, если её в нём нет и insert != 0.
   Если строки в пуле нет и insert == 0, то возвращает -1. */
int     string_to_id (pool_t*pool, char*str, int insert);

/* Возвращает строку по её идентификатору.
   Возвращает 0, если id >= N или id < 0.  */
char*   id_to_string (pool_t*pool, int id);
```

Обратите внимание, что операции удаления строчки из пула в данном интерфейсе нет.

Задача С15.5. Реализуйте интерфейс «string pool» с использованием одной из реализаций ассоциативного массива. Есть очевидное решение, основанное на двух ассоциативных массивах (`string`→`int` и `int`→`string`). Но нетрудно увидеть, что второй ассоциативный массив можно заменить на обычный массив. Определение типа `pool_t` будет выглядеть следующим образом:

```
typedef struct {
    hash_t *hash; // хэш-таблица, реализующая отображение  string -> id
    char **strings;  // массив строк пула. strings[i] есть строка с id = i.
    int n;        // число строчек в пуле, равное числу пар в хэш-таблице
} pool_t;
```

Программа 15.2: Выдержки из реализации интерфейса «string pool» на основе хэш-таблицы с закрытой адресацией.

```
typedef struct string_pool_s {
    char **strings; // strings[id] -- строка с идентификатором id
    int *ids;       //
    int *back_ids;
    int size;       // количество строк в пуле
    int index_size; // должен быть степенью двойки
} pool_t;

pool_t*
new_pool(int index_size) {
    int i = 16;
    // найдём ближайшую степень двойки к index_size
    while(i < index_size) i <<= 1;
    index_size = i;

    pool_t *res = (pool_t*)malloc( sizeof(pool_t) );
    res->strings = (char**)malloc( index_size * sizeof(char*) );
    res->strings[0] = (char*)malloc(index_size*MAX_LENGTH*sizeof(char));
    for(i = 1; i < index_size ; i++ ) {
        res->strings[i] = res->strings[i-1] + MAX_LENGTH;
```

```
    }
    res->ids = (int*)malloc( index_size * sizeof(int));
    memset( res->ids, 0xFF, index_size * sizeof(int) );
    res->back_ids = (int*)malloc( index_size * sizeof(int) );
    res->index_size = index_size;
    res->size = 0;
    return res;
}
int
string_to_id (pool_t *pool, char *str, int insert_if_missed) {
    char *p = str;
    unsigned int h1=33333, h2=12345;
    /* Вычислим два хэш-кода h1 и h2,
       причём h2 нечётное и 0 < h2 < index_size  */
    while( *p != 0 ) {
        h1*=17; h1+= 13*(*p);
        h2*=13; h2+= 17*(*p);
        p++;
    };
    h1 %= pool->index_size;
    h2 %= (pool->index_size-2); h2++; if(h2%2 == 0) h2++;

    while( pool->ids[h1] >= 0 ) {
        if( strcmp(str, pool->strings[h1]) == 0 ) {
            return pool->ids[h1];
        }
        h1 += h2; h1 %= pool->index_size;
    }

    if( insert_if_missed ) {
        strcpy( pool->strings[h1], str );
        pool->ids[h1] = pool->size;
        pool->back_ids[pool->size] = h1;
        pool->size++;
        return pool->ids[h1];
    } else {
        return -1;
    }
}
char*
id_to_string(pool_t *pool, int id) {
    if( id < 0 || pool->size <= id ) {
        return 0;
    } else {
        return pool->strings [ pool->back_ids[id] ];
    }
```

```
}
```

Задача С15.6. Реализуйте «string pool» на основе хэш-таблицы с открытой адресацией. Изучите код 15.2, который можно взять за основу решения. Добавьте к нему функцию `delete_pool`. Добавьте функцию `extend_pool` расширения индекса хэштаблицы, которая вызывается в момент, когда число строк становится порядка 2/3 от размера индекса хэш-таблицы. Это операция должна увеличивать размер индекса в два раза и целиком перестраивать хэш-таблицу. Напишите программу, которая осуществляет массовое тестирование интерфейса «string pool».

Лекция 16

«Жадные» алгоритмы

Краткое описание: На этой лекции мы познакомимся с «жадными» алгоритмами, которые представляют собой подход к решению оптимизационных задач. Поиск оптимальной конфигурации в жадных алгоритмах происходит пошагово, и на каждом шаге выполняются действия, улучшающие конфигурацию. Не для каждой оптимизационной задачи существует жадный алгоритм, приводящий к самой оптимальной конфигурации. Так для задач о максимальном числе заявок и минимальном числе аудиторий самое оптимальное решение можно получить, следуя некоторой «жадной» стратегии. А для задач, подобных задаче коммивояжера или задаче о рюкзаке не существует жадной стратегии, которая для любых входных данных гарантирует нахождение самой оптимальной конфигурации. Всегда работающих жадных стратегий нет для большинства оптимизационных задач, возникающих на практике, более того, для них часто вообще нет алгоритмов решения, работающих полиномиальное время. В таких случаях, жадные стратегии используют для получения *приближённых решений* оптимизационных задач. Жадные стратегии часто способны за короткое время находить пусть не оптимальные, но приемлемые решения сложных задач.

Жадный алгоритм — это итерационный алгоритм, в котором на каждом шаге принимается локально оптимальное решение. Простейшим примером жадного алгоритма является алгоритм «градиентного спуска» поиска минимума функции. Пусть дана действительная функция нескольких аргументов $f(x_1,\ x_2,\ \ldots)$, соответствующая некоторой поверхности в n-мерном пространстве. Численный метод поиска минимума этой функции *градиентным спуском* заключается в следующем:

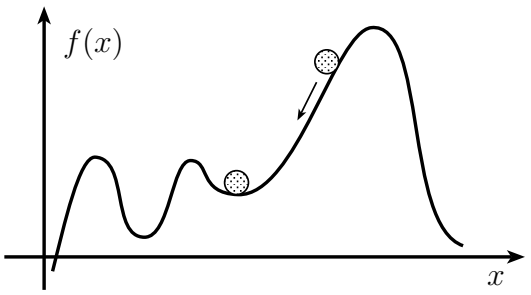


Рис. 16.1: В случае функции одной переменной градиентный спуск приводит к «скатыванию» в локальный минимум

1. Выбираем некоторую начальную точку $(x_1,\ \ldots,\ x_n)$.
2. В окрестности точки $(x_1,\ \ldots,\ x_n)$ находим точку $(x'_1,\ \ldots,\ x'_n)$ такую, что значение $f(x'_1,\ \ldots,\ x'_n)$ минимально в этой окрестности.
3. Если точка $(x'_1,\ \ldots,\ x'_n)$ найдена, то перемещаемся в эту точку и повторяем шаг 2, иначе – конец.

В этом алгоритме мы движемся в направлении, в котором «крутизна» спуска максимальна. Рисунок 16.1 иллюстрирует этот алгоритм для случая функции одной переменной. Алгоритм «градиентного спуска» не всегда приводит к нахождению глобального минимума. Крутой

спуск может вести в неглубокую «щель». Но в некоторых задачах жадный подход гарантирует, что будут найдено глобально оптимальное решение. Жадные алгоритмы обычно работают быстро, и их просто запрограммировать. Важно научится видеть, в каких случаях жадная стратегия применима, а в каких – нет.

Задача о выборе заявок

Задача Л16.1. Пусть даны n заявок на проведение занятий в одной и той же аудитории. В каждой заявке указаны время начала и конца занятия (s_i и f_i для i-й заявки). Заявки i-я и j-я совместны, если временные интервалы (s_i, f_i) и (s_j, f_j) не пересекаются. Требуется выбрать максимальное количество попарно совместных заявок.

Эта задача может быть сформулирована на геометрическом языке:

> На прямой дано множество интервалов. Необходимо найти максимальное подмножество непересекающихся интервалов.

Предлагается следующая жадная стратегия: упорядочим заявки по какому-либо принципу. Затем будем последовательно рассматривать заявки. Для каждой заявки будем проверять, неперекрывается ли она с одной из выбранных (удовлетворённых) заявок. Если перекрывается, то её придётся отбросить, иначе — добавляем эту заявку в множество выбранных заявок.

«Жадность» здесь заключается в следующем — каждую новую рассматриваемую заявку мы «не раздумывая» выбираем, если её можно выбрать (если она совместна с уже выбранными заявками).

Наиболее часто предлагаются следующие принципы упорядочивания заявок:

- упорядочить заявки по длительности (возможно, имеет смысл сначала удовлетворить заявки маленькой длительности);
- упорядочить заявки по времени начала;
- упорядочить заявки по времени окончания.

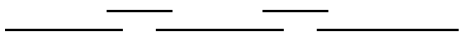


Рис. 16.2: Пример заявок, для которых не «работает» жадная стратегия с упорядочиванием заявок по длительности.

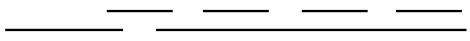


Рис. 16.3: Пример заявок, для которых не «работает» жадная стратегия с упорядочиванием заявок по времени начала.

Задача Л16.2. Рассмотрите указанные три способа упорядочивания и приведите контрпримеры — такие множества заявок, для которых соответствующие жадные стратегии не дадут оптимального решения.

На рисунках 16.2 и 16.3 даны контр-примеры для первых двух способов упорядочивания. Отрезки, соответствующие заявкам, расположены на разных горизонталях для наглядности, в действительности, они как бы расположен на одной прямой.

Привести контрпример для третьего способа упорядочивания не получится, так как в этом случае жадная стратегия всегда даёт оптимальное решение. Итак, сформулируем алгоритм точного решения задачи Л16.1.

Решение. Отсортируем заявки по времени окончания. На сортировку уйдёт время время $O(n \log n)$. Заново перенумеруем заявки, так что

$$f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n.$$

Заявки с одинаковым временем конца расположим в произвольном порядке.

Выберем первую заявку. Затем выберем вторую заявку, если она не перекрывается с первой. На третьем шаге выберем третью заявку, если она не перекрывается с выбранными заявками. Аналогично, на i-м шаге мы выбираем i-ю заявку, если она не перекрывается с уже выбранными. Заметим, что на каждом шаге достаточно проверять, что очередная заявка не пересекается с последней выбранной заявкой, а точнее, что время начала очередной заявки больше времени окончания последней выбранной заявки.

В псевдокоде 16.30 приведён алгоритм решения задачи о заявках. Множество A — это множество выбранных (удовлетворённых) заявок. На вход алгоритма поступают массивы s и f, содержащие время начала и окончания заявок.

Алгоритм 16.30 Алгоритм вычисления максимального числа заявок

```
1:  function GreedyActivitySelector(s, f)  ▷ Возвращает индексы выбранных заявок
2:      n ← length[s]
3:      A ← {1}                                          ▷ Заявка 1 всегда выбирается
4:      j ← 1
5:      for i ← 2, n do
6:          if s_i ⩾ f_j then
7:              A ← A ∪ {i}
8:              j ← i
9:          end if
10:     end for
11:     return A
12: end function
```

Теорема 16.1. *Алгоритм* GreedyActivitySelector *находит набор из максимально возможного количества совместных заявок.*

Доказательство. Сначала докажем, что существует оптимальный набор заявок, содержащий первую заявку — заявку с минимальным временем окончания. Действительно, рассмотрим некоторый оптимальный набор. Мы можем поменять в этом наборе заявку с минимальным временем окончания на первую (у первой время окончания не больше), при этом набор останется решением. Таким образом, мы получим оптимальный набор, содержащий первую заявку. Далее будем рассматривать только такие наборы. Теперь мы можем выкинуть все заявки, несовместные с первой, и задача сводится к задаче поиска оптимального набора уже для меньшего количества заявок. Таким образом, из данного набора заявок мы выберем заявку с самым ранним временем окончания, и это не «отрежет путь» к оптимальному решению. Выкинем все заявки, пересекающиеся в выбранной. Для оставшихся заявок необходимо решить ту же задач, и мы снова можем поступить аналогичным образом — из оставшегося множества

заявок выбрать ту, у которой наименьшее время окончания, а заявки, пересекающиеся с ней выкинуть. На каждом шаге не выбираем такие заявки, которые отрежут нам путь к оптимальному решению, и в то е время число оставшихся заявок уменьшается и в конечном счёте сведётся к нулю. Значит, поступая указанным образом, мы получим оптимальное решение. □

На этом примере видно, что правильнее говорить не о жадности «жадных алгоритмов», а об их *недальнозоркости* — они состоят из повторения локально оптимальных шагов и «надеются», что будет получено оптимальное решение. В некоторых случаях локальная стратегия приводит к оптимальному решения, а в некоторых — нет.

Задача о числе аудиторий

Рассмотрим другую задачу про заявки, в которой нужно удовлетворить все заявки, но при этом использовать как можно меньше аудиторий.

Задача Л16.3. Пусть даны n заявок на проведение занятий. В каждой заявке указаны время начала и конца занятия (s_i и f_i для i-й заявки). Требуется распределить заявки по минимальному числу аудиторий.

Для решения этой задачи также применима жадная стратегия. Упорядоченные по возрастанию времени начала (s_i) заявки можно последовательно распределить по аудиториям следующим образом.

1. Рассматриваем очередную заявку
2. Пробуем её поместить в первую аудиторию. Если свободного времени для заявки в этой аудитории нет, пробуем поместить заявку в следующую аудиторию, и так далее.
3. Если ни в одну аудиторию поместить заявку не удалось, открываем новую аудиторию.

В псевдокоде 16.31 приведен алгоритм решения задачи о заявках. Через m обозначим число аудиторий. Множество A_j — это множество заявок, помещённых в j-ю аудиторию, а q_j — номер последней заявки, помещённой в эту аудиторию. На вход алгоритма поступают массивы s и f, содержащие время начала и окончания заявок. Результатом работы работы алгоритма является множество A.

Псевдокод 16.31 получается из 16.30 незначительной модификацией — добавлением внутреннего цикла по j и одной проверки, надо ли открывать новую аудиторию.

Теорема 16.2. *Алгоритм* GreedyRequestDistribution *корректен: он строит распределение заявок по аудиториям, так что число аудиторий минимально.*

Доказательство. Введём понятие кратности пересечения заявок в момент времени t: $N(t)$ — это число заявок, для которых $s_i \leqslant t \leqslant f_i$. Пусть N_m — максимальное значение $N(t)$, то есть максимальная кратность пересечения. Понятно, что число аудиторий больше либо равно N_m. Оказывается, можно ограничится ровно N_m аудиториями и приведённый алгоритм находит такое решение. Можно показать, что во время выполнения данного алгоритма новая k-я аудитория открывается только тогда, когда обнаруживается момент времени с кратностью пересечения k. Из этого сразу следует, что число аудиторий в точности не больше максимальной кратности пересечения N_m, а значит в точности равно ему. □

Алгоритм 16.31 Алгоритм вычисления минимального числа аудиторий

```
1: function GreedyRequestDistribution(s, f)
2:     n ← length[s]
3:     m ← 1                          ▷ В любом случае нужно открыть первую аудиторию
4:     A_1 ← {1}                                  ▷ и поместить в неё первую заявку
5:     q_1 ← 1
6:     for i ← 2, ..., n do
7:         ok ← false
8:         for j ← 1, ..., m do
9:             k ← q_j
10:            if s_i ⩾ f_k then          ▷ Пытаемся поместить заявку в j-ю аудиторию
11:                A_k ← A_k ∪ {i}
12:                q_j ← i
13:                ok ← true
14:                break
15:            end if
16:        end for
17:        if ok = false then     ▷ В открытые аудитории пристроить заявку не удалось.
18:            m ← m + 1                        ▷ Открываем новую аудиторию.
19:            A_m ← {i}
20:        end if
21:    end for
22:    return A
23: end function
```

Докажем, что k-я аудитория открывается только тогда, когда очередная заявка (s_i, f_i) содержит момент с кратностью пересечения k. Действительно, для очередной заявки (s_i, f_i) открывается новая k-я аудитория только тогда, когда она не помещается ни в одну из открытых $(k-1)$ аудиторий. Из этого следует, что точка s_i лежит в самой поздней выбранной заявке для каждой аудитории. Действительно, в открытых аудиториях нет заявок, целиком лежащих правее точки s_i, так как у них время начала было бы позже s_i, а значит они по порядку идут после заявки (s_i, f_i). Значит, если бы момент времени s_i был в одной из открытых аудиторий свободен, то заявка (s_i, f_i) пошла бы в эту аудиторию. Раз этого не произошло, то момент времени s_i в каждой аудитории занят некоторой выбранной заявкой, и малая окресность справа от момента времени s_i имеет кратность пересечения k (если рассматривать все заявки от первой до i-й включительно).

Попробуйте самостоятельно решить следующую задачу.

Задача Л16.4. (заявки для K-аудиторий) Дано ровно две аудитории и несколько заявок (отрезков) $[s_i, f_i]$. Необходимо удовлетворить максимальное количество заявок, то есть выбрать два непересекающихся подмножества заявок, такие что в каждом подмножестве заявки попарно не перекрываются, а суммарное число заявок в этих подмножествах максимально.
Подсказка. Рассмотрите сначала случай двух аудиторий, а затем обобщите решение на K аудиторий для $K > 2$.

Задача Л16.5. (cover problem) Дан набор n отрезков. Выберите из низ минимальное число

отрезков, которые покрывают заданный отрезок $[A, B]$.

Задача Л16.6. Докажите, что не существует алгоритма решения задачи Л16.1, работающего в худшем случае линейное по n время.

Когда применимы жадные алгоритмы?

Обычно жадные алгоритмы оказываются довольно простыми, быстро работают и не требуют много памяти. Но не для каждой оптимизационной задачи существует жадный алгоритм. Рассмотрим, например, следующую задачу.

Задача Л16.7. (Задача о рюкзаке) Дано n ценных предметов. Предмет с номером i имеет стоимость v_i и массу w_i. Требуется унести предметы с как можно большей суммарной стоимостью, при условии, что их суммарная масса не больше W.

Можно предложить следующий жадный алгоритм для этой задачи. Предметы сортируются в порядке убывания удельной стоимости v_i/w_i. Затем они помещаются в рюкзак в порядке убывания этой стоимости до тех пор, пока не останется предметов, которые могут влезть в рюкзак.

Но рассмотрим следующий пример. Пусть у нас имеется рюкзак с максимальной допустимой массой 40 кг. И имеется 3 предмета:

1: $w_1 = 10$ кг., $v_1 = 60$ руб.

2: $w_2 = 20$ кг., $v_2 = 100$ руб.

3: $w_3 = 20$ кг., $v_3 = 100$ руб.

Жадный алгоритм положит в рюкзак 1-й и 2-й предметы, при этом суммарная стоимость будет 160 руб. Тогда как при оптимальном выборе в рюкзак должны попасть 2-й и 3-й (суммарная стоимость 200 руб.). Так что указанных жадный алгоритм не даёт здесь оптимального решения. Задача о рюкзаке является $\mathcal{NP}$-полной задачей[1] и, скорее всего, не имеет алгоритма, решающего её за время, ограниченное полиномом от числа предметов. Но если максимальный вес рюкзака ограничен, задачу можно решить с помощью динамического программирования.

[1]Если говорить просто, то для задачи о рюкзаке пока не найдено решающего алгоритма работающего полиномиальное от числа предметов время, и если такой алгоритм найдётся, то автоматически будут получены полиномиальные алгоритмы для целого ряда сложных и важных задач, решаемых на настоящее время перебором.

Точно **класс** $\mathcal{NP}$ определяется как класс задач, в которых нужно получить ответ «Да» или «Нет» (такие задачи называются задачами с выбором – decision problems), и которые решаются на недетерминированной машине Тьюринга за полиномиальное время. Это означает примерно следующее: класс $\mathcal{NP}$ состоит из задач с выбором, для которых можно написать алгоритм со следующими свойствами: 1) он работает полиномиальное от размера входа время; 2) в алгоритме используется функция f, генерирующая случайный бит; 3) для любых входных данных существует положительная вероятность того, что алгоритм выдаст правильный ответ «Да». Класс $\mathcal{NP}$ задач по сути состоит из задач, решаемых перебором всех возможных конфигураций некоторого объекта, причём длина описания конфигурации ограничена полиномом (многочленом) от длины входа, а число возможных конфигураций растёт более, чем полиномиально. Функция f, генерирующая случайные биты, может быть использована как раз для генерации некоторой случайной конфигурации, а даже алгоритм должен за полиномиальное время проверить, что эта конфигурация удовлетворяет решению.

Задача Л16.8. Верно ли что при условии монотонности (чем тяжелее предмет, тем он ценнее) есть точный жадный алгоритм?

Задача Л16.9. Рассмотрим непрерывную задачу о рюкзаке. Она отличается от обычной тем, что предметы можно разрезать на части и при этом стоимость каждой части пропорциональна весу этой части. Докажите, что в этом случае, жадный алгоритм работает.

В общем случае нет рецептов для определения, есть ли для данной задачи жадный алгоритм или нет, и иногда его не очень просто придумать. Также нетривиальным бывает доказательство правильности работы жадного алгоритма. Обычно оно производится следующим образом. Сначала мы показываем, что на первом шаге, совершая жадный выбор, мы не закрываем пути к оптимальному решению. Для этого показывается, что для любого оптимального решения есть решение не хуже, которое согласованно с жадным выбором. Затем, рассматривая только такие решения, мы сводим задачу к предыдущей, но уже с меньшим количеством элементов.

Довольно часто (хотя и не всегда) установить, что данный жадный алгоритм даёт оптимум, можно с помощью теории матроидов. Вкратце об этом разделе комбинаторики рассказано в главе 17.4 книги [1]. Отметим также, что задача о выборе заявок теорией матроидов не покрывается.

Приближённые алгоритмы

Приведём две задачи, для которых, видимо[2], не существует полиномиального алгоритма, но для которых несложно придумать жадную стратегию, которая работает быстро, но не гарантирует оптимального результата.

Попробуйте самостоятельно найти контрпримеры, для которых жадные алгоритмы находят неоптимальное решение.

Задача Л16.10. (Вариант задачи коммивояжера, salesman problem) Дан граф, в котором вершины обозначают города, а рёбра — дороги между городами. Каждой дороге назначено число — цена билета для данной дороги. Необходимо минимизировать сумму потраченных денег и побывать в каждом городе ровно один раз, начав путешествие с некоторого заданного города.

Класс $\mathcal{P}$ — это задачи с выбором, решаемые за полиномиальное время на обычной машине Тьюринга, то есть это просто задачи с выбором, для которых существует полиномиальный алгоритм решения.

Класс $\mathcal{NP}$**-полных задач** — это такие задачи из класса $\mathcal{NP}$, при наличии полиномиального решения которых доказано существование полиномиального решения всех остальных задач из класса $\mathcal{NP}$.

В настоящее время не известно, существуют ли задачи, которые принадлежат классу $\mathcal{NP}$ и при этом не принадлежат классу $\mathcal{P}$. Проблема «$\mathcal{NP} = \mathcal{P}$ или $\mathcal{NP} \neq \mathcal{P}$?» сводится по сути к вопросу: «Есть ли полиномиальное решение для какой-либо $\mathcal{NP}$-полной задачи?»

Следует отметить, что задача с рюкзаком не является задачей с выбором. Ответ для ней не имеет вид «Да» или «Нет», а является описанием того, какие предметы нужно положить в рюкзак, чтобы их суммарная ценность была максимальной. Когда говорят, что задача о рюкзаке $\mathcal{NP}$-полная, имеют ввиду следующий вариант задачи с выбором: «Можно ли положить в рюкзак предметов суммарной ценности более L?». В этом варианте задачи присутствует ещё один дополнительный аргумент L.

[2]В предположении $P \neq \mathcal{NP}$.

Жадный алгоритм 1. Жадный алгоритм для данной задачи может быть следующий: в каждый момент времени будем искать город, в котором мы ещё не были и до которого цена билета минимальна.

Жадный алгоритм 2. Рассмотрим все дороги. Будем перебирать дороги в порядке возрастания длины и выбирать те дороги, выбор которых не приводит к появлению циклов и ветвлений в сети выбранных (на текущий момент) дорог. На каждый момент граф городов с рёбрами, соответствующими выбранным дорогам, представляет собой набор цепочек и изолированных вершин. На каждом шаге в результате появления еще одного ребра может происходить одно из трёх действий: соединение двух изолированных вершин, соединение изолированной вершины с концом цепочки, соединение концов двух цепочек выбранным ребром. В конечном итоге все вершины будут соединены выбранными дорогами в цепочку.

Жадный алгоритм 3. Будем хранить некоторый путь, проходящий по части городов. Изначально этот путь состоит из одной произвольной дороги, начинающейся в данном городе. На каждом шаге мы будем искать непосещённый город w и дорогу (u, v) в текущем пути такие, что замена дороги (u, v) на пару дорог (u, w) и (w, v) приводит к минимально возможному увеличению длины маршрута. То есть будем искать самое «дешёвое» ответвление от текущего пути в один из непосещённых городов. Осуществляя пошагово такие замены получим некоторый путь, проходящий через все города.

Задача Л16.11. (Сумма подмножества, subset-sum problem) Дан набор натуральных чисел $\{a_1, a_2, \ldots, a_n\}$ и некоторое число S. Найдите такое подмножество чисел, сумма которых меньше S, но при этом максимально близко к S. С этой задачей можно провести следующую жизненную аналогию: необходимо диск фиксированной ёмкости максимально до конца заполнить некоторым подмножеством имеющихся у вас видео-файлов.

Жадный алгоритм для данной задачи может быть следующий: упорядочим числа по величине и будем по порядку их перебирать, выбирая только те, которые не приводят к тому, что сумма выбранных чисел превосходит S.

Семинар 16

«Жадные» алгоритмы

Краткое описание: Мы рассмотрим классические задачи, решаемые жадными алгоритмами: задачу о максимальном числе заявок, и о минимальном числе аудиторий. Рассмотрим также ряд других задач, часть из которых имеет точный жадный алгоритм, а часть могут быть решены жадным алгоритмом лишь приближенно. В последнем случае важно научится из множества возможных приближенных алгоритмов выбирать лучший.

Задача о выборе заявок

Задача С16.1. Прочитайте задачу Л16.1 и изучите псевдокод 16.30 . На основе функции `select`, представленной в коде 16.1, напишите программу, решающую задачу о выборе заявок. В коде 16.1 структура `activity_t` соответствует заявке. Функция `select` получает на вход массив заявок, и возвращает максимальное число совместных заявок, которое можно получить методом жадного выбора. Поле `selected` устанавливается в `1`, если заявка выбрана, в `0` иначе. Время работы функции `select` равно $\Theta(n)$. Но заявки необходимо правильно предварительно упорядочить. С учётом процедуры упорядочивания суммарное время работы вашей программы не должно превосходить $O(n \log n)$. Попробуйте два способа упорядочивания заявок — по времени начала и по времени конца. Найдите такие входные данные, для которых эти способы упорядочивания дают различное число выбранных заявок.

Программа 16.1: Функция, вычисляющая максимальное возможное число заявок.

```
typedef struct {
    int s, f;      // времена начала и конца заявки
    char selected; // выбрана заявка или нет
} activity_t;

int select(activity_t *a, int n){
    int i, j = 0, count = 1;
    a[0].selected = 1;
    for (i = 1; i <= n; i++){
        if ( a[i].s >= ac[j].f ) {
            a[i].selected = 1;
            j = i;
            count++;
        } else {
            a[i].selected = 0;
```

```
        }
    }
    return count;
}
```

Задача о числе аудиторий

Задача С16.2. Напишите программу, решающую задачу о минимальном числе аудиторий Л16.3. Программа получается модификацией кода предыдущей задачи, аналогичной модификации псевдокода 16.30 в псевдокод 16.31.

Задача про атлетов

Задача С16.3. В город N приехал цирк с командой атлетов. Они хотят удивить горожан города и продемонстрировать башню из атлетов максимальной высоты. Башня — это цепочка атлетов, первый стоит на земле, второй стоит у него на плечах, третий стоит на плечах у второго и т.д. Каждый атлет характеризуется силой s_i (в кг.) и массой m_i (в кг.). Сила — это максимальная масса, которую атлет способен держать у себя на плечах. Известно, что если атлет тяжелее, то он и сильнее: то есть если $m_i > m_j$, то $s_i > s_j$. Напишите программу, которая находит башню атлетов максимальной высоты.
Вход. В первой строчке дано число атлетов N. Затем идёт N строчек, в каждой из которой дано два числа — масса и сила атлета.
Выход. В первой строке выведите число M атлетов в башне, а затем M строчек с параметрами атлетов, начиная с нижнего атлета до верхнего.

Задача С16.4. В коде 16.2 представлены две функции `max_athlets1` и `max_athlets2`, вычисляющие башню атлетов. Структура `athlet_t` соответствует атлету. Её поля: `s` — сила атлета, `m` — его масса, `used` — флаг[1], равный единице, если атлет участвует в построении башни, и нулю — в противном случае. Изучите работу этих функций. Находят ли они всегда самую высокую башню атлетов? Какая из них лучше? Есть ли такие входные данные, для которых данные функцию возвращают разный результат а) выбраны разные атлеты б) отличается число выбранных атлетов? Проведите массовое тестирование этих функций.
Подсказка. В функции `max_athlets2` мы строим башню снизу вверх. Переменная `cs` соответствует запас силы башни атлетов, – максимальная масса, которую удержит уже построенная башня. На каждом шаге из атлетов, не занятых в башне, мы выбираем такого, что запас силы новой конструкции максимален.

Программа 16.2: Функции для задачи С16.4.

```
typedef struct athlet {
    int s,    // сила атлета
        m,    // масса
        used; // флаг -- участвует или нет
```

[1]Переменную называют *флагом*, если она принимает одно из двух возможных значений — 0 или 1.

```
} athlet_t;

int cmp(const athlet_t *a, const athlet_t *b) {
        return a->s - b->s;
}
int max_athlets1(athlet_t *a, int n) {
    int i,
        m,     // суммарная масса задействованных атлетов
        count; // число задействованных атлетов

    // упорядочим атлетов по возрастанию силы
    qsort(a, n, sizeof(athlet_t), cmp);

    m = 0 ; count = 0;
    for(i = 0; i < n ; i++) {
        if( a[i].s >= m ) {
            m += a[i].m;
            c++;
            a[i].used = 1;
        } else {
            a[i].used = 0;
        }
    }
    return c;
}


int cmp_reverse(const athlet_t *a, const athlet_t *b) {
        return b->s - a->s;
}
int max_athlets2(athlet_t *a, int n, athlet_t* res) {
    int i, j;
    int cs,  // запас силы выстроенной нижней части башни
        count; // число задействованных атлетов

    // упорядочим атлетов в порядке убывания слилы
    qsort(a, n, sizeof(athlet_t), cmp_reverse);

    // Сразу же возьмём атлета наибольшей силы (он же и наибольшей массы).
    // Он нам точно не "отрежет" путь к оптимальному решению.
    a[0].used = 1;
    cs = a[0].s;
    count = 1; j = 1;
    while (j > -1){
        int cr = 0; // значение запаса силы для следующего шага
        j = -1;
```

```
        // В цикле по i подбираем следующего атлета так, чтобы
        // "запас силы" на следующем шаге был наибольшим
        for (i = 1; i < n; i++){
            if ( !a[i].used && min(cs - a[i].m, a[i].s) > cr){
                cr = min(cs - a[i].m, a[i].s);
                j = i;
            }
        }
        if (j > -1) {
            a[count++] = a[j];
            cs = cr;
            a[j].used = 1;
        } else {
            a[j].used = 0;
        }
    }
    return count;
}
```

Принцип «жадного» выбора и перебор

Задача С16.5. Дано $2n$ точек плоскости. Требуется соединить их отрезками так, чтобы

1) каждая точка была концом ровно одного отрезка;
2) сумма длин отрезков была наименьшей.

Напишите программу, которая решает эту задачу.
Вход. Число точек $N = 2n$ и затем N строчек, в которых даны координаты точек. $2 \leqslant N \leqslant 50$.
Выход. Сумма длин получившихся отрезков и N строчек, в каждой из которых указаны два номера (от 0 до $N-1$) точек, которые соединены отрезком. Попробуйте понять, в каком случая программа будет работать дольше и оцените время её работы в худшем случае. насколько.
Подсказка. В общем случае «жадный» алгоритм эту задачу не решает, однако его использование здесь может быть полезным. Будем решать задачу перебором с отсечением. Напишем рекурсивную процедуру, которая будет выбирать из множества точек две, запуская себя на множестве остальных точек. Такая процедура построит все возможные допустимые соединения. Нам надо лишь выбрать соединение с наибольшей суммой длин отрезков. Очевидна оптимизация этого алгоритма. В процессе построения будем хранить наилучшее найденное соединение, сумму длин отрезков этого соединения, S_{best}, и сумму длин отрезков соединения, которое мы строим — $S_{current}$. Если $S_{current} \geqslant S_{best}$, то текущее соединение точно не будет минимальным. Продолжать строить текущее соединение уже бессмысленно. В самом начале у нас нет наилучшего найденного соединения. Время работы алгоритма сильно зависит от того, как быстро и какое соединение мы найдем первым, потому что по параметрам первого соединения будет сделана значительная часть отсечений. Можно предложить множество различных эвристик поиска начального соединения. Такой эвристикой может быть следующий «жадный» алгоритм:

1) выбираем из множества точек две ближайшие друг к другу;
2) добавляем полученный отрезок в наше начальное соединение;
3) рассматриваем оставшиеся точки и переходим к шагу 1), если такие есть.

Как показывает практика, выбор«жадным» образом начального соединения уменьшает время работы алгоритма в несколько раз.

Задача про минимальное число квадратов

Задача C16.6. Имеется прямоугольник с целыми сторонами, требуется разбить его на минимальное число квадратов так, чтобы они полностью покрывали этот прямоугольник. Предложите один из жадных алгоритмов решения этой задачи и реализуйте его в виде программы на Си. Найдите пример прямоугольника (если такой существует), для которого ваша программа найдёт не самое оптимальное разбиение.

Подсказка. Одна из жадных стратегий основана на алгоритме Евклида. Пусть a, b — длины сторон прямоугольника, а S — конструируемое разбиение (множество квадратов). Рассмотрим следующий алгоритм.

while true do
  if $a = 0$ или $b = 0$ **then**
    break ▷ заканчиваем вычисления
  else if $a > b$ **then**
    $a \leftarrow a - b$)
    добавляем квадрат со стороной b в S
  else if $a \geqslant b$ **then**
    $b \leftarrow b - a$.
    добавляем квадрат со стороной a в S.
  end if
end while

На каждой итерации цикла мы работаем с некоторым прямоугольником A размера $a \times b$; на первой — с исходным. От прямоугольника мы отрезаем с краю квадрат со стороной, равной меньшей стороне прямоугольника, разбивая прямоугольник на квадрат и некоторый прямоугольник A'. На следующих итерациях мы работаем с прямоугольником A'. На каждом шаге мы отрезаем от прямоугольника наибольший квадрат, который в нём помещается. Это и есть принцип «жадного» выбора.

Задача C16.7. Даёт ли описанный выше жадный алгоритм разбиение минимального размера? Проверьте работу алгоритма на прямоугольниках размера 11×3, 5×6, 12×7 и приведите рисунки, отображающие работу алгоритма. Можно ли получить разбиение меньшего размера, если на некоторых шагах отрезать от прямоугольника не один квадрат со стороной b, а два квадрата со стороной $b/2$?

Задача C16.8. Изучите определение 21.7 сложности $C(q)$ рационального числа q, данное на странице 372. Докажите, что $C(a/b)$ равна размеру минимального разбиения прямоугольника $a \times b$.

Задачи на «жадные» алгоритмы

Предложите жадные стратегии решения следующих задач и реализуйте их на языке Си. Докажите, что сконструированные алгоритмы получают оптимальные решения.

Задача С16.9. Дано n действительных точек на прямой. Требуется покрыть эти точки минимальным количеством отрезков длины 1.

Задача С16.10. Дано n отрезков на прямой, с концами в целых точках (координаты по модулю меньше 2^{31}). Выберите из них минимальное число отрезков, которые покрывают некоторый заданный отрезок $[a, b]$.

Задача С16.11. У нас есть один компьютер и n программ. Для каждой программы известно время выполнения на этом компьютере и время к которому эта программа должна быть выполнена (deadline). Требуется составить расписание выполнения программ, при котором будет выполнено их максимальное количество.

Задача С16.12. Изучите условие задачи Л16.10 и предложенные к ней описания жадных алгоритмов. Реализуйте два из этих алгоритмов и осуществите их массовое тестирование. Какой из алгоритмов работает лучше для N точек на плоскости, случайно распределённых а) внутри некоторого круга; б) по нормальному распределению вокруг некоторого центра. Приведите примеры входных данных, для которых ваш алгоритм находит неоптимальное решение (для этого имеет смысл придумать второй более сложный алгоритм и с помощью тестирования на большом числе примеров найти случай, когда он возвращает более оптимальное решение).

Лекция 17

Алгоритмы на графах: обход графа в ширину и в глубину

Краткое описание: Графы являются полезной математической абстракцией, которая широко используемой в дискретной математике и теории программирования. Мы сделаем короткий обзор ключевых понятий теории графов и рассмотрим алгоритмы обхода графов в глубину и в ширину. Эти алгоритмы несут в себе идеи, активно используемые в большинстве задач на графах. В частности, на основе алгоритма обхода графа в глубину строится решение задачи топологической сортировки и поиска сильно связных компонент графа. А с помощью обхода графа в ширину решается задача поиска кратчайшего пути в графе.

Определение графа. Основные понятия и обозначения

Введём основные понятия теории графов[1].

Определение 17.1. **Ориентированный граф** $G = (V, E)$ *определяется как пара* (V, E)*, где* V *— конечное множество, а* E *множество пар элементов из* V*, то есть подмножество множества* $V \times V$*. Элементы множества* V *называются* **вершинами графа***, а элементы множества* E *—* **рёбрами графа***. Если пара* (u, v) *принадлежит* E*, то говорят, что из вершины* u *идёт ребро в* v*.*

Граф называется **неориентированным***, если у рёбер нет ориентации, то есть рёбра представляют собой неупорядоченные пары элементов из* V*. В случае неориентированных графов говорят, что ребро* (u, v) *соединяет вершины* u *и* v*.*

Если $u = v$*, то ребро* (u, v) *называется* **петлёй***.*

Обычно граф представляют в виде рисунка, на котором вершины обозначают точками, а рёбра — соединительными линиями. В случае ориентированных графов на линиях ставят стрелку, которая направлена от первой вершины ко второй. Линии, соответствующие рёбрам, могут пересекаться в некоторых точках, не являющихся вершинами. Это не имеет никакого значения. Также не имеет значения взаимное расположение точек на плоскости. Важен лишь сам факт, что некоторые пары вершин соединены ребром.

- Вершина v называется **смежной** с вершиной u, если в графе существует ребро (u, v). Множество всех вершин смежных с u будем обозначать $Adj[u]$.

[1]Следует иметь в виду, что терминология здесь не вполне устоялась, несмотря на то, что теория графов начала развиваться давно.

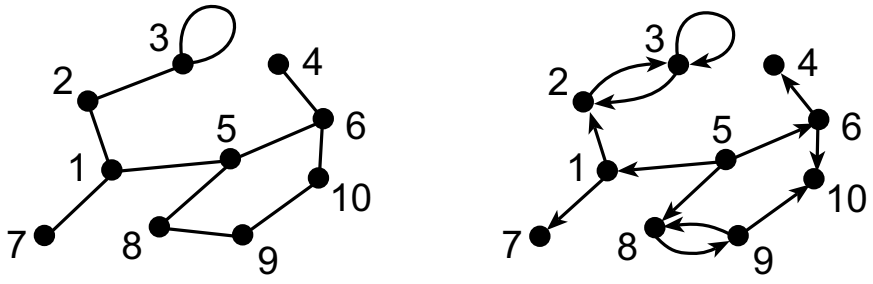


Рис. 17.1: Примеры неориентированного (а) и ориентированного (б) графа. В неориентированном графе (а) есть путь, соединяющий любую пару вершин. В ориентированном графе (б) не для любой пары вершин (u, v) существует путь из u в v. Например, пути из вершины 1 в вершину 6 не существует. В неориентированном графе (а) два цикла (3) и (5, 6, 10, 9, 8). В ориентированном графе (б) три простых цикла: (3), (8, 9) и (2, 3).

Как обычно $|M|$ обозначает количество элементов множества M. Выражения $|Adj[u]|$, $|V|$, $|E|$ означают соответственно число вершин смежных с u (**степень вершины** u), число вершин в графе и число рёбер в графе.

Для неориентированного графа $\sum_{v \in V} |Adj(v)| = 2 \cdot |E|$, а для ориентированного графа $\sum_{v \in V} |Adj(v)| = |E|$.

- **Путь** из вершины u в вершину v — это последовательность рёбер графа $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, $e_3 = (v_2, v_3)$, $\ldots$, $e_k = (v_{k-1}, v_k)$, в которой $v_0 = u$, $v_k = v$ и $e_i \in E$ для всех $i = 1, \ldots, k$. Путь называется **простым**, если вершины $v_0, \ldots, v_k$ попарно различны. Число k называется **длиной пути**.

  В ориентированном графе путь называется **циклом**, если $v_0 = v_k$. Цикл называется **простым**, если вершины $v_0, \ldots, v_{k-1}$ попарно различны (то есть, если цикл не имеет «самопересечений»).

  В неориентированном графе путь называется **циклом**, если $v_0 = v_k$, и все рёбра в этом пути различны.

- Неориентированный граф называется **связным**, если для любой пары вершин существует путь из одной в другую. Множество вершин неориентированного графа, до которых существует путь из вершины u будем называть **связной компонентой вершины** u. Для каждой пары вершин одной связной компоненты существует путь из одной в другую. Граф однозначно разбивается на связные компоненты. Для пары вершин из разных компонент не существует соединяющего их пути. Связным называется граф, если он состоит из одной связной компоненты.

- Число рёбер в кратчайшем пути между двумя вершинами u и v мы будем называть **расстоянием** между этими вершинами и обозначать $\delta(u, v)$. Если вершины u и v не связаны, то будем считать, что $\delta(u, v) = \infty$. Расстояние между смежными вершинами равно 1. Для функции расстояния $\delta(u, v)$ выполняются следующие свойства:

$$\delta(u, v) = 0 \Leftrightarrow u = v$$
$$\delta(u, v) \leqslant \delta(u, v') + \delta(v', v)$$

- Неориентированный граф G называется **деревом**, если он связный и не имеет циклов.
- **Графом с взвешенными рёбрами** мы будем называть граф $G = (V, E)$, в котором каждому ребру приписано некоторое число (вес). Вес ребра (u, v) будем обозначать как $w(u, v)$.

На рисунке 17.2 показано несколько примеров неориентированных графов.

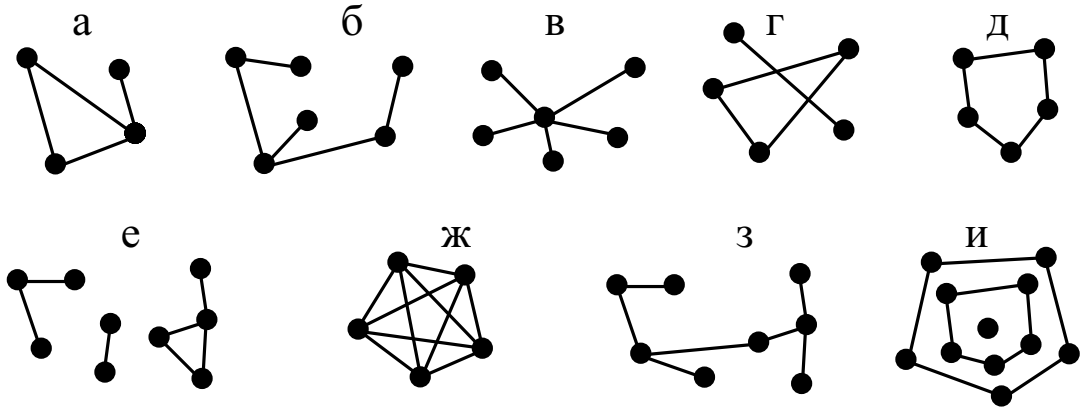


Рис. 17.2: Примеры изображений неориентированных графов на плоскости. Графы (б), (в) и (з) не имеют циклов и являются деревьями. Граф (г) имеет две связные компоненты, графы (е) и (и) имеют три связные компоненты, а остальные графы односвязные. Граф (ж) является полным графом размера 5.

Перечислим несколько классических алгоритмических задач на графах:

Задача Л17.1. (проверка на связность) Вход: неориентированный граф. Определить, является граф связным или нет. Если граф не связный, перечислить его связные компоненты.

Задача Л17.2. (является ли граф деревом) Вход: неориентированный граф. Определить, есть в графе циклы или нет.

Задача Л17.3. (кратчайший путь) Вход: неориентированный граф и две выделенные вершины графа u и v. Найти кратчайший путь из u в v.

Задача Л17.4. (цикл Эйлера) Вход: неориентированный граф. Проверить, существует ли цикл, который проходит по всем рёбрам графа ровно один раз. Если такой цикл есть, выведите его. Примечание: задача равносильна задаче нарисовать фигуру, не отрывая ручки от бумаги; при этом необходимо вернуться в исходную точку, и запрещено проводить одну и ту же линую дважды.

Задача Л17.5.* (цикл Гамильтона) Вход: неориентированный граф. Проверить, существует ли цикл, который проходит по всем вершинам графа ровно один раз. Если такой цикл есть, выведите его.

Задача Л17.6.* (проверка на планарность) Вход: неориентированный граф. Определить, можно ли нарисовать данный граф на плоскости без пересечений рёбер.

Некоторые из этих задач простые (проверка на связность, является ли граф деревом, кратчайший путь), другие сложнее (цикл Эйлера, проверка на планарность), третьи практически неразрешимы для больших графов (цикл Гамильтона).

Прежде, чем перейти к обсуждению алгоритмов решения различных задач на графах, рассмотрим различные способы представления графов в памяти.

Представление графа в памяти

Существует два основных метода представления графов в памяти: в виде массива множеств смежных вершин или в виде матрицы смежности.

Если вершины графа являются сложными объектами, то следует в первую очередь, поставить в соответствие вершинам идентификаторы от 0 до $|V|-1$, либо использовать одну из реализаций ассоциативного массива. Кроме того, существует несколько различных способов хранения множества смежных вершин. Например, его можно реализовать односвязный список или как ассоциативный массив, сопоставляющий значение **true** только смежным вершинам.

У каждого из указанных методов есть свои преимущества и недостатки.

Когда мы используем представление графа $G = (V, E)$ в виде **списков смежных вершин**, мы храним в памяти массив Adj, содержащий для каждой вершины $u \in V$ список всех смежных с ней вершин $Adj[u]$. Суммарное число элементов во всех списках составляет $|E|$ для ориентированных графов и $2|E|$ для неориентированных. Поэтому для хранения графа в таком виде, требуется $O(V + E)$ памяти. Этот способ особенно удобен, когда мы имеем дело с разреженными графами.

Заметим, что это представление вполне пригодно для хранения графов с весами. Для этого мы должны в списке, соответствующем вершине u, хранить пары $(v, w(u, v))$. Такое представление не позволяет быстро проверить, есть ли в графе ребро (u, v). Действительно, для этого нам придётся пробежаться по всему списку $Adj[u]$, который может иметь размер $|V|-1$. Поэтому в некоторых случаях удобно для каждой вершины хранить хэш-таблицу, содержащую отображение вершины v на множество пар $(v, w(u, v))$.

Есть второй, более простой способ представления графов в памяти, в котором описание графа хранится в виде **матрицы инцидентности**, то есть в виде массива размерности 2. В этом способе вершины графа необходимо пронумеровать числами $0, \ldots, |V|-1$. Элемент a_{ij} матрицы инцидентности показывает, соединены вершины i и j ребром или нет:

$$a_{ij} = \begin{cases} 1, & \text{если есть ребро из } i\text{-й вершины в } j\text{-ю}, \\ 0, & \text{если такого ребра нет.} \end{cases}$$

Для неориентированных графов матрица инцидентности симметрична. Этот способ требует $O(V^2)$ памяти для хранения графа. Поэтому его используют в основном для небольших и/или **плотных графов** — графов, в которых число ребер достаточно велико, а именно, $|E|$ сравнимо с $|V|^2$. Для случая графов с взвешенными рёбрами граф представляют в виде матрицы весов — для ребра (i, j) вместо 1 в матрице храним вес $w(i, j)$ этого ребра. Для обозначение ребра, отсутствующего в графе, обычно используется некоторый специфический вес (0, -1 или ∞).

Следующая таблица обобщает сведения о представлениях графов в памяти.

Представление	Списки смежности	Матрица инцедентности
Требуемая память	$O(V + E)$	$O(V^2)$
Преимущества	требует мало памяти в случае разреженных графов	простой и удобный способ доступа к данным

Поиск в ширину. Кратчайший путь в графе

Поиск в ширину (breadth-first search, BFS) — один из базовых алгоритмов на графах. Пусть задан граф $G = (V, E)$ и некоторая начальная вершина s. Поиск в ширину осуществляет просмотр вершин графа в порядке возрастания расстояния от s. Напомним, что расстоянием от вершины s до вершины u называется длина кратчайшего пути из s в вершину u.

Введём обозначения:

- $d[u]$ — расстояние от s до u.
- $c[u]$ — цвет вершины.
- $\pi[u]$ — вершина, которая будет идти предпоследней в кратчайшем пути из s в u. Эту вершину мы будем называть **предшественником** u.

Алгоритм использует очередь Q, для которой определены две операции:

- Enqueue(Q, v) — добавляет элемент v в конец очереди;
- Dequeue(Q) — возвращает в качестве результата первый элемент очереди и при этом удаляет его из очереди.

Алгоритм 17.32 Поиск в ширину

```
 1: procedure BFS(G, s)
 2:     for всех u ∈ V[G] \ {s} do
 3:         d[u] ← ∞
 4:         c[u] ← white
 5:         π[u] ← nil
 6:     end for
 7:     d[s] ← 0
 8:     c[s] ← grey
 9:     Q ← s
10:     while Q ≠ ∅ do
11:         u ← Dequeue(Q)
12:         for всех v ∈ Adj[u] do
13:             if c[v] =white then
14:                 d[v] ← d[u] + 1
15:                 π[v] ← u
16:                 Enqueue(Q, v)
17:                 c[v] ← grey
18:             end if
19:         end for
20:         c[u] ← black
21:     end while
22: end procedure
```

В строках 2–9 производится инициализация данных. В очередь Q помещается вершина, из которой мы ищем путь. В основном цикле программы (строки 10–10) последовательно извлекаются вершины из очереди Q и для каждой вершины определяются те вершины, в которые можно попасть из данной, но в которых мы ещё не были. Эти новые обнаруженные вершины

добавляются в конец очереди. Алгоритм заканчивает работу, когда очередь становится пустой, то есть когда все обнаруженные вершины рассмотрены.

Массив c цветов вершин используется для того, чтобы отмечать обнаруженные и не добавлять их в очередь повторно. Вначале все вершины кроме вершины s белые (white). Когда алгоритм находит какую-нибудь белую вершину, он добавляет её в очередь и делает серой (grey).

Множество белых вершин — это необнаруженные пока вершины, то есть «белые пятна на карте». Серые вершины — это обнаруженные вершины, «окрестности» которых ещё не исследованы. И наконец, чёрные вершины — это обнаруженные вершины с исследованными «окрестностями».

Для каждой обнаруженной вершины v значение $\pi[v]$ указывает на вершину u, из которой вершина v была обнаружена. Вершина u является предшественником вершины v. У каждой вершины кроме s есть ровно один предшественник. Массив $\pi[u]$ задает дерево на множестве вершин графа. Корнем дерева является вершина s. Цепочка предшественников

$$u \to \pi[u] \to \pi[\pi[u]] \to \pi[\pi[\pi[u]]] \to \dots$$

оборвётся на вершине s, для которой $\pi[s]$ неопределено ($\pi[s] = \texttt{nil}$).

Массив π позволяет восстановить путь от данной вершины s к любой вершине u. Как будет показано далее, этот путь является одним из кратчайших путей между s и u.

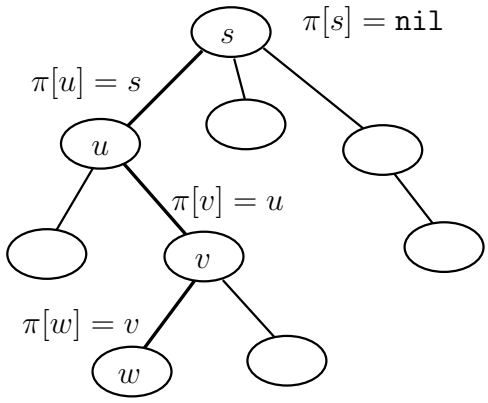


Рис. 17.3: Дерево предшественников, задаваемое массивом π.

Поясним, почему этот алгоритм находит кратчайший пусть. Рассмотрим вершины в последовательности, в какой он попадали в очередь. Величина d в этой последовательности не убывает. Сначала в очередь попадают *все* вершины, в которые можно за один шаг попасть непосредственно из вершины s. Для таких вершин $d = 1$. Доставая их из очереди мы ищем в смежные им белые вершины. Такие вершины будут помещены в конец очереди и для них значение d будет равной 2. Поскольку в очередь изначально шла группа *всех* вершин с $d = 1$, то обработав их окрестности, мы найдём *все* вершины с $d = 2$. Аналогично обработав все вершины с $d = 2$ мы найдём все вершины с $d = 3$ и так далее. Группы вершин с одинаковыми значениями d идут через очередь сплочёнными группами. Группа с $d = K$ представляют собой «фронт волны», распространяющейся из вершины s, на шаге K. Обрабатывая последовательно вершины из этого фронта, алгоритм находит и кладёт в конец очереди вершины из фронта с $d = K + 1$.

Строгое доказательство того, что d, вычисляемое алгоритмом поиска в ширину, равняется расстоянию до вершины, дано в [1].

Оценим сложность описанного алгоритма BFS. Будем считать, что граф представляется в виде списков смежности. На инициализацию данных требуется $O(V)$ операций. Каждая обнаруженная вершина будет добавлена в очередь и затем будет рассматриваться в цикле не более одного раза. При рассмотрении вершины мы проверяем все смежные с ней вершины. Поэтому строчки 13-17 внутри всех циклов будут выполнены не более $\sum_{v \in V} |Adj(v)| = O(E)$ раз. Поэтому сложность алгоритма равна $O(V + E)$.

Алгоритм поиска в ширину очень часто применяется на практике. В основном он используется для нахождения кратчайший путей в графах с одинаковыми весами рёбер.

Задача Л17.7. Как изменится время работы алгоритма, если мы будем использовать представление графа в виде матрицы смежности (изменив соответствующим образом процедуру BFS)?

Задача Л17.8. Постройте алгоритм который по заданному $\pi[u]$ для произвольной вершины u строит за время $O(V)$ кратчайший путь из s в u.

Простейшим примером использования алгоритма DFS является задача поиска кратчайшего пути лабиринта. Лабиринт это квадратное поле размера $n \times n$. Часть ячеек лабиринта проходима, а часть – нет. Необходимо найти кратчайший путь из верхнего левого угла лабиринта, в нижний правый.

Проверьте, что алгоритм основанный на DFS, работает в худшем случае время, пропорциональное площади лабиринта.

Поиск в глубину. Топологическая сортировка

Алгоритм поиска в глубину в отличается от поиска в ширину тем, что он пытается идти «вглубь», пока это возможно. При обнаружении некоторой вершины алгоритм сразу переходит к её обработке, и возвращается к исходной вершине, только после её обработки.

Следующая схема описывает алгоритм поиска в глубину (Depth First Search, DFS)

Так же как и в алгоритме поиска в ширину, здесь используется цвет $c[u]$ для обозначения статуса вершин. Белый цвет означает, что вершина ещё не обнаружена (белые пятна на карте). Серый цвет означает, что вершина обнаружена, но ещё не обработана (новая «земля» обнаружена, но её окрестности ещё не исследованы). Черный цвет означает, что вершина обнаружена и обработана («земля» обнаружена и её окресности исследованы).

В алгоритме присутствуют две процедуры. Поиск в глубину осуществляет только рекурсивная процедура DFS-Visit(v). Процедура DFS нужна лишь для того, чтобы пройти все вершины в случае, когда граф состоит из нескольких компонент связности.

Ещё один важный момент: здесь введены дополнительные переменные $d[u]$, $f[u]$, $time$. Переменная $time$ увеличивается перед каждым присваиванием в строках 3 и 12 процедуры DFS-Visit. Эта переменная «считает» суммарное число входов и выходов из процедуры DFS-Visit и её можно интерпретировать как время (на «вход» в вершину или «выход» из вершины тратится один такт). Элемент $d[u]$ содержит время начала обработки вершины u, а элемент $f[u]$ — время окончания. Вершина u будет белой до момента $d[u]$, серой — между $d[u]$ и $f[u]$, и черной — после $f[u]$. Эти переменные не используются алгоритмом поиска в глубину, но понадобятся нам в дальнейшем, когда мы будем рассматривать применения алгоритма DFS.

Значение $\pi[u]$ для вершины u указывает на вершину, из которой мы пришли в u. И с помощью $\pi[u]$ мы можем восстановить путь из вершины, в которой начинался поиск, в данную. При обходе графа в глубину (в отличие от обхода в ширину) найденные пути могут оказаться не кратчайшими.

Вершину u будем называть предком v, если между моментом как алгоритм «зашёл» в вершину u, и моментом, когда он «вышел» из неё, была посещена вершина v.

Сложность алгоритма

Оценим теперь сложность алгоритма DFS. Опять будем считать, что мы храним граф в виде списков смежности. Для каждой вершины процедура DFS-Visit будет вызвана ровно

Алгоритм 17.33 Поиск в глубину

```
 1: procedure DFS(G)
 2:     for всех u ∈ V[G] do
 3:         c[u] ← white
 4:         π[u] ← nil
 5:     end for
 6:     time ← 0
 7:     for всех вершин u ∈ V[G] do
 8:         if c[u] =white then
 9:             DFS-Visit(u)
10:         end if
11:     end for
12: end procedure
```

```
 1: procedure DFS-Visit(u)
 2:     c[u] ← grey
 3:     time ← time + 1
 4:     d[u] ← time
 5:     for всех v ∈ Adj[u] do
 6:         if c[v] =white then
 7:             π[v] ← u
 8:             DFS-Visit(v)
 9:         end if
10:     end for
11:     c[u] ← black
12:     time ← time + 1
13:     f[u] ← time
14: end procedure
```

один раз. Следовательно, на вызовы этой процедуры потребуется $O(V)$ операций. На выполнение непосредственно циклов в строчках 2-5 и 7-11 процедуры $DFS(G)$ также потребуется время $O(V)$. Цикл внутри процедуры DFS-Visit(u) будет выполнен $|Adj[u]|$ раз. Так как $\sum_{u \in V} |Adj[u]| = O(E)$, то общее время, которое потребуется на выполнение циклов в DFS-Visit составит $O(E)$. Поэтому суммарное время работы алгоритма $O(V + E)$.

Свойства обхода в глубину

Теорема 17.1. При обходе графа $G = (V, E)$ в глубину, для любых $u, v \in V$ справедливо ровно одно из следующих утверждений:

1. $[d[u], f[u]] \subset [d[v], f[v]]$ и вершина u является потомком v при поиске в глубину;
2. $[d[v], f[v]] \subset [d[u], f[u]]$ и вершина v является потомком u при поиске в глубину;
3. $[d[u], f[u]]$ и $[d[v], f[v]]$ не пересекаются и ни одна из вершин u и v не является предком другой.

Доказательство. Если u — потомок v, то к обработке вершины u мы переходим через цепочку рекурсивных вызовов DFS-Visit, которая начинается в 5-й строчке процедуры, обрабатывающей v. Таким образом, выполнение DFS-Visit(u) происходит между присваиваниями $d[v]$ и $f[v]$. Следовательно, $[d[u], f[u]] \subset [d[v], f[v]]$.

Аналогично рассматривается случай, когда v — потомок u.

Если же из u и v ни одна из вершин не является предком другой, то обработка одной вершины в процедуре DFS-Visit должна полностью завершиться до того, как начнется обработка другой. Поэтому $[d[u], f[u]]$ и $[d[v],\ f[v]]$ не пересекаются. □

Теорема 17.2. (теорема о белом пути) *Вершина v является потомком вершины u в лесе поиска в глубину, тогда и только тогда, когда в момент обнаружения вершины u существует путь из u в v, проходящий только по белым вершинам.*

Доказательство. Так как рекурсивные вызовы выполняются только для белых вершин, то если такого пути не существует, v не может являться потомком u.

Предположим теперь, что существует такой путь из белых вершин и пусть v не является потомком u. Тогда к концу обработки u вершина v останется белой. Пусть v' — последняя черная вершина на рассматриваем пути из u в v. Тогда у v' существует соседняя вершина, которая и осталась белой после обработки v', что невозможно. Следовательно, v будет потомком u. □

Малые дополнения к алгоритму обхода графа в глубину позволяют решить задачу разбиения графа на связные компоненты, проверить, является ли граф деревом, осуществить топологическую сортировку вершин графа и решить множество других алгоритмических задач.

Задачи и упражнения

Задача Л17.9. Каким будет время работы алгоритма $DFS(G)$, если граф будет представлен в виде матрицы смежности?

Задача Л17.10. Может ли вершина оказаться единственной в лесе обхода графа в глубину, если в графе есть как входящие в неё, так и исходящие из неё ребра?

Задача Л17.11. Для неориентированного графа $G = (V, E)$ построить алгоритм, определяющий все его связные компоненты.

В этом и следующем разделах мы рассмотрим две классические задачи, для решения которых можно использовать алгоритм поиска в глубину. Первая задача — это задача о топологической сортировке (Topological Sort).

Топологическая сортировка

Задача Л17.12. Пусть имеется ориентированный граф $G = (V, E)$ без циклов. Требуется указать такой порядок на множестве вершин V, что любое ребро ведёт из вершины с меньшим номером в вершину с большим.

Алгоритм 17.34 осуществляет топологическую сортировку вершин ациклического графа G.

Алгоритм 17.34 Топологическая сортировка

```
 1: procedure TOPOLOGICAL-SORT(G)
 2:     L ← ∅
 3:     for всех u ∈ V[G] do
 4:         c[u] ← white
 5:     end for
 6:     for всех вершин u ∈ V[G] do
 7:         if c[u] = white then
 8:             DFS-VISIT(u)
 9:         end if
10:     end for
11:     return L
12: end procedure
```

```
 1: procedure DFS-VISIT(u)
 2:     c[u] ← grey
 3:     for всех v ∈ Adj[u] do
 4:         if c[v] = white then
 5:             DFS-VISIT(v)
 6:         end if
 7:     end for
 8:     c[u] ← black
 9:     добавить u в начало списка L
10: end procedure
```

Приведённый алгоритм осуществляет обычный обход графа в глубину, только в момент, когда вершина становится чёрной, она добавляется в конец списка L.

Теорема 17.3. *Алгоритм* TOPOLOGICAL-SORT *правильно осуществляет топологическую сортировку ориентированного графа $G = (V, E)$ без циклов.*

Доказательство. Рассмотрим произвольное ребро $(u, v) \in E$. Докажем, что u находится в списке L перед v. Так как в графе G нет циклов, то в нём не существует пути из v в u. Поэтому к началу обработки u вершина v в дереве поиска не может быть серой. Если вершина v белая, то она становится ребенком u и поэтому будет добавлена в список раньше u(то есть окажется после u). Если же v — черная, то это значит, что к началу обработки u, она уже добавлена в список L. То есть в любом случае u будет расположена перед v в L. □

Задача Л17.13. Постройте алгоритм, который за время $O(V)$ определяет, есть ли в данном неориентированном графе $G = (V, E)$ циклы.

Задача Л17.14. Верно ли, что алгоритм TOPOLOGICAL-SORT для ориентированного графа с циклами находит такой порядок следования вершин, при котором число рёбер, идущих из вершины с большим номером в вершину с меньшим, минимально?

*Сильно связные компоненты

Ещё одним важным применением алгоритма обхода графа в глубину является задача о разбиении ориентированного графа на **сильно связные компоненты**.

Определим понятие сильной связности. Две вершины ориентированного графа **сильно связаны**, если существует путь из u в v и из v в u. То, что вершины u и v сильно связаны, будем обозначать как $u \sim v$. Отношение сильной связности **транзитивно**, то есть

$$(u \sim v \;\&\; v \sim t) \Rightarrow u \sim t.$$

Благодаря этому свойству вершины графа разбиваются на группы вершин, такие, что в каждой группе любая пара вершин сильно связана, а любые две вершины из разных групп не являются сильно связанными. Такие группы называются **сильно связными компонентами** графа.

Ориентированный граф называется **сильно связным**, если из любой его вершины существует путь в любую другую вершину. Сильно связный граф представляет собой одну сильно сильно связную компоненту.

Нам потребуется понятие транспонированного графа, который получается из изходного обращением направления всех рёбер:

Определение 17.2. *Граф $G^T = (V, E^T)$, где $E^T = \{(u, v) | (v, u) \in E\}$, называется транспонированным к графу $G = (V, E)$.*

Транспонированный граф можно легко построить за время $O(V + E)$ (если оба графа заданы списками смежности).

Алгоритм 17.35 Сильно связные компоненты

1: **procedure** SCC(G)
2: Вызвать $DFS(G)$, и
найти время окончания обработки $f[u]$ для каждой вершины u.
3: Построить граф G^T.
4: Вызвать $DFS(G^T)$, при этом во внешнем цикле
вершины нужно перебирать в порядке убывания $f[u]$.
5: Сильно связными компонентами будут деревья рекурсивного обхода вершин графа, построенные на предыдущем шаге.
6: **end procedure**

Докажем, что этот алгоритм находит сильно связные компоненты графа G.

Для начала нам понадобится несколько вспомогательных утверждений.

Лемма 1: Транспонированный граф имеет те же самые сильно связные компоненты, что и исходный граф.

Лемма 2: Если две вершины принадлежат одной сильно связной компоненте, то никакой путь между ними не выходит за пределы этой компоненты.

Докажите эти леммы самостоятельно.

Лемма 3: В процессе поиска в глубину все вершины одной сильно связной компоненты попадут в одно дерево поиска.

Доказательство. Рассмотрим какую-нибудь сильно связную компоненту. Пусть $u \in U$ первая вершина этой компоненты, которая была обработана. Из вершины u существует путь в

любую другую вершину этой компоненты не выходящий за её пределы. Такая вершина u — первая, то по теореме о белом пути все вершины компоненты U попадут в одно дерево, и u будет их общим предком. □

Теорема 17.4. *Алгоритм* SCC *находит сильно связные компоненты неориентированного графа.*

Доказательство. Рассмотрим какое-нибудь из деревьев U, получившихся в 4-й строке алгоритма. Докажем, что это дерево является сильно связной компонентой. Пусть u — вершина этого дерева. Из u существует путь в любую вершину дерева в графе G^T. Следовательно для любой $v \in U$ существует путь в графе G из v в u. Осталось показать, что из u можно попасть в любую вершину U в графе G. Далее мы будем рассматривать только вызов $DFS(G)$ во 2-й строчке алгоритма.

Пусть $U' \subset U$ — некоторая сильно связная компонента. И пусть v — вершина этой компоненты, у которой значение $f(v)$ наибольшее. Так как u — корень дерева U, то $f[v] < f[u]$. Возможны два случая:

1. $d[v] < d[u]$, то есть к моменту, когда мы начали обрабатывать v, вершина u ещё была белой. Рассмотрим путь из v в u. Так как v имеет наибольшее значение $f[v]$, то она начала обрабатываться первой из вершин компоненты U' и, следовательно, на пути из v в u не может быть серых вершин (серые вершины находятся с v в одной связной компоненте). Также там не может быть и чёрных вершин, так как в этом случае u сама должна быть черной. Следовательно, к началу обработки v существует путь из белых вершин из v в u, и поэтому вершина v должна быть предком u, то есть $f[v] > f[u]$. Следовательно, этот случай невозможен.

2. $d[v] < d[u]$, в этом случае выполняется $d[u] < d[v] < f[v] < f[u]$. Поэтому по теореме о вложениях интервалов для потомков v — является потомком u в некотором дереве поиска. Следовательно существует путь из u в v в графе G. Таким образом, $u \in U'$.

Так как U' произвольная компонента, то всё дерево U является одной сильно связной компонентой. □

Задача Л17.15. Как может изменится число компонент сильно связного графа при добавлении к нему одного ребра?

Семинар 17

Алгоритмы на графах: обход графа в ширину и в глубину

Простейший пример графа из жизни — это города (вершины графа), связанные дорогами (рёбрами графа). Если все дороги имеют односторонее движение, то граф **ориентированный**, а соответствующие им рёбра называют **направленными**. Дорогу с двусторонним движением удобно рассматривать как две дороги с односторонним движением.

Каждому ребру в графе дорог естественно сопоставить число — длину дороги или среднее время движения по этой дороге. Это число мы будем называть длиной или весом ребра.

Первая важная задача на графах — это задача о кратчайшем пути между двумя вершинами:

Задача С17.1. (кратчайший путь) Дан ориентированый граф и две выделенные вершины графа s и t. Найти путь из s в t, состоящий из минимального числа рёбер (путь с минимальным числом шагов).

Задача С17.2. (кратчайший взвешенный путь) Дан ориентированый граф с взвешенными рёбрами и две выделенные вершины графа s и t. Найти путь из s в t, для которого сумма весов входящих в него рёбер минимальна (путь с минимальной суммарной длиной шагов).

Для решения первой задачи используется процедура обхода графа в ширину. Эффективное решение второй задачи основано на жадной стратегии, и его построением мы займёмся на следующем семинаре.

Обход графа в ширину. Задача «Лабиринт»

Цель процедуры обхода графа заключается в том, чтобы посетить каждую вершину графа. Обычно во время этого обхода осуществляют какие-либо дополнительные вычисления с данными, прикреплёнными к вершинам и рёбрам графа. Очень часто процедура обхода графа является лишь каркасом для решения самых различных алгоритмических задач.

На основе процедуры обхода графа в ширину решается задача С17.1 поиска кратчайшего пути в графе (длина пути измеряется в количестве рёбер).

Программа 17.1: Реализация поиска в ширину на языке Си.

```
typedef enum { WHITE, GREY, BLACK } colors_t;
const int INFTY = 0x7fffffff; // = (2^31-1)
typedef struct {
    int V, E;  // число вершин и число рёбер;
    int **adj; // Adj[u] = массив вершин смежных с u;
```

```
    int *ne;   // ne[u] = размер массива вершин смежных с u
    int *pi;   // pi[u] = вершина из которой мы "шагнули"
               //         в вершину u;
    int *color; // color[u] = цвет вершины u;
    int *d;    // d[u]  = длина кратчайшего пути до вершины u.
} graph_t;
/*
 Пропущены описания функций для работы с очередью.
 Очередь содержит элементы типа int - идентификаторы вершин.

 Пропущена также функция main, где осуществляется инициализация
 графа, заданного переменной g.
*/
void bfs(graph_t *g, int s) {
    // g - граф
    // s - идентификатор вершины, с которой начинается поиск.
    int i, u, v;
    for(i = 0; i < g->V; i++) {
        g->color[i]  = WHITE; // Изначально все вершины белые,
        g->d[i]  = INFTY;     // расстояние до них неизвестно,
        g->pi[i] = -1;        // предшественники не заданы.
    }
    queue_t *q = new_queue (g->V);
    enqueue(q, s);
    g->d[s] = 0;
    g->color[s] = GREY;
    while( dequeue(q, &u) ) {
        for(i = 0 ; i < g->ne[u]; i++) {
            v = g->adj[u][i];
            if (g->color[v] == WHITE) {
                g->pi[v] = u;
                g->d[v] = g->d[u]+1;
                g->color[v] = GREY;
                enqueue(q, v);
            }
        }
        g->color[v] = BLACK;
    }
    delete_queue(q);
}
```

В приведённом коде пропущен код функций, для работы с очередью:

- `queue_t *new_queue (int n)`:
  Создать очередь. В аргументе указывается максимальный ожидаемый размер очереди.
- `void delete_queue (queue_t *q)`:
  Удалить очередь (освободить выделенную под очередь память).

- `int dequeue(queue_t *q, int *p)`:
  Извлекает идентификатор вершины из очереди и помещает его по адресу `*p`. Функция возвращает 0, если очередь пуста, иначе — 1.

- `int enqueue(queue_t *q, int x)`:
  Помещает элемент `x` в очередь `q`.

В приведённой программе пропущена также инициализация графа, заданого глобальной переменной `graph_t *g`.

Задача С17.3. (Трассировка[1] «перекрашивания» в BFS) Изучите псевдокод 17.32 обхода графа в ширину на странице 316. Проведите аналогию с принципом распространения волны — вершины, которые находятся в очереди являются фронтом волны. Начальную вершину можно рассматривать как точечный источник волны, которая постепенно начинает расширятся и на каждом шаге представляет собой границу некоторой окрестности начальной вершины. Из этой аналогии и пошло название процедуры обхода в ширину.

На основе кода 17.1 напишите программу, которая считывает описание графа, осуществляет обход его в ширину и при этом выводит информацию о вершинах, которые «перекрашиваются» в серый и чёрный цвета. На вход программы подаётся описание графа в следующем формате: в первой строчке указывается число вершин N и число рёбер M, затем идёт M строчек, каждая из которых содержит описание одного ребра, а именно, пару идентификаторов вершин от 0 до $N-1$.

Забегая вперёд, проведём сравнительную аналогию обхода в ширину и обхода в глубину. В то время как поиск в ширину ассоциируется с одновременным распространением частиц во все стороны из некоторого точечного источника, то обход в глубину моделирует перемещение одного человека в лабиринте; этот человек поставил себе целью обойти весь лабиринт и использует для этого клубок ниток (чтобы всегда знать дорогу назад) и карандаш, чтобы помечать те места в лабиринте, где он уже побывал. Важно заметить, что алгоритм обхода в глубину можно получить из алгоритма обхода в ширину простой заменой структуры данных «очередь» на «стек».

Задача С17.4. (лабиринт) Напишите программу, которая находит кратчайший путь в лабиринте. Лабиринт это квадратное поле $n \times n$, разбитое на квадратные ячейки. На вход поступает число n и карта лабиринта — n строчек по n символов (каждая строчка завершается символом «`\n`»). Символ '.' означает проходимую ячейку, символ '#' — стенку, символ 'S' — начало пути, символ 'F' — конец пути. Каково максимальное время работы вашего алгоритма? Убедитесь, что время работы алгоритма в худшем случае растёт пропорционально площади лабиринта. Используйте в программе структуру данных `cell_t`:

```
typedef struct cell_s {
    int x,y; // координаты ячейки
} cell_t;
```

Используйте очередь из элементов этого типа для хранения «фронта волны». Заведите также два глобальных двумерных массива `pi` и `color` и в элементах `pi[x][y]` и `color[x][y]` храните информацию о родительской ячейке и цвете ячейки с координатами (x, y). После

[1] Трассировка (англ. *tracing*) — слежение за ходом выполнения. В программировании трассировкой называют вывод информации о выполняемых действиях и текущих значениях переменных.

процедуры обхода в глубину нужно, используя данные массива `pi[x][y]`, восстановить кратчайший путь (восстанавливать нужно, начиная с конечной вершины).

Обход графа в глубину. Задача «Одежда профессора»

Задача C17.5. (Трассировка «перекрашивания» в DFS) Изучите алгоритм обхода графа в глубину, записанный в виде псевдокода 17.33 на странице 319. На основе кода 17.2 напишите программу, которая выводит информацию о вершинах, которые «перекрашиваются» в серый и черный цвета. Вход такой же, как и в задаче C17.3.

Программа 17.2: Реализация топологической сортировки на языке Си

```
typedef enum {WHITE, GREY, BLACK} color_t;
/*
    здесь пропущены объявления и инициализация
    глобальной переменной n (число вершин графа) и
    глобальных массивов color, adj и ne:
        color[v] - цвет вершины v
        adj[v]   - список смежных вершин вершины v
        ne[v]    - число смежных вершин вершины v */

/* dfs_visit(v) - вывести те вершины графа, которые должны
   быть выведены перед вершиной v, а затем и саму вершину v. */
void dfs_visit(int v) {
    int i;
    color[v] = GREY;
    for(i = 0 ; i < ne[v] ; i++ ) {
        int u = adj[v][i];
        if (color[u] == 0) {
            dfs_visit(u);
        }
    }
    printf("%d ", v);
    color[v] = BLACK;
}
/* tsort() - Вывести вершины графа в топологическом порядке */
void tsort() {
    int v;
    for(v = 0; v < n; v++)
        color[v] = WHITE;
    for(v = 0; v < n; v++)
        if (color[v] == WHITE)
            dfs_visit(v);
}
```

Задача C17.6. (Обход графа с использованием стека для хранения серых вершин) В про-

грамме, которая была написана в задаче С17.3, замените очередь на стек. Какой тип обхода получится в результате?

Задача С17.7. («Одежда профессора») Дан список элементов одежды профессора с указанием зависимостей вида «галстук должен быть одет после рубашки». Эта зависимость на входе записана в виде одной строки

```
галстук рубашка
```

Необходимо проверить, можно ли удовлетворить всем данным зависимостям и одеть профессора. В первой строчке входа задано число зависимостей E. Затем следует E строчек, в каждой из которых дана пара слов, означающих элементы одежды. Выход должен содержать слово «`YES`», если профессора можно одеть, а затем должны быть перечислены все элементы одежды в том порядке, в каком их следует одевать. Если одеть все элементы одежды нельзя, выведите одно слово «`NO`». Длина названий элементов одежды не более 30 символов. Всего возможных зависимостей не более 1000. Примечание: заведите глобальные переменные, в которых будет хранится словарь различных встретившихся названий элементов одежды и их идентификаторов. Идентификаторы элементов одежды должны быть числами от 0 до $n-1$, где n – число различных встретившихся элементов одежды. Реализуйте функцию `int getid(char *)`, которая возвращает идентификатор элемента одежды по данному названию. Если данное в аргументе название встречается первый раз, функция `getid` должна добавить его в словарь. Реализуйте эту функцию с помощью хэш-таблицы с открытой адресацией. Используйте в качестве «затравки» код 17.2 на странице 327. Добавьте в этот код проверку того, что данный ориентированный граф не содержит циклов, а именно, в функции `dfs_visit` в теле цикла проверяйте, что все вершины, смежные с вершиной `v`, имеют белый или чёрный цвет, но не серый. Если среди смежных вершин окажется серая (при движении мы наткнулись на собственный «хвост»), то в графе есть циклы, и все элементы одежды одеть невозможно.

Задача С17.8.* («Профессору холодно») Решите задачу С17.7, в которой в случае невозможности одеть все элементы одежды вычисляется максимальное число элементов одежды, которое можно одеть. Примечание: не существует алгоритма, который решает поставленную задачу за время $O(n^k)$ для любого k (задача полиномиально неразрешима). Предложите эвристический алгоритм, который решает данную задачу приближённо.

Задача С17.9. (рекурсивный обход и обход с использованием стека) Сравните два типа обхода графа — обход, реализованный в коде 17.2 и обход, описанный в задаче С17.6. Верно ли, что последовательности обхода, которые они дают, всегда совпадают? Обоснуйте ответ. Проведите численный эксперимент.

Лекция 18

«Жадные» алгоритмы на графах

Краткое описание: Метод «жадного выбора» широко применяется при решении задач на графах. Мы изучим классический алгоритм Дейкстры вычисления кратчайшего пути в графе, который допускает аналогию с «жадным алгоритмом» распространения света в пространстве. Мы также изучим два жадных алгоритма вычисления минимального остовного дерева — алгоритмы Крускала и Прима. Эти алгоритмы несут в себе важные идеи, которые применимы для приближённого решения ряда других задач, например, задачи коммивояжера и задачи кластеризации точек в пространстве.

Алгоритм Дейкстры поиска кратчайшего пути

В задаче «Лабиринт» мы использовали метод вычисления кратчайшего пути, основанный на понятии «фронт волны» — множество ячеек, до которых можно добраться за n шагов и нельзя добраться за меньшее количество шагов. Используя структуру данных «очередь» мы пробегали по текущему фронту волны и вычисляли следующий фронт волны (см. задачу С17.4 на стр. 326 и псевдокод 17.32 на стр. 316). Мы показали, что время работы этого алгоритма ограничено площадью лабиринта.

Лабиринт — это частный случай графа. Вершины графа соответствуют свободным ячейкам лабиринта, а рёбра соединяют соседние ячейки. Длина (вес) всех рёбер в этом графе одинакова и равна 1.

Рассмотрим задачу про кратчайший путь в графе, в котором ребра могут иметь различную длину (действительное положительное число).

Алгоритм 18.36 Алгоритм Дейкстры

```
1: комментарии:
   d, π — глобальные массивы, результат работы алгоритма:
     d[u] — длина кратчайшего пути из вершины s до вершины u,
     π[u] — предшественник вершины u в кратчайшем пути от s к u.
   V, E — множество вершин и рёбер графа G.
   w(u, v) — длина ребра e = (u, v), u, v ∈ V.
2: function DIJKSTRA(G, w, s)
3:     INITIALIZE-SINGLE-SOURCE(G, s)
4:     U ← ∅
5:     Q ← V
6:     while Q ≠ ∅ do
7:         u ← EXTRACT-MIN(Q)              ▷ Извлекаем вершину с минимальным d
8:         U ← U ∪ {u}                     ▷ Помещаем её в множество «зажжёных» вершин
9:         for всех вершин v ∈ Adj[u] do
10:            RELAX(u, v, w)              ▷ Обновляем значение d для всех смежных вершин
11:        end for
12:    end while
13: end function
14: function INITIALIZE-SINGLE-SOURCE(G, s)
15:    for всех вершин v ∈ V do
16:        d[v] ← ∞
17:        π[v] ← nil
18:    end for
19:    d[s] ← 0
20: end function
21: function RELAX(u, v)
22:    if d[v] > d[u] + w(u, v) then
23:        d[v] ← d[u] + w[u, v]            ▷ Помечаем, что в кратчайшем пути до v,
24:        π[v] ← u                        ▷ предпоследняя вершина есть u.
25:    end if
26: end function
```

Задача Л18.1. (Кратчайший путь) Дан ориентированный граф с взвешенными рёбрами и две выделенные вершины графа s и t. Найти путь из s в t, для которого сумма весов (длин) входящих в него рёбер минимальна (путь с минимальной суммарной длиной шагов).

Алгоритм Дейкстры, который решает эту задачу, основан на простой «жадной» стратегии. Пусть есть некоторое множество вершин U, до которых мы знаем как добраться из вершины s кратчайшим способом. Изначально это множество состоит только из одной вершины s, длина кратчайшего пути до которой равна 0. Опишем шаг добавления к этому множеству одной

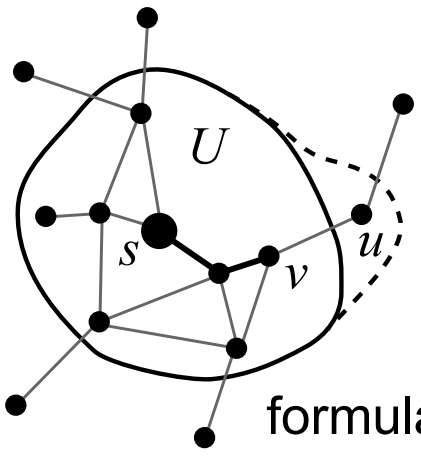


Рис. 18.1: Алгоритм Дейкстры: шаг дополнения множества вершин U.

вершины. Длины кратчайших путей до вершин множества U обозначим как $d(s,v)$, $v \in U$. Среди вершин, смежных с вершинами U, найдем такую вершину u ($u \notin U$), до которой можно добраться быстрее всего из вершины s, используя вершины U как транзитные, то есть найдём такую вершину v, для которой достигается минимум

$$\min_{v \in U,\ u \notin U} d(s,v) + w(v,u).$$

После этого обновим множество U: $U \leftarrow U \cup \{u\}$ и повторим шаг поиска вершины u. Будем осуществлять такие шаги, пока множество вершин U не станет равно связной компоненте вершины s.

Задача Л18.2. Используя метод математической индукции, покажите, что алгоритм Дейкстры находит длины кратчайших путей из вершины s до всех остальных вершин графа.

Можно провести аналогию между алгоритмом Дейкстры и явлением распространения света. Множество вершин U можно интерпретировать как множество точек неоднородного пространства, до которых «добрался» свет из точечного источника света s. Следуя принципу Гюйгенса, каждая точка, до которой «добрался» свет, сама становится источником света и испускает лучи в смежные части пространства. На каждом шаге алгоритм Дейкстры находит точку, которая «загорится» следующей.

Это лишь идея алгоритма Дейкстры. Точная формулировка алгоритма представлена в виде псевдокода 18.36. На первых шагах необходимо инициализировать массив длин кратчайших путей d. Изначально известно лишь, что расстояние от s до s равно нулю: $d[s] = 0$. Остальные расстояния устанавливаются равными бесконечности. Множество вершин U – множество вершин, до которых мы знаем длину кратчайшего пути из s, – изначально пусто, а множество остальных вершин Q совпадает с множеством всех вершин V. Затем (строки 6-12) начинается цикл извлечения из множества Q вершины u, с минимальным значением $d[u]$. В этом цикле поддерживается следующий *инвариант*:

$$U \cup Q = V.$$

Задача о минимальном покрывающем дереве

Прекрасной иллюстрацией применения жадных является задача о **задача о минимальном покрывающем дереве**. Рассмотрим для начала такую задачу:

Задача Л18.3. В некоторой стране имеется n городов. Требуется соединить все города телефонной связью так, чтобы суммарная длина всех проводов была минимальна, и чтобы двигаясь по проводам можно было добраться из любого города в любой другой. Каждый телефонный провод соединяет два города, то есть провода не могут разветвляться вне городов. Города считаются просто точками на плоскости. Для любой пары городов известна известна длина провода, которой потребуется на прокладку телефонного соединения между ними.

В терминах теории графов эта задача может быть сформулирована следующим образом:

Задача Л18.4. Имеется связный неориентированный граф $G(V,E)$. Для каждого ребра (u,v) задан неотрицательный вес $w(u,v)$ (длина провода). Требуется найти подмножество $T \subset E$,

связывающее все вершины графа и имеющее минимальный суммарный вес

$$w(T) = \sum_{(u,v)\in T} w(u,v).$$

Задачу можно переформулировать и таким образом — «выкинуть» из связного графа рёбра как можно большего суммарного веса, но сохранить при этом свойство связности.

В случае неотрицательных весов подмножество T можно считать деревом, так как, если у нас есть цикл, то из него можно просто удалить ребро, от этого связность графа не нарушится, а суммарный вес не увеличится. Связный граф без циклов и есть дерево.

Связный подграф G, являющийся деревом и покрывающий все вершины G, называется **покрывающим деревом** (или остовным деревом). Поэтому задача состоит в нахождении **покрывающего дерева с минимальным суммарнымо весом рёбер** (minimum-spanning-tree). Отметим ещё такой факт, что как и любое дерево, состоящее из n вершин, покрывающее дерево всегда содержит ровно $n-1$ ребро.

Задача Л18.5. Докажите, что любое дерево (связный граф без циклов) из n вершин имеет $n-1$ ребро.

Общая схема алгоритма

Здесь мы рассмотрим два алгоритма, решающих задачу о минимальном покрывающем дереве: алгоритмы Прима и Крускала. Оба алгоритма являются жадными, и они отличаются тем, что по разному осуществляют жадный выбор. Но общая схема этих алгоритмов будет следующей.

Искомое дерево строится постепенно. К изначально пустому множеству A на каждом шаге добавляется одно ребро. Причём это делается так, чтобы A всегда оставалось подмножеством некоторого минимального покрывающего дерева. То есть ребро (u,v) добавляется к A, если $A\cup\{(u,v)\}$ будет также подмножеством некоторого минимального остова. Будем называть такое ребро **безопасным**. Общая схема алгоритмов Прима и Крускала представлена в псевдокоде 18.37.

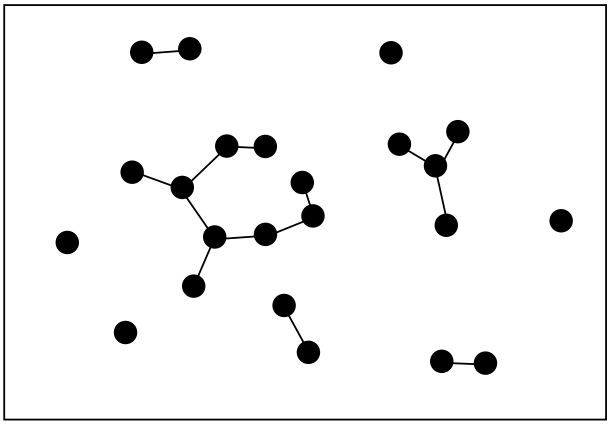
Алгоритм 18.37 Общая схема алгоритма MST

```
1: function MST-Generic(G, w)
2:     A ← ∅
3:     while A не является остовным деревом do
4:         найти безопасное ребро (u, v) для A
5:         A ← A ∪ {(u, v)}
6:     end while
7:     return A
8: end function
```

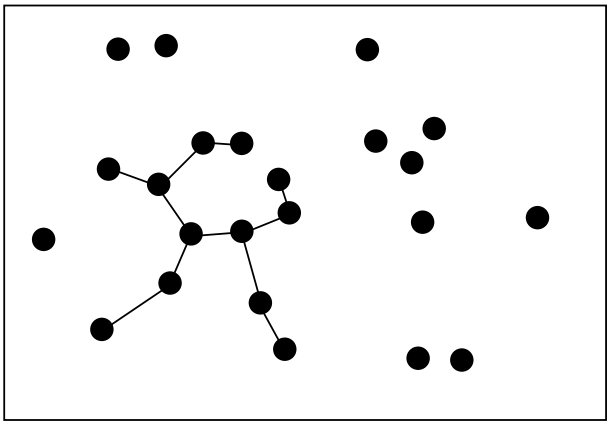
Если A является подмножеством некоторого минимального остовного дерева, то для него всегда существует безопасное ребро (любое ребро из минимального остова, не входящее в A является безопасным).

В процессе работы алгоритма, A всегда остаётся подмножеством некоторого минимального остова. Поэтому в конце работы алгоритма A и будет искомым минимальным покрывающим

деревом. Основной сложностью, конечно, здесь является поиск безопасного ребра. И рассматриваемые алгоритмы отличаются как раз тем, как они осуществляют этот поиск.



kruskal



prim

Рис. 18.2: Рост минимального покрывающего дерева в алгоритмах Крускала и Прима

(а) На каждом шаге алгоритма Крускала выбрано множество рёбер A, длина (вес) которых меньше некоторой фиксированой величины w. Но при этом некоторые рёбра с длиной (весом) менее w не выбраны, так как их добавление приводит к появлению циклов. Обычно во время работы алгоритма Крускала появляется несколько «ростков» (лес деревьев), которые постепенно «растут», присоединяя к себе вершины из окрестности, и «срастаются» друг с другом.

(б) В алгоритме Прима имеется только один «росток», который на каждом шаге присоединяет к себе вершину, присоединение которой «дешевле всего стоит». То есть ищется ребро с минимальным весом, которое соединяет одну из вершин «ростка» с одной из вершин, не принадлежащих ростку.

Алгоритм Крускала

В алгоритме Крускала множество выбранных рёбер A в любой момент представляет собой лес, состоящий (возможно) из нескольких связных компонент. И на каждом шаге добавляется ребро минимального веса, среди всех рёбер, концы которых принадлежат разным компонентам.

Здесь FindSet(u) возвращает главного представителя множества содержащего элемент u. Две вершины u и v принадлежат одному множеству, если FindSet(u) = FindSet(v). Объединение деревьев выполняется процедурой Union.

В начале работы алгоритма множество A пусто, и имеется $|V|$ деревьев, каждое из которых состоит из одной вершины. Далее рёбра сортируются по неубыванию весов (строка 6). Затем в таком порядке они просматриваются в цикле (строки 7-12). Для каждого ребра проверяется, лежат ли его концы в одном дереве. Если да, то ребро отбрасывается. В противном случае оно добавляется к A, и два соединённых им дерева объединяются в одно.

Можно так реализовать FindSet(u) и Union(u, v), что время работы алгоритма будет составлять $O(E \log E)$.

Алгоритм 18.38 Алгоритм Крускала

```
1: function MST-Kruskal(G, w)
2:     A ← ∅
3:     for каждой вершины v ∈ V[G] do
4:         MakeSet(v)
5:     end for
6:     Упорядочить рёбра E по весам
7:     for (u, v) ∈ E (в порядке возрастания веса) do
8:         if FindSet(u) ≠ FindSet(u) then
9:             A ← A ∪ (u, v)
10:            Union (u, v)
11:        end if
12:    end for
13:    return A
14: end function
```

Алгоритм Прима

В алгоритме Прима A всегда является некоторым деревом и на каждом шаге к нему добавляется ребро наименьшего веса среди рёбер, соединяющих вершины этого дерева с вершинами не из дерева.

Данный алгоритм получает на вход граф G, весовую функцию w и вершину r, которая будет корнем построенного остовного дерева. Алгоритм использует очередь с приоритетами Q. Приоритет вершины v определяется значением $key[v]$. Функция Extract-Min возвращает элемент с минимальным значением ключа и при этом удаляет его из кучи. Функция Decrease-Key изменяет значения ключа у одного из элементов кучи.

Кроме того здесь мы не храним явно множество A. Но при необходимости его всегда легко восстановить как

$$A = \{(v, \pi[v]) : v \in V \backslash \{r\} \backslash Q\}.$$

В конце работы алгоритма очередь пуста, поэтому множество

$$A = \{(v, \pi[v]) : v \in V \backslash \{r\}\}$$

есть множество рёбер минимального покрывающего дерева.

Время работы алгоритма Прима зависит от того как реализована очередь Q. Если реализовать её с помощью двоичной кучи, то на операцию Extract-Min требуется время $O(\log V)$. На операцию Decrease-Key также потребуется $O(\log V)$. Цикл $While$ выполнится $|V|$ раз. Цикл for выполнится $O(E)$ раз, так как сумма степеней всех вершин равно $2|E|$. Таким образом получаем, что сложность алгоритма будет $O(V \log V + E \log V) = O(E \log V)$. Используя фибоначчиевы кучи эту оценку можно улучшить до $O(E + V \log V)$.

Осталось проверить, что алгоритм Прима действительно решает поставленную задачу.

Теорема 18.1. *Алгоритм Прима правильно находит минимальное покрывающее дерево для неориентированного графа G.*

Доказательство. Нам достаточно показать, что ребро выбираемое на шаге 9, всегда будет безопасным. Пусть у нас уже построено дерево A, которое содержится в минимальном остове

Алгоритм 18.39 Алгоритм Прима

```
1: function MST-Prim(G, w. r)
2:     A ← ∅
3:     for каждой вершины u ∈ Q do
4:         key[u] ← ∞
5:     end for
6:     key[r] ← 0
7:     π[r] ← NIL
8:     while Q ≠ ∅ do
9:         u ← Extract-Min(Q)
10:        for каждой вершины v ∈ Adj[u] do
11:            if v ∈ Q и w(u, v) < key[v] then
12:                π[v] ← u
13:                key[v] ← w(u, v)
14:                Decrease-Key(Q, v)
15:            end if
16:        end for
17:    end while
18:    return A
19: end function
```

T и пусть на 9 шаге выбрано ребро (u, v) ($u \in A$, $v \notin A$). Предположим, что T не содержит это ребро, поскольку в противном случае доказываемое утверждение очевидно. Так как T покрывающее дерево, то существует путь из u в v по рёбрам этого дерева. Кроме того, так как $u \in A$, $v \notin A$, то на этом пути найдётся ребро $(x, y) \in T$, такое что $x \in A$, $y \notin A$.

Удалим теперь из T ребро (x, y) и добавим (u, v). Получим новое покрывающее дерево

$$T' = (T \backslash \{(x, y)\}) \cup \{(u, v)\}.$$

Из условия выбора (u, v) следует, что $w(u, v) \leqslant w(x, y)$. Отсюда следует, что $w(T') \leqslant w(T)$. Это означает, что T' также будет минимальным покрывающим деревом. $A \cup \{(u, v)\} \subset T'$ поэтому ребро (u, v) будет безопасным. □

Задача Л18.6. Пусть (u, v) — ребро минимального веса. Покажите, что оно принадлежит некоторому минимальному остову.

Задача Л18.7. Пусть граф $G = (V, E)$ задан матрицей смежности. Постройте простую реализацию алгоритма Прима, время работы которой $O(V^2)$.

Задача Л18.8. Докажите, что алгоритм Крускала правильно находит минимальное покрывающее дерево.

Семинар 18

«Жадные» алгоритмы на графах

Алгоритм Дейкстры поиска кратчайшего пути

Задача С18.1. Изучите код 18.1, реализующий алгоритм Дейкстры, представленный в виде псевдокода 18.36 на стр. 330. На основе этого кода напишите программу `dijkstra`, которая считывает описание взвешенного графа, идентификатор вершины `s` и выводит значения массива `d`. Создайте систему тестов — около 20 файлов, содержащих описания взвешенных графов с числом вершин от 5 до 500, и напишите программу (скрипт) которая запускает программу вычисления кратчайшего пути для всех тестов[1].

Программа 18.1: Алгоритм Дейкстры поиска кратчайших путей из одной вершины

```
typedef enum { REACHED, NOTREACHED } colors_t;
typedef struct edge {
    int v;
    double length;
} edge_t;
typedef struct {
    int V, E;     // число вершин и число рёбер графа;
    edge_t **adj; // adj[u] = массив ребер исходящих из u;
    int *ne;      // ne[u] = размер массива вершин смежных с u
    int *pi;      // pi[u] = вершина из которой мы "шагнули"
                  //         в вершину u;
    color_t *color; // color[u] = REACHED если вершина лежит в U,
                    //          = NOTREACHED - иначе;
    double *d;    // d[u] = длина кратчайшего пути до вершины u.
} graph_t;

// Множество "зажжённых" вершин U -- это множство
// таких v, для которых color[v] == REACHED;
void diikstra(graph_t *g, int s) {
    int u, v;
    // Инициализация начальных значений d, pi и color
    for(v = 0 ; v < g->V; v++) {
        g->pi[v]    = -1;
```

[1]На языке `bash` запустить программу `dijkstra` на серии тестов (файлов), которые хранятся в директории `tests`, можно с помощью следующей команды:
`for t in tests/*; do ./dijkstra <$t > $t.result; done; .`

```
        g->d[v]     = 1E+100; //infinity
        g->color[v] = NOTREACHED;
    }
    g->d[s] = 0;
    // Цикл расширения множества вершин U
    while(extract_min(g, &u)) {
        int i;
        g->color[u] = REACHED; // U <- U \/ {u}
        for(i = 0 ; i < g->ne[u]; i++) {
            edge_t e = g->adj[u][i];
            if(g->d[e.v] > e.length + g->d[u]) {
                g->d[e.v] = e.length + g->d[u];
                g->pi[e.v] = u;
            }
        }
    }
}
int
extract_min(graph_t *g, int *u) {
    int v, v_min = -1;
    double d_min = 2E+100;
    for(v = 0 ; v < g->V; v++) {
        if(g->color[v] == NOTREACHED && g->d[v] < d_min) {
            d_min = g->d[v];
            v_min = v;
        }
    }
    if(v_min == -1) {
        return 0;
    } else {
        *u = v_min;
        return 1;
    }
}
```

Задача С18.2. Алгоритм Флойда-Уоршолла позволяет найти кратчайшие пути для всех пар вершин. Он работает время пропорциональное кубу от числа вершин – $O(|V|^3)$, в то время как алгоритм Дейкстры работает время, пропорциональное квадрату числа вершин – $O(|V|^2)$. Реализация алгоритма Флойда-Уоршолла выглядит очень просто. Предположим, что число вершин равно `n` и двумерный массив `d[n][n]` содержит длины рёбер. А именно, элемент `d[i][j]` равен длине ребра между `i`-й и `j`-й вершиной, если такое ребро есть, либо очень большому числу, условно соответствующему бесконечности, если ребра нет. Тогда после выполнения строчек, представленных в коде 18.2, массив `d` будет содержать уже длины кратчайших путей. Напишите программу `floyd.c`, которая считывает описание взвешенного графа и выводит таблицу значений `d[i][j]` длин кратчайших путей. Для тестовых взвешенных графов, подготовленых при решении задачи С18.1, запустите программу `floyd` и проверьте, что значения

`d[s][j]` в алгоритме Флойда-Уоршалла совпадают со значениями `d[j]` в алгоритме Дейкстры для всех значениях `j`, `0 <= j < n`. Напишите программу на языке Си или на языке bash, которая запускает программы `floyd` и `dijkstra` и проверяет, что массив расстояний, возвращаемый программой `dijkstra` совпадает со строкой `s` в таблице, возвращаемой программой `floyd`.

Программа 18.2: Алгоритм Флойда-Уоршолла вычисления длин кратчайших путей для всех пар вершин

```
for(k = 0 ; k < n; k++)
    for(i = 0 ; i < n; i++)
        for(j = 0 ; j < n; j++)
            if(d[i][j] > d[i][k]+d[k][j])
                d[i][j] = d[i][k]+d[k][j];
```

Задача о минимальном покрывающем дереве

Минимальное покрывающее дерево графа $G = (V, E)$ — это задача о поиске подмножества ребер $T \subseteq E$ связного графа G, которые обеспечивают связность графа и, в то же время, имеют минимальный суммарный вес (см. задачу Л18.4).

Граф $G' = (V, T)$ получается деревом, множество вершин которого совпадает с множеством вершин исходного графа, а множество рёбер является подмножеством рёбер исходного графа.

Алгоритм Крускала — один из простых алгоритмов, позволяющих решить эту задачу. Он основан на жадной стратегии — рёбра графа перебираются в порядке возрастания веса и добавляются в искомый остов графа, если их добавление не приводит к появлению циклов. Во время работы алгоритма получается множество деревьев («ростков»), которые постепенно растут и срастаются друг с другом.

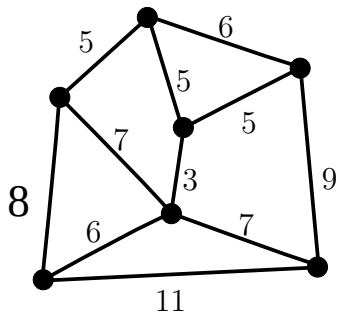


Рис. 18.1: Пример графа с взвешенными рёбрами.

Когда рассматривается очередное по весу ребро, необходимо проверить, приведёт ли её добавление к появлению циклов. Если концы ребра принадлежать разным росткам, то ребро можно добавлять в остов. Если же концы ребра принадлежат одному ростку, то его нужно пропустить, и не добавлять в остов.

Необходимо хранить следующую структуру — разложение множества вершин V на непересекающие подмножества $V_1, \ldots, V_n$, $\bigcup V_i = V$. При этом необходимо быстро получать ответ на вопрос о том, принадлежат данные две вершины одному и тому же множеству или разным, а также уметь быстро осуществлять операцию слияния двух подмножеств в одно.

В приведённом коде 18.3 используется следующая идея. Каждому подмножеству вершин (ростку) V_i ставится в соответствие один из его представителей, который называется главным представителем. Вводится специальная связь «родитель», и каждое подмножество V_i представляет собой дерево по этой связи. В корне этого дерева находится главный представитель множества V_i — прародитель всех этого множества.

Таким образом, чтобы найти главного представителя множества, которому принадлежит вершина u, необходимо двигаться по связи «родитель» и найти прародителя вершины u. Это

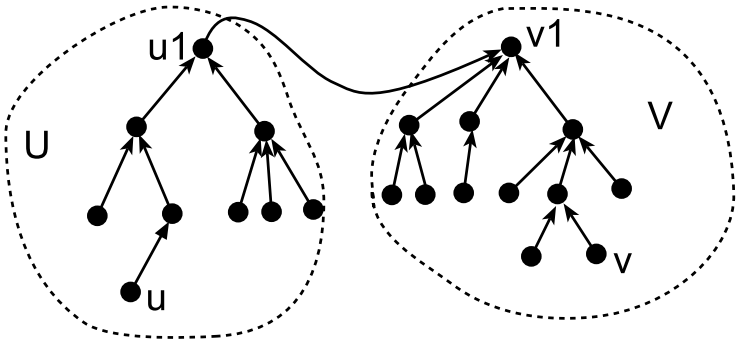


Рис. 18.2: Представление системы множеств в виде деревьев
Вершина u' является прародителем вершины u.
Вершина v' является прародителем вершины v.
Операция объединения множеств V и U заключается в создании связи «родитель» от корня одного u' к корню дерева v'. В результате получается дерево с одним корнем v'.

реализовано в функции `set_find(parent, u)`. Она находит главного представителя подмножества (ростка), которому принадлежит `u` — рекурсивно определяет прародителя, у которого нет родителей. Две вершины `u` и `v` принадлежат одному ростку, если верно тождество `set_find(parent, u) == set_find(parent, v)`. Связь «родитель» задается одним массивом `parent`. Элемент `parent[u]` этого массива равен родителю элемента `u`. Считаем, что элемент является корнем дерева, если выполнено равенство `parent[u]==u`.

Задача С18.3. Изучите код 18.3, реализующий алгоритм Крускала вычисления минимального покрывающего дерева. Нарисуйте последовательные состояние леса деревьев (ростков) в графе вершин со со связями родитель-ребёнок для графа, показанного на рисунке 18.1.

Программа 18.3: Алгоритм Крускала вычисления минимального покрывающего дерева

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
typedef struct {
    int from, to;  // начальная и конечная вершины ребра
    double w;      //  вес (длина) ребра
} edge_t;
// функция сравнения ребер по весу
int cmp(edge_t *a, edge_t *b) {
    if(a->w > b->w ) return 1;
    if(a->w < b->w ) return -1;
    return 0;
}
// Вершина C, не имеющая "родителя" (parent[C] == C), является
// идентификатором (цветом) множества всех своих "потомков".

// Определение пра-пра...родителя (идентификатора множества) вершины node.
int set_find(int *parent, int node) {
    if(parent[node] == node) {
        return node;
    } else {
        int c = find_set(parent, parent[node]);
```

```
        parent[node] = c;
        return c;
    }
}
// Объединение двух множеств, котрым принадлежат вершины u и v
// сводится к вычислению вершин деревьев, к которым принадлежат
// u и v, и прикреплении вершины одного дерева к вершине другого
int set_union(int *parent, int u, int v) {
    u = find_set(parent, u);
    v = find_set(parent, v);
    parent[u] = v;
}
void take_edge(edge_t e) {
    printf("Используем ребро %d->%d, вес=%lf\n", e.from, e.to, e.w);
}
int main() {
    int i, component_count;
    int edge_count, node_count; // число рёбер и число вершин
    edge_t *E;              // массив рёбер
    int *parent;            // массив идентификаторов "родителей" вершин
    double W = 0;           // вес покрывающего дерева

    scanf("%d%d", &node_count, &edge_count);
    E = (edge_t*) malloc(edge_count * sizeof(edge_t));
    parent = (int*) malloc(node_count * sizeof(int));
    for(i = 0; i < edge_count ; i++) {
        scanf("%d%d%lf", &E[i].from, &E[i].to, &E[i].w);
    }

    // Отсортируем рёбра по весу.
    qsort(E, edge_count, sizeof(edge_t),
              (int (*)(const void*, const void*)) cmp);

    // Инициализируем массив parent.
    // Изначально имеем множество одноэлементных множеств.
    for(i = 0; i < node_count ; i++)  parent[i] = i;

    component_count = node_count;
    for(i = 0;  i < edge_count;  i++) {
        if( find_set(parent,E[i].from) == find_set(parent,E[i].to) ){
             // Ребро соединяет вершины из одного подмножества.
             continue; // Пропускаем его.
        } else { // Объединяем подмножества, которым принадлежат концы ребра.
            set_union(parent, E[i].from, E[i].to );
            component_count--;
            take_edge(E[i]);
```

```
            W += E[i].w;
        }
    }
    if(component_count != 1)
        fprintf(stderr, "warning: Граф не односвязный.\n");
    printf("Вес минимального покрывающего дерева = %lf.\n", W);
    free(parent); free(E);
    return 0;
}
```

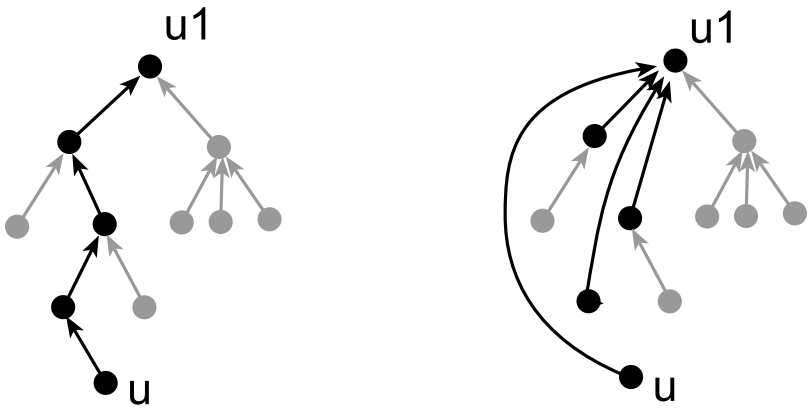


Рис. 18.3: Метод «сжатия» путей в алгоритме Крускала.
При определении главного представителя множества U, которому принадлежит u, в функции `set_find` осуществляется рекурсивное перемещение по цепочке $u \to$ `parent[`u`]` $\to$ `parent[parent[`u`]]` $\to \ldots \to u'$ к родителю, прародителю и т.д. до элемента, у которого нет родителя. После этого осуществляется сжатие этой цепочки, а именно все стрелки «`parent`» для элементов этой цепочки устанавливают указывающими на u' — главного представителя множества U.

Интересная оптимизация заключается в том, что после прохождения по цепочке

$$\texttt{u} \to \texttt{parent[u]} \to \texttt{parent[parent[u]]} \to \ldots$$

прародителей вершины `u` можно обновить значения элементов массива `parent` и для всех элементов цепочки в качестве родителя указать вершину дерева — главного представителя ростка. Это позволит в следующий раз для этих вершин не двигаться по длинной цепочке связей «родитель» и сразу попасть в корень дерева. Эта оптимизация называется **методом сжатия путей** и достигается с помощью одной строки «`parent[u] = c;`» в функции `set_find`.

Более простая реализация системы непересекающихся подмножеств выглядит следующим образом. Заведём массив `color` в котором элемент `color[u]` равен идентификатору (цвету) подмножества. Изначально `color[u]=u`, то есть каждая вершина имеет свой уникальный «цвет» — всё множество вершин разбито на одноэлементные подмножества. Реализации функций `set_find` и `set_union` выглядели бы следующим образом.

```
int set_find(int *color, int u) {
    return color[u];
}
```

```
int set_union(int *color, int u, int v) {
    int cu = color[u];
    int cv = color[v];
    if (cv != cu) {
        int i;
        for(i = 0 ; i < node_count; i++) {
            if (color[i] == cv) {
                color[i] = cu;
            }
        }
    }
    return cu;
}
```

В этой реализации функция `set_find` определения «цвета» элемента вычисляется быстро (за $O(1)$). Зато на операцию слияния двух множеств мы тратим время $O(|V|)$, где $|V| =$ `node_count` — множество вершин графа. Действительно, при слиянии двух множеств в одно, необходимо найти все вершины второго множества и перекрасить их в цвет вершин первого множества. В представленной реализации алгоритма Крускала ситуация обратная — процедура слияния двух подмножеств выполняется быстро (за $O(1)$), а в функции определении «цвета» элемента осуществляются сложные действия. Амортизационно реализация системы непересекающихся множеств с помощью деревьев с сжатием путей оказывается более эффективным и суммарное время работы функций `set_find` и `set_union` в алгоритме Крускала не превосходит $O(|V| \log |V|)$.

Задача С18.4. Используя численный эксперимент, определите асимптотику роста времени работы алгоритма Крускала в зависимости от размера графа. Для тестирования используйте случайные полные графы, вершины которых являются случайными точками на плоскости (равномерно распределёнными в круге или по нормальному закону вокруг некоторого центра), а вес ребра равен евклидову расстоянию между точками, которые соответствуют концам ребра. Изучите два случая — с методом сжатия путей и без (закомментируйте одну строчку в функции `set_find` программы 18.3, представленной на стр. 339).

Лекция 19

Структуры данных: двоичная куча

Интерфейс «очередь с приоритетом»

Интерфейс «очередь с приоритетом» (priority queue, PQ) — это абстрактный интерфейс хранилища записей (ключ, значение) с двумя функциями:

Insert — добавить запись (key, value);

Extract-Minimum — извлечь запись (key, value) с минимальным значением ключа.

В стандартных реализациях очереди с приоритетами время выполнения обеих функций есть $O(\log n)$, где n — число хранимых записей. Это, в частности, позволяет на основе очереди с приоритетами осуществить сортировку элементов, которая будет работать и в среднем и в худшем случае время $O(n \log n)$. С помощью очереди с приоритетами можно эффективно решать задачи, подобные следующей:

Задача Л19.1. На вход поступает поток действительных чисел (порядка 10^9 штук). Необходимо периодически быстро отвечать на запрос: вывести K минимальных чисел из всех поступивших на вход ($K \approx 1000$).

Кроме того, очередь с приоритетами возникает в целом ряде известных задач и алгоритмов:

- Алгоритм Дейкстры поиска кратчайших путей в графе из данной вершины. В этом алгоритме в очереди с приоритетом естественно хранить множество вершин, до которых ещё «не добрался луч света», а значением ключа сделать длину известного на данный момент пути до вершины;
- Алгоритм Прима вычисления минимального покрывающего (остовного) дерева, который подобен алгоритму Дейкстры.
- Планировщик задач — организация переключений между процессами в операционных системах[1]. В этой задаче роль ключей играет время бездействия процесса, либо какая-нибудь сложная функция от времени бездействия и приоритета процесса.
- Алгоритм построения сжимающих кодов Хаффмана.

Простейшая, но не самая эффективная реализация очереди с приоритетами, основана на списке, в котором хранятся упорядоченные по возрастанию ключей записи. В этом случае реализация функции Extract-Minimum проста — необходимо просто извлечь (удалить) из списка первый элемент и вернуть его в качестве результата. Операция добавления несколько сложнее — необходимо двигаться по списку и искать для добавляемой записи подходящее место в отсортированном списке. Возможно, придётся дойти до конца списка. Среднее время работы функции добавления равно $O(n)$, где n — количество хранимых записей.

Двоичная куча

Структура «двочная куча» (binary heap) является простейшей реализацией очереди с приоритетами и функции Insert и Extract-Minimum выполняет за время $O(\log n)$.

Определение 19.1. **Структура данных «двоичная куча»** — *это структура данных «двоичное дерево», в узлах которой хранятся записи* $(key, value)$*, причём, для каждого узла выполнено* **свойство кучи***: ключи детей больше либо равны ключу родителя. Более того, обычно предполагается, что двоичное дерево хранится просто в виде массива, в котором у элемента с индексом* i *детьми считаются элементы с индексами* $2i+1$ *и* $2i+2$ *(индексы начинаются с* 0*). Соответственно, у элемента с индексом* i *индекс родителя вычисляется по формуле* $(i-1)/2$ *(деление нацело без остатка).*

На рисунке 19.1 показан пример двоичной кучи из 12 элементов. В узлах указаны их ключи, а рядом с узлами — индексы в массиве, которые им соответствуют.

Реализация двоичного дерева в виде массива сильно упрощает операции с ним. Но надо понимать, что если дерево слабо ветвится, то в массиве будут множество незаполненных (незадействованных) ячеек. Но в нашем случае дерево будет максимально заполненным и представлять собой непрерывный массив без пустот. Все n записей будут идти подряд и занимать в массиве первые n ячеек.

Итак, опишем функции Heap-Insert и Heap-Extract-Minimum.

[1]Поясним суть задачи, решаемой планировщиком задач. У большинства современных компьютеров есть только одно центральное вычисляющее устройство — процессор. Бывают двухпроцессорные и многопроцессорные машины. В отдельный момент времени один процессор может выполнять только одну программу. Если необходимо *одновременно* выполнять несколько программ (слушать музыку, работать с интернет-браузером, копировать файлы из сети, обслуживать сетевые соединения и т.д), то необходимо идти на специальные ухищрения. Архитектура современных операционных систем такова, что изначально ориентирована на выполнение многих задач, операционные системы многозадачны, и число одновременно выполняемых задач может быть больше тысячи. Слово «одновременно» не совсем отражает действительность. В действительности, в операционной системе присутствует *планировщик задач*, который распределяет процессорное время между выполняемыми задачами (процессами). Большую часть времени задачи (процессы) не выполняются, а ждут, когда придёт их черёд. В планировщиках задач часто используется система приоритетов. Задачам назначается приоритет, и планировщик задач наиболее приоритетные задачи чаще, вне очереди, «подпускает» к процессору, а менее приоритетные — реже. В современных операционных системах планировщики задач много сложнее, чем описываемая здесь очередь с приоритетами. Тем не менее, базовая идея алгоритмов планирования часто основана на очереди с приоритетами и её реализации с помощью двоичной кучи.

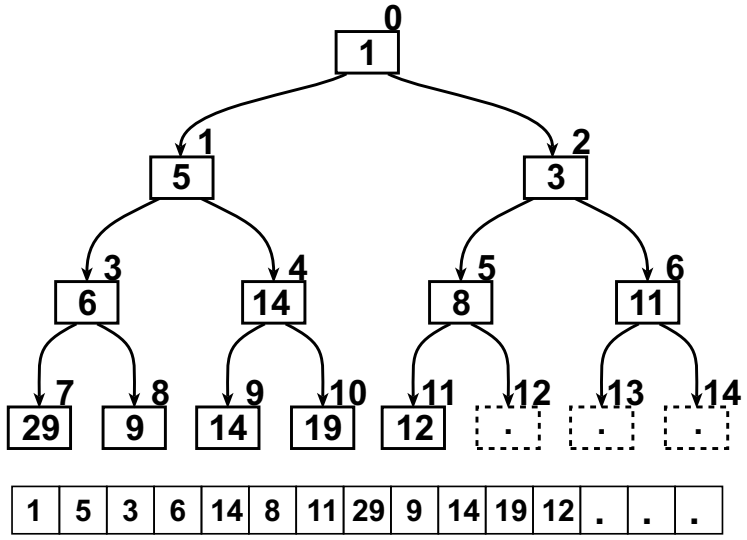


Рис. 19.1: Двоичная куча из 12 элементов. Ключи не убывают при движении от корня к листьям. Справа от каждого узла дерева указан его индекс в массиве. Снизу показано представление бинарной кучи в памяти компьютера в виде массива.

Во-первых, заметим, что в корне дерева, которое в данном случае называется **вершиной кучи**, хранится запись с минимальным ключём, так как у её детей ключи не меньше, и у детей детей ключи тоже не меньше и т.д.

Записи в двоичной куче упорядочены по ключам сверху вниз, а именно, при движении от вершины кучи вниз к любому из листьев значения ключей не уменьшаются. Но при этом отношение двух ключей из разных поддеревьев может быть любым. В частности, ничего нельзя сказать про порядок записей, находящихся на одном уровне глубины дерева.

Пусть в куче хранится n записей, которые хранятся в массиве a: $a[0], a[1], \ldots, a[n-1]$. Функция Heap-Add$(key, value)$ устроена следующим образом. Добавим запись $x = (key, value)$ в конец массива: $a[n] \leftarrow (key, value)$. Посмотрим на родителя элемента $a[n]$ (это элемент с индексом $p = (n-1)/2$) и проверим, выполнено ли свойство кучи — ключ родителя должен быть не больше ключа его ребёнка, то есть должно быть выполнено условие $a[p].key \leqslant a[n].key$. Если оно выполнено, то процесс добавления закончен. Если нарушено, то просто переставим местами элементы $a[n]$ и $a[p]$, и будем проверять, не нарушено ли свойство кучи для элемента $a[p]$, куда переместилась добавляемая запись. Свойство кучи может быть, по-прежнему, нарушено и снова требуется меняться местами с родителем. Этот процесс называется «всплытием». Новая запись добавляется в конец массива, а потом происходит серия обмен местами элементов массива, вызывающих перемещение этой записи по дереву вверх до подходящего места в вертикальной цепочке от листа до корня дерева. Если в добавляемой записи ключ самый маленький, то она «всплывёт» до самого верха и окажется в вершине кучи. В конце следует увеличить размер кучи на 1: $n \leftarrow n + 1$

Число операций обмена элементов во время «всплытия» ограничено сверху высотой дерева. Несложно увидеть, что высота дерева в точности равна $\lfloor \log_2 n + 1 \rfloor$ (целой части $\log_2(n+1)$). Таким образом, время выполнения функции добавления есть $O(\log n)$.

Функция Heap-Extract-Minimum должна просто вернуть запись (пару $(key, value)$), хранящуюся в нулевом элементе массива: $result \leftarrow a[0]$. Но функция Heap-Extract-Minimum должна также удалить его из кучи. После его удаления появится дырка в «массиве», кото-

рую нужно «заделать». Сделаем это следующим образом — переместим последний элемент $x = a[n-1]$ массива на место удалённого элемента: $a[0] \leftarrow a[n-1]$. Свойство кучи в результате будет, скорее всего, нарушено — в вершине будет находится запись x с ключём большим, чем ключ одного из его детей (или обоих). Начнём «топить» элемент x, пока он не займет надлежащего места по вертикали. Для начала посмотрим на детей $a[1]$ и $a[2]$, выберем из них того, у которого наименьший ключ, и поменяемся с ним местами. Таким образом, запись x спустится на один уровень дерева вниз. Для ячейки массива, в которую переместилась запись x, снова проверим свойство кучи. Оно по-прежнему может быть нарушено, и, возможно, потребуется дальнейший спуск. Снова посмотрим, есть ли дети у элемента, в который переместилась запись x (индексы детей известны, и если индекс ребёнка больше либо равен n, то ребёнка нет), посмотрим какие у них ключи, и выполнено ли свойство кучи. Если свойство кучи нарушено, поменяем местами x и наименьшего из детей (запись x и запись ребёнка с наименьшим ключём). Так будем «топить» запись x, пока есть дети и свойство кучи не выполнено. Возможно, придётся дойти до самого низа кучи и проделать h операций обмена, где h — высота кучи. Рекурсивная функция «топления» вершины приведена в псевдокоде 19.40.

Алгоритм 19.40 Рекурсивная функция «топления» вершины

```
function Heapify(массив a, индекс i)
    l ← 2i + 1
    r ← 2i + 2
    if l < a.length then
        if r < a.length и a[l].key > a[r].key then
            k ← l
        else
            k ← r
        end if
        if a[k].key < a[i].key then
            Swap(a[i], a[k])
            Heapify(a, k)
        end if
    end if
end function
```

Время выполнения функции Heap-Extract-Minimim также ограничено высотой кучи, то есть $O(\log n)$.

Итак, обе функции – Heap-Insert и Heap-Extract-Minimum – выполняются за время $O(\log n)$ в худшем случае.

Алгоритм сортировки HeapSort

Алгоритм сортировки, основанный на очереди с приоритетами, прост. Возьмём реализацию очереди с приоритетами, в которой обе функции выполняются за время $O(\log n)$, например, двоичную кучу.

ШАГ 1: Добавим последовательно все элементы, которые нам нужно отсортировать, как ключи в очередь с приоритетами. Никаких значений к этим ключам не будем привязывать — поле *value* просто исключим из записей.

ШАГ 2: Извлечём последовательно все элементы с помощью функции Extract-Minimum. Ясно, что они будут извлекаться в порядке возрастания.

И первый и второй шаг представляют собой циклы, в которых n раз выполняется одно действие с очередью. Время выполнения этого действия ограничено $O(\log n)$. Поэтому суммарно на каждый шаг тратится время $O(n \log n)$, а значит и на два шага тратится время $O(n \log n)$.

Функция Build-Heap

Пусть дан массив записей $(key, value)$. Двоичную кучу набора записей можно получить, добавляя их последовательно в пустую (изначально) кучу с помощью функции Insert. Время работы этого алгоритма $O(n \log n)$. Есть более быстрый алгоритм. Функция Build-Heap превращает массив в двоичную кучу за $O(n)$.

Алгоритм 19.41 Функция Build-Heap, «превращающая» массив в двоичную кучу

```
function BUILD-HEAP(массив a)
    i ← (n − 1)/2
    for i ← n/2 downto 0 do
        HEAPIFY(a, i)
    end for
end function
```

Докажите самостоятельно, что этот алгоритм работает линейное по размеру массива время.

Семинар 19

Структуры данных: двоичная куча

Реализация двоичной кучи

Задача С19.1. В коде 19.1 показан заголовочный файл структуры данных «двоичная куча». Реализуйте интерфейс двоичной кучи, заданный в этом файле. Используйте фрагменты, представленные в коде 19.2. Модифицируйте представленную в фрагментах функцию `bheap_insert` так, чтобы куча автоматически подстраивала размер динамически выделенного массива под хранимые записи (чтобы ни пользователю бинарной кучи, ни программисту не нужно было гадать, каков будет размер кучи). Для этого используйте функцию `realloc`.

Программа 19.1: Интерфейс двойчной кучи

```
typedef struct {
   key_t  key;
   value_t value;
} pair_t ;

typedef struct {
    pair_t *data;
    unsigned size;
    unsigned data_size;
} bheap_t;

bheap_t* bheap_new    ( int initial_data_size );
void     bheap_delete ( bheap_t *h );
void     bheap_insert ( bheap_t *h, pair_t v );
int      bheap_extract_min ( bheap_t *h, pair_t *v );
```

Задача С19.2. В коде 19.2 функции `heapify_down` и `heapify_up` реализованы методом итераций. Перепишите их, использовав метод рекурсии.

Программа 19.2: Фрагменты реализации двоичной кучи

```
#include "bheap.h"
/* Объявление функций, используемых для внутренних нужд */
void  bheap_heapify_up   ( bheap_t *h, unsigned int c );
void  bheap_heapify_down ( bheap_t *h, unsigned int p );


/* Реализации функций */
```

```
void
bheap_insert( bheap_t *h, pair_t v ) {
    h->data[ (h->size)++ ] = v ;
    bheap_checkup( h, h->size - 1 );
}
int
bheap_extract_min( bheap_t *h, pair_t *v ) {
    if( h->size == 0 ) return 0;
    *v = h->data[0];
    h->data[0] = h->data[ --(h->size) ];
    bheap_heapify_down( h, 0 );
    return 1;
}
void
bheap_heapify_up( bheap_t *h, int c ) {
    int p;
    for( p = (c-1) / 2;  p > 0 ; c = p , p = (c-1) / 2 ) {
        if( h->data[p].key > h->data[c].key ) {
            pair_t t = h->data[p];
            h->data[p] = h->data[c]; h->data[c] = t;
        } else {
            break;
        }
    }
}
void
bheap_heapify_down( bheap_t *h, int p ) {
    int c;
    for( c = 2 * p + 1 ;  c < h->size  ; p = c, c = 2 * p + 1 ) {
        if( c + 1 < h->size && h->data[c + 1].key < h->data[c].key ) c++;
        if( h->data[c].key < h->data[p].key ) {
            pair_t  t =  h->data[c];
            h->data[c] = h->data[p]; h->data[p] = t;
        } else {
            break;
        }
    }
}
```

Сортировка методом двоичной кучи

Задача С19.3. Проверьте как работает сортировка, основанная на двоичной куче (см. код 19.3), Оформите отдельно функцию сортировки целых чисел с сигнатурой

```
void bheap_sort(int *a, int size);
```

Проведите массовое тестирование функции `bheap_sort`: сгенерите несколько больших массивов (размера 10^6 и больше) с различной степенью упорядоченности элементов (упорядоченный массив, упорядоченный в обратном порядке, массив случайных чисел, массив содержащий упорядоченные кусочки и т.п.) и отсортируйте их с помощью `bheap_sort`. Сверяйте результат с верным результатом, получаемым с помощью функции `qsort`. Сравните времена работы функций `qsort` и `bheap_sort` на случайном массиве 500000 чисел. Объясните результаты сравнения.

Программа 19.3: Сортировка на основе двоичной кучи

```
#include <stdio.h>
#include "bheap.h"

int main() {
    bheap_t *h;
    int n, i;
    pair_t v = {0, 0};

    scanf("%u", &n);
    h = bheap_new(n);
    for( i = 0; i < n; i++ ){
        scanf("%d", &v.key);
        v.value = i;
        bheap_insert(h, v);
    }

    while( bheap_extract_min(h, &v) ) {
        printf("%d ", v.key);
    }
    printf("\n");
    bheap_delete( h );
    return 0;
}
```

Функция построения кучи

Задача С19.4. Известно, что функция сортировки n чисел в среднем занимает время $\Theta(n \log n)$. Двоичная куча представляет собой более слабую упорядоченность. Ключи не убывают при движении вверх от детей к и родителям, то есть упорядочены лишь цепочки вида

$$a[n_0] \geqslant a[n_1] \geqslant a[n_2] \geqslant a[n_3] \geqslant \ldots \geqslant a[0],$$

где

$$n_1 = (n_0 - 1)/2,\ n_2 = (n_1 - 1)/2,\ n_3 = (n_2 - 1)/2,\ \ldots$$

Оказывается, есть алгоритм превращения массива объектов в двоичную кучу за время $O(n)$. Он представляет собой цикл по элементам массива от $a[n/2]$ до $a[0]$ в котором вызывается рекурсивная функция «топления элемента» `heapify_down`. Докажите, что описанный

алгоритм любой массив «превращает» в двоичную кучу. Реализуйте его в виде функции `build_heap(int *a, unsigned size)`.

Реализация очереди с приоритетами с помощью дерева поиска

Задача С19.5. Реализуйте очередь с приоритетами на основе одного из сбалансированных деревьев поиска и функции извлечения самого левого элемента дерева. Оцените времена выполнения функций `insert` и `extract_min`.

Задача С19.6. Расширьте интерфейс очереди с приоритетом функциями удаления записи с заданным ключём `delete_key(key_t key)` и изменением ключа записи `change_key(key_t key, value_t value, key_t newkey)` и реализуйте его. Считайте, что ключи в данном случае могут повторятся, а поля `value` уникальны для каждой записи.

Часть IV

Дополнительные материалы

Лекция 20

Алгебра булевых функций

Краткое описание: Материал этой лекции следует рассматривать как дополнительный. Здесь затрагиваются такие важные понятия как **выразимость** и **полнота**. Выбирая какой-либо набор операций (какую либо формальную систему вычисления или описания), мы получаем набор объектов, которые можно описать (реализовать) с помощью этого набора операций (системы). Системы, с помощью которых можно описать (реализовать) все возможные объекты, называются полными. Так среди множества исполнителей машина Тьюринга является полной формальной системой (на машинах Тьюринга можно реализовать все вычислимые функции), а конечные автоматы — не полная система. В этой лекции понятия полноты и выразимости продемонстрированы на примере булевых функций.

На лекции 4 мы рассмотрели калькулятор арифметических выражений в обратной польской нотации. В определённом смысле эта нотация является полной — в ней можно записать любую функцию, которая представляется арифметическим выражение от своих аргументов.

Для вычислительных машин более естественным являются другие функции – булевы функции. Это функции работающие с битами, где бит — это ячейка памяти, которая может принимать значение 0 или 1. Можно сказать, что булевы функции получают на вход набор бит и возвращают в качестве результата тоже набор бит.

Полные системы булевых функций

Пусть B — некоторое конечное множество, которые мы будем называть алфавитом, а его элементы – буквами. Любая конечная последовательность букв называется словом. Множество слов длины n обозначается как B^n. Множество всех слов обозначается как B^*.

Определение 20.1. **Булева функция** — *это функция, которая на вход получает бинарное слово фиксированной длины m и возвращает бинарное слово фиксированной длины n, то есть имеет вид*

$$f : B^m \to B^n,$$

где $B = \{0, 1\}$ — алфавит из двух букв. Логические бинарные операции — это булевы функции для случая $m = 2$, $n = 1$:

$$f : B^2 \to B.$$

Задача Л20.1. Сколько всего различных логических бинарных операций?

Эту задачу проще всего решить в общем виде.

Задача Л20.2. Рассмотрим отображения вида $f : A \to B$. Это функции, у которых аргумент — элемент множества A, а результат функции — элемент множества B. Известны размеры $|A|$ и $|B|$ множеств A и B? Сколько существует различных отображений вида $f : A \to B$?

Каждое отображение — это спопоставление каждому из возможных аргументов A одного из элементов множества B. Подсчитаем число всех таких сопоставлений. Когда мы конструируем отображение мы для каждого элемента A выбираем, какой элемент из B ему сопоставить, то есть делаем $|A|$ выборов, каждый раз выбирая из $|B|$ вариантов. Поэтому ответ $|B|^{|A|}$. Множество всех отображений вида $f : A \to B$ условно обозначается как B^A. Поэтому мы можем записать:

$$|B^A| = |B|^{|A|}.$$

Логические операции — это отображения вида $f : B^2 \to B$, поэтому их число можно посчитать по формуле

$$|B|^{|B^2|} = 2^4 = 16.$$

Среди логических операций наиболее популярны три: AND, OR и XOR. Эти операции получают на вход пары бит, то есть слова длины 2:

$$00,\ 01,\ 10,\ 11,$$

а возвращают один бит:

$$0,\ 1.$$

Приведём таблицы сопоставления входов и выходов для указанных трёх операций:

$x_1\ x_2$	$\mathrm{AND}(x_1, x_2)$
00	0
01	0
10	0
11	1

$x_1\ x_2$	$\mathrm{OR}(x_1, x_2)$
00	0
01	1
10	1
11	1

$x_1\ x_2$	$\mathrm{XOR}(x_1, x_2)$
00	0
01	1
10	1
11	0

Из этих таблиц видно, что операция AND возвращает 1 только тогда, когда оба бита единичны, а операция OR возвращает 0 только тогда, когда оба бита нулевые. Операция XOR равна 1 только тогда, когда один из входных бит единичный, а второй — нулевой.

Рассмотренные три операции являются **бинарными** операциями — они из двух бит «делают» один бит. Есть также **унарные** операции, которых на вход получают один бит и возвращают один бит. Унарная операция NOT инвертирует входной бит: 0 превращает в 1, а 1 — в 0:

x	$\mathrm{NOT}(x)$
0	1
1	0

Оказывается, с помощью этих четырёх операций можно задать произвольную булеву функцию. Это означает, что набор из четырёх логических операций

$$\{\mathrm{NOT},\ \mathrm{AND},\ \mathrm{OR},\ \mathrm{XOR}\}$$

является *полным* на множестве булевых функций. Понятие полноты — очень важное понятие в математике и теории вычислений. Рассмотрим его подробнее.

Определение 20.2. *Пусть дана некоторая* **система логических операций** $\mathcal{B} = \{O_1, O_2, \ldots\}$ *и некоторая булева функция* $f : B^m \to B^n$. *множество входных бит обозначим как* x_1, x_2, $\ldots$, x_m, *а множество выходных бит как* y_1, y_2, $\ldots$, y_n. *Зафиксируем специальные биты: нулевой бит* $z_0 = 0$ *и единичный бит* $z_1 = 1$. *Разрешено вводить любое число новых бит* z_2, z_3, ..., *вычисляя их значения с помощью операций из системы* $\mathcal{B}$ *над уже вычисленными или входными битами. Затем значения* y_i *нужно также определить как значение операций из системы* $\mathcal{B}$ *над вычисленнымих или входными битами. Например, функцию* $f : B^4 \to B$, *заданную формулой*

$$f(x_1, x_2, x_3, x_4) = \mathrm{XOR}(\mathrm{AND}(x_1, x_2), \mathrm{AND}(x_3, x_4)),$$

можно вычислить с помощью следующей последовательности шагов:

$$\begin{aligned} z_2 &= \mathrm{AND}(x_1, x_2), \\ z_3 &= \mathrm{AND}(x_3, x_4), \\ y_1 &= \mathrm{AND}(z_2, z_3). \end{aligned}$$

Множество булевых функций, которые можно вычислить с помощью описанной схемы, называются **вычислимыми в системе** $\mathcal{B}$. *Если в системе* $\mathcal{B}$ *все булевы функции вычислимы, то система называется* **полной**.

Примечание:. Можно не ограничиваться бинарными и унарными логическими операциями и распространить понятие полной системы на систему произвольных булевых функций (а не только функций вида $f : B^2 \to B$ и $f : B \to B$). Какие операции считать элементарными и рассматривать как базис по сути определяется испольнителем, на котором производятся вычисления, а исполнители бывают разные.

Приведённая схема пошагового вычисления булевых функций напрямую связана с *логическими схемами*.

Логический элемент — это преобразователь входных сигналов в выходной согласно одной из логических функций (сигнал может иметь значение 0 или 1)[1]. **Логическая схема** — это множество логических элементов, входы и выходы которых некоторым образом соединены друг с другом. Выход элемента может быть соединён проводом только с входом некоторого другого элемента, то есть для проводов задано направление движение сигнала, и двигаясь в этом направлении нельзя вернуться в точку, где мы уже побывали. Кроме того, с каждым входом может быть соединён только один выход. Есть несколько точек-входов, на которые поступает сигналы извне, от них идут провода на входы некоторых элементов схемы. И есть несколько логических элементов, выходы которых рассматриваются как выходные значения.

Логических элементов NOT, OR, AND и XOR достаточно, чтобы составить логическую схему, вычисляющую любую булеву функции. Эту систему можно редуцировать до двух элементов — NOT и AND. Более того, достаточно иметь всего лишь один логический элемент — элемент NOT $\circ$ AND, который соответствует элементу AND, выход которого дополнительно инвертировали.

Считается, что есть специальная точка, из которой всегда выходит единичный сигнал. Логическую операцию, которая для любых входов возвращает 1 (удобно считать, что у неё вообще входов нет), обозначим жирной единицей **1** и будем считать, что она присутствует по умолчанию в любой системе.

[1]Иногда рассматривают более широкое множество логических элементов, у которых несколько входов и несколько выходов.

Утверждение 20.1. *Системы*

$$\mathcal{B}_1 = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}, \quad \mathcal{B}_2 = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}\},$$
$$\mathcal{B}_3 = \{\mathrm{XOR}, \mathrm{AND}\}, \quad \mathcal{B}_4 = \{\mathrm{NOT}, \mathrm{AND}\}, \quad \mathcal{B}_5 = \{\mathrm{NOT} \circ \mathrm{AND}\}$$

являются полными. При этом первые две системы **редуцируемые**, *то есть некоторые элементы из них могут быть исключены, но они по-прежнему останутся полными. Последние три системы не редуцируемые. Не редуцируемые полные системы называются* **базисами**.

Базис — это такое множество объектов, из которых можно сконструировать любой объект из выбранного множества. Это свойство называется свойством полноты базиса. Кроме того, базис обладает свойством минимальности — из него нельзя ничего выкинуть, не нарушив свойство полноты.

Операция $\mathrm{NOT} \circ \mathrm{AND}$ (рис. 20.1) есть композиция унарной операции NOT и бинарной операции AND. **Композиция операций** означает, что выход одной поступает на вход другой:

$$(\mathrm{NOT} \circ \mathrm{AND})(x_1, x_2) = \mathrm{NOT}(\mathrm{AND}(x_1, x_2)).$$

$x_1\ x_2$	$(\mathrm{NOT} \circ \mathrm{AND})(x_1, x_2)$
00	1
0 1	1
1 0	1
1 1	0

Рис. 20.1

Обозначим операцию AND знаком амперсанд (&), операцию XOR — знаком плюс в кружочке ($\oplus$), операцию OR — вертикальной палочкой (|), а унарную операцию NOT знаком $\neg$:

$$\mathrm{NOT}(x) = \neg\, x, \quad \mathrm{AND}(x, y) = x \,\&\, y, \quad \mathrm{OR}(x, y) = x \mid y, \quad \mathrm{XOR}(x, y) = x \oplus y.$$

Для того, чтобы зафиксировать последовательность операций будем использовать круглые скобки.

Задача Л20.3. Докажите, что AND, OR и XOR коммутативные и ассоциативные операции:

$$x \oplus y = y \oplus x, \quad (x \oplus y) \oplus z = x \oplus (x \oplus y),$$
$$x \,\&\, y = y \,\&\, x, \quad (x \,\&\, y) \,\&\, z = x \,\&\, (y \,\&\, z),$$
$$x \mid y = y \mid x, \quad (x \mid y) \mid z = x \mid (y \mid z)$$

Задача Л20.4.
а) Покажите, что $x_1 \oplus x_2 \oplus \ldots \oplus x_n$ равно 1 тогда, когда среди x_i нечётное число единиц. б) Покажите, что $x_1 \,\&\, x_2 \,\&\, \ldots \,\&\, x_n$ равно 1 тогда и только тогда, когда все x_i единицы. в) Покажите, что $x_1 \mid x_2 \mid \ldots \mid x_n$ равно 0 тогда и только тогда, когда все x_i нули.

Обозначим операцию $\mathrm{NOT} \circ \mathrm{AND}$ как $*$:

$$x * y \equiv \mathrm{NOT}(\mathrm{AND}(x, y)).$$

Эта операция коммутативна, но не ассоциативна.

Задача Л20.5. Докажите, что $x * x = \neg\,(x)$, $(x * y) * (x * y) = x \,\&\, y$, $(x * x) * (y * y) = x \mid y$. Выразите с помощью операции $*$ операцию $x \oplus y$.

Задача Л20.6. Запишите выражение $x \oplus y \oplus z$, пользуясь только операцией $*$.

Задача Л20.7. Докажите, что $1 \oplus x = \neg\,(x)$, $\quad (x \,\&\, y) \oplus x \oplus y = x \mid y$.

Доказательство полноты системы $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$

Логическое выражение — это запись булевой функции $f : B^n \to B$ с помощью логических операций и булевых переменных, соответствующих входными битам.

Докажем, что система булевых функций $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ является полной. Любую булеву функцию можно выразить через эти функции, то есть представить в виде логического выражения от булевых переменных, в котором используются только операции AND, OR и NOT.

Любую булеву функцию можно записать в виде таблицы, в которой для каждого возможного набора значений аргументов указано значение функции. Приведем для примера описание некоторой булевой функции $g : B^3 \to B$ в виде таблицы, приведённой справа. Опишем алгоритм построения логического выражения, которое будет реализовывать функцию, заданную таблицей.

$x_1\ x_2\ x_3$	$g(x_1, x_2, x_3)$
000	1
001	0
010	1
011	0
100	1
101	1
110	1
111	0

Определение 20.3. *Назовём* AND**-выражением** *логическое выражение, в котором несколько булевых переменных соединены операцией* AND*; некоторые из переменных в этом выражении могут быть взяты с отрицанием. Примеры* AND*-выражений:*

$$x_1 \,\&\, x_3 \,\&\, \neg\, x_2, \quad x_3, \quad x_2 \,\&\, x_3, \quad \neg\, x_1 \,\&\, \neg\, x_2 \,\&\, \neg\, x_3 \,\&\, \neg\, x_4.$$

ШАГ 1. Найдём в таблице строчки, в которых значение функции равно 1. В нашем примере, это строчки:

$x_1\ x_2\ x_3$	$g(x_1, x_2, x_3)$
000	1
010	1
100	1
101	1
110	1

ШАГ 2. Для каждой такой строчки сформируем AND**-выражение** от булевых переменных, представляющее собой операцию AND от всех булевых переменных, среди которых те, которые равны нулю в данной строчке, взяты с операцией отрицания « $\neg$ »:

$x_1\ x_2\ x_3 \to$	AND-выражение
000	$\to (\neg\, x_1 \,\&\, \neg\, x_2 \,\&\, \neg\, x_3)$
010	$\to (\neg\, x_1 \,\&\, x_2 \,\&\, \neg\, x_3)$
100	$\to (x_1 \,\&\, \neg\, x_2 \,\&\, \neg\, x_3)$
101	$\to (x_1 \,\&\, \neg\, x_2 \,\&\, x_3)$
110	$\to (x_1 \,\&\, x_2 \,\&\, \neg\, x_3)$

Каждое из логических выражений справа равно 1 в только том случае, когда аргументы равны тем значениям, которые указаны слева. Например, логическое выражение

$$(x_1 \,\&\, x_2 \,\&\, \neg\, x_3)$$

можно интерпретировать как

«Аргументы функции в точности равны (1, 1, 0).»

ШАГ 3. Теперь нам нужно записать следующее выражение:

«Функция g равна 1, когда аргументы точности равны
(0, 0, 0) или (0, 1, 0) или (1, 0, 0) или (1, 0, 1) или (1, 1, 0).»

Это утверждение запишем с помощью операции OR от всех AND-выражений:

$$\begin{aligned} g(x_1,x_2,x_3) = \; & (\neg x_1 \,\&\, \neg x_2 \,\&\, \neg x_3) \mid \\ & (\neg x_1 \,\&\, \, x_2 \,\&\, \neg x_3) \mid \\ & (\, x_1 \,\&\, \neg x_2 \,\&\, \neg x_3) \mid \\ & (\, x_1 \,\&\, \neg x_2 \,\&\, \, x_3) \mid \\ & (\, x_1 \,\&\, \, x_2 \,\&\, \neg x_3) \end{aligned}$$

Таким образом, мы на частном примере показали, как по таблице, описывающей функцию, построить соответствующее функции логическое выражение.

Определение 20.4. *Представление функции в виде логического выражения, в котором* AND*-выражения соединены с помощью операции* OR*, называется* **коньюктивной нормальной формой** *(КНФ).*

Итак по таблице, задающей функцию, несложно построить коньюктивную нормальную форму — логическое выражение, реализующее данную функцию. В КНФ используются только операции AND, OR и NOT. Таким образом, система $\mathcal{B} = \mathrm{AND}, \mathrm{OR}, \mathrm{NOT}$ является полной системой.

Задача Л20.8. Покажите, что верны следующие соотношения

$$\neg\,(a \,\&\, b) = \neg\, a \mid \neg\, b, \quad \neg\,(a \mid b) = \neg\, a \,\&\, \neg\, b.$$

$$\neg\,(x_1 \,\&\, x_2 \,\&\, \ldots) = \neg\, x_1 \mid \neg\, x_2 \mid \ldots, \quad \neg\,(x_1 \mid x_2 \mid \ldots) = \neg\, x_1 \,\&\, \neg\, x_2 \,\&\, \ldots.$$

Задача Л20.9. а) Запишите $\mathrm{AND}(x, y)$ через операции OR и NOT.
б) Запишите $\mathrm{OR}(x, y)$ через операции AND и NOT.

Задача Л20.10. Покажите, что $\mathcal{B} = \{\mathrm{AND}, \mathrm{NOT}\}$ и $\mathcal{B} = \{\mathrm{OR}, \mathrm{NOT}\}$ являются базисами.

Полиномы Жегалкина

Система $\mathcal{B} = \{\mathrm{AND}, \mathrm{XOR}\}$ представляет особый интерес. Она является базисом. Давайте операцию AND обозначать знаком умножения (точкой) а XOR — знаком сложения (+). Их действительно можно интерпретировать как операции умножения и сложения, только результат рассматривать по модулю два. В арифметике по модулю[2] 2 мы следим лишь за чётностью/-нечётностью результата. Сравните приведённые таблицы чётности результатов сложения и умножения чисел с таблицами операций AND и XOR.

[2]В арифметике по модулю M числа отождествляются с их остатками при делении. Два числа считаются эквивалентными, если их разность делится на M. Всего получается M классов эквивалентности, которые можно обозначить M числами 0, 1, …, $M-1$. Легко показать, что в арифметике по модулю выполнены законы коммутативный, ассоциативный и дистрибутивный.

$x_1\ x_2$	$x_1 \cdot x_2$
ЧЁТ ЧЁТ	ЧЁТ
ЧЁТ НЕЧ	ЧЁТ
НЕЧ ЧЁТ	ЧЁТ
НЕЧ НЕЧ	НЕЧ

$x_1\ x_2$	$x_1 + x_2$
ЧЁТ ЧЁТ	ЧЁТ
ЧЁТ НЕЧ	НЕЧ
НЕЧ ЧЁТ	НЕЧ
НЕЧ НЕЧ	ЧЁТ

Из этого непосредственно следует что для AND и XOR верен распределительный закон:

$$a \cdot (b + c) = a \cdot b + a \cdot c.$$

В этом несложно убедится непосредственно, просто проверив все 8 возможных наборов значений булевых переменных a, b и c.

Переменные, которые принимают значение 0 или 1 принято называть **булевыми переменными**. Вместо термина «бит» мы будем использовать термин «булева переменная».

Для удобства будем как обычно считать, что у умножения (операции AND) больший приоритет, нежели у сложения. В частности, $x + y + x \cdot y = x + y + (x \cdot y)$.

Утверждение, что система операций $\mathcal{B} = \{\mathrm{AND}, \mathrm{XOR}\}$ является базисом, по сути означает, что любую булеву функцию можно выразить в виде полинома от булевых переменных. Запишем для примера несколько простых булевых функций в виде полиномов:

$$\begin{aligned}
&\mathrm{XOR}(x, y) = x + y\\
&\mathrm{AND}(x, y) = x \cdot y\\
&\mathrm{OR}(x, y) = x \cdot y + x + y\\
&\mathrm{NOT}(x) = x + 1\\
&\mathrm{NOT}(\mathrm{AND}(x, y)) = x \cdot y + 1
\end{aligned}$$

В арифметике по модулю 2 верны следующие необычные тождества:

$$x^2 = x \cdot x = x, \quad x - y = x + y$$

Это значит что степени переменных можно отбрасывать ($x^n = x$, для $n \geqslant 1$), а операция вычитания, обратная операции сложения, совпадает с самим сложением:

$$x + y = z, \quad z + y = x + y + y = x + 2 \cdot y = x + 0 \cdot y = x.$$

Поэтому все возможные многочлены от переменных x и y можно представить в виде

$$P(x, y) = a + b \cdot x + c \cdot y + d \cdot (x \cdot y),$$

где константы $a,\ b,\ c,\ d$ равны 0 либо 1. Перечислим все многочлены от двух переменных:

$$\begin{aligned}
&0, \quad 1, \quad x, \quad y, \quad x \cdot y,\\
&1 + x, \quad 1 + y, \quad 1 + x \cdot y, \quad x + y, \quad x + x \cdot y, \quad y + x \cdot y,\\
&1 + x + y, \quad x + y + x \cdot y, \quad 1 + x + x \cdot y, \quad 1 + y + x \cdot y\\
&1 + x + y + x \cdot y.
\end{aligned}$$

Их 16 штук. А мы с вами уже подсчитали, что множество всех булевых операций (функций вида $f : B^2 \to B$) 16 штук. Многочленов тоже оказалось 16 штук. Причём разные многочлены не могут совпадать как функции, так как их разность есть ненулевой многочлен, а среди указанных многочленов, отличных от $P(x, y) = 0$, нет ни одного, который был бы равен 0 при всех возможных x и y.

Определение 20.5. *Представления булевых функции в базисе* $\mathcal{B} = \{\text{AND, XOR}\}$ *называются* **полиномами Жегалина**[3].

Задача Л20.11. Дан многочлен $P(x_1, x_2, \ldots, x_n)$ от булевых переменных, в котором раскрыты все скобки, убраны все степени и приведены все подобные слагаемые. Покажите, что если он не равен тождественно нулю (как многочлен), то найдутся такие значения переменных, при котором он равен 1 (как вычисленное значение при подставленных значениях переменных). Проведите доказательство методом математической индукции.

Итак, все булевы функции вида $f : B^2 \to B$ представимы в виде полинома Жегалкина. Аналогичное утверждение можно доказать и для функций вида $f : B^n \to B$, при любом натуральном n.

Полиномы Жегалкина — это **формальная система записи булевых функций**, которая *полна*, то есть в этой системе можно записать любую булеву функцию. Более того, после раскрытия скобок, приведения слагаемых и избавления от степеней вид полинома (с точностью до перестановки слагаемых) однозначно определяет булеву функцию.

Схема доказательства этого утверждения следующая. Каждый многочлен после раскрытия скобок, убирание лишних степеней и приведения подобных превращается в сумму мономов. **Моном** — это произведение некоторого подмножества булевых переменных. Если булевых переменных n штук, то число возможных мономов есть 2^n (количество различных подмножеств множества из n элементов). Свободный член – единица – соответствует пустому подмножеству.

Каждый из этих мономов может либо присутствовать, либо отсутствовать в многочлене, и всего получается 2^{2^n} многочленов.

Далее, давайте возьмём два различных многочлена и вычтем один из другого (напомним, что вычитание в арифметике по модулю два совпадает со сложением). В результате (после сокращения некоторых слагаемых) получится многочлен, неравный нулевому многочлену. В задаче Л20.11 утверждается, что этот многочлен при некоторых значениях не равен нулю. Значит разные многочлены реализуют разные булевы функции.

Но число всех различных булевых функций вида $f : B^n \to B$ равно как раз 2^{2^n}. Значит многочлены покрывают всё множество функций.

Задача Л20.12. Сколько слагаемых будет в многочлене

$$P(x_1, \ldots, x_{10}) = (x_1 + x_2) \cdot (x_3 + x_4) \cdot \ldots \cdot (x_9 + x_{10})$$

после того, когда будут раскрыты все скобки и приведены подобные слагаемые?

Задача Л20.13. Булева функция $f : B^3 \to B$ задана таблицей:

[3]Напомним, что в рамках этой лекции единичную функцию **1** мы считаем неявно присутствующей в каждом базисе. Соответственно в полиномах Жегалкина кроме операций $\cdot$ (AND) и $+$ (XOR) разрешено использовать единицу.

$x_1\ x_2\ x_3$	$f(x_1, x_2, x_3)$
000	1
001	1
010	0
011	1
100	0
101	1
110	0
111	1

Выразите функцию f, используя операции
а) OR, AND, NOT; б) XOR, AND.

*Задача SAT

Мы подошли к важной задаче теории сложности вычислений. Пусть дано некоторое логическое выражение от n булевых переменных x_1, x_2, …, x_n, записанное с помощью логических операций (AND, OR, NOT, XOR и возможно других) и скобок, например:

$$x_3 \,\&\, ((x_1 \,\&\, x_2) \mid (\neg\, x_2 \,\&\, x_5) \mid \neg\, (x_3 \,\&\, \neg\, x_4 \,\&\, x_5)).$$

Задача SAT: Определить, есть ли такие значения булевых переменных, при которых данное логическое выражение равно 1 (истина).

Задача SAT[4] — это главная $\mathcal{NP}$-полная задача — задача, для которой не найдено (и, видимо, не существует) эффективного быстрого алгоритма, а именно, такого алгоритма, время работы которого было бы ограничено полиномом какой-либо степени от длины выражения (в символах).

Множество $\mathcal{NP}$-полных задач определяются через эту задачу — задача X является по определению $\mathcal{NP}$-полной, если она сводится к задаче SAT, а задача SAT в свою очередь сводится к задаче X[5]

Давайте разберемся, в чём заключается сложность задачи SAT.

Она выглядит не такой уж и сложной. Во-первых можно перебрать все возможные значения переменных. Например, пять булевых переменных могут принимать $2^5 = 32$ возможных значений (наборов значений). Вычислим данное выражение непосредственно для каждого из 32 вариантов и ответим на вопрос. Если хотя бы в одном случае получится 1, то ответ «Да», иначе — «Нет».

Проблема в том, что если у нас 100 булевых переменных, то перебирать придётся 2^{100} вариантов. Это очень много: $2^{100} > (2^{10})^{10} = 1024^{10} > 10^{30}$.

Пусть логическое выражение имеет длину N символов. Это число ограничивает сверху как количество использованных в выражении переменных, так и количество логических операций в выражении.

[4]Три буквы SAT взяты от английского слова *satisfy*.

[5]О понятии сводимости одной задачи к другой можно подробно прочитать в книге [12].

Существует ли алгоритм, решающий задачу SAT, который работает время, ограниченное некоторым полиномом от N?

Ответ на этот вопрос не известен. Полиномиального алгоритма не найдено, но и не доказано, что такого алгоритма нет.

Давайте рассмотрим один простой непереборный алгоритм решения задачи SAT. Превратим данное нам выражение в полином Жегалкина. Для этого операции AND заменим на знак «$\cdot$», а XOR — на знак «$+$». Операцию OR можно записать через AND и XOR:

$$x \mid y = x \cdot y + x + y.$$

Также и остальные логические операции можно выразить через эти две и единичный элемент.

После всех замен мы получим многочлен, записанный с помощью операций умножения, сложения, единицы и скобок. Осталось только раскрыть все скобки и привести подобные слагаемые.

Логическое выражение равно нулю при всех значения булевых переменных тогда и только тогда, когда в полученном многочлене сократятся все слагаемые и он тождественно будет равен нулю.

Почему описанный алгоритм не является приемлемым решением задачи SAT?

Лекция 21

Сложность вычислений

Краткое описание: Сложность вычислений является важнейшим понятием теории вычислений. Различают несколько типов сложности: комбинационную, описательную и алгоритмическую. При этом особо выделяют асимптотическую алгоритмическую сложность, которая соответствует степени роста времени работы алгоритма с размером входных данных. Процессорное время в настоящий момент является одним из самых важных вычислительных ресурсов. Поэтому на практике значительное внимание уделяется асимптотической алгоритмической сложности задач. Мы рассмотрим идеи корректного определения этого понятия.
Комбинационная и описательная сложность относятся к задачам проектирования вычислительных устройств, работающих согласно заданной логике. Одной из актуальных современных задач такого типа является реализация логики работы процессора вычислительного устройства на кристалле минимального размера.

Типы сложности

Комбинационная сложность

Комбинационная сложность относится не к области программирования алгоритмов, а к области конструирования вычислительных схем.

Рассмотрим операции AND (логическое «и»), OR (логическое «или»), XOR (логическое «исключающее или») на двумя битами, результатом выполнения которых является один бит:

$x_1\ x_2$	$\mathrm{AND}(x_1, x_2)$
00	0
01	0
10	0
11	1

$x_1\ x_2$	$\mathrm{OR}(x_1, x_2)$
00	0
01	1
10	1
11	1

$x_1\ x_2$	$\mathrm{XOR}(x_1, x_2)$
00	0
01	1
10	1
11	0

Множество значений одного бита есть $B = \{0, 1\}$. Множество значений пары бит обозначают как $B^2 = \{00, 01, 10, 11\}$. Указанные операции AND, OR, XOR имеют вид[1] $f : B^2 \to B$.

Из этих элементов можно конструировать различные схемы, которые на вход получают m бит, а возвращают n бит, то есть являются *реализацией* функций вида $B^m \to B^n$. Такие функции называются **булевыми функциями**.

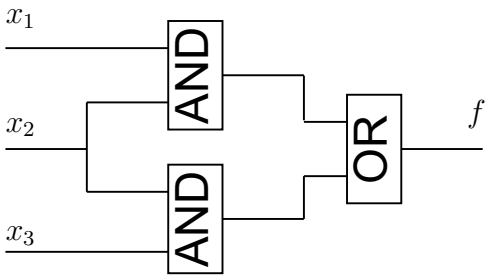


Рис. 21.1: Пример реализации логической функции в виде схемы.

Например, функцию

$$f(x_1, x_2, x_3) = \mathrm{OR}(\mathrm{AND}(x_1,\ x_2),\ \mathrm{AND}(x_2,\ x_3))$$

вида $B^4 \to B^1$ можно визуально представить в виде схемы, показанной на рисунке 21.1.

Возникают следующие вопросы:

1. Любую ли булеву функцию можно реализовать с помощью данного набора элементов?

2. Какое минимальное число элементов требуется для реализации некоторой данной булевой функции $f : B^m \to B^n$?

Ответ на первый вопрос отрицательный. Не все булевы функции можно реализовать в виде композиции элементов AND, OR, XOR. Чтобы иметь возможность описывать все функции, необходимо добавить элемент $\mathrm{NOT} : B \to B$, который 0 отображает в 1, а 1 отображает в 0, а также специальный элемент, обозначаемый как **1**, который ничего не получает на вход, а возвращает значение 1. Наборы элементов, с помощью которых можно реализовать (в виде схемы) произвольную булеву функцию, называются **полными**. В частности набор $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}, \mathbf{1}\}$ является полным. Также полным является набор $\{AND, \mathrm{NOT}, 1\}$. Более подробно о булевых функциях можно узнать в дополнительной лекции 20.

Второй вопрос непосредственно касается комбинационной сложности.

Определение 21.1. **Комбинационная сложность** *булевой функции f относительно набора элементов $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}, \mathbf{1}\}$ — это минимальное число элементов указанного типа, с помощью которых можно реализовать функцию f в виде схемы.*

Безусловно, можно определять комбинационную сложность относительно произвольного набора элементов. Обычно берут полные наборы, чтобы сложность была определена для всех элементов.

[1]Тот факт, что функция f получает на вход элемент множества X, а возвращает в качестве результата элемент множества Y, коротко записывают как $f : X \to Y$.

Понятие комбинационной сложности определяется не только на множестве булевых функций. Его можно определять, например, на множестве натуральнозначных функций натурального аргумента, то есть функций вида $f : \mathbb{N} \to \mathbb{N}$. В качестве элементарных элементов можно взять функции умножения, сложения, вычитания, а также элементы без входов — **0** и **1**, которые всегда равны числам 0 и 1, соответственно.

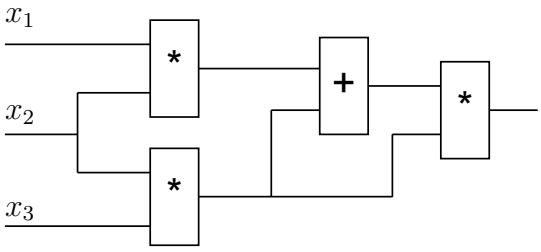


Рис. 21.2: Арифметическая схема, вычисляющая выражение $f = ((x_1 \cdot x_2) + (x_2 \cdot x_3)) \cdot (x_2 \cdot x_3)$.

Таким схемам естественно сопоставить арифметические выражения (см. рис. 21.2). Но при этом, число операций в арифметическом выражении, возможно, будет больше, чем число элементов в соответствующей схеме. Это связано с тем, что в схеме выход из каждого элемента можно разветвить на несколько и направить его в виде входа в несколько других элементов, а в арифметическом выражении подвыражение может «передать» своё значение только в то место, где оно встретилось. Если его значение нужно еще в каком-либо месте, то придётся его просто переписать.

Комбинационная сложность относится к задачам проектирования вычислений в железе, в виде процессоров и других электронных вычислительных подблоков. Комбинационная сложность равна минимально возможному размеру схемы, реализующую заданную логику.

Алгоритмическая сложность

Для задачи вычисления арифметических выражений удобно ввести абстрактного исполнителя. Назовём его «Арифмометр». Он получает на вход арифметическое выражение (для него это выражение будет играть роль программы) и последовательно, операция за операцией, согласно расставленным скобкам осуществляет его вычисление. Число операций, которое проделает этот исполнитель (число знаков операций в выражении) соответствует сложности данного выражения-программы.

Задачами для этого исполнителя являются функции, значение которых могут быть записано как арифметическое выражение от аргументов. И сложность задачи для данного исполнителя — это минимальная сложность выражения, которое реализует функцию.

Определение 21.2. *Пусть дан исполнитель И и набор элементарных действий, которые он совершает. Результат f может получен в результате последовательности элементарных действий. Алгоритмическая сложность f относительно исполнителя И — это минимальное число элементарных действий, с помощью которых можно реализовать результат f.*

В то время, как комбинационная сложность относится к задачам проектирования устройств (hardware), алгоритмическая сложность относится к задаче программирования алгоритмов.

Здесь важно отметить следующую деталь. В случае, когда исполнитель понимает инструкции подобные циклу **while** или условному оператору **if**, количество элементарных действий начинает зависеть от входных данных, и тогда задача поиска минимального числа элементарных действий в приведённой формулировке теряет смысл.

Например, какова сложность задачи вычисления НОД(a, b)? Ясно, что время вычисления зависит от входных данных (аргументов a и b). Если зафиксировать входные данные,

то пропадает смысл осуществлять какие-либо вычисления — проще просто вернуть заранее вычисленный результат, соответствующий именно этим входным данным.

Для исполнителей, у которых время вычислений может зависеть от входных данных, используется другое определение алгоритмической сложности, а именно, сложностью называется *средняя степень роста* (*асимптотика роста*) времени вычислений в зависимости от параметров задачи.

Асимптотическая сложность

Как было сказано, алгоритмы должны обладать свойством массовости, то есть получать входные данные и осуществлять с ними-либо преобразования (вычисления). Время вычислений в общем случае зависит от входных данных.

Например, в задаче «Ханойские башни» для перемещения n дисков требуется время $2^n - 1$ (см. алгоритм 1.3). В этой задаче входом является число n и время вычисления растёт примерно как n-я степень двойки. В таких случаях говорят, что асимптотическая сложность задачи есть $\Theta(2^n)$. Запись $\Theta(f)$ используют тогда, когда речь идёт о функции, *примерно пропорциональной* f.

Для того, чтобы освоится с понятием асимптотической сложности, давайте оценим, как растёт время вычисления НОД(a, b) в следующем алгоритме.

Алгоритм 21.42 Рекурсивный улучшенный алгоритм Евклида

```
1: function НОД(a, b)
2:     if a > b and b > 0 then
3:         return НОД(b, min(a mod b, b − a mod b))
4:     else if a < b then
5:         return НОД(b, a)
6:     else
7:         return a
8:     end if
9: end function
```

Выражение (a mod b) означает остаток при делении a на b. Операция mod имеет больший приоритет, чем операции сложения и вычитания. Выражение $\min(a \bmod b,\ b - a \bmod b))$ равно расстоянию на числовой прямой от числа a до ближайшего числа, кратного b.

Задача Л21.1. Докажите, что данный алгоритм действительно вычисляет НОД(a, b).

Обратите внимание на то, что при вычислении НОД(a, b) рекурсивно вызывается вычисление функции НОД с другими значениями аргументов, причём первый аргумент a заведомо больше b. Это будет верно для всех последующих рекурсивных вызовах и условие оператора **if** будет выполнено. В результате при каждом рекурсивном вызове новое значение a будет равно старому b, а новое значение b — расстоянию от старого a до ближайшего числа, кратного старому b. Это расстояние меньше либо равно половине старого b. Ясно, что значение b с каждым следующим рекурсивным вызовом становится меньше старого как минимум в два раза. Рекурсивные вызовы прекратятся, когда b достигнет значения 0. Поэтому, можно сказать, что число шагов рекурсии ограничено сверху некоторым числом, которое равно тому, сколько раз нужно разделить исходное b (а точнее $\min(a, b)$) на 2 без остатка, чтобы уменьшить его до

0. Это ни что иное, как округлённые до целого числа логарифм по основанию 2 от числа b. Поэтому асимптотическая сложность приведённого алгоритма есть

$$O(\log_2(\min(a, b))).$$

Выражение $O(f)$ используется обозначения класса функций, которые растут пропорционально функции f или слабее, а точнее, ограничены сверху функцией вида $A \cdot f + B$, где A и B — некоторые константы.

Сложность описания

Определение 21.3. *Сложность описания объекта — это длина его описания на некотором формальном языке.*

Соответственно, описательная сложность вычислительной задачи относительно заданного формального языка описания алгоритмов — это длина описания самой короткой программы, которая решает данную задачу.

Эта сложность, также как во всех предыдущих случаях, имеет относительный характер — она зависит от выбранного исполнителя (языка описания алгоритмов).

Длина программы — не самый важный показатель её эффективности. Безусловно, степень использования ресурсов компьютера (процессорного времени, памяти, сетевых устройств, ...) является более важным показателем с практической точки зрения. И, конечно, не следует стремится писать короткие программы. Гораздо важнее, чтобы они были эффективными, понятными, правильно структурно организованными.

Но бывают задачи, где сложность описания играет существенную роль. В частности, графические изображения можно рассматривать как результат выполнения «рисующих» алгоритмов, записанных на некотором специальном языке. Существуют целый ряд интересных алгоритмов сжатия, основанных на том, что программа сжатия старается найти алгоритм как можно меньшего размера, выполнения которого приводит к данному изображению. В этом случае не так важно, сколько времени будет работать алгоритм, а важно, чтобы описание алгоритма было как можно меньше.

Основные понятия теории сложности вычислений

Теория сложности вычислений занимается оценкой вычислительной сложности различных вычислительных задач в рамках различных исполнителей. И асимпототическая сложность алгоритмических задач является центральным понятием этой теории.

Сложность вычислимой функции естественно определить через сложность *самой лучшей* программы для какого-то исполнителя. При этом, как мы уже отметили, можно рассматривать различных исполнителей и различные аспекты сложности: размер и сложность кода программы, требуемую для вычислений память или время вычисления. Время вычисления является наиболее важным аспектом сложности, именно на него обращают внимание в первую очередь при разработке алгоритмов. В качестве эталонного исполнителя обычно рассматривают машину Поста, машину Тьюринга, а чаще, некоторое известное реальное (а не абстрактное) вычислительное устройство.

Здесь мы встречается со следующей проблемой: нельзя определить сложность задачи как сложность *самой лучшей программы* решающей эту задачу, так как самой лучшей программы может просто не быть: одна программа хороша для входных данных одного типа, а другая — для входных данных другого типа.

Обойти эту трудность помогает рассмотрение *асимптотик* времени вычисления (или требуемой памяти) как функций от размера входа.

Рассмотрим случай, когда входной и выходной алфавиты равны $B = \{0, 1\}$, то есть на вход поступает бинарное слово и результат вычислений — тоже бинарное слово.

Размером входа называется длина двоичного слова на входе.

Пусть P — данная вычислимая функция (задача), а И1 — некоторый исполнитель. Рассмотрим программу $\mathcal{M}$ для этого исполнителя, которая решает поставленную задачу P.

Пусть $t(\mathcal{M}, x)$ обозначает время работы программы $\mathcal{M}$ на входном слове x.

Определим функцию $T(\text{И1}, \mathcal{M}, n)$ максимального времени работы программы $\mathcal{M}$ от размера входа n.

$$T(\mathcal{M}, n) = \max_{|x| \leqslant n,\, x \in B^*} t(\mathcal{M}, x).$$

Здесь

- B^* — множество слов в алфавите B;
- $t(\mathcal{M}, x)$ — время работы программы $\mathcal{M}$ на входе $x \in B^*$;
- $|x|$ — длина слова x.

Обычно не существует программы, который работает быстрее всех других программ на всех возможных входах.

Можно попробовать найти программу $\mathcal{M}_b$, которая для всех возможных n даёт $T(\mathcal{M}_b, n)$ меньше, чем $T(\mathcal{M}, n)$ для любой другой программы $\mathcal{M} \neq \mathcal{M}_b$. То есть $T(\mathcal{M}_b, n)$ равно

$$T_P(n) = \inf_{\mathcal{M} \in \mathcal{S}(P)} T_{max}(\mathcal{M}_b, n).$$

Здесь $\mathcal{S}(P)$ есть множество программ, решающих задачу P. Возможные значения времени работы алгоритма обычно дискретное множество на положительной полуоси, поэтому инфимум достигается на некоторой программе $\mathcal{M}_b$. Она в указанном смысле является *самой лучшей* данного n. Для других значений n программа $\mathcal{M}_b$ будет другой. Это неправильный подход к определению сложности. Определённая указанным образом функция $T_P(n)$ слабо отображает реальную сложность задачи, так как при фиксированном ограничении на размер входа можно написать программу, специально оптимизированную для всех значений входных данных меньше n.

Множество различных «сложностей» — это не множество чисел, а множество функций от длины входа n. Причём естественно рассматривать только положительные неубывающие функции.

Приведём определение **верхней и нижней оценки сложности задачи**. Пусть дана неубывающая функция $f : \mathbb{N} \to \mathbb{R}$. Сложность задачи P для данного исполнителя меньше (больше) f, если существует (не существует) программа $\mathcal{M}$, время работы которой на входных данных размера меньше n ограничено сверху функцией $f(n)$.

Сложностью задачи можно было бы назвать самую маленькую оценку сверху, или самую большую оценку снизу (особенно хорошо, если эти оценки совпадают). Но дело в том, что

естественного линейного порядка на неубывающих функциях нет, и нет возможности выбрать самую большую или самую маленькую функцию из данного множества.

Здесь на помощь может прийти рассмотрение классов эквивалентности положительных неубывающих функций. Например:

Определение 21.4. *Две неубывающие функции f, g асимптотически эквивалентны, если существуют константы A_1, A_2, B_1, B_2 такие, что*

$$f(n) < A_1 \cdot g(n) + B_1, \quad g(n) < A_2 \cdot f(n) + B_2.$$

Класс эквивалентности функции $f(n)$ обозначается как $\Theta(f(n))$.

В теории алгоритмов часто встречаются следующие классы:

$$\Theta(\log n), \quad \Theta(n), \quad \Theta(n \cdot \log n), \quad \Theta(n^\alpha), \quad \Theta(a^n).$$

Класс $\Theta(f(n))$ можно грубо называть классом функций *примерно пропорциональных* $f(n)$.

Кроме $\Theta(g(n))$ используется ещё символ $O(g(n))$, который означает класс всех функций, которые ограничены сверху, начиная с некоторого $n = N$, функцией $A \cdot g(n) + B$, для некоторых положительных констант A и B.

Есть задачи, для которых их вычислительную сложность естественно связывать с некоторым классом $\Theta(f(n))$. А именно:

Определение 21.5. *Сложность задачи A равна $\Theta(f(n))$ если в классе $\Theta(f(n))$ существуют функции $h_1(n)$ и $h_2(n)$, являющиеся верхней и нижней оценкой сложности задачи A.*

Для некоторых задач такого класса может посто не найтись. В таких случаях приходится рассматривать более крупные классы эквивалентности.

Задача Л21.2.* **(сложность простейших алгоритмических задач)** Рассмотрим абстрактного исполнителя псевдокода. В псевдокоде разрешено использовать любое число целочисленных переменных и массивов, операторы структурного программирования **while**, **if**, **for**, определение функций и рекурсию, арифметические операции с целыми числами, оператор присваивания, и оператор обращения к элементу массива.

Чему равна сложность следующих задач (оцените сверху и снизу):

а) вычисление НОД(a, b) и НОК(a, b) (заметьте, что длина входа в битах равна $(\log_2 a + \log_2 b)$);

б) вычисление наименьшего делителя данного числа a (длина входа есть $n = \log_2 a$);

в) вычисления минимума введённых чисел;

г) вычисление минимального числа, которое присутствует в массиве более, чем один раз;

д) сортировка массива;

е) проверка данного числа a на простоту[2];

ж) удаление из неотсортированного массива дубликатов;

В пунктах б, в, г, д считайте, что сравнение и проверка на равенство двух элементов происходит за $O(1)$, а множество всех возможных элементов бесконечно велико.

Рассмотрите один из четырёх возможных случаев:

[2]В 2002 году было показано, что задача проверки натурального числа на простоту имеет полиномиальную сложность (от числа цифр в представлении числа), см. статью Manindra Agrawal, Neeraj Kayal and Nitin Saxena, «PRIMES is in P».

- переменные могут хранить сколь угодно большие числа, а арифметические операции с ними выполняются за одну условную секунду, либо переменные и арифметические операции есть только для маленьких чисел (например, ограниченных по модулю числом 2^{31});
- обращение к элементу массива происходит за одну условную секунду либо за время, пропорциональное логарифму от числа хранимых в нём элементов.

Какой из указанных случаев наиболее близок к реальным условиям вычисления на компьютерах?

На практике приходится пользоваться иными методами оценки времени работы алгоритмов. Современные программы имеют большое количество параметров скрытых даже в самых элементарных операциях. Детали реализации алгоритма и особенности архитектуры используемого компьютера могут сильно повлиять на время работы алгоритма. Даже оценив асимптотику времени работы программы, тяжело рассчитать реальное время её работы, потому что коэффициенты в оценочной функции сложно вычислить теоретически. От программиста скрыты заложенные в процессор алгоритмы оптимизации выполнения кода, работы с памятью и внешними устройствами. Даже если бы он их знал, формула реалистичной оценочной функции была бы очень сложной. Поэтому на практике пользуются следующими методами сравнения алгоритмов:
1. Тестирование на конечном наборе тестовых примеров, приближённых к реальным входным данным.
2. Профилирование — оценка реального времени работы каждой части программы с последующей интроспекцией (рассмотрением всё более мелких частей программы) и выявлением зависимостей от размера входных данных.

Скорость работы не является единственным критерием отбора алгоритмов. В реальной ситуации также важны простота реализации (сложность описания), расширяемость (возможность расширения класса решаемых задач) и безопасность (защита от некорректных данных и сбоев).

*Комбинационная и алгоритмическая сложности рациональных чисел

Поясним введённые понятия алгоритмической и комбинационной сложности на примере задачи вычисления (конструирования) рациональных чисел. Рассмотрим вычислителей, для которых множество решаемых задач есть рациональные числа. Назовём их $\mathbb{Q}$-исполнителями и $\mathbb{Q}$-схемами.

Это довольно специальный искусственный случай исполнителей и схем, но он позволяет увидеть особенности различных типов сложности.

$\mathbb{Q}$-исполнитель №1

Пусть наш $\mathbb{Q}$-исполнитель №1 имеет одну ячейку памяти x, в которой может хранится одно рациональное число[3], и пусть он умеет выполнять две элементарные операции: увеличить число на 1 и обратить число:

$$1)\ x \leftarrow x+1, \quad 2)\ x \leftarrow 1/x.$$

Изначально в ячейке находится число 0.

Будем считать, что операция взятия обратного числа нисколько не стоит.

Определение 21.6. *Алгоритмической сложностью рационального числа q относительно $\mathbb{Q}$-исполнителя №1 назовём минимальное число операций прибавления единицы, необходимых для того, чтобы получить число q.*

Задача Л21.3. Покажите, что алгоритмическая сложность $C_1(q)$ числа $q \in \mathbb{Q}$ непосредственно связана с его цепным разложением, а именно, если

$$q = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ldots}}},$$

где числа a_0, a_1, ... натуральные, и число a_0 также может равняться 0, то алгоритмическая сложность q относительно $\mathbb{Q}$-исполнителя №1 равна

$$C_1(q) = a_0 + a_1 + \ldots$$

Интересно отметить, что алгоритмическая сложность числа $q = a/b$, где $a, b \in \mathbb{N}$, в для данного исполнителя в точности равна количеству рекурсивных вызовов при вычислении НОД(a,b) согласно рекуррентной формуле 1.1, а также количеству итераций в алгоритме 1.2.

Если же считать, что операция добавления единицы ничего не стоит, а операция обращения стоит 1, то сложность оказывается равна количеству рекурсивных вызовов при вычислении по формуле

$$\text{НОД}(a,b) = \begin{cases} a, \text{ если } a = b \text{ или } b = 0, \\ b, \text{ если } a = 0, \\ \text{НОД}(b,\ \ a \bmod b), \text{ если } a = b, \end{cases}$$

$\mathbb{Q}$-исполнитель №2

$\mathbb{Q}$-исполнитель №2 вычисляет арифметические выражения, в которых разрешено использовать число 1, операцию сложения, операцию взятия обратного числа и скобки, для указания приоритета операций. Для удобства исключим 0 из множества задач этого исполнителя.

Алгоритмической сложностью $C_2(q)$ рационального числа q относительно такого исполнителя естественно назвать минимальное число операций в выражении, которое равно q. Например, $q = 5/6 = (1+1)^{-1} + (1+1+1)^{-1}$, поэтому $C_2(5/6) \leqslant 6$ (четыре операции сложения и две операции обращения)

[3]Будем считать, это в эта ячейка памяти элементарна и может принимать конечное число состояний и в ней могут хранится не все, а лишь некоторые рациональные числа. А именно, будем считать, что модуль числителя и знаменателя ограничены некоторым фиксированным числом.

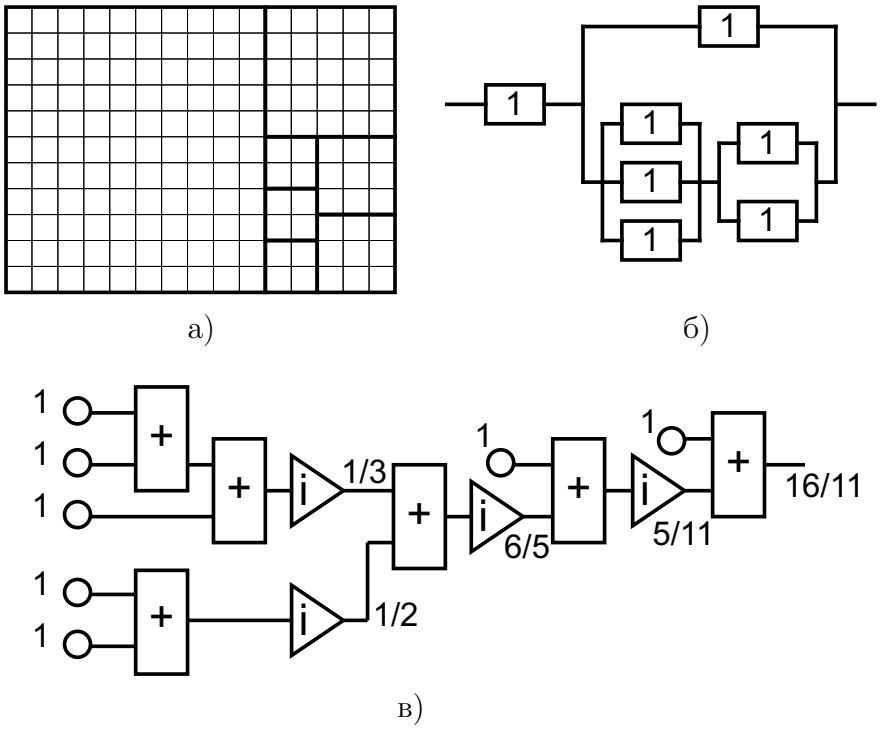


Рис. 21.3: Различные способы представления рационального числа $q = 16/11$. На рисунке (а) показано разбиение прямоугольника с отношением сторон 16/11 на квадраты. На рисунке (б) показана соответствующая разбиению электрическая схема из единичных резисторов с сопротивлением $R = 1 + (1^{-1} + ((1^{-1} + 1^{-1})^{-1} + (1^{-1} + 1^{-1} + 1^{-1})^{-1})^{-1})^{-1} = 16/11$. На рисунке (в) показана одна из возможных схем вычисления числа 16/11.

Запишем очевидные соотношения для функции C_2:

$$\begin{array}{lcl} C_2(q_1 + q_2) & \leqslant & C_2(q_1) + C_2(q_2) + 1, \\ C_2(1/q) & \leqslant & C_2(q) + 1, \\ C_2(1) & = & 0. \end{array}$$

Определение 21.7. *Рациональные числа можно получать как выражения, сконструированные из единиц, скобок, операций сложения, и функции $i(x) = 1/x$. Например $6/5 = i(i(1 + 1) + i(1 + 1 + 1))$. Минимальное число единиц в выражении, равном рациональному числу q, назовём* **эффективной сложностью рациональных чисел** *или просто* **сложностью рациональных чисел**. *Обозначим эту сложность как $C(q)$.*

Соотношения для эффективной сложности C будут отличаться от соотношений для C_2:

$$\begin{array}{lcl} C(q_1 + q_2) & \leqslant & C(q_1) + C(q_2), \\ C(1/q) & = & C(q), \\ C(1) & = & 1. \end{array}$$

Интересно заметить, что сложность C соответствует минимальному значению единичных резисторов, из которых можно сконструировать параллельно-последовательную схему с сопротивлением равным q.

Кроме того, $C(a/b)$, где $a, b \in \mathbb{N}$, равно минимальному количеству квадратов, на которое можно разрезать прямоугольник со сторонами a и b, если разрезать параллельно сторонам от края до края разрезаемого кусочка.

На рис. 21.3 (а,б) показан пример разбиения прямоугольника размера 16 и соответствующая этому разбиению электрическая схема из единичных резисторов.

$\mathbb{Q}$-схема №1

Рассмотрим схемы, в которых провода несут сигнал типа «рациональное число», и можно использовать три элемента: **1**, SUM и INVERSE.

Первый элемент является просто источником единичного сигнала. Обозначим его маленьким кругом. Следующий элемент SUM (сумматор) имеет два входа и один выход. Он суммирует два числа, которые получает на вход. Второй элемент INVERSE, обозначаемый треугольником, имеет один вход и один выход. Он инвертирует рациональное число, то есть меняет числитель и знаменатель местами:

$$\mathrm{SUM}(x_1, x_2) = x_1 + x_2, \quad \mathrm{INVERSE}(x) = 1/x.$$

Например, на рис. 21.3 (в) показана схема, вычисляющая 16/11 по формуле

$$\frac{16}{11} = 1 + \cfrac{1}{1 + \cfrac{1}{\cfrac{1}{1+1+1} + \cfrac{1}{1+1}}}.$$

В данном случае имеем следующее определение комбинационной сложности рационального числа.

Определение 21.8. *Комбинационной сложностью рационального числа q относительно элементов* {**1**, SUM, INVERSE} *называется минимальное число этих элементов, которых достаточно, чтобы сконструировать схему, вычисляющую q.*

Схему вычисления 16/11 на рис. 21.3 (в) можно значительно сократить, если использовать ветвление выходов. Попробуйте найти схему, с минимальным числом элементов, которая вычисляет 16/11.

Лекция 22

К теореме Геделя

В книге [13] дано определение понятия *формул языка первого порядка*, и их частный случай — *формул арифметики* (это выражения, состоящие из булевых операций И, ИЛИ, НЕ, символа равенства, переменных x_1, x_2, ..., принимающих значения из множества $\mathbb{N}$, кванторов $\exists$ и $\forall$, цифр, знаков умножения и сложения, скобок, ...).

От проблемы останова к теореме Геделя

В лекции 3 была представлена схема доказательства неразрешимости проблемы останова (невычислимости функции Is-Applicable).

Но давайте рассмотрим алгоритм 22.43, который претендует быть решением этой задачи. Попробуйте самостоятельно найти причину, по которой данный алгоритм не есть решение задачи останова.

Алгоритм 22.43 Алгоритм, который *почти* решает задачу останова.

```
1: function Is-Applicable'(M, X)
2:     A ← Program-To-Formula(M, X)          ▷ «Программа M остановится.»
3:     A_n ← '¬ (' + A + ')'                 ▷ «Программа M не остановится.»
4:     proof ← ∅
5:     while true do
6:         proof ← Next-Proof(proof)
7:         if Is-Proof(proof, A) then
8:             print 'Программа M остановится.'
9:             break
10:        end if
11:        if Is-Proof(proof, A_n) then
12:            print 'Программа M никогда не остановится.'
13:            break
14:        end if
15:    end while
16: end function
```

Приведённый алгоритм Is-Applicable' основывается на двух утверждениях.

1. Пусть дана программа $\mathcal{M}$ и входные данные X для некоторого Тьюринг полного исполнителя $\mathbf{I}_1$. Тогда можно записать формулу $\alpha(X)$, соответствующую утверждению «Программа $\mathcal{M}$ применима к входным данным X». Более того, существует алгоритм

PROGRAM-TO-FORMULA($\mathcal{M}, X$), который генерирует эту формулу. Описание того, как получить эту формулу дано в книге [12].

2. Процесс доказательства формализуем, а именно, можно разработать формальные языки формул и доказательств и алгоритм IS-PROOF($proof, A$), который проверяет, что слово $proof$ является корректным доказательством и оно доказывает формулу A.

Неочевидно, что данные утверждения верны, но они доказаны. Доказано также, что задача останова неразрешима, поэтому алгоритм IS-APPLICABLE' не может её решать. Это может значить только одно: среди утверждений вида «Программа $\mathcal{M}$ никогда не остановится» встречаются такие, которые нельзя ни доказать, ни опровергнуть. Причём ясно, что в этих случаях программа $\mathcal{M}$ не останавливается, поскольку если программа останавливается, это всегда можно доказать, предъявив лог пошаговой работы исполнителя от начала работы до момента останова.

Утверждения, которые нельзя ни доказать, ни опровергнуть, присутствуют в любой теории, содержащей в себе теорию арифметики. Для этих утверждений либо A либо $\neg A$ можно взять в качестве дополнительной аксиомы.

В нашем случае ясно, что только $\neg A$ может быть взято в качестве дополнительной аксиомы, так как соответствующая программа $\mathcal{M}$ точно не останавливается (как уже было сказано, если программа останавливается, то это всегда можно доказать).

Но при этом, если положить, что программа $\mathcal{M}$ останавливается, противоречия в теории не возникает (иначе бы мы получили доказательство того, что программа не останавливается, методом от противного).

Это довольно странно.

Добавляем аксиому о том, что программа $\mathcal{M}$ останавливается. Шаг на котором программа якобы останавливается, мы предъявить не можем. А если предъявим, сразу получим противоречие. Поэтому не предъявляем, а развиваем себе теорию дальше, как ни в чём не бывало.

«Заявили о существовании некого объекта. Верим, что он существует, и из этого строим теорию дальше. Но при этом прекрасно понимаем, что мы не сможем его предъявить. Не потому, что пока не нашли. А потому, что мы знаем, что его нельзя найти. Если мы его предъявим, то сразу получим противоречие.» – какая-то жутко неконструктивная математика.

Ниже приведён парадокс, который ещё ярче демонстрирует эту странность. Он касается утверждения вида «Программа $\mathcal{M}$ останавливается», которое не может быть ни ложным, ни истинным.

Остановится ли данная программа?

Будем считать далее, что входные данные X есть пустая стока, и будем опускать этот аргумент в функции PROGRAM-TO-FORMULA.

Рассмотрим функцию MAIN в алгоритме 22.44. Закончит ли она своё выполнение или зависнет? Ясно, что остановиться она не может, так как вариант останова один — если будет найдено доказательство того, что она зависнет (строка 23). Значит она не останавливается, а зависает. Итак, методом от противного мы доказали, что она зависнет. Значит доказательство её зависания есть и, следовательно, эта программа должна остановиться (строка 28). Получили противоречие.

Или с другого конца. Пусть она останавливается. То, что какая-то программа останавливается всегда можно доказать, просто предъявив лог пошаговой работы исполнителя. Значит в некоторый момент времени переменная *proof* будет содержать это доказательство, и алгоритм ... зависнет. Стоп! Мы предположили, что она останавливается, и пришли к тому, что она зависнет. Значит предположение не верно и она зависнет. Это и есть доказательство того, что она зависнет. Это доказательство существует. Значит когда-нибудь переменная *proof* будет содержать это доказательство и ... программа остановится.

Итак, имеем парадокс.

Удивительным образом время работы строго заданного детерминированного механического процесса вычислений оказывается неопределённым — то ли конечным, то ли бесконечным.

Программы и исполнители живут независимо от того, какие мы строим теории у себя в головах. Логика работы исполнителя строго задана и она не меняется от того, добавляем мы новые аксиомы в теорию арифметики или нет. И казалось бы каждая из них должна либо точно останавливаться на некотором шаге, либо точно зависать. Третьего не дано.

Варианты объяснения парадокса:

- Тьюринг-полного исполнителя, для которого возможна реализация подобного псевдокода, не существует. Например, невозможно реализовать функцию а) Generate-Program; б) Program-To-Formula; в) ...
- При доказательстве мы использовали трюк, который нельзя формализовать и/или написать для него вычислимую функцию Is-Proof.
- ...

Попробуйте самостоятельно разрешить этот пародокс. Следующие книжки могут оказаться полезными: [].

Доказательство вычислимости Program-To-Formula самое сложное. Оно приведено в [12] и оно, видимо, верное. Нужно проверить подходит ли оно для данного случая.

Доказательство существования Generate-Program, видимо, привести несложно. Необходимо показать, что для любой функции `main` можно сконструировать подходящую функцию `generate`. Нужно просто подробно описать Тьюринг-полного исполнителя, для которого программа представляет некоторое подобие языка Си (для удобства и Тьюринг-полноты без указателей, без оператора взятия адреса, без символов «бэкслеш» и новой строки) и явно описать алгоритм конструирования. Идеи можно взять из приведённых ниже кодов 22.1 и 22.2. В этих кодах сделано несколько переносов строк, которые следует исключить. Перед такими переносами стоит символ «бэкслеш».

В коде 22.1 приведён пример программы на языке Си, которая печатает саму себя.

Программа 22.1: Программа, печатающая свой текст.

```
int f(int x); int main() { \
char q = '"'; char *s = "int f(int x); int main() { \
char q = '%c'; char *s = %c%s%c; printf(s, q, q, s, q); } \
int f(int x) { return 2 * x; }"; \
printf(s, q, q, s, q); } \
int f(int x) { return 2 * x; }
```

В коде 22.2 приведён ещё один пример программы на языке Си, которая печатает саму себя. В ней показано как можно справится с символом «бэкслеш» и символом новой строки.

Алгоритм 22.44 Алгоритм–загадка

```
1: function Generate-Program
2:     Возвращает текст самой себя и всех приведённых ниже функций.
3: end function
4: function Program-To-Formula(M)
5:     Возвращает слово, являющееся формальной записью утверждения «Программа M
       (без входных данных) останавливается.»
6: end function
7: function Is-Proof(proof, A)
8:     Возвращаем true, если proof является корректной формальной записью доказа-
       тельства утверждения A.
9:     Иначе, возвращаем false.
10: end function
11: function Next-Proof(proof)
12:     Возвращаем следующее за proof (в лексикографическом порядке) слово, являю-
        щееся корректной записью доказательства какого-то утверждения.
13: end function
14: function Main
15:     M ← Generate-Program()
16:     proof ← ∅
17:     A ← Program-To-Formula(M)          ▷ Формула «Программа M остановится.»
18:     A_n ← '¬ (' + A + ')'              ▷ Формула «Программа M не остановится.»
19:     while true do
20:         proof ← Next-Proof(proof)
21:         if Is-Proof(proof, A) then
22:             print 'Программа M остановится'.
23:             while true do                          ▷ Бесконечный цикл
24:             end while
25:         end if
26:         if Is-Proof(proof, A_n) then
27:             print 'Программа M никогда не остановится'.
28:             break
29:         end if
30:     end while
31: end function
```

Программа 22.2: Программа, печатающая свой текст (вариант 2).

```
#include <stdio.h>
int f(int x); int main() { \
char e='\\'; char r='\n'; char q = '"'; \
char *s = "#include<stdio.h>%cint f(int x); int main() { \
char e='%c%c'; char r='%cn'; char q = '%c'; char *s = %c%s%c; \
printf(s, r, e, e, e, q, q, s, q); } int f(int x) { return 2 * x; }"; \
printf(s, r, e, e, e, q, q, s, q); } int f(int x) { return 2 * x; }
```

Лекция 23

Разбор грамматик

Краткое описание: На данной лекции мы рассмотрим классические задачи «правильные скобочные выражения», «калькулятор», «схема сопротивлений» и рассмотрим общие принципы разработки программ, решающих задачу разбора выражений на некотором формальном языке, заданном с помощью правил вывода.

Для усвоения материала данной лекции необходимо представлять, что такое грамматика и правила вывода (см. лекцию 4 на стр. 76), а также понимать устройство потоков ввода вывода (см. стр. 122).

Задача «Правильные скобочные выражения»

Приведём точную формулировку задачи «правильные скобочные выражения».

Задача Л23.1. (Правильные скобочные выражения) Напишите программу, которая получает на вход слово в алфавите $B = \{(,\ )\}$, и выводит `YES` или `NO`, в зависимости от того, принадлежит введённое слово языку правильных скобочных выражений или нет. Язык правильных скобочных выражений задаётся с помощью правил вывода[1]:

```
S ::= ∅
S ::= '(' S ')' S
```

stdin	stdout
`()()`	`YES`
`)(`	`NO`
`((()))()()()`	`YES`
`((()))(((())`	`NO`

Одно из простейших решений этой задачи основано на следующем утверждении:

Утверждение 23.1. *В любом префиксе правильного скобочного выражения открывающих скобок больше, чем закрывающих.*

Это утверждение несложно доказывается по индукции. Оно позволяет провести полезную аналогию между скобочным выражением и путём в графе. Посмотрите на рис. 23.1. Справа

[1] В данных правилах скобки являются терминальными символами, то есть символами с фиксированным значением. Символ `S` является нетерминальным символом, он соответствует некоторому множеству слов в алфавите терминальных символов, а именно, множеству правильных скобочных выражений. Говорят, что правильные скобочные выражения выводятся из символа `S`.

на рисунке изображён направленный граф и выделен один путь из A в B. При движении от точки A к точке B мы следуем простому правилу — если очередная скобка открывающая, то движемся по диагонали вверх, если закрывающая — то по диагонали вниз. Номер уровня l в первом случаем увеличивается, а во втором — уменьшается. Выражение является правильным, если в конце мы оказываемся на нулевом уровне (число скобок открывающих равно количеству скобок закрывающих), в во время движения ни разу не опускались ниже нулевого уровня.

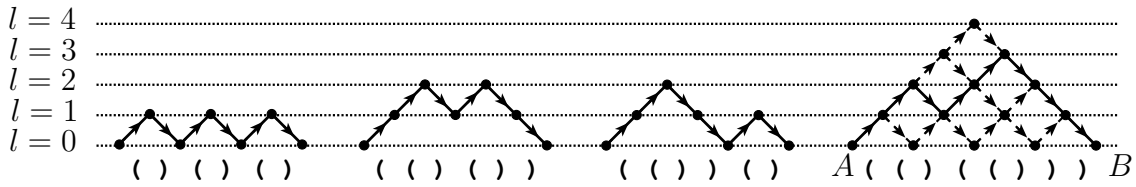


Рис. 23.1: Отображение правильных скобочных выражений в пути в графе.

В коде 23.1 показана программа на Си, решающая поставленную задачу.

Обратите внимание на то, что предложенный алгоритм определения правильности скобочного выражения *однопроходный*. Символы по очереди считываются из входного потока и сразу «забываются» программой. Для получения ответа достаточно на каждом шаге хранить значение уровня l.

Программа 23.1: Простейшее решение задачи Л23.1.

```
#include<stdio.h>
int main() {
    int c, l = 0;
    do {
        c = getc(stdin);
        if ( c == '(' ) l++;
        if ( c == ')' ) l--;
        if ( l < 0 ) break;
    } while ( c != EOF && c != '\n' );
    if ( l == 0 )
        printf("YES\n");
    else
        printf("NO\n");
   return 0;
}
```

Приведённый простой код, решает поставленную задачу. Но этот код не даёт ключа к решению задач подобного типа. Давайте усложним язык скобочных выражений и добавим новые типы скобок.

Задача Л23.2. (Правильные сложные скобочные выражения) Напишите программу, которая получает на вход слово в алфавите $B = \{$`(`, `)`, `[`, `]`, `{`, `}`$\}$, и выводит `YES` или `NO`, в зависимости от того, принадлежит введённое слово языку правильных скобочных выражений или нет. Язык правильных скобочных выражений задаётся с помощью правил:

```
S ::= ∅
S ::= '(' S ')' S
S ::= '[' S ']' S
S ::= '{' S '}' S
```

stdin	stdout
[()]	YES
[)	NO
([]{})[]{}[]	YES
([)]	NO

Решение этой задачи невозможно получить «малыми изменениями» из решения предыдущей задачи. Классическое решение (см. код 23.2) данной задачи основано на структуре данных «стек». В стек помещаются лишь открывающие скобки. Логика работы проста. Если очередной считанный символ `c` является открывающей скобкой, то он помещается в стек. Если он является закрывающей скобкой, то из стека вынимается скобка `d`. Это и есть парная скобка для считанной закрывающей скобки и эти скобки должны быть одного типа. При поступлении символа конца потока, следует проверить что стек пуст. Это будет означать, что не осталось открывающих скобок, для которых не нашлось пары. Если во время работы алгоритма возникает ситуация, когда стек пуст, а на вход приходит закрывающая скобка, то цикл считывания прекращается и выводится сообщение об ошибке. В программе 23.2 предполагается, что функция `pop` возвращает `0`, если стек пуст. Реализуйте функции `pop` и `push` самостоятельно.

Программа 23.2: Простейшее решение задачи Л23.2.

```
#include<stdio.h>
/* Пропущено определение функций pop и push */
int main() {
    int c, l = 0;
    do {
        c = getc(stdin);
        if ( c == '('  || c == '[' ||  c == '{' ) push(c);
        if ( c == ')'  || c == ']' ||  c == '}' ) {
            int d = pop();
            switch ( d ) {
                case '(';  if (c == ')') continue; else break;
                case '[';  if (c == ']') continue; else break;
                case '{';  if (c == '}') continue; else break;
            }
            break; // d is not correct
        };
    } while (c != EOF && c != '\n');

    if ( pop() == 0 && (c == EOF || c == '\n') )
        printf("YES\n");
    else
        printf("NO\n");
```

```
    return 0;
}
```

Рекурсивный разбор грамматик

Решение задач Л23.1 и Л23.2 можно построить непосредственно по правилам вывода описанных в задачах грамматик.

Поставим в соответствие символу `S` функцию `int read_S()`, которая считывает из потока ввода правильное скобочное выражение и возвращает 1.

Функция `read_c(int c)` предназначена для считывания одного ASCII-символа. Если говорить в терминах теории грамматик, эта функция считывает фиксированный *терминальный* символ. Она получает в аргументе символ `c`, который требуется взять из потока ввода. Если во входном потоке следующий символ равен данному, то функция считывает его и возвращает результат 1, если же во входном потоке следующий символ другой, то функция оставляет его в потоке несчитанным и возвращает результат 0.

Основная идея заключается в прямом отображении правила

`S ::= '(' S ')' S`

в логическое выражение на языке Си:

```
read_c( '(' ) && read_S() && read_c( ')' ) && read_S();
```

Если есть несколько правил вывода символа `S`, то соответствующие им логические выражения следует объединить логическим оператором «ИЛИ». Правилу «`S` $\to\varnothing$» соответствует выражение 1, поэтому функцию `read_S` можно было бы определить так:

```
(read_c( '(' ) && read_S() && read_c( ')' ) && read_S())   ||   1;
```

Соответственно, при вычислении этого логического выражения программа попробует прочитать значение символа `S` сначала согласно правилу вывода «`S` $\to$`'(' S ')' S`», и затем, если не получится[2], согласно правилу вывода «`S` $\to\varnothing$».

Конечный код несколько сложнее, так как в случае неуспешного считывания символа `s` согласно первому правилу, необходимо *считывающий курсор* переместить на прежнее место. Решение задачи Л23.1 с помощью функций `read_S` и `read_c` показано в коде 23.3. Вместо потока ввода в приведённой программе используется глобальная переменная `char *p`, которая указывает на текущий считываемый символ.

[2]Напомним, что в языке Си выражение вида «`( A1 || A2 || A3 || A4 || ... )`» вычисляется следующим образом: последовательно вычисляются логические выражения `A1`, `A2`, ... до первого выражения, равного значению «истина» ($\neq 0$). Выражения, которые следуют после истинного, не вычисляются, и возвращается значение «истина». Значение «ложь» возвращается лишь если все выражения цепочки имеют значение «ложь».

Вычисление выражение вида «`( A1 && A2 && A3 && A4 && ... )`» происходит по другому алгоритму: последовательно вычисляются все логические выражения `A1`, `A2`, ... до первого выражения, равного значению «ложь» ($= 0$). Выражения, которые следуют после ложного, не вычисляются, и возвращается результат «ложь». Значение «истина» возвращается, если все выражения цепочки истинны.

Программа 23.3: Второе решение Л23.1 на основе функции `read_s`.

```
#include<stdio.h>
#define N 1000
char str[N];
char *p = str;
int read_c(int c) {
    if ( *p == c) {
        p++;
        return 1;
    } else {
        return 0;
    }
}
int read_S() {
    char *start = p; // текущая позиция в строке символов
    return
    ((read_c('(') && read_S() && read_c(')') && read_S()) ||
                ((p=start),0) )
    || 1 ;
}
int main() {
    fgets(str, N, stdin);
    if ( read_S() &&  (*p == '\n' || *p == 0)) {
        printf("YES\n");
    } else {
        printf("NO\n");
    }
    return 0;
}
```

Программа сначала целиком зачитывает выражение, а затем, перемещаясь по считанной строке с помощью инкрементирования указателя `p`, осуществляет проверку, непосредственно следуя правилу вывода «`S → '(' S ')' S`».

Указатель `p` после вызова функции `read_S` сместится вправо только в том случае, если начало строки является правильным скобочным выражением. Функция `read_S` старается «отъесть» (считать) от введённой строки как можно больше символов, являющихся в совокупности правильным скобочным выражением.

После запуска функции `read_S` необходимо просто проверить, что достигнут конец строки, то есть указатель `p` указывает на символ `'\n'` или `'\0'`.

С помощью малого изменения кода 23.3 можно получить решение задачи Л23.2. Для этого необходимо заменить определение функции `read_S` на следующее:

```
int read_S() {
    char *start = p; // текущая позиция в строке символов
    return
      ( (read_c('(') && read_S() && read_c(')') && read_S()) ||
          ((p=start),0) )
```

```
   || ( (read_c('{') && read_S() && read_c('}') && read_S()) ||
          ((p=start),0) )
   || ( (read_c('[') && read_S() && read_c(']') && read_S()) ||
          ((p=start),0) )
   || 1 ;
}
```

Функция `read_S` всегда возвращает 1 (**true**), поскольку пустое слово является корректным выражением и его всегда можно «считать» (то есть просто ничего не считать).

Не совсем правильно применять зачитывание строки целиком в память. Один из недостатков заключается в том, что программа может обрабатывать строки ограниченной длины (в нашем примере не более 1000).

А можно работать непосредственно с потоком ввода, как мы это делали в программе 23.1. При этом придётся использовать функцию `ungetc(int c, FILE *stream)`[3].

Для удобства, несколько изменим смысл функции `read_S`. Пусть если поток ввода содержит неправильное скобочное выражение, то функция `read_S` считает лишь первую часть выражения, которая *может быть началом правильного скобочного выражения*. Она считывает из потока ввода символ за символом, пока считанная последовательность символов может быть началом правильного скобочного выражения. В момент, когда достигается конец строки или появляется символ, который «портит строку» (в нашем случае это лишняя закрывающая скобка, которая не сможет быть спарена ни с какой открывающей), функция `read_S` заканчивает работу, предварительно вернув считанный «плохой» символ в поток ввода.

Если то, что ей удалось считать, является целым правильным скобочным выражением, она возвращает 1, а иначе — 0. В коде 23.4 приведено решение задачи Л23.1, основанное на функции `read_S` с указанной логикой работы. Эта программа не использует зачитывание всего выражения в память и может корректно работать на очень длинных входных словах.

Программа 23.4: Третье решение задачи Л23.1.

```
#include<stdio.h>
int read_c(int c) {
    int a = getc(stdin);
    if ( a == c) {
        return 1;
    } else {
        if (c != EOF) ungetc(a, stdin);
        return 0;
    }
}
int read_S() {
     int empty = 1;
     return
     ( read_c('(') &&
           ( ((empty = 0),1) && read_S() && read_c(')') && read_S() ) )
```

[3]Кроме функции `ungetc(c, stream)` есть ещё функция `fseek( stream, offset, SEEK_SUR )`, которая позволяет сместить считывающий курсор на одну или несколько позиций назад. Но не все потоки позволяют выполнять с ними указанные действия.

```
     || empty ;
}
int main() {
    int c;
    if ( read_S() && (c = getc(stdin)) == '\n' ) {
        printf("YES\n");
    } else {
        printf("NO\n");
    }
    return 0;
}
```

Задача «Схема сопротивлений»

Задача Л23.3. (Схема сопротивлений) Параллельно-последовательная схема сопротивлений (ПП-схема) это или параллельно соединённые ПП-схемы (символ `A`), или последовательно соединённые ПП-схемы (символ `B`), или один резистор (символ `R`). Описания ПП-схем выводятся из символа `S` согласно следующим правилам[4]:

```
S ::= A | B | R
A ::= '(' S+ ')'
B ::= '[' S+ ']'
R ::= REALNUMBER
```

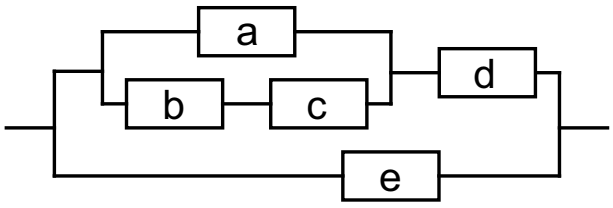


Рис. 23.2: Визуальное представление схемы с описанием «`[([a (b c)] d) e]`».

В этих правилах подразумевается, что повторяющиеся описания `S`, соответствующие выражению `S+` во втором и третьем правиле, отделяются друг от друга пробельными символами. Более точно, второе и третье правила записываются так:

```
A     ::= '(' S (SPACE S)* ')'
B     ::= '[' S (SPACE S)* ']'
SPACE ::= (' ' |  '\t' |  '\n')+
```

На вход поступает строчка с описанием ПП-схемы. Необходимо найти сопротивление схемы. Сопротивление резистора (символ `REALNUMBER`) соответствует формату `"%lf"` записи чисел типа `double`. Сопротивление схемы определяется рекурсивным способом. Сопротивление R схемы, составленной из последовательно соединённых схем с сопротивлениями R_1, R_2, …, R_m, равно $R = R_1 + R_2 + \ldots + R_m$. Сопротивление R схемы, составленной из параллельно соединённых схем с сопротивлениями R_1, R_2, …, R_m, равно $R = (R_1^{-1} + R_2^{-1} + \ldots + R_m^{-1})^{-1}$.

Будем действовать согласно намеченной схеме — писать рекурсивные функции, логика работы которых максимально близка к заданным правилам. В данной задаче возникает одна

[4]Здесь, также как и раньше, мы используем специальные символы «`|`» (вертикальная черта), для обозначения союза ИЛИ, и «`+`» (знак плюс) для обозначения повторяемости символа 1 или большее число раз.

дополнительная особенность — необходимо не просто проверить входное выражение на корректность, а вернуть результат.

Результат, в который должен трансформироваться некоторый символ (в нашем случае `S`, `A`, `B` или `R`), называется **значением символа**. Мы будем писать функции, которые получают в аргументе адрес, куда следует поместить полученное значение. Возвращать они будут 1 или 0 в зависимости от того, удалось считать символ или нет.

Основной «каркас» программы приведён в коде 23.5. Этот код почти не нуждается в комментариях. Функция `read_S` определена прямо в соответствии с правилами вывода, функция `read_c` осталась прежней. Самый непростой код у функции `read_R`, которая должна считать с позиции, на которую указывает переменная `p`, действительное число, поместить результат по адресу `r` и сместить указатель `p` до первого символа после считанного числа. Для этого используется функция `sscanf`, которая работает также как и обычная функция `scanf`, только считывает данные не из стандартного потока входа, а из строки, указатель на которую получает в первом аргументе. Также используется специальный формат `%n`, который ничего не считывает, а соответствует количеству символов, считанных от момента начала выполнения функции `scanf` (`sscanf`, `fscanf`) до момента достижения символов `%n` в строке формата ввода.

Программа 23.5: Решение задачи Л23.3.

```
#include<stdio.h>
#include<ctype.h>
#define N 1000
char str[N];
char *p = str;
int read_S(double *r);
int read_A(double *r);
int read_B(double *r);
int read_R(double *r);
int read_c(int c);
int eat_space() {
   while ( isspace(*p) ) p++;
   return 1;
}
int read_S(double *r) {
    eat_space();
    return read_A(r) || read_B(r) || read_R(r);
}
int read_R(double *r) {
    int shift;
    // формат %n "возвращает", сколько символов считано функцией scanf
    if ( sscanf(p, "%lf%n", r, &shift ) !=0 ) {
        p += shift;
        return 1;
    } else {
        return 0;
    }
}
```

```
int read_c(int c) {
    if ( *p == c) { p++; return 1; }
    else return 0;
}
int main() {
    double r = 0;
    fgets(str, sizeof(str), stdin);
    if ( read_S(&r) ) {
        printf("Result = %lf\n", r);
    } else {
        printf("Parse error.");
    }
    fprintf(stderr, "Unread tail is = %s\n", p);
    return 0;
}
```

Две функции — `read_A` и `read_B` несут в себе ключевую логику подсчёта сопротивления схемы. Приведём код лишь одной из них (см. код 23.6). Код второй функции несложно написать, действуя по аналогии.

Программа 23.6: Функция `read_A`.

```
int read_A(double *r) {
    char *start = p;
    eat_space();
    if ( *p == '(' ) {
        double q;
        p++;
        *r = 0;
        if ( read_S( r ) ) {
            while ( read_S( &q ) ) {
                *r += q;
            }
            if ( *p == ')' ) {
                p++;
                return 1;
            }
        }
    }
    p = start;
    return 0;
}
```

Приведём напоследок интересную реализацию (см. код 23.7) функций `read_A` и `read_B`, основанную на функции общего назначения `read_sequence`. Эта функция соответствует символу кратности повторения `+`. Она возвращает 1, если ей удаётся прочитать указанный символ 1 или большее число раз. При этом она производит указанные вычисления с последовательностью прочитанных значений, а именно, с помощью указанной в её аргументе функции `aggregate` она *агрегирует* вычисленные значения последовательности в одно результирующее

значение. Первый аргумент этой функции равен некоторой функции `read_x`, которая отвечает за чтение одного символа.

Программа 23.7: Улучшенный вариант функций `read_A` и `read_B`.

```
int read_sequence( int read_x(double*),
                   double aggregate(double,double),
                   double *r ) {
    if ( read_x( r ) ) {
        double x;
        while ( eat_space() && read_x( &x )) {
            *r = aggregate( *r, x );
        }
        return 1;
    } else {
        return 0;
    }
}
double aggregate_sum(double agr, double x) {
    return agr + x;
}
double aggregate_invsum(double agr, double x) {
    return 1/(1/agr + 1/x);
}
int read_A(double *r) {
    char *start = p;
    return  (    read_c( '(' )
              && read_sequence(read_S, aggregate_sum, r)
              && read_c( ')' )
             ) ||  ((p = start),0) ;
}
int read_B(double *r) {
    char *start = p;
    return  (    read_c( '[' )
              && read_sequence(read_S, aggregate_invsum, r)
              && read_c( ']' )
             ) ||  ((p = start),0) ;
}
```

В коде 23.7 несложно увидеть прямое соответствие логики функций правилам вывода.

В данном примере мы использовали технику **функций обратного вызова** (**callback functions**). Суть этой техники заключается в создании функций общего назначения, логика работы которых зависит от некоторых базовых действий. Эти базовые действия представлены указателями на функции и передаются функции общего назначения в аргументах.

Функции общего назначения `read_sequence` передаются два «базовых действия»: считывание некоторого символа и агрегация значения, и она с их помощью, считывает последова-

тельность символов, вычисляя их «суммарное» значение.

Аналогичная техника используется в функциях обхода деревьев `traverse` (см. задачу С12.13 на стр. 239). Эти функции «обходят» все узлы дерева и применяют указанное в аргументе действие к каждому узлу дерева.

Полезна также следующая аналогия — функции общего назначения X передаются некоторые данные на обработку, а также функция *обратной связи* Y. Тот, кто вызывает функцию X, предоставляет средства для обратной связи. Посредством вызовов функции Y из функции X можно как бы «связываться» с тем, кто вызвал X.

Задача «Калькулятор»

Задача Л23.4. (Калькулятор) Напишите программу, которая получает на вход арифметическое выражение — слово в алфавите $B = \{\text{␣}, \texttt{.}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{0}, \ldots, \texttt{9}, \texttt{(}, \texttt{)}\}$ — и выводит значение этого арифметического выражения, если оно корректно, или строку `Parse error`, если не корректно.

```
S ::= (S ('+' | '-') )? B
B ::= (B ('*' | '/') )? A
A ::= REALNUMBER | '(' S ')'
```

stdin	stdout
1.5+2*3	7.5
-1+2*(3+4*5)*6+7	282
-1+2*(3+4/5)/6+7	7.26667
-2-5+	Parse error.
2.0*3/4/5	0.3

Условно назовём указанные правила вывода арифметических выражений *левоассоциативными*. Давайте попробуем решить эту задачу также, как и задачу расчёта сопротивления схемы. Грамматические правила отображаются в следующий код:

Программа 23.8: Функции, соответствующие левоассоциативным правилам.

```
int read_S(double *r) {
    char *start = p;
    double q;
    return
   ( (read_S(r) && read_c('+') && read_B(&q) && (*r += q, 1) ||
                        ((p=start),0) )
|| ( (read_S(r) && read_c('-') && read_B(&q) && (*r -= q, 1) ||
                        ((p=start),0) );
}
int read_B(double *r) {
    char *start = p;
    double q;
    return
   ( read_B(r) && read_c('*') && read_A(&q) && (*r *= q, 1) ||
```

```
                        ((p=start),0) )
|| ( read_B(r) && read_c('/') && read_A(&q) && (*r /= q, 1) ||
                        ((p=start),0) );
}
int read_A(double *r) {
    char *start = p;
    double q;
    return
      ( read_double(r)  || ((p=start),0) )
   || ( read_c('(') && read_S(&r) && read_c(')') ||
                        ((p=start),0) );
}
```

Рекурсивные функции (см. код 23.8), соответствующие левоассоциативным правилам вывода, не работают. В частности, при выполнении функция `read_S` каждый раз рекурсивно вызывает себя, не меняя при этом состояния (значения указателя `p`), что приводит к бесконечной цепочке вложенных рекурсивных вызовов `read_S`.

Таким образом, на данном примере мы видим, что не всегда непосредственное отображение правил вывода в код рекурсивных функций может привести к требуемому результату.

Попробуем разрешить возникшую проблему. Для того нам потребуется разобраться с понятиями право- и левоассоциативности.

Рассмотрите два набора правил, описывающих синтаксис арифметических выражений — левоассоциативные и правоассоциативные:

```
S ::= (S ('+' | '-') )? B        S ::= B ( ('+' | '-') S ) ?
B ::= (B ('*' | '/') )? A        B ::= A ( ('*' | '/') B ) ?
A ::= REALNUMBER | '(' S ')'     A ::= REALNUMBER | '(' S ')'
```

Чем они отличаются друг от друга?

Ответ: различие заключается в последовательности выполнения операций. Рассмотрим для примера следующее выражение:

```
1 + 2 * 3 / 4 / 5 - 6 - 7 - 8.
```

Согласно первым правилам вычисления будут проходить так, как будто скобки расставлены следующим образом:

```
(((1 + (((2 * 3) / 4) / 5)) - 6 ) - 7) - 8).
```

Такая расстановка скобок, фиксирующая последовательность выполнения операторов, считается правильной. Согласно второму набору правил, скобки расставятся другим образом:

```
(1 + ((2 * (3 / (4 / 5))) - (6 - (7 - 8)))).
```

Бинарные операторы, для которых скобки неявно расставляются первым способом, называются *левоассоциативными*, а если скобки неявно расставляются вторым способом, то *правоассоциативными*.

Приведём более точное определение. Пусть символ * обозначает один из операторов с *равным приоритетом*. Рассмотрим выражение без скобок:

```
a * b * c * ... * z.
```

Используемые операторы называются **левоассоциативными**, если поледовательность выполнения операторов определяется следующей расстановкой скобок:

```
((..((a * b) * c) * ...) * z)
```

И используемые операторы называются **правоассоциативными**, если поледовательность выполнения операторов определяется следующей расстановкой скобок:

```
(a * (b * (c * ... * z)...)))
```

Указанная последовательность выполнения имеет место только в случае, если приоритеты операций одинаковы. Если же в выражении используются операции с разным приоритетом, то сначала вычисляются части выражения (последовательности элементов выражения), связанные бинарными операциями, имеющими самый высокий приоритет. Затем рассматриваются операции со следующим приоритетом, и так далее.

Арифметические операторы (сложение, вычитание, умножение, деление) принято рассматривать как левоассоциативные.

Определение 23.1. **Синтаксическим деревом разбора выражения** *называется дерево, в узлах которого находятся токены, на которое разбилось данное выражение. Узлы, не являющиеся листьями, содержат токены, обозначающие операторы. В листьях дерева находятся токены, являющиеся элементарными. В случае арифметических выражений в листьях дерева находятся числа или переменные, а в остальных узлах символы, обозначающие арифметические операции. С каждым узлом дерева связано некоторое значение или действие. Значение (действие) каждого узла рекурсивно определяется через значения (действия), соответствующие дочерним узлам. В случае бинарной операции сложения, в узел с токеном* `'+'` *будет помещено значение, равное сумме значений, полученных в правом и левом ребёнке этого узла. Степень арности оператора определяется количеством детей соответствующего узла.*

На рисунке 23.3 показаны деревья синтаксического разора выражения `a*b*c*...*z`, и выражения `1+2*3/4/5-6-7-8` при разных предположениях о типе ассоциативности используемых операторов.

Непосредственное отображение левоассоциативных правил в код рекурсивных функций приводит к неработающему коду (коду, который вызывает переполнение стека из-за бесконечной последовательности рекурсивных вызовов).

Непосредственное отображение правоассоциативных правил в код рекурсивных функций приводит к работающему коду, но операторы сложения и умножения при этом работают как правоассоциативные, что не соответствует общепринятому соглашению, и, в частности, приводит к неверной последовательности выполнения операторов в выражении $2 * 3/4/5$ и неверному результату.

Как разрешить возникшую проблему? Одна из идей заключается в следовании правоассоциативным правилам, но при этом двигаться по токенам выражения нужно не слева направо, а справа налево. Так имеет смысл поступать, если все операторы, используемые в выражении, являются левоассоциативными. Неприятность заключается в том, что человеку удобнее некоторые операторы делать правоассоциативными, а другие — левоассоциативными.

Рассмотрим, например, выражение языка Си:

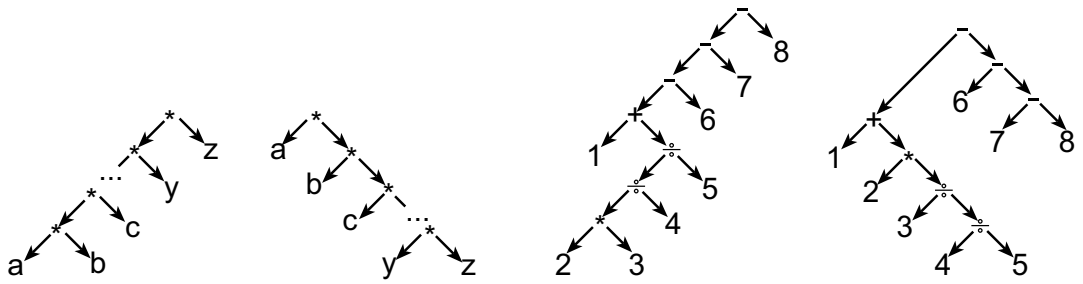


Рис. 23.3: (а,б) Синтаксическое дерево разбора выражения `a*b*c* ...*z` для случая, когда операция `*` левоассоциативная (а) и правоассоциативная (б).
(в,г) Синтаксическое дерево разбора выражения `1+2*3/4/5-6-7-8` с учётом того, что приоритет операторов «`*`» и «`/`» выше приоритета операторов «`+`» и «−», для случаев, когда операции левоассоциативны (в) и правоассоциативны (г).

```
a = b =  1 + c * d / e / f
```

В этом выражении используется три различных бинарных оператора: операторы умножения, деления, сложения и оператор присваивания. Приоритет операторов умножения и сложения здесь самый высокий. Следующий приоритет у оператора сложения. Затем у оператора присвоения. Оператор присвоения является правоассоциативным, а остальные операторы — левоассоциативными. В итоге, вычисления будут происходить в последовательности, задаваемой следующей расстановкой скобок:

```
(a = (b =  (1 + (((c * d) / e) / f)) ))
```

В случае, когда используются операторы с разным типом ассоциативности, следует использовать более хитрые приёмы. Один из таких хитрых[5] приёмов заключается в следующем. Пусть дан набор левоассоциативных операций с равным приоритетом. Напишем рекурсивную функцию, которая соответствует правоассоциативному правилу разбора этих операций. Добавим к этой функции ещё два аргумента — значение выражения слева (аргумент `left`) и ссылку на функцию, осуществляющий оператор стоящий слева от считывающего курсора (аргумент `op`).

Так, например, для сложения получим следующий код:

```
double OPPLUS (double a, double b) {return a+b;}
double OPMINUS(double a, double b) {return a-b;}
int read_S( double *r, double *left, double op(double,double)) {
    char *start = p;
    double l;
    return
        read_B(&l, NULL, NULL)
          && ((left!= NULL && l = op(*left, &l)) || 1)
             && read_c('+')
                && read_S( r, &l, OPPLUS)
```

[5]Описанный хитрый приём не часто используется на практике, так как он не обладает необходимой общностью применения.

```
                || ((p=start),0))
        ||
        read_B(&l, NULL, NULL)
          && ((left!= NULL && l = op(*left, &l)) || 1)
             && read_c('-')
                && read_S( r, &l, OPMINUS)
                    || ((p=start),0));
}
```

Эта функция считывает очередной символ `B` состоящий из элементарных элементов, связанных более высокоприоритетными операторами. Его значение помещается в переменную l. Это значение и значение символа `B`, который был прочитан ранее и который стоит слева от читаемого функцией read_S места, «соединяются» с помощью оператора `op`.

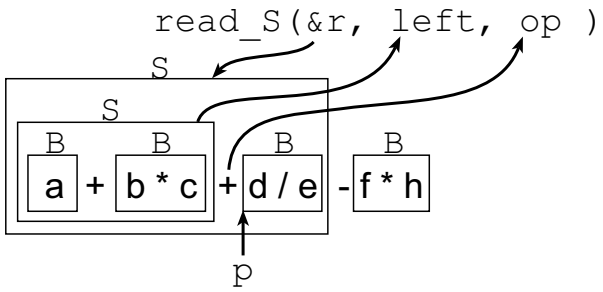


Рис. 23.4: Этап разбора выражения `a+b*c+d/e-f*h`. Считывающий курсор (указатель `p`) указывает на символ `d`. Вызвана функция `read_S`.

На рисунке 23.4 показано один из этапов разбора выражения `a+b*c+d/e-f*h`. Указатель показывает на символ `d`. Уже прочитано выражение `a+b*c`. Указатель на переменную, которое хранит значение этого выражения передаётся при очередном вызове функции `read_S` в аргументе `left`. Кроме того, функции передаётся оператор, который следует применить к `*left` и значению следующего прочитанного символа `B`. Следующий прочитанный символ `B` соответствует выражению `d/e`, его значение будет помещено в переменную `l`. Результирующее значение `op(*left, l)` будет помещено в ту же переменную `l` (в данном случае `l=*left+l`) и адрес на эту переменную будет передан в качестве аргумента `left` при следующем вызове функции `read_S`.

*Калькулятор общего назначения

Рассмотрим следующую задачу

Задача Л23.5. Напишите калькулятор общего назначения. На вход калькулятор получает список бинарных операторов и выражение, состоящее из чисел, идентификаторов переменных, скобок и бинарных операций из этого списка, обозначаемых одним символом. Каждый бинарный оператор, подаваемый на вход калькулятору, снабжается следующей информацией:

```
struct binop_info {
    int sym;  /* символ, обозначающий оператор */
    int prio; /* приоритет оператора */
```

```
    int left; /* 1- лево-,   0 -- правоассоциативный */
    /* указатель на функцию, выполняющую оператор */
   double (*op)(struct ast_node *left, struct ast_node *right);
};
```

Также на вход подаётся строка с выражением. Результатом должен быть результат вычислений данного выражения с использованием описанных операторов. При этом общее описание синтаксиса выражений таково:

```
S ::= '(' S ')'
S ::= S binop S
S ::= REALNUMBER | IDENTIFIER
```

Второе правило задаёт рекурсивное правило конструирования выражений, но не содержит информации о приоритетах и ассоциативности операций. Эта информация предоставляется в нашей задаче отдельно, во входных данных.

Например, на вход может быть подан список операторов, включающий арифметические операции, оператор возведения в степень, запятую, оператор и присвоения. Выражение, может быть, например, таким: «`a = b = 1 + 2^3 / (4 + 5 * 6) - 7, b = (b + 8) * a`»

Дерево синтаксического разбора этого выражения показана на рисунке 23.5.

Поскольку информация о приоритетах и ассоциативности операций нам заранее неизвестна, использовать напрямую метод рекурсивного синтаксического разбора не получится. Для решения данной задачи, нам потребуется больше информации о пройденной части выражения. Наиболее общим способом решения подобных задач является построение дерева синтаксического разбора, или абстрактное синтаксическое дерево. Вычисления в данном способе делаются после того, как всё выражение разобрано, а информация об операциях и значениях хранится в дереве.

c][cc]expr`a = b = 1 + 2^3 / (4 + 5 * 6) - 7, b = (b + 8) * a`

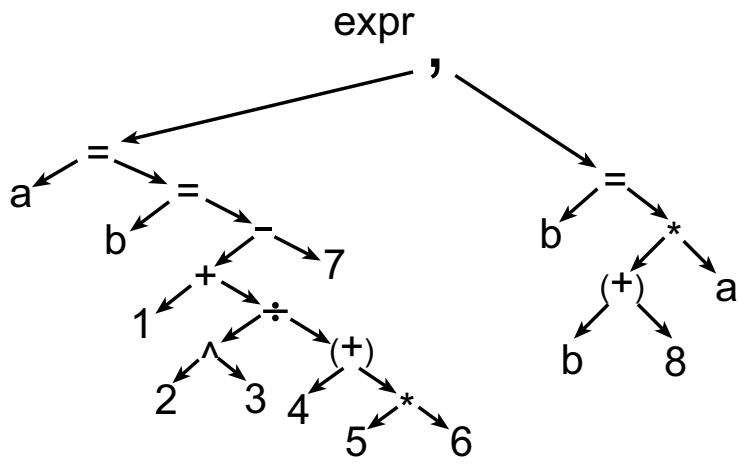


Рис. 23.5: Синтаксическое дерево разбора выражения.

Для начала, введём описание узла дерева синтаксического разбора:

```
struct ast_node {
```

```
    int type; /* 0 -- число, 1 -- идентификатор,
               * 2 -- скобочное выражение
               * 3 -- бинарный оператор
               */
    union {
        struct binop_info *binop; /* ссылка на описание */
        double realnumber; /* число */
        int identifier; /* идентификатор */
    }
    struct ast_node *left; /* левое поддерево */
    struct ast_node *right; /* правое поддерево */
};
```

Синтаксическое дерево мы будем строить пошагово, читая на каждом шаге очередной оператор и правый операнд. Изначально синтаксическое дерево будет состоять из одного самого первого операнда.

Узлам синтаксического дерева будем назначать приоритеты. Если в узле хранится бинарный оператор, то приоритет узла равен приоритету этого оператора.

Узлы, которым соответствует число, переменная или корень синтаксического дерева выражения в скобках, имеют одинаковый наивысший приоритет.

Алгоритм построения синтаксического дерева будет таким, что узлы правого склона дерева упорядочены по приоритету (чем ниже узел, тем выше приоритет). В самом низу склона находится лист, имеющий наивысший приоритет.

Также верно, что правоассоциативные операторы с равным приоритетом, расположены на склоне дерева друг за другом и идут в том же порядке, что они расположены в выражении, если двигаться по склону вниз.

То же самое верно и для левоассоциативных операторов, если двигаться по склону вверх.

Предположим, что мы уже разобрали часть строки слева от считывающего курсора и построили для неё дерево. Также мы прочитали следующий за ним бинарный оператор `bop` и курсор находится вначале описания правого операнда этого оператора.

На правом склоне дерева найдём самый верхний узел `t`, который

- имеет приоритет строго выше приоритета оператора `bop`, если `bop` правоассоциативный;
- имеет приоритет не меньше приоритета оператора `bop`, если `bop` левоассоциативный.

Такой узел `t` обязательно найдётся, так как внизу склона находится лист с наивысшим приоритетом.

Перестроим дерево так, чтобы узел с оператором `bop` стал родителем узла `t` и узла, соответствующего правому операнду оператора `bop`.

В коде 23.9 показан пример реализации функции `read_S`. В этом коде на каждом шаге поиск узла `t` осуществляется с корня дерева.

Программа 23.9: Функция построения дерева синтаксического разбора.

```
int read_S(struct ast_node *left,
           struct binop_info *bop,
           struct ast_node **result) {
    struct binop_info *binop;
```

```
struct ast_node *node, **parent;
struct ast_node *t = left;
if ( ( read_c('(') && read_S(NULL,0,&node) && read_c(')') )
     || read_number(&node)
     || read_identifier(&node) ) {
    /* Обновим тип, если было считано
       выражение в скобках */
    if ( node->type == 3 ) node->type = 2;
    if ( !left ) {
        /* левой части нет, поэтому текущим результатом
         * объявляем полученный узел.
         */
        *result = node;
    } else {
        /* проходим вниз по правому краю левого дерева,
         * сравнивая приоритеты операций
         */
        /* в переменной parent хранится указатель на адрес
         * родительского узла, который мы будем замещать
         * вновь созданным узлом.
         */
        parent = result;
        while ( t ) {
            if ( t->type==0  || t->type==1 || t->type==2
                || t->prio > bop->prio
                || (t->prio == bop->prio && bop->left)) {
                /* ЕСЛИ
                 * это число, переменная или скобочное
                 * выражение, либо приоритет левого
                 * оператора выше нашего, либо оператор
                 * левоассоциативен и того же приоритета,
                 * ТОГДА
                 * создаем узел, соответствующий бинар-
                 * ному оператору и выходим из цикла.
                 */
                create_binop_node(parent, bop, t, node);
                break;
            } else {
                /* приоритет левого оператора ниже нашего
                 * либо приоритет одинаков, но
                 * оператор правоассоциативен,
                 * ТОГДА  проходим вниз по "склону"
                 */
                parent = &t->right;
                t = t->right;
            }
```

```
            }
        }
        /* считываем следующий бинарный оператор */
        if ( read_binop(&binop) ) {
            return read_S(*result, binop, result);
        } else {
            return 1;
        }
    } else {
        return 0;
    }
}
```

В этом коде мы воспользовались функциями `read_binop`, `read_number`, `read_identifier`, а также функцией `create_binop_node`.

Функция с прототипом

```
int read_binop(struct binop_info **binop);
```

считывает бинарный оператор и возвращает указатель на структуру `struct binop_info`, соответствующую этому оператору.

Функции с прототипами

```
int read_number(struct ast_node **);
int read_identifier(struct ast_node **);
```

считывают число и идентификатор соответственно и возвращают соответствующим образом подготовленный узел дерева разбора.

Функция `create_binop_node` создает узел дерева разбора и принимает на вход четыре аргумента:

- `struct ast_node **parent` — указатель, по которому нужно поместить адрес созданного узла;
- `struct binop_info *bop` — указатель на структуру с описанием бинарного оператора;
- `struct ast_node *left` — левый ребёнок;
- `struct ast_node *right` — правый ребёнок.

Задача Л23.6. Реализуйте перечисленные четыре функции и протестируйте их.

Задача Л23.7. Покажите, что указанная реализация функции `read_S` с использованием поиска нужного узла от корня дерева сверху работает в худшем случае время, пропорциональное $O(n^2)$, где n — число бинарных операторов в выражении. Рассмотрите пример, состоящий из последовательности выражений, разделенных одним и тем же правоассоциативным оператором (например, оператором присваивания).

Предложите реализацию функции `read_S`, работающую линейное время относительно количества последовательных бинарных операторов. Усложнится ли при этом структура `struct ast_node`?

Задача Л23.8.* **(Расстановка скобок)** Напишите программу, которая находит расстановку скобок в данном выражении, задающую правильную последовательность выполнения бинарных операторов. При этом требуется, чтобы программа работала линейное время от размера

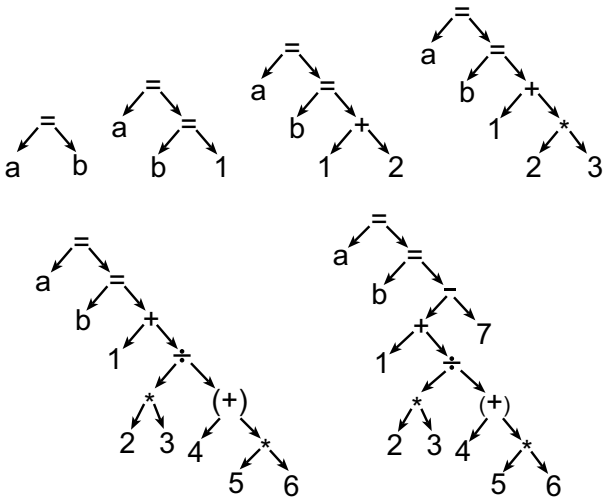


Рис. 23.6: Шаги построения синтаксического дерева разбора выражения «a=b=1+2^3/(4+5*6)-7,b=(b+8)*a».

входных данных. Убедитесь в этом, осуществив тестирование программы. Проверьте как работает ваша программа на больших входных данных, состоящих из большого количества только правоассоциативных (левоассоциативных) операторов.

Формат входа. На вход подаётся список бинарных операторов, с указанием их приоритета и типа ассоциативности, а также выражение, использующее эти операторы. Список операторов представлен в виде набора строчек, каждая из которых начинается либо со слова `RIGHT`, либо со слова `LEFT`. После идёт набор символов или пар символов, обозначающих бинарные операторы. Операторы разделяются пробелом. Номер строчки соответствует приоритету. В последней строчке ввода дано выражение. Количество бинарных операторов менее 20. Для их обозначения используются символы `+ - * / | & , = ; ! % ^` и их пары. Длина входного выражения не превосходит 100000 символов. Входное выражение состоит из бинарных операторов, чисел и имён переменных — идентификаторов, состоящих из латинских букв, знака подчёркивания и цифр. Идентификатор не может начинаться на цифру. Входное выражение не содержит скобок.

Формат выхода. Выведите данное выражение, в котором поставлены скобки. Число пар скобок, должно соответствовать количеству бинарных операторов в выражении.

Примеры ввода/вывода:

stdin	stdout
LEFT + - LEFT * / RIGHT ^ 1+2*3/4/5+6^7^8	((1+(((2*3)/4)/5))+(6^(7^8)))
LEFT , RIGHT = += *= LEFT + - a=b*=c,d=e+=f	((a=(b*=c)),(d=(e+=f)))

Калькулятор на bison

Существуют системы, которые по набору правил вывода могут генерировать программы для синтаксического разбора текстов на языке, заданном этими правилами.

Одна из простейших и наиболее популярных систем — программа `bison`.

Документацию по этой программе можно получить по команде

```
> info bison
```

Файл с правилами bison

Для получения программы-калькулятора средствами `bison` нам понадобится написать два файла: файл `calc.y` с правилами грамматики и файл `calc.c` содержащий функцию `main` и функцию `yylex` чтения терминальных символов из потока ввода. Программа `bison` получит на вход файл `calc.y` и сгенерирует два файла: `calc.tab.c` и `calc.tab.h`

В глобальной переменной `yylval` (объявляемой в файле `calc.tab.c`) хранится *значение* последнего считанного символа (терминального или нетерминального). Тип этой переменной равен `YYSTYPE`. В случае калькулятора под *значением* считанных символв (кусочков арифметического выражения) естественно понимать результат их вычисления, а тип `YYSTYPE` определить как `double`.

Между `%{` и `%}` пишется код на Си, который будет помещен в начало создаваемого файла `calc.tab.c`. Обычно этот код состоит из директив препроцессора `#include` и `#define`, а также объявления глобальных переменных.

Затем после `%}` идут декларации `bison`. Команды `%left` и `%right` позволяют задавать ассоциативность бинарных и унарных операторов. Операторы перечисляются в порядке возрастания приоритета. Операторы, описанные в одной строке, имеют одинаковый приоритет.

Правила грамматики начинаются после `%%`. Левая часть каждого правила есть идентификатор одного нетерминального символа. Правая часть правила отделяется от левой символом двоеточия. В конце правила ставится точка с запятой. Символ `|` используется как союз ИЛИ.

```
/* calc.y Инфиксный калькулятор*/
%{
#define YYSTYPE double
#include <math.h>
%}
/* Декларации BISON */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* унарный минус          */
%right '^'    /* возведение в степень   */
/* Правила грамматики */
%%
input:   /* пустая строка */
       | input line /* вход рекурсивно определяется как
                       множество строк */
       ;
```

```
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
        ;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
        ;
%%
```

Файл на языке Си с функциями `main` и `yylex`

Функция `yylex` разбивает входной поток символов на «неделимые атомы», называемые токенами.

В простейшем случае токены — это ASCII символы. Иногда удобно считать, что токены — это строки непробельных символов, а пробельные символы рассматривать как разделительные символы между токенами. Обычно токены — это слова, которые имеют определённую семантику (значение). В случае программы-калькулятора токенами удобно сделать символы бинарных операций и записи действительных чисел.

Можно сказать, что токен — это последовательность символов, представляющая собой некоторый цельный кусок входных данных, внутренняя структура которого скрыта от программы синтаксического анализа. Для описания этой структуры не пишется правил вывода. Внутренняя структура токена либо очень проста (например, фиксированное слово из ASCII символов или один символ), либо декларируется способом, отличным от правил вывода (например, для считывания чисел используется стандартная функция `scanf` и программисту не имеет смысла включать в грамматику своего языка грамматику описания действительных чисел).

С точки зрения программиста, **токен** — это последовательность байт, которые обозначают терминальный символ рассматриваемой грамматики.

За один вызов функция `yylex` считывает из потока ввода один токен и возвращает его идентификатор. При этом *значение* токена помещается в глобальную переменную `yylval`.

В случае считанного числа, функция возвращает идентификатор `NUM`, а в глобальную переменную `yylval` помещает его значение.

Функция `yylex` вызывается в цикле в функции, `yyparse`, которая генерируется автоматически по набору грамматических правил программой `bison`. Функция `yyparse` вызывается из функции `main`.

```
/*
  file:  calc.c
  title: Функции токенацзера:
```

```
          yylex   -- функция разбиения потока на токены.
          yyerror -- функция обработки ошибок
          main    -- главная функция
*/
#include <stdio.h>
#include <ctype.h>
#include "calc.tab.h"
int yyparse(void);

int yyerror (const char *s) {
    printf ("%s\n", s);
}

/* Функция yylex считывает очередной токен.
  В случае считанного числа возвращает NUM, а значение числа
  помещает в глобальную переменную yylval.
  Пробельные символы пропускает и рассматривает как
  разделители между токенами. В случае считанного символа
  возвращает ASCII код символа.  По достижении конца потока,
  возвращает 0. */
int yylex(void) {
    int c;

    /* пропустить пробельные символы, разделяющие токены  */
    while ((c = getchar()) == ' ' || c == '\t')
      ;
    /* токен, равный действительному числу */
    if (c == '.' || isdigit(c)) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUM;
     }
     /* токен соответствующий концу файла  */
     if (c == EOF)
        return 0;
     /* токен, соответствующий одному символу */
     return c;
}
int main(void) {
    return yyparse();
}
```

Команды компиляции

Для получения исполняемой программы-калькулятора необходимо выполнить следующие команды.

```
> bison -d calc.y    # создаёт файлы calc.tab.c и calc.tab.h
> gcc calc.tab.c calc.c -lm -o calc # создаёт запускаемый
                                    # файл calc
```

Пример использования калькулятора

```
> calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
-56 + 2
-54
3 ^ 2
9
<Ctrl+D>
```

Более подробно формальные языки и принципы построения синтаксических анализаторов изучаются на курсе «Теория и реализация языков программирования».

Задачи

Задача Л23.9. (Грамматика AB) Язык задан грамматикой

```
S     ::= A | B
A     ::= '(' B SPACE B ')' | 1
B     ::= '[' A SPACE A ']' | 1
SPACE ::= (' ' | ' \t')+
```

Напишите программу, которая распознаёт слова, выводимые из символа S, а именно, считывает строку и выводит YES, если слово в строке выводимо из S, или NO — иначе. Используйте при этом
а) метод рекурсивных функций; б) bison.
При решении данный задач, возможно, придётся перейти к другой эквивалентной грамматике.

Задача Л23.10. (Перевод из инфиксной в постфиксную) Напишите программу перевода арифметического выражения из инфиксной формы в постфиксную, используя
а) рекурсивные функции; б) нерекурсивные функции; в) bison.
Проверьте, как работает программа на входных данных `1-2-3`, `1+2*3/4/5/6-7-8`, `1*2*3*4*5*6`.

Задача Л23.11. (Расчет сопротивления с помощью bison) Решите задачу расчета сопротивления параллельно-последовательной схемы, используя программу `bison`.

Задача Л23.12. (Визуализация синтаксического дерева) Напишите программу, которая для данного инфиксного выражения строит дерево разбора. Для описания дерева используйте формальный язык `dot`, а для получения изображения программу `dot` из системы `graphviz`[6].

[6] `GraphViz` (`http://www.graphviz.org`) — свободно распространяемый набор инструментов для визуализации графов.

```
digraph t {
node [shape="circle"];
o1 [label="*"]; o2 [label="+"];
o3 [label="*"]; o4 [label="/"];
o5 [label="/"]; o6 [label="-"];
o7 [label="-"];
node [shape="box"];
a1 [label="1"]; a2 [label="2"];
a3 [label="3"]; a4 [label="4"];
a5 [label="5"]; a6 [label="6"];
a7 [label="7"]; a8 [label="8"];
o1->a1; o1->a2;
o5->a6; o5->o4;
o4->o3; o4->a5
o3->a3; o3->a4;
o2->o1; o2->o5;
o6->o2; o6->a7;
o7->o6; o7->a8;
}
```

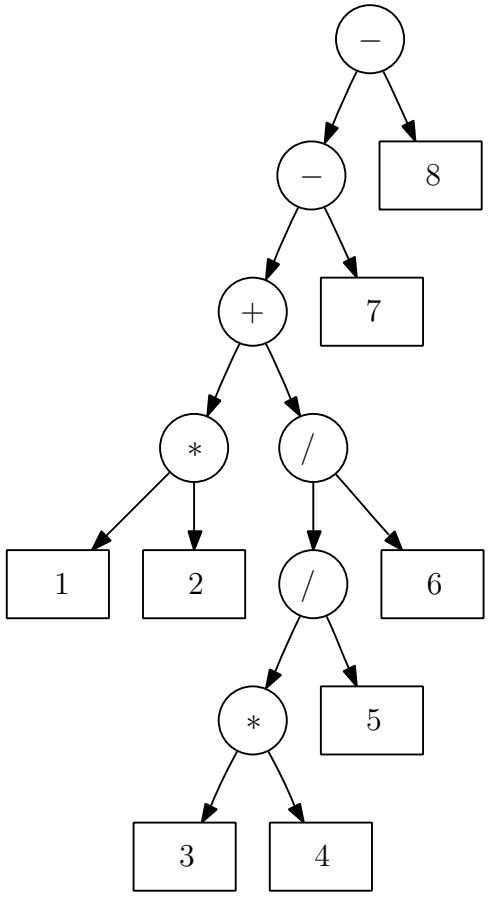


Рис. 23.7: Описание синтаксического дерева выражения `1*2+3*4/5/6-7-8` в формате `dot` и его визуализация, построенная с помощью программы `graphviz/dot`.

Например, при получении на вход выражения `1*2+3*4/5/6-7-8` программа должна выводить описание, представленное на рис. 23.7.

Лекция 24

Продукционное программирование и регулярные выражения Perl

Краткое описание: Продукционное программирование — это метод описания алгоритмов с помощью правил преобразования. В лекции 3 был рассмотрен абстрактный вычислитель «машина Маркова», который является простейшим, но при этом Тьюринг-полным, продукционным вычислителем. На этой лекции мы рассмотрим расширенный вариант машины Маркова, который называется «регулярные выражения Perl» (Perl compatible regular expressions, PCRE).
Важным примером является *задача вычисления арифметических выражений с помощью правил*. Очевидный алгоритм вычисления выражения, к которому прибегает обычно человек, основан на правилах, а именно, человек пошагово упрощает выражения выполняя отдельные элементарные операции умножения и сложения пар чисел. Но описание этого алгоритма в виде правил не так очевидно. Проблема возникает из-за необходимости учёта приоритетов операций (у умножения приоритет больше).
Алгоритмы, основанные на продукционном подходе, иногда имеют трудно прослеживаемую логику и ведут себя неожиданно. Но несмотря на это, они имеют широкое применение на практике. Это связано, в значительной степени, с продукционной природой мышления человека и продукционной природой исходного описания логики действия механизмов (программ).
Мы разберём различные стороны продукционного программирования и определим, в каких случаях следует прибегать к продукционному программированию, а в каких — нет.
Ключевые слова: регулярные выражения, шаблоны, правила подстановки, продукционное программирование.

Продукционное программирование — это метод описания алгоритмов с помощью правил преобразования. Каждое правило (продукция) состоит из двух частей — левой и правой:

<что искать> → <на что заменить>

Например,

- найти подстроку «hello» и заменить её на подстроку «good bye»;

В машине Маркова это правило записывается как «`hello` → `good bye`». С помощью таких простых замен можно осуществить любые преобразования текста. Несложно, например, убрать из текста знаки препинания, найти и раскрыть аббревиатуры, большие буквы сделать маленькими, удалить все союзы, предлоги и междометия. Более того, как мы уже отмечали, с

помощью замен можно осуществлять арифметические действия с числами и вообще *произвольные вычисления*, которые реализуемы на машине Тьюринга (и только эти). Но для реализации численных вычислений требуется большое число правил и они довольно сложны. На практике удобно прибегать к преобразованиям с помощью правил с расширенными возможностями. Приведём примеры правил, заданных в виде описаний на естественном языке:

1. Найти подстроку вида «dd.mm.yyyy», где символы d, m и y обозначают произвольную цифру и преобразовать её в строку вида mm/dd/yyyy.

2. Найти подстроку вида «(<несколько цифр>)» и удалить скобки.

3. Найти подстроку вида «$<число>» и заменить её на «X руб.», где $X =$ <курс доллара> $\cdot$ <число>.

4. Найти текст между строками «`<math>`» и «`</math>`» и преобразовать его согласно некоторой заданной функции.

5. Найти в тексте телефонные номера с кодом города 095 и заменить этот код на 495, в частности, номера `+7(095)1234567`, `8(095)123-45-67`, `8-095-123-45-67` должны быть заменены на `+7(495)123-45-67`.

Эти правила формально также можно разделить на левую и правую части, но эти части имеют более общий вид:

$$\left\langle \begin{array}{c}\text{шаблон искомой}\\ \text{подстроки}\end{array} \right\rangle \rightarrow \left\langle \begin{array}{c}\text{алгоритм конструирования новой подстроки, на которую}\\ \text{следует заменить найденную}\end{array} \right\rangle$$

Левая часть правила не просто искомая фиксированная строка, а *шаблон искомой строки*, то есть описание того, как устроена искомая строка, из каких частей состоит, какой (примерно) имеет вид. Шаблон строки задает не одну строку, а целый класс строк.

Правая часть также не просто строка символов, а строка, сконструированная из частей искомой подстроки, или, в общем случае, алгоритм конструирования строки, которая заменит найденную подстроку. Этот алгоритм обычно достаточно прост и состоит из небольшого числа элементарных операций без циклов. Так в примере 3 в правой части правила выполняется одна операция умножения.

Расширенные правила уже могут успешно применены на практике и они реализованы в регулярных выражениях языка Perl.

Регулярные выражения Perl

Регулярные выражения Perl позволяют осуществлять довольно сложные преобразования текста. Операция применения правила к слову (тексту), хранящемуся в переменной, выглядит следующим образом:

```
<имя переменой> =~  s/шаблон подстроки/на что заменить/;
```

Имена переменных в языке Perl начинаются со знака доллара. Следующая программа на языке Perl выведет строчку `<ba>cabc`:

```
$x = "abcabc";       # инициализировать строковую переменную $x
$x =~ s/ab/<ba>/;    # заменить в строке  $x  "ab" на "<ba>"
print $x;            # вывести зачение переменной $x
```

Сохраните эти три строчки кода в файле `a.pl` и выполните команду[1]

```
> perl a.pl
```

Правило `$x =~ s/ab/<ba>/` заменит первую слева подстроку `ab` на `<ba>`. Для того, чтобы произошла замена всех экземпляров искомой подстроки, необходимо указать символ `g` после последней косой черты:

```
$x = "abcabc";
$x =~ s/ab/<ba>/g;     # даст в результате $x == <ba>c<ba>c.
```

После символа решётки `#` до конца строки можно писать произвольный комментарий.

Символы `. * + ? \ ^ $`, употребляемые в правой части правила, имеют особый смысл. Есть специальные обозначения для классов символов — шаблонов, которые соответствуют одному символу из некоторого заданного множества. Приведём некоторые из них:

`\s` — пробельный символ (пробел, табуляция и несколько других специальных символов);

`\S` — не пробельный символ;

`\d` — цифра;

`\w` — буква или символ подчёркивания;

`\W` — символ, который не является буквой или символом подчёркивания;

`.` — любой символ, кроме символа новой строки;

`[<символы>]` — символ из указанного множества символов;

`[^<символы>]` — символ, не принадлежащий указанному множеству символов;

Конструкция `[...]` является универсальной, с её помощью можно задать произвольные множества символов. Например, шаблон `[0123456789]` эквивалентен шаблону `\d`. В квадратных скобках можно использовать символ «тире» для задания диапозонов символов, и это позволяет сократить запись. Например, `[0123456789]` можно коротко записать как `[0-9]`, а шаблон `[a-zA-Z0-9]` соответствует латинской букве (заглавной или маленькой) или цифре.

Примеры шаблонов:

`\s\w\w\w\w\s` — шаблон для слова из 4 букв, окруженного пробельными символами;

`[Пп]о-видимому[^,.]]\s` — шаблон для слова «по-видимому», начинающегося с заглавной или маленькой буквы, после которого не стоит ни запятая, ни точка;

`[,.;:!?]` — шаблон для знаков препинания;

[1]Для успешного выполнения команды в системе должен быть установлен интерпретатор языка Perl.

Есть также специальные 0-шаблоны (шаблоны нулевой длины), которые соответствуют не символу или набору символов, а промежутку между символами определённого типа.

`^` — начало строки (начало стоки или сразу после символа новой строки `\n`);

`$` — конец строки (конец строки или перед символом новой строки `\n`);

`\b` — граница слова;

`\B` — промежуток, не являющийся границей слова;

Под словом в данном случае понимается последовательность символов типа буква, цифра или знак подчёркивания.

Шаблоны можно комбинировать с помощью союза «или». Для этого используются круглые скобки и знак «`|`»:

`(a|the|this) (table|chair)` — шаблон 6 словосочетаний: `the table`, `a table`, `this table`, `the chair`, `a chair`, `this chair`;

`(net|rule|rate|bank|price)[.\n]{,50}(capital|financ|oblig)` — шаблон для определения английских текстов, относящихся к экономике. Если одна из указанных пар слов ($3 = 15$ возможных пар) встречается в тексте и между словами находится 50 символов или меньше, то велика вероятность того, что текст относится к экономике.

Здесь используется конструкция `{n,m}`, которая задаёт степень повторяемости элемента шаблона, стоящего слева от неё. Есть три варианта использования этой конструкции:

`X{n,m}` — повторение шаблона `X` не менее n раз и не более m раз;

`X{n,}` — повторение шаблона `X` не менее n раз;

`X{,m}` — повторение шаблона `X` не более m раз.

Для указания степени повторяемости элемента шаблона используются также символы `* + ?`. В частности, `X+` — это шаблон для строки, которая удовлетворяет шаблону `X`, либо шаблону `XX`, либо шаблону `XXX`, и так далее, где шаблон `X` может быть повторён любое число раз больше 0.

X+ — повторение шаблона `X` 1 или более раз (равносильно `{1,}`);

X* — повторение шаблона `X` 0 или более раз (равносильно `{0,}`);

X? — повторение шаблона `X` 0 или 1 раз (равносильно `{0,1}`).

Если же требуется в правой части правила указать символы `\ . * + ? { } ( )`, то следует перед ними ставить символ `\` (бэкслэш), при этом говорят, что символ бэкслэш *экранирует* специальные символы, «превращая» их в обычные, не специальные символы. Два символа бэкслэш `\\`, идущих подряд, означают просто символ бэкслеш.

Примеры регулярных выражений:

- `(\n|^d)\s*\n+` — пустая строка;

- `\d+\.?\d*` — цифры (более одной), затем, возможно, символ точки, и затем снова цифры или ничего;
- `(19\d\d|20\d\d)` — целое число из промежутка [1900, 2099].
- `\+7 ?\(?495\)? ?\d\d\d-?\d\d-?\d\d` — номера, телефонов, которые начинаются на « `+7 (495)` », причём круглых скобок и пробелов может не быть, а в номере могут присутствовать тире между третьей и четвертой, пятой и шестой цифрой;
- `\([^()]\)` — набор символов, отличных от круглых скобок, в круглых скобках.

Инвертирование бит

На языке программирования Perl с помощью регулярных выражений несложно реализовать логику работы машины Маркова. Рассмотрим, к примеру, алгорифм Маркова, который инвертирует бинарное слово:

```
A1  →  0A
A0  →  1A
A   →. ∅
∅   →  A
```

Ему соответствует следующая программа на языке Perl.

```
# file: invert.pl
print "Input  :";  # вывести приглашение
$_ = <>;           # считать строку
s/\s//g;         # удалить в этой строке все пробельные символы
my $step =1;     # создать (команда my) новую локальную переменную
                 # $step и инициализировать её единицей
while (
    s/A1/0A/ ||
    s/A0/1A/ ||
   (s/A// && print("Ответ  : $_\n") && exit) ||
                       # так записываются стоп-правила
    s/^/A/             #  добавить в начало слова букву A
) { print "Шаг  $step: $_\n";  sleep 1; $step++; }
print "Ни одно из правил не применимо.\n Ответ  :$_\n";
```

Операция применения правила `s/../../` возвращает логическое значение «истина»[2], если правило применено. Логическая бинарная операция «или», обозначаемая как `||`, работает следующим образом: вычисляется левый операнд; если он истиностный, то второй операнд не вычисляется, иначе — вычисляется. Логическая операция «и», обозначаемая как `&&`, работает по-другому: вычисляется левый операнд; если он равен логическому значению «ложь», то второй операнд не вычисляется, иначе — вычисляется. Функция `print` в языке Perl возвращает истиное значение, если удалось напечатать то что требуется в указанный поток вывода.

Пример работы программы `a.pl`:

[2]В языке Perl, также как и в Си, нет специальных булевых значений. Значением «ложь» считается число равное 0, пустая строка и неопределенное значение (значение, которое в языке Perl имеет неинициализированная переменная). Остальные значения считаются истинными.

```
> perl invert.pl
Input  : 011000101
Шаг   1: A011000101
Шаг   2: 1A11000101
Шаг   3: 10A1000101
Шаг   4: 100A000101
Шаг   5: 1001A00101
Шаг   6: 10011A0101
Шаг   7: 100111A101
Шаг   8: 1001110A01
Шаг   9: 10011101A1
Шаг   10: 100111010A
Ответ  : 100111010
```

Правило ∅→A срабатывает первым и добавляет символ A в начало строки. Затем начинают работать первые два правила, за счет которых символ A начинает перемещаться вправо и «перепрыгивая» через цифру, инвертировать его. Когд справа от A не оказывается цифры, срабатывает завершающее правило A→. ∅ и вычисления прекращаются.

Перевёртывание строки

Решим задачу перевёртывания бинарного слова (переписывание слова в обратном порядке символов — от последнего к первому) с помощью обычных правил. Эта задача решается следующим алгорифмом Маркова:

```
AG   →   A
A1   →   1A
A0   →   0A
A    →.  ∅
G00  →   0G0
G01  →   1G0
G10  →   0G1
G11  →   1G1
GG   →   A
∅    →   G
```

Как вы уже, наверное, отметили, читать алгорифмы Маркова правильнее с последних правил, так как часто именно они срабатывают в первую очередь. Первым применяется правило ∅ → G — оно добавляет в начало слова букву G. Затем, за счет применения правил с 5-го по 8-е, эта буква перемещается вправо, и «тащит» впереди себя цифру. Когда первая цифра будет перенесена в конец, снова сработает правило ∅ → G и в начале слова слова появится буква G. Она потащит впереди себя следующую цифру пока эта цифра не упрётся в букву G. Следующие цифры будут переносится не в конец, а до первой встретившейся буквы G. В конечном итоге слово будет перевернуто, только везде в промежутках между цифрами будут стоять буквы G. Осталось от избавиться от лишних букв G. Последнее правило срабатывает два раза и в начале слова появляется две буквы GG, которые правилом GG→A заменяются на букву A. Буква A за счет применения первых четырёх правил начнёт перемещаться вправо, перепрыгивая через цифры и стирая букву G. В конечном итоге, буква A будет удалена с помощью 4-го

правила и вычисления остановятся.

Разобраться в том, как работает этот алгорифм без комментариев было бы сложно. Это один из недостататков продукционных алгоритмов — их описания плохо читаются, логику их работы сложно понять без глубокого анализа и тестирования на примерах.

Программа 24.1: Программа перевёртывания входного слова

```
#!/usr/bin/perl
$_ = <>;
s/\s//g;
while(
    s/AG/A/    ||
    s/A1/1A/   ||
    s/A0/0A/   ||
   (s/A// && print("Result  :$_\n") && exit) ||
    s/G00/0G0/ ||
    s/G01/1G0/ ||
    s/G10/0G1/ ||
    s/G11/1G1/ ||
    s/GG/A/    ||
    s/^/G/
) { print $_, "\n"; sleep 1; }

print "Result=", $_, "\n";
```

Задача Л24.1. Во время работы алгорифма, записанного в виде программы 24.1, наступает момент, когда слово имеет длину в два раза больше, чем длина исходного слова. Напишите алгорифм, который переворачивает слово и во время преобразований увеличивает длину данного слова не более, чем на три символа.

Продукционный подход к задаче «калькулятор»

Рассмотрим задачу вычисления арифметических выражений над целыми числами, в которых встречаются целые числа, бинарные операции сложения, вычитания, умножения, целочисленного деления, а также, круглые скобки.

Попробуем решить эту задачу с помощью продукционного подхода. Ограничимся для начала только сложением и умножением. На первый взгляд решение очевидно. Необходимо задать следующие три правила:

- если скобки окружают число, то их можно отбросить;
- если встречается выражения вида «число * число», то его следует заменить на результат умножения;
- если встречается выражения вида «число + число», то его следует заменить на результат сложения.

Формально, эти правила можно описать так:

```
(число)       →  $1
число+число   →  $1+$2
число*число   →  $1*$2
```

Выражения `$1` и `$2` здесь обозначают численные значения первого и второго числа. На языке регулярных выражений Perl они выглядят не так наглядно, так как символы `+` и `*` являются специальными и их необходимо экранировать (чтобы они обозначали просто символы, а не символы, указывающие кратность повторения):

```
while (
   s/   \((число)\)    /   $1     /xe ||
   s/(число)\*(число)  /   $1*$2  /xe ||
   s/(число)\+(число)  /   $1+$2  /xe
) { }
```

Вместо слова «`число`» в регулярных выражениях следует писать шаблон числа. Шаблон целого положительного неотрицательного числа прост — `\d+`. Шаблон действительного неотрицательного числа несколько сложнее[3]. Но мы пока будем работать только с неотрицательными целыми числами.

Здесь в каждом правиле после символа косой черты указан символ `x`. Это специальный символ, который разрешает использовать пробельные символы в левой и правой стороне правила, которые игнорируются транслятором регулярных выражений.

Кроме того, после косой черты указан символ `e`. Он говорит транслятору, что правую часть правило следует интерпретировать как выражение на языке Perl, а не как строку. В выражении можно использовать любые функции и операторы язык Perl. Части шаблона, окружённые скобками (не экранированными), называются группами. Строки, которые соответствуют этим частям шаблона помещаются в специальные переменные `$1`, `$2`, `$3`, ...

В нашем случае мы окружили скобками шаблоны чисел, так как значения этих чисел мы хотим использовать в правой части правила.

Целиком программа выглядит следующим образом:

```
$_ = <>;        # считать строчку
$_ =~ s/\s//g; # убрать все пробельные символы
my $step = 1;
while (
   s/\(((\d+)\)     /    $1     /ex ||
   s/(\d+)\*(\d+)   /    $1*$2  /ex ||
   s/(\d+)\+(\d+)   /    $1+$2  /ex
) { print "Шаг  $step: $_\n"; $step++;}
print "Ответ =$_\n";
```

[3]Это шаблон, которому должны удовлетворять такие записи: `123`, `0.123`, `2.`, `.123`, `12.34`. Шаблон `\d*\.?\d*` не годится. Он соответствует следующему описанию: несколько цифр (возможно, нисколько), затем, возможно, точка, а затем снова несколько цифр (возможно, нисколько). Этому описанию подходит и просто точка, что не верно. Необходимо указать, что в записи числа должна присутствовать по крайней мере одна цифра и не более одной точки. Одно из решений такое: `(\d+\.?\d*|\d*\.?\d+)`. Но есть и другие форматы записи чисел. Например, запись `1.5E-7` соответствует числу $1.5 \cdot 10^{-7}$.

Ни одно из правил не помечено как правило останова. Результат будет печататься, когда ни одно из правил не применимо.

Приведём примеры работы этой программы:

```
(1+2)*(3+4)
Шаг  1: (3)*(3+4)
Шаг  2: 3*(3+4)
Шаг  3: 3*(7)
Шаг  4: 3*7
Шаг  5: 21
Ответ =21
```

```
2*3+4*5
Шаг  1: 6+4*5
Шаг  2: 6+20
Шаг  3: 26
Ответ =26
```

Во втором примере калькулятор следует правилу большего приоритета операции умножения. Но оказывается, **указанные правила работают неверно**, а именно калькулятор не всегда следует правилу большего приоритета операции умножения. Ошибка возникает, например, при вычислении выражения `1+2*(3+4)`:

```
1+2*(3+4)
Шаг  1: 3*(3+4)
Шаг  2: 3*(7)
Шаг  3: 3*7
Шаг  4: 21
Ответ =21
```

Указанная ошибка возникает всякий раз, когда некоторое число x умножается с одной стороны на выражение в скобках (причём в скобках есть сложение), а с другой стороны стоит знак `+` и некоторое число. Добавьте к указанным трём правилам ещё несколько правил (их следует поставить первыми), чтобы любые выражения с неотрицательными целыми числами вычислялись корректно. Приоритет операции умножения должен быть выше приоритета операции сложения.

Одно из решений основано на добавлении правил вида:

```
(число+число)          →  $1+$2
(число+число+число)    →  $1+$2+$3
+число+число+          →  "+($1+$2)+"
```

Задача Л24.2. Реализуйте с помощью правил калькулятор, в котором допускаются отрицательные числа и присутствуют все четыре операции арифметики: `+ * - /`. Какие трудности возникают с унарным минусом и как их можно преодолеть?

Примеры применения продукционного подхода

Продукционный подход широко применяется на практике и не в компьютерных системах, но и в различных формальных описаниях законов и правил общества.

1. *Конфигурационные файлы программ.*

Многие компьютерные программы имеют конфигурационные файлы — файлы, содержащие значения различных параметров, которые определяют логику работы программы.

Обычно, конфигурация состоит из набора пар (имя параметра, значение параметра). В конфигурацию сложных систем могут входить более сложные конструкции, и часто этими конструкциями являются правила, описывающие то, как должна вести себя программа в той или иной ситуации.

Наиболее ярким примером является конфигурация программы `iptables`, которая отвечает за фильтрацию, изменение и перенаправление пакетов с данными, которые приходят на различные сетевые устройства.

Конфигурация этой программы целиком состоит из правил, которые описывают как по свойствам пакета определить, на какое сетевое устройство его перенаправить, отфильтровать его или пропустить. Правила объеденненны в цепочки последовательно исполняемых правил. Этот элемент императивного программирования значительно облегчает восприятие человеком описанной правилами логики.

2. *Системы автоматического построения синтаксических анализаторов.*

Другим важным примером применения продукционного подхода являются грамматики — описания формальных языков с помощью правил вывода. Существуют системы, которые позволяют программисту описывать формальный язык с помощью правил вывода, а программа, которая осуществляет синтаксический разбор текстов на этом языке генериться автоматически.

В лекции 23 на стр. 23 приведён пример использования системы `bison`.

3. *Конвертация данных из одного формата в другой.*

Задача конвертации данных из одного формата в другой нередко возникает на практике. И если два используемых формата достаточно близки, эту задачу проще всего решать с помощью правил преобразования. Это могут быть, в частности, правила преобразования, основанные на регулярных выражениях Perl.

Если данные записаны на одном из XML-языков[4], то для «программу преобразования» естественно писать на языке шаблонов XSLT. Программирование на XSLT является довольно специальным видом продукционного программирования, сильно отличающимся от программирования вообще.

4. *Средства описания логики сборки крупных программистских проектов — make utils.*

Пакет инструментов make utils — одно из самых простых и мощных средств описания логики сборки множества исходных файлов в исполняемые файлы (запускаемые файлы, статические и динамические библиотеки).

Так, например, в специальном файле `Makefile` можно написать следующие строки:

```
test: libmy1.a libmy2.a
    gcc test.c -lmy1 -lmy2 -o test

libmy1.a: my1.c my1.h
    gcc -c -o libmy1.a my1.c

libmy2.a: my2.c my2.h
    gcc -c -o libmy2.a my2.c
```

[4]XML-языки — семейство языков разметки данных, основанных на представлении всех данных в виде иерархии вложенных друг в друга элементов, где каждый элемент снабжён набором пар (атрибут, значение). Существует множество средств обработки и анализа XML-данных. В значительной степени XML направлен на обмен данными между приложениями и передачи данных по интернет.

Эти строки означают, что для достижения цели `test` необходимо выполнить команду «`gcc test.c -lmy1 -lmy2 -o test`», при этом цели `libmy1.a` и `libmy2.a` должны быть уже достигнуты. Для достижения цели *libmy1.a* необходимо, чтобы выполнить команду «`gcc -c -o libmy1.a my1.c`», при этом цели `my1.c` и `my1.h` должны быть уже достигнуты. Названия целей — это имена файлов. В простейшем случае цель считается достигнутой, если существует соответствующий ей файл. Зависимость цели X от цели Y удовлетворена, если существует файл Y и дата модификации файла Y раньше даты модификации файла X.

В действительности, сфера применения Makefile'ов шире, нежели описание зависимостей между Файлами проекта и определение последовательности сборки частей проекта. Это общее средство описания зависимостей между целями, где роль целей могут играть как команды компиляции, так и произвольные команды оболочки, в том числе запуск любых программ.

5. *Социальные науки.*

Продукционный подход к описанию логики действий распространён в различных социальных науках. Так, в частности, прикладная медицина в значительной степени состоит из правил вида (набор симптомов, рецепт).

В юриспруденции многие законы также часто имеют вид правил. Такая форма законов удобна для их описания, но как показывает практика, в продукционных описаниях сложно ориентироваться и сложно контроллировать их логическую непротиворечивость и полноту (наличие алгоритма действия для всех ситуаций, которые могут возникнуть).

6. *Экспертные системы и другие системы, основанные на знаниях.*

Наиболее полно программирование на основе правил представлено в экспертных системах и системах основанных на знаниях. В таких системах используются языки логического программирования, подобные языку Prolog или CLIPS. Они позволяют создавать программы, которые распознают ситуацию по набору входных данных, в частности прикладные программы для медицины, определяющие болезнь по набору симптомов, программы, распознающие причину поломки сложных устройств по набору признаков, и, в общем случае, программы осуществляющие эффективный поиск значений булевых параметров, удовлетворяющих набору данных условий.

Достоинства и недостатки продукционных алгоритмов

Мы уже отметили два недостатка продукционных алгоритмов:

- *сложность понимания логики алгоритма по его описанию* — по множеству правил, составляющих алгоритм, сложно догадаться о задаче, решаемой алгоритмом, о последовательности применения правил, о том, какой смысл имеют отдельные группы правил и специальные символы; даже сам разработчик этих правил по прошествии времени может забыть о том, какую роль выполняют те или иные правила;

- *неэффективность алгоритмов* — продукционные алгоритмы часто работают медленнее, чем их императивные аналоги; так, например, задача калькулятор решается с помощью рекурсивного разбора за линейное по длине входа время, а алгоритм, основанный на правилах, работает в худшем случае квадратичное время. То же самое верно и про задачу переворачивания слова.

Эти недостатки часто существенны. Но здесь следует сделать следующее замечание.

Во многих задачах правила являются единственным разумным решением поддержания краткого и естественного описания логики работы алгоритма. Попытки превращать продукционные описания алгоритмов в императивные приводят к запутанным программам с многократно вложенными друг в друга условными операторами `if`, повторениями кода, которые сложно модифицировать и поддерживать.

Понятие **поддержка кода** возникает в крупных программистских проектах. Крупные проекты посвящены решению сложных задач, для которых идеальное решение часто недостижимо. Такие проекты постоянно развиваются, их программный код модифицируется и дополняется. Понимание программистами того, что с кодом, который они пишут, им ещё не раз придётся работать — дополнять, анализировать, искать в нём ошибки, — сильно влияет на проектирование архитектуры системы и стиль написания кода. Поддержка кода — это анализ, модификация и дополнение кода в связи с новыми требованиями к программе, расширением функционала программы, или исправлением найденных ошибок.

Лекция 25

Функциональное программирование на примере языка Haskell

Краткое описание: На примере языка Haskell мы познакомимся функциональным программированием — методом описания алгоритмов, отличающимся от обычного *императивного* программирования. Программы на функциональном языке программирования выглядят как определения того, что нужно вычислить, а последовательность элементарных действий, на которые раскладывается программа, остаётся скрытой. Говорят, что функциональное программирование позволяет реализовать давнюю мечту всех программистов мира — «я описываю, *что* мне нужно получить, а уж компьютер сам должен догадаться, *как* это сделать». Мы уточним, что имеется ввиду под этим выражением, обсудим различные стороны функционального программирования и определим несколько важных свойств, которыми различаются языки программирования.
Ключевые слова: функциональное программирование, Haskell, ленивые вычисления, лямбда-конструкция, вычисления методом редукции.

Дистрибутивы Haskell

Есть несколько интерпретаторов языка Haskell как под Windows, так и под Linux. Все они распространяются бесплатно. Рекомендуем начать с маленького и удобного для обучения интерпретатора Hugs, который можно взять на сайте `http://haskell.org/hugs`.

Различие императивного и функционального программирования

Языки программирования, в которых алгоритмы описываются как последовательность действий, называются **императивными**. Язык программирования Си (также как и Pascal, Java, C#) является императивным языком программирования.

Программист, когда пишет программу на императивном языке программирования, мыслит в терминах *действий*. Он группирует последовательности действий в процедуры (макродействия), использует условные и безусловные переходы (низкоуровневое программирование на языке ассемблера) или организует их в виде иерархии вложенных блоков с помощью операторов структурного программирования `while`, `for` и `if`. Объектно-ориентированное программирование на языках Си++ или Java также можно рассматривать как разновидность

императивного программирования[1].

Базовой единицей программы в функциональном языке программирования является определение функции (function definition), причём имеется в виду математическое определение, а не то, что обычно понимается под этим в императивных языках.

Приведём для примера математическое определение функции, вычисляющей наибольший общий делитель (НОД) двух натуральных чисел:

$$gcd(a,b) = \begin{cases} gcd(a-b,b), & \text{если } a > b, \\ gcd(b-a,a), & \text{если } a < b, \\ a, & \text{если } a = b. \end{cases}$$

На функциональном языке программирования Haskell это определение запишется следующим образом:

```
gcd a b   | a > b = gcd (a-b) b
          | a < b = gcd (b-a) a
          | otherwise  = a
```

Этот код выглядит как формальная запись данного математического определения. Но это «работающее» определение — функцией `gcd` можно пользоваться при вычислениях. Вертикальная черта и следующее за ней условие называется *охранной конструкцией*. Определение, заданное справа от охранной конструкции будет использовано, только если условие охранной конструкции выполнено.

Для сравнения напомним, как выглядит соответствующий код на языке Си.

```
int gcd(int a, int b) {
    if (a > b) {
        return gcd(a-b, b);
    } else if (a < b) {
        return gcd(b-a, a);
    } else {
        return a;
    }
}
```

Кто-то возможно посчитает, что разница приведённых кодов только в синтаксисе, и не более того. Но давайте пойдём дальше. Мы могли описать НОД прямо в соответствие с его исходным математическим определением:

```
НОД(a, b) = max (  пересечение множеств A и B ),
              где A = Делители ( a ),
                  B = Делители ( b )
Делители (a) = числа x из множества {1,2, ..., a},
                 такие что a mod x = 0.
```

[1]Объектно-ориентированное программирование расширяет императивное программирование новыми принципами — группировка данных и методов по классам, наследование, переопределение методов, деление данных и методов на общедоступные (public) и личные (private) и др. Но основной принцип остаётся прежним — программист описывает методы как последовательность действий.

Что, в переводе на язык Haskell даст

```
-- Haskell:
gcd    a b = max c
          where c = intersect as bs
             where as = divisors a
                   bs = divisors b
divisors a = [x | x <- [1..a], rem a x == 0]
-- функция rem x y  возвращает
-- остаток от деления x на y
```

Функция `gcd` определена в стандартных пакетах Haskell, поэтому, чтобы не было конфликта, новую функцию следует назвать как-нибудь по другому.

Кроме языка Haskell есть множество других функциональных языков, например, Mathematica и Lisp.

```
Mathematica:
Gcd[a_, b_] := Max[ Intersection[ Divisors[a], Divisors[b] ] ];
Divisors[n_] := Select[ Range[1, n], Mod[n, #] == 0& ];
```

«Функциональный программист» мыслит в терминах функций и зависимостях функций друг от друга. «Императивный программист» мыслит в терминах действий и объединения последовательностей действий в процедуры.

Написать программу на функциональном языке значит записать выражение, которое должно быть вычислено, а также описать функции, которые используются в этом выражении. Акцент при этом делается не на то, *в какой последовательности* и какие команды будут исполняться, а на то, *что должно быть получено* и как функции выражаются друг через друга. Очень типично, когда «функциональный программист» просто не представляет последовательность выполнения элементарных действий своей программы. Функциональным образом мы мыслим, когда работаем с табличным редактором, — мы просто описываем зависимости ячеек таблицы друг от друга, и в одной из ячеек получаем нужный нам результат.

Программа на языке Haskell представляет собой одно выражение. Причём можно явно выделить две части: до слова «where» и после него. Например, программа

```
1+(x+y)*2 where x = 1; y = 2;
```

в качестве результата вернёт 7. Функции, которые используются в выражении, должны быть определены после слова `where`.

Важно заметить, что *переменных в Haskell просто нет*! Переменных нет, но можно определять функции, которые не получают аргументов и возвращают числа. Именно так следует интерпретировать символы `x` и `y` в последнем примере — это функции, а не переменные. Знак «=» имеет в Haskell другое значение, нежели операция присваивания «=» в языке Си (или аналогичная операция «:=» в языке Pascal). В языке Си эта операция интерпретируется следующим образом: вычислить то, что указано справа от знака «равно», и результат поместить в переменную (ячейку памяти), которая указана слева от знака «равно». Строка

```
x = x + 2
```

в языке Си интерпретируется как команда «увеличить значение переменной x на 2». В языке Haskell смысл этой команды совсем другой — «определить функцию x следующим образом: результат функции равен сумме результата вычисления функции x и числа 2». То есть в языке Haskell эта строка является определением рекурсивной функции с именем x. Функция x определена через себя, и использование этой функции приведёт к бесконечной цепочке рекурсивных вызовов и к ошибке переполнения стека «stack overflow»:

```
> x where x = x + 2
ERROR - stack overflow.
```

В языке Haskell нет переменных, а значит нет и понятия состояния, потому что состояние — это множество значений всех переменных.

Типы в языке Haskell

В языке Haskell есть базовые типы: `Integer` (целое число), `Char` (символ), `Float` (число с плавающей точкой), `Rational` (дробное). Есть специальные конструкции «`()`», «`[]`» и «`->`», которые позволяют определять новые типы на основании существующих.

Пусть `a` и `b` являются некоторыми типами данных. Тогда конструкция «`[a]`» означает новый тип — список элементов типа `a`. В частности тип «`String`» есть синоним типа «`[Char]`».

Конструкция «`(a,b)`» означает тип пары элементов типов `a` и `b`. Соответственно можно задавать типы троек, четвёрок и произвольных наборов (кортежей) из n элементов.

Конструкция «`a->b`» соответствует типу функций, которые получают на входе элемент типа `a` и возвращают элемент типа `b`.

Примеры типов:

`Integer->Integer`	целочисленная функция целого аргумента;
`[Integer]->Float`	функция, которая получает список целых чисел, а возвращает действительное число типа `Float`;
`Float->Float->Float`	функция, которая получает на входе два действительных числа и возвращает действительное число;
`(Float,Integer)->[(Float,Float)]`	функция, которая получает на входе пару чисел типа `Float` и `Integer` и возвращает список пар чисел типа `Float`.

При конструировании типов можно использовать круглые скобки, чтобы обозначить неделимые элементы. Например, тип `(Float->Float)->Float` соответствует функции, которая получает на вход функцию типа `Float->Float` и возвращает действительное число.

Интересно заметить, что при конструировании новых типов с помощью операций `[a]`, `(a,b)` и `a->b` не обязательно вместо `a` и `b` подставлять конкретные существующие типы. Можно использовать маленькие латинские буквы, означающие произвольный тип. В частности, тип `a->b->[(a,b)]` означает функцию, которая получает на входе два элемента типов `a` и `b` и возвращает список пар элементов типа `a` и `b`.

Параметризация функций

Рассмотрим тип `Float->Float->Float`.
Его можно интерпретировать двумя способами:

- функция, которая получает на вход два числа типа `Float` и возвращает действительное число;
- функция, которая получает на вход одно число типа `Float` и возвращает функцию типа `Float->Float`.

Действительно, если у функции с двумя аргументами зафиксировать один аргумент, то получится функция от одного аргумента. Про языки программирования, в которых есть возможность задания у функции лишь части аргументов, говорят, что они имеют *параметризацию функций*.

Пример 1. Функция `inc` увеличивает число на единицу. Она определяется следующим образом:

```
inc n = n+1   -- функция типа a->a
```

Комментарии в Haskell начинаются с двух тире. Выражение «`inc (inc 3)`» будет редуцировано (упрощено, вычислено) до 5. Этот факт мы будем записывать так:

$$\texttt{inc (inc 3)} \Rightarrow \texttt{5}.$$

Есть возможность явно *типизировать* функцию с помощью следующего объявления

```
inc  :: Integer->Integer
```

Пример 2. Функция `add` находит сумму данных ей чисел. Аргументами этой функции являются два числа:

```
add :: Integer->Integer->Integer
add x y = x + y
```

Функцию `inc` можно было бы определить через функцию `add`:

```
inc = add 1
```

Зафиксировав один аргумент у функции `add`, мы получили функцию с одним аргументом.
Заметьте, что круглые скобки для окружения аргументов функции не используются и аргументы разделяются просто пробелами, а не запятыми, как в языках Си или Pascal.

Пример 3. Есть два способа обозначения списков – квадратные скобки, в которых перечислены элементы через запятую, или круглые скобки, в которых элементы разделены двоеточием. В частности записи `[1,2,3]`, `(1:2:3)` и `(1:(2:(3)))` эквивалентны. Символ двоеточие «:» означает операцию **присоединения элемента к списку слева**. Пусть `x` есть элемент, а `xs` — некоторый список элементов того же типа, что и `x`. Тогда выражение `x:xs` есть список, полученный из списка `xs` с помощью добавления элемента `x` в начало. Но интересно заметить, что конструкцию `(x:xs)` можно использовать и слева от знака «равно», где она соответствует **операции отщепления первого элемента** от списка. Это позволяет рекурсивно определить функцию `length`, которая измеряет длину списка элементов неопределённого типа:

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs
```

Строка «`len [] = 0`» означает, что длина пустого списка равна 0. Строка «`len (x:xs) = 1 + len xs`» означает, что длина списка, от которого отщепили один элемент, равна 1 плюс длина оставшегося списка. Стандартная функция, вычисляющая длину списка, имеет имя `length`.

Лямбда-конструкция

В языке Haskell, также как и во всех остальных функциональных языках, есть возможность конструирования функций прямо в выражении. Для этого используют конструкцию, которая называется **лямбда-конструкцией**:

```
(\x -> выражение от x)
```

Выражение в скобках обозначает функцию от одного аргумента, условно обозначенного как `x`. Выражение от `x`, которое идёт после стрелки, является определением этой функции.

С помощью лямбда-конструкции функции можно конструировать «на лету» прямо в выражениях. Для фиксирования аргумента функции используется символ «\». В частности, «`\x -> x * x *x`» означает функцию $f(x) = x^3$, а выражение «`(\x -> x * x) 3`» равно 9.

Примеры:

```
    (\x -> x*x ) 2  ⇒  4.
    (\x -> x*x ) y  ⇒  y * y.
(\x -> 2*x + 1 ) 2  ⇒  5.
```

Лямбда-конструкции часто используются в первом аргументе функции `map`, которая получает на вход функцию и список элементов, а в качестве результата возвращает список элементов, к которым применена данная функция. Приведём определение функции `map`[2]:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Слева от знака «равно» символ «:» означает «отщепить», а справа от знака «равно» — «присоединить». В частности, выражение

```
map (\x -> x + 10) [1,2,3] ⇒ [11,12,13].
```

означает «прибавить к каждому элементу списка [1,2,3] число 10». В данном сучае можно было использовать не лямбда-конструкцию, а параметризацию функции `add`:

```
map (add 10) [1,2,3] ⇒ [11,12,13].
```

Ещё один пример:

[2]Функция `map` определена в стандартных пакетах Haskell, поэтому, если вы захотите опробовать, как работает данное определение, используйте имя, отличное от `map`.

```
map (\x -> x*x*x ) [1,2,3] ⇒ [1,8,27].
```

А вот простой способ найти первые 5 степеней двоек:

```
take 5 powers where powers = map (2 ^) [1..] ⇒ [2,4,8,16,32].
```

Есть другой способ определения функции `map`: «`map f xs = [ f x | x <- xs ]`». Это соответствует математической записи

$$\texttt{map}(f,\ xs) = \{f(x) \mid x \in xs\}.$$

Конструкция «`x<-xs`» называется «генератором», её следует интерпретировать как «элемент `x` берётся из списка `xs`». Здесь выражение `[1..]` обозначает список *всех* натуральных чисел.

Работа с бесконечными последовательностями

А сейчас мы перейдём к вещам, пугающим и шокирующих «императивных» программистов. Хотите верьте, хотите – нет, но в Haskell есть возможность оперировать бесконечными объектами. Можно завести функцию, которая возвращает бесконечную последовательность натуральных чисел или бесконечную последовательность чисел Фибоначчи, или какую-нибудь другую бесконечную последовательность.

Например, следующая конструкция

```
ones = 1 : ones
```

определяет функцию `ones`, которая возвращает бесконечную последовательность единичек. Действительно, если мы начнём раскрывать это рекурсивное определение, то получим такие выражения:

```
ones = 1 : 1 : ones,
ones = 1 : 1 : 1 : ones,
ones = 1 : 1 : 1 : 1 : ones,
...
```

Это последовательность, которая остаётся равна сама себе после добавления единицы в начало. Неленивые языки, которым «не терпится» сделать сразу то, что от них просят, очень скоро получат переполнение стека или памяти. «Ленивый» язык Haskell не спешит раскрывать определение, данное ему справа от знака «равно», а раскрывает его по мере необходимости. Такое «равно» называют «ленивым равно», оно по сути означает определение функции, а не операцию присваивания. Есть функциональные языки, в которых есть два типа «равно» — ленивое (определение функции) и неленивое (вычисление выражения справа и присваивание результата переменной, что слева от знака «равно»). Например, в языке Mathematica (`http://wolfram.com`) для определения функций используется «`:=`», а для присваивания — просто «`=`».

Рассмотрим функцию `numsFrom`, которая получает один аргумент – целое число n – и возвращает список *всех* целых чисел, которые больше либо равны n:

```
numsFrom n = n : numsFrom (n+1)
```

Используя эту функцию, можно получить бесконечную последовательность квадратов целых чисел:

```
squares = map (^2) (numsfrom 0)
```

Выражение «`(^2)`» означает функцию, которая возводит данное число в квадрат. Получить первые элементы последовательности можно с помощью функции `take`:

`take 5 squares` $\Rightarrow$ `[0,1,4,9,16]`.

Функцию `take` можно было определить рекурсивно:

```
take :: Integer -> [a] ->[a]
take 1 (x:xs) = [x]
take n (x:xs) = x : take (n-1) xs
```

Задача представления числа в виде суммы степеней двойки

Есть специальная операция композиции двух функций, которая обозначается с помощью точки. Если $h(x) = f(g(x))$, то в Haskell это запишется так:

```
h = f . g
```

Интересно заметить, что операция композиции может быть определена в Haskell как обычная функция:

```
(.) :: (b->c) -> (a->b) -> (a->c)
f . g = \x -> f (g x)
```

Она получает на вход две функции и возвращает одну.

Используя операцию композиции, напишем функцию `toDigits`, которая для данного целого числа находит список разрядов двоичного представления, и функцию `countUnits`, которая считает число единиц в двоичной записи натурального числа. В частности для $n = 11$ функция `countUnits` должна вернуть 3, так как $11 = 1+2+8 = 2^0+2^1+2^3$, а для 255 ответ должен быть равен 8, так как $255 = 1 + 2 + 4 + \ldots + 64 + 128$ есть сумма восьми различных степеней двойки.

Введём следующие определения.

```
toDigsI :: Integer->[Integer]
toDigsI n  | n == 0    = [ ]
           | otherwise = (n `mod` 2) : toDigsI (n `div` 2)
countUnits = sum . toDigsI
toDigits   = reverse . toDigsI
```

Функция `toDigsI` для данного числа n находит список его разрядов в двоичном представлении в направлении справа налево. Стандартная функция `reverse` обращает список: получает на вход список и возвращает тот же список, в котором элементы идут в обратном порядке, начиная с последнего до первого:

```
   toDigsI (1+8+16) ⇒ [1,0,0,1,1],
reverse [1,0,0,1,1] ⇒ [1,1,0,0,1],
  toDigits (1+8+16) ⇒ [1,1,0,0,1],
countUnits (1+8+16) ⇒ 3,
     countUnits 255 ⇒ 8.
```

Для того, чтобы найти сами степени двойки, в сумму которых разлагается число, можно использовать операцию «`zipWith (*)`» поэлементного умножения двух списков, а именно, списка разрядов двоичного разложения и списка степеней двойки:

```
powers = (2 ^ ) [0..]
toPowers' = (zipWith (*) powers) . toDigsI
```

Вообще «`zipWith (*)` [a_1, a_2, a_3, ...] [b_1, b_2, b_3, ...]» равно [$a_1 * b_1$, $a_2 * b_2$, ...], где вместо звёздочки может стоять произвольная операция. В частности, «`zipWith (+) [1,2,3] [10,100,1000]`» даст в результате «`[11, 102, 1003]`».

В итоге имеем

```
toPowers' (16+8+1) ⇒ [1,0,0,8,16].
```

Осталось добавить шаг фильтрации нулевых элементов:

```
-- первый способ:
toPowers = (filter (/=0))
           . (zipWith (*) powers)
           . toDigsI
```

Функция `filter` получает на вход булеву функцию (функцию, возвращающую «правду» или «ложь») и список, а возвращает список только из тех элементов, на которых значение этой функции равно «правде». В данном случае мы оставляем только ненулевые элементы:

```
toPowers (16+8+1) ⇒ [1,8,16].
```

Функция «`zipWith`» получает три аргумента. Если указано только два аргумента, то она превращается в функцию от одного аргумента. Это позволяет использовать выражение «`(zipWith (*) powers)`» как функцию от одного аргумента и поместить в цепочку композиции с функцией `toDigsI`. Аналогичная ситуация с функцией `filter`: мы задали для неё первый аргумент – «`(/=0)`» – это функция сравнения с нулём. Второй аргумент остался неопределённым. Он достанется ей по цепочке как значение функции «`(zipWith (*) powers)`» на том, что вернёт ей функция `toDigsI`, применённая к тому, что даст пользователь в качестве аргумента функции `toPowers`. Точки в определении функции `toPowers` играют роль операции «|» (pipe) в стандартной оболочке Linux. С помощью этой операции происходит передача результатов вычисления одной функции на вход другой. В нашем случае была выстроена цепочка из трёх функций.

Функцию `toPowers` можно определить и по-другому:

```
-- второй способ:
toPowers =\n -> filter (/=) (zipWith (*) powers (toDigsI n))

-- третий способ:
toPowers n = filter (/=) (zipWith (*) powers (toDigsI n))
```

В этих способах мы явно вводим аргумент n и используем его в выражении и скобки располагаем уже совсем другим образом.

Если f_1, f_2, f_3 есть функции, то функцию h, равную их композиции, $(h(x) = f_1(f_2(f_3(x))))$ можно определить в Haskell тремя способами:

```
h = f1 . f2 . f3
h x = f1 (f2 (f3 x ) )
h = \x -> f1 (f2 (f3 x ) )
```

Быстрая сортировка

Выше мы рассматривали простые примеры, которые далеки от реальных практических задач. Давайте рассмотрим первый серьёзный алгоритм – алгоритм быстрой сортировки. Несмотря на свою «серьёзность», выглядит он подозрительно просто:

```
-- Быстрая сортировка  на языке Haskell
qsort [ ] = [ ]
qsort (x:xs) =  qsort [y | y <- xs, y<x ]
                ++ [x] ++
                qsort [y | y <- xs, y>=x]
```

Поясним смысл этих строчек. Запись «`qsort [] = []`» означает, что если на вход дан пустой список `[]`, то и в результате будет пустой список. В следующей строчке рассматривается случай, когда список не пуст и от него можно отщепить первый элемент `x`. Оставшаяся часть списка обозначена как `xs`. Выражение «`[y | y <- xs, y<x ]`» равно множеству элементов списка `xs`, которые строго меньше `x`. Выражение «`[y | y <- xs, y>=x ]`» равно элементам списка `xs`, которые больше либо равны `x`. Далее мы сортируем эти два списка с помощью самой же функции `qsort` и склеиваем три списка: список «`qsort [y | y <- xs, y<x ]`», одноэлементный список «`[x]`» и список «`qsort [y | y <- xs, y>=x]`».

Тот же самый алгоритм на языке Си занимает далеко не 4 строчки:

```
/* Быстрая сортировка на языке Си */
void qsort( a, lo, hi )
int a[], hi, lo; {
  int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l < h) && (a[l]<= p))
          l = l+1;
      while ((h > l) && (a[h]>= p))
          h = h-1;
      if (l < h) {
          t = a[l];
```

```
            a[l] = a[h];
            a[h] = t;
        }
      } while (l < h);
      t = a[l];
      a[l] = a[hi];
      a[hi] = t;

      qsort( a, lo, l-1 );
      qsort( a, l+1, hi );
    }
}
```

Таким образом, на языке Haskell многие алгоритмы можно запрограммировать существенно быстрее, нежели на Си или Pascal.

В одной обзорной лекции сложно дать полноценное введение в язык программирования. Большинство приведённых примеров интуитивно ясны, но их, безусловно, недостаточно, чтобы самому начать писать программы на Haskell.

Попробуйте выполнить представленные примеры на интерпретаторе Haskell. Просмотрите обучающие материалы, выложенные на сайтах `http://haskell.org`, `http://wtk.norilsk.net/pm/fp/haskell.html` и `http://www.haskell.ru`.

Зачем нужно функциональное программирование?

Создатели языка Haskell позиционируют Haskell как язык с *чистой функциональная парадигмой*. Этот язык иначально являлся академической разработкой математиков для математиков. Но эта ситуация меняется и он находит себе применение и в промышленные проектах.

Сегодня отмечаются такие достоинства языка Haskell

- проще писать сложные программы, и программы получаются существенно короче;
- программы имеют ясный и "читаемый" код, их несложно понять, даже не зная деталей языка Haskell;
- меньше делается ошибок, так как синтаксис языка Haskell защищает программиста от совершения многих типичных ошибок;
- короче и проще этап проектирования и разработки программ — программист должен просто понять, что ему нужно, и затем описать это на формальном математическом языке;
- создаются адаптивные, легко изменяемые и расширяемые программы.

Кроме того, есть мнение, что благодаря строгой типизации языка, в программах на Haskell не случается системных ошибок и не бывает аварийных ситуаций (`core dump`). Разработчики языка также утверждают, что программы на Haskell получаются более модульные и встраиваемые и предоставляют больше возможностей для повторного использования (re-use). В

частности, представленная программа быстрой сортировки на Haskell (в отличие от программы на Си) может сортировать не только целые числа, но и числа типа `Float` и любые другие объекты, на которых определена операция сравнения.

Обратите внимание на то, что в списке достоинств не указаны такие моменты, как эффективность кода, или экономичное использование памяти, или скорость работы программ. Это не потому, что этих достоинств нет. Просто сегодня акценты индустрии языков программирования переместились в другую сторону. Уже мало кого интересует скорость работы программ или возможность писать супероптимальный код. Ясно, что на практике возникает необходимость ускорить работу некоторых функций, так как они часто вызываются и/или играют важную роль. Но таких кусочков кода не слишком много и им можно уделить отдельное внимание. Например, важные функции, от которых требуется высокая скорость работы, можно реализовать на языке Си, оформить в виде библиотеки и подключить к основному приложению, написанному на удобном для человека языке программирования (языке быстрой разработки), подобному Haskell или Python.

Сегодня важно иметь средства для разработки действительно сложных систем, где важно, чтобы программисты «не увязли» в собственном коде и были способны понять текст программ, который был написан месяц или несколько лет назад. Именно поэтому многие разработчики языков программирования в качестве одного из важных достоинств языка указывают его близость к естественному языку (обычно английскому).

Где используется функциональное программирование

Идеи функционального программирования были оценены по достоинству, и сегодня многие языки программирования содержат те или иные возможности функционального программирования. В частности широко используется лямбда-конструкция и параметризация функций. Примеси функцонального программирования присутствуют в языках Python, Ruby, Perl, JavaScript и многих других языках, которые не являются чисто функциональными, а поддерживают несколько парадигм программирования.

Один из самых «старейших» функциональных языков Lisp по прежнему держит лидирующие позиции, различные его диалекты (в первую очередь, scheme) используются в промышленных проектах.

Перечислим некоторые из них.

- Software AG, одна из главных программистских компаний Германии, разработала на функциональном языке экспертную систему Natural Expert. Пользователи с огромным удовольствием пишут для этой системы свои приложения. Система работает на мейнфреймах IBM.

- Компания Ericsson разработала функциональный язык Erlang для создания системы управления телефонных станций.

- Программа emacs является одной из самых функциональных и гибких (настраиваемых) сред разработки широкого назначения. Функциональность этой программы легко расширяется с помощью добавляемых модулей, написанных на языке Lisp.

- Исследователи в корпорации MITRE используют Haskell для *прототипирования* приложений обработки цифровых сигналов.

Функциональное программирование часто используется на этапах *проектирования* логики программы и *прототипирования*. **Прототипирование** — это быстрая «черновая» реализация базовой функциональности для анализа работы системы в целом. Для прототипирования используют языки высокого уровня абстракции (Java, Perl, Python, Haskell, ...). После этапа прототипирования обязательно следуют этапы пересмотрения архитектуры системы, разработки, реализации и тестирования конечного продукта. На этапе разработки подготавливают систему тестов, по работе которых буду судить о качестве продукта. При реализации решения обычно используют другой, «более машинноориентированный» язык программирования (Си, Си++, ...), пишут более аккуратный, документированный код, а на тестирование системы тратят сравнительно большое количество усилий для достижения качественного результата.

На этапе прототипирования выявляются важные архитектурные ошибки, вносятся поправки в интерфейсы модулей (перераспределяется функциональность между кусочками системы). Прототипирование по мнению многих программистов является самым приятным этапом разработки, так как малыми усилиями создается нечто более-менее работающее. Кроме того, во время прототипирования на программистов обычно «снисходит понимание» и они начинают «видеть», как система должна быть устроена.

Для многих программистов не секрет, что на процедурных языках можно писать объектно-ориентированным образом, а на объектно-ориентированных языках писать программы, следуя процедурному стилю программирования. Аналогично, практически на всех языках можно использовать функциональный стиль программирования. Это связано с тем, что создатели языков стараются сделать их достаточно универсальными, чтобы они успешно использовались при решении разных задач. Абсолютной универсальности достичь невозможно. Хотя есть некоторые удачные экземпляры, такие как язык Python, которые покрывают большой диапазон стилей программирования и в то же время имеют достаточно простой синтаксис. Универсальность языка не всегда является плюсом. Часто она влечёт за собой сложность синтаксиса и неоднозначность языковых конструкций. Конечно, сам язык (транслятор языка) все конструкции интерпретирует вполне однозначно, а вот программист, если язык слишком универсальный, может запутаться.

Поэтому, ограничения, которые вы будете встречать в языках программирования, следует уважать. Они спасают вас от множества проблем, которые могли бы возникнуть, если бы вы писали на «слишком универсальном» языке программирования.

Когда Haskell предпочтительнее Си?

У функциональных языков есть свои недостатки. Так, например, программы, написанные на функциональных языках программирования, работают медленее, чем программы, написанные на Си. В значительной степени это связано с тем, что большинство трансляторов функциональных языков программирования являются интерпретаторами, а не компиляторами.

Коротко, преимущества функциональных языков программирования можно выразить так: *программы пишутся быстрее, но работают они медленнее*. И это хороший компромис — человеко–часы существенно дороже часов работы компьютера.

Представленная искусная реализация функции быстрой сортировки на Си, придуманная Хоаром, безусловно, выигрывает по эффективности у реализации на Haskell (как по времени работы, так и по необходимой для вычислений памяти). Но код на Haskell существенно проще.

Следует отметить, что обе реализации имеют сложность $O(n \log n)$, то есть время работы на списке (массиве) длины n в среднем растёт пропорционально $n \log n$, и в этом «асимптотическом смысле» обе реализации одинаково хороши.

Активно развиваются алгоритмы трансляции функциональных языков, и по эффективности ассемблерного кода они постепенно начинают догонять императивные языки. Здесь следует отметить функциональный язык OCAML, для которого создан удивительно быстрый интерпретатор.

Самое важное достоинство языка Haskell заключается в потенциальной возможности сделать умные трансляторы этого языка. А именно, в *трансляторы* языка Haskell со временем можно будет добавить алгоритмы, которые по данным определениям функций *смогут сами находить наиболее эффективные алгоритмы* их вычисления.

Так например, для вычисления некоторых функций транслятор сам мог бы «догадаться» использовать метод динамического программирования или жадную стратегию. Сегодня теория алгоритмов уже настолько развита, что можно выделить ряд шаблонов алгоритмических задач, и «научить» трансляторы функциональных языков программирования «видеть их» в определениях функций и применять известные эффективные алгоритмы вычисления.

Конечно, язык Си предпочтительнее в целом ряде случаев: системное программирование, драйверы устройств, приложения, от которых требуется высокая производительность, компьютерные игрушки с качественной графикой и др. Кроме того, привязка к языкам Си и Си++ часто связана с наличием большого количество разработанных библиотек, которыми удобно пользоваться, а не создавать необходимую функциональность с нуля. Но всё это довольно специальные случаи. Существует целый класс задач, в которых не требуется высокая скорость работы или специальная уникальная библиотека. Часто оптимизация требуется лишь для небольшого числа функций, а остальная логика может быть запрограммирована на удобном для человека языке программирования, пусть не совсем эффективном с точки зрения скорости работы и использования памяти. Кроме того, все языки программирования стараются делать *расширяемыми*. И язык Haskell тоже расширяемый. Есть возможность, с помощью модулей, написанных на языке Си или Си++, и создавать «родные» для языка Haskell библиотеки функций.

Лекция 26

Виртуализация исполнителей. Архитектура компьютера

Краткое описание: На этой лекции мы снова обратимся к абстрактным исполнителям, но в этот раз в контексте темы «компьютерная архитектура и виртуализация». На примере виртуальной машины BF изучим понятия «адресное пространство кода», «адресное пространство данных», «программный стек». Также рассмотрим базовые принципы построения программных средств: самодокументированность (self documentation) и самоверифицируемость (self verification), которые позволяют повысить надежность (reliability) и способность к развитию (supportability) программ.

Парадигмы, языки программирования и исполнители

Для начала систематизируем понятия, связанные с исполнителями и парадигмами программирования.

Рассмотрим таблицу 26.1. В ней указаны четыре базовые парадигмы программирования – императивная, автоматная, продукционная и функциональная – и для для каждой из них указаны соответствующие ей языки программрования и исполнители. Сразу следует отметить, что приведенная таблица не совсем точна. Языки и исполнители нередко являются мультипарадигменными. Не существует чисто императивного исполнителя или чисто функционального языка. В значительной степени это связано с нечёткостью существующих определений парадигм программирования[1]. Тем не менее, попытка привести парадигмы и языки программирования в систему, несколько огрубив ситуацию, может дать полезное понимание.

С автоматной и продукционной парадигмами программирования мы познакомились на примерах исполнителей машин Тьюринга и алгорифмов Маркова. В первом случае, программист мыслит в терминах состояний и переходов между состояниями. Логика программы описывается как функция переходов: в каком состоянии и при каком условии в какое новое состояние переходить и какие при этом выполнять элементарные действия. В автоматной парадигме программирования важно, что логика работы исполнителя сильно зависит от состояния исполнителя (состояния запоминающих устройств исполнителя).

Продукционная парадигма программирования соответствует случаю, когда логика действий описывается с помощью правил. Правила упорядочены и на каждом шаге исполнитель находит первое правило, которое можно применить, то есть такое правило, у которого левая

[1]Задача классификации парадигм (стилей) программирования подобна задаче классификации видов животных. Деление на царства, группы и семейства в значительной степени условно, реальную значимость имеет лишь полный список свойств вида.

парадигма	**исполнитель**	**языки, инструменты, ...**
императивная	BF	BF, язык ассемблера, Си, Pascal, ..
автоматная	исп. Тьюринга	VHDL, Verilog
продукционная	исп. Маркова	регулярные выражения Perl, make, CLIPS, Prolog
функциональная	комбинаторы, λ-исчисление	Haskell, Lisp

Таблица 26.1: Таблица парадигм программирования.

часть удовлетворяет текущему состоянию. Эта парадигма в действии показана в лекции **??** на примере регулярных выражений языка Perl.

Про функциональное программирование рассказывается в лекции **??**, где на примере языка Haskell показана основная особенность функционального программирования: программа есть набор математических определений, где функции выступают наравне с числами и могут выступать как в роли аргументов, так и в роли результатов вычисления функций. Функциональное программирование очень близко к продукционному. Действительно, всякое математическое определение по сути является правилом. У определения есть левая часть (что определяется) и правая часть (каким образом определяется). Так же как и в продукционном программировании, в функциональном отсутствует возможность явно управлять последовательностью выполнения элементарных действий. Программист пишет определения (правила), которые верны, а задачу определения последовательности выполнения элементарных действий, необходимых для получения результата согласно совокупности определений, оставляет интерпретатору языка.

Но следует сказать, что исполнитель языка Haskell нельзя назвать элементарным функциональным исполнителем. Это высокоуровневый язык программирования промышленного уровня, который позволяет писать программы, похожие по внешнему виду на обычные императивные. На Haskell, в частности, можно писать согласно объектно-ориентированнойъ парадигме. Грубо говоря, Haskell является слишком высокоуровневым, чтобы быть чистым представителем некоторой парадигмы. В роли *канонического функционального исполнителя* может выступать формальная система преобразований в *комбинаторной логики*. В этой формальной системе определен класс объектов, называемых комбинаторами. Каждый комбинатор является отображением множества комбинаторов на себя. После уточнения способа записи входных данных (как с помощью базовых комбинаторов записывать натуральные числа) и описания алгоритма вычислений эту формальную систему можно интерпретировать как исполнитель. Подробнее об этом можно прочитать в [14], [17, Функциональное программирование].

И наконец, мы ещё никого не определили на роль канонического императивного исполнителя. Эту роль с успехом может выполнять исполнитель программ на языке BF.

Далее мы напишем виртуальный исполнитель BF на языке Си и проведем его анализ.

Виртуализация

Деление на языки и исполнители чисто условное. В действительности, с каждым языком программирования неявно связан абстрактный исполнитель этого языка. Как именно действует этот исполнитель описывается в спецификации языка, при этом термин «исполнитель» может и не использоваться.

Для некоторых языков существуют их исполнители — виртуальные (программа) или физические (в виде устройства). Например, для Java байт-кода существует виртуальный исполнитель «Virtual Java Machine», для языка «машинный код i386» существует физические устройства-исполнители: процессоры.

Определение 26.1. **Виртуальный исполнитель $\mathbf{I}_1$** *(виртуальная машина $\mathbf{I}_1$) — это программа для некоторого исполнителя $\mathbf{I}_2$, которая получает на вход программу для исполнителя $\mathbf{I}_1$ и выполняет её. При этом необходимо наличие у исполнителя $\mathbf{I}_2$ всех устройств, присутствующих у исполнителя $\mathbf{I}_1$. Отсутствующие устройства должны эмулироваться с помощью существующих.*

Несложно увидеть связь виртуализации с понятием эмуляции, данным в лекции 3 на странице 61. Эти понятия почти синонимичны, отличие лишь в том, что понятие «виртуализация» более общее и употребляется в ряде других контекстов: «виртуальный диск с данными», «виртуальный рабочий стол», «виртуальный компьютер», «виртуальные деньги» и др.

Виртуальный исполнитель BF

Решим задачу создания виртуального исполнителя одного из простейших императивных языков программирования — языка BF[2].

Исполнитель языка BF включает в себя следующие компоненты:

- два потока данных — входной (input) и выходной (output);
- лента данных, ячейки которой хранят числа типа `unsigned char`;
- лента с кодом, на которой записана программа на языке BF;

Лента данных бесконечна в обе стороны. Лента с кодом бесконечна в одну сторону, но можно также считать, что она ограничена и в точности равна размеру заложенной в исполнитель программы программы.

Исполнитель перемещается по ленте с кодом и последовательно исполняет команды, записанные в ячейках. Указатель на текущую ячейку ленты кода обозначим как `code_pt`. Есть также указатель на текущую ячейку ленты данных — `data_pt` Есть 8 допустимых команд, каждая из которых представляет один символ:

```
+ - < > , . [ ]
```

На рисунке 26.1 показано соответствие этих команд командам языка Си.

[2]Мы рассмотрим модифицированную версию языка BF. Отличие будет в логике выполнения циклов, кроме того, будут введены дополнительные команды.

команда BF	соответствующая команда языка Си
>	++data_pt;
<	--data_pt;
+	(*data_pt)++;
-	(*data_pt)--;
.	putc(*data_pt);
,	*data_pt = getc();
[	do {
]	} while (*data_pt);

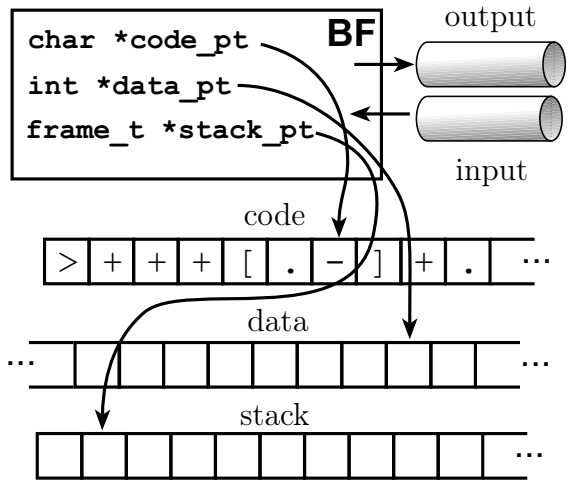


Рис. 26.1: Устройство исполнителя BF: бесконечная в обе стороны лента данных (data), бесконечная в одну сторону лента с программным кодом (code), бесконечная в одну сторону лента информацией о начатых циклах (stack), потоки ввода и вывода (input, output).

Язык BF придуман не для того, чтобы на нём программировать. Он представляет собой некоторую простую модель вычислений, логически целостную и тьюринг-полную. Язык BF удобен исключительно в учебных целях. Причём, практика программирование на BF, возможно, не так важна, как практика создания и тестирования виртуального исполнителя BF.

Язык BF является простейшим языком императивного программирования. Он содержит две простейшие арифметические операции, две простейшие операции перемещения по памяти, команды для посимвольного ввода и вывода данных из потоков, и две операции для организации цикла. Этих операций достаточно, чтобы запрограммировать любую вычислимую функцию.

Исполнитель BF проще для понимания нежели исполнители машин Тьюринга и алгорифмов Маркова, и может рассматриваться как ещё один канонический исполнитель, с помощью которого можно определить понятие вычислимости.

Основной цикл обработки команд виртуального исполнителя мог бы выглядеть следующим образом:

```
while ( 1 ) { // потенциально бесконечный цикл
    char command = get_next_command();
    switch( command ) {
        case '+': (*data_pt)++; break;
        case '-': (*data_pt)--; break;
        case '>': data_pt++; break;
        case '<': data_pt--; break;
        case '.': putc( *data_pt ); break;
        case ',': *data_pt = getc(); break;
        case '[': /* ???? */; break;
        case ']': /* ???? */; break;
        case 0  : goto STOP;
```

```
        default:
          printf("Некорректная команда '%c'.\n", command);
    }
}
```

Большая часть команд программируется прямо в соответствии с их определением, без каких либо дополнительных построений. На ленте `data` храняться байты. С помощью команд `>` и `<` можно перемещаться по ленте, а с помощью команд `+` и `-` можно увеличивать и уменьшать значения ячеек на 1 (при этом $255 + 1 \to 0$ и $0 - 1 \to 255$).

Важно понимать, что при выполнении команд ввода и вывода символам ставятся в соответствие их ASCII коды: при считывании символа в ячейку записывается его ASCII код, а при выполнении команды вывода печатается символ, ASCII код которого равен хранящемуся в ячейке числу.

Особый интерес представляют команды `[` и `]`. Эти команды позволяют организовать циклы. Из описания следует, что квадратные скобки должны быть сбалансированными, то есть, если из программы убрать все символы кроме скобок, должно остаться правильное скобочное выражение. То, что находится между парными скобками, естественно назвать телом цикла.

Команда `[` является меткой начала цикла, к которой нужно перейти при достижении парной закрывающейся скобки. Возвращаться к началу цикла нужно только в том случае, если активная ячейка на ленте `data` содержит не нулевое число (`*data_pt != 0`). Если же активная ячейка содержит 0, цикл заканчивается и выполнение передается следующей за циклом команде. Необходимо уметь обрабатывать и вложенные циклы. Отсюда следует, что исполнитель BF должен хранить информацию о вложенных циклах, и ясно, что структурой хранения этой информации должен быть стек (вспомните алгоритм проверки правильности скобочного выражения, основанный на стеке).

Итак, у BF есть ещё одно устройство хранения данных — стек. Он показан на рисунке 26.1. При выполнении команды `[` в стек помещается её адрес на ленте `code` (адрес начала цикла). Когда выполняется команда `]`, исполнитель «смотрит» значение активной ячейки на ленте `data`. Если оно не равно нулю, то из вершины стека берётся (без извлечения) значение адреса начала цикла, и управление передается команде, следующей за командой начала цикла `[`. Если же значение активной ячейки на ленте `data` равно нулю, то цикл заканчивается, из стека извлекается значение адреса начала цикла (за ненадобностью), и осуществляется переход к следующей за циклом команде.

Код обработки циклов может быть таким:

```
switch( command ) {
    /* ... */
    case '[': push( code_pt ); break;
    case ']': {
                if ( *data_pt ) {
                    // top возвращает вершину стека,
                    // не извлекая её из стека
                    code_pt = top();
                } else {
                    // удаляем адрес начала цикла
                    pop();
                }
```

```
                    break;
                }
        /* ... */
    }
```

Осталось реализовать функции для работы со стеком (`push`, `pop`, `top`) и функцию `get_next_command`. Кроме того, необходимо определится с методом передачи данных. Исполнителю BF необходимо передать текст программы, входные данные для программы, и получить результат вычислений. Предлагается программу хранит в виде файла и передавать её исполнителю BF в виде аргумента в командной строке. А входные и выходные данные передавать посредством стандартных потоков ввода/вывода. Вот как будет выглядеть вызов программы `a.bf`, в которую пользователь ввёл `AAAAAA` и получил в результате `ABCDEF`:

```
> ./bf a.bf
AAAAAA   # ввёл пользователь
ABCDEF   # вывела программа
```

Программа 26.1 является **прототипом** виртуального исполнителя BF. Это именно прототип, обладающий целым рядом недостатков, которые мы подробнее разберём ниже.

Программа 26.1: Прототип виртуальной машины BF

```
#include <stdio.h>
#include <stdlib.h>
#define N 10000

typedef unsigned char BYTE;

int Debug = 0;             // равно 1 в режиме отладки
BYTE  data[2*N];
BYTE *data_pt = data + N; // устанавливаем указатель на середину
                           // выделенной области, чтобы можно
                           // было двигаться и влево, и вправо
char code[N];
char *code_pt = code;      // указатель на исполняемую команду

char* stack[N];
char** stack_pt = stack;  // указатель не первую
                           // пустую ячейку
/* Работа со стеком */
char* pop() {
    return *(--stack_pt);
}
void push(char* code_p) {
    *(stack_pt++) = code_p;
}
char* top() {
    return *(stack_pt - 1);
}
```

```
/* Следующая команда */
int get_next_command() {
    return *(code_pt++);
}
int main(int argc, char *argv[]) {
    int i;
    char *filename = NULL;
    FILE *f = NULL;
    if( argc <= 1 ) {
        _log(1,"Мало аргументов.\n");
        exit(1);
    }
    for( i = 1 ; i < argc; i++ ) {
        if( strcmp("-d", argv[i]) == 0 ) {
            Debug = 1;
        } else {
            if( filename != NULL ) {
                _log(2, "Задано несколько файлов.\n");
            }
            filename = argv[i];
        }
    }
    f = fopen(filename, "r");
    if( f == NULL ) {
        _log(1, "Не могу открыть файл '%s'.\n", filename);
        exit(2);
    }
    // Считываем программу в память
    while( !feof(f) ) {
        *(code_pt++) = fgetc(f);
    }
    // Ставим символ конца программы - нулевой байт
    *code_pt = 0;
    // Указатель code_pt устанавливаем на начало программы
    code_pt = code;
    while( 1 ) { // потенциально бесконечный цикл выполнения
        char command = get_next_command();
        switch( command ) {
            case '+': (*data_pt)++; break;
            case '-': (*data_pt)--; break;
            case '>': data_pt++; break;
            case '<': data_pt--; break;
            case '.': fputc(*data_pt, stdout); break;
            case ',': *data_pt = fgetc(stdin); break;
            case '[': push( code_pt ); break;
            case ']': {
```

```
                    if( *data_pt ) {
                        code_pt = top();
                    } else {
                        pop();
                    }
                    break;
                }
            case ' ':          // пропускаем
            case '\n':         // пробельные
            case '\t': break; // символы
            case 0: goto STOP;
            default:
             _log(2, "Некорректная команда %c.\n", command);
        }
        if( Debug ) { //
            _log(3,"Code shift = %d.\n", code_pt - code);
            _log(3,"Data shift = %d.\n", data_pt - data - N);
            _log(3,"Value = %d (%c).\n", *data_pt, *data_pt);
        }
    }
STOP:
    return 0;
}
```

Тем не менее данный прототип достаточно успешно выполняет несложные программы и позволяет увидеть язык BF в действии. Для удобства программа принимает опцию `-d`, которая включает режим отладочного вывода. Для этого используется функция `_log`, определённая ниже в коде 26.2.

Архитектура исполнителя BF является более близкой к реальным архитектурам, в нём присутствует важное разделение внутренней памяти и потоков ввода и вывода. Язык BF неизбежно требует от исполнителя наличия стека для хранения адресов начал циклов. И здесь можно проводить аналогию с программным стеком, используемым для организации исполнения подпрограмм в реальных архитектурах.

Журналирование

Журналирование (англ. *logging*) — вывод информации о выполняемых действиях, а также значений данных программы в поток вывода (файл), который называется журналом (log).

Журналирование играет ключевую роль при отладке программ. Если речь идёт не об одной программе, а о сложной системе, в которой одновременно работает несколько взаимодействующих программ (процессов), то журналирование становится практически единственным инструментом отладки.

Какую отладочную информацию имеет смысл выводить в журнал? Здесь нужно найти правильный баланс — не писать слишком много, но в то же время достаточно, чтобы при

возникновении проблем, по журналу можно было локализовать их причину. Если записей в журнале будет слишком много, в них можно запутаться. Конечно, есть инструменты, подобные UNIX-программе `grep`, которые позволяют значительно облегчить поиск нужных записей по ключевым словам.

Есть простая, но эффективная идея, — указывать уровень важности сообщения (критичности возникшей ошибки). Обычно сообщения делят на критические ошибки, ошибки, предупреждения и информационные сообщения. Соответственно определяются функция `_log`, которая пишет в журнал (роль журнала может играть поток ошибок `stderr`). Она ожидает те же аргументы, что и функция `printf`, только имеет дополнительный первый аргумент — уровень важности сообщения. Число её аргументов неопределено, также как и у функции `fprintf`. В коде 26.2 показано, как следует определять такие функции. Дополнительную информацию можно получить в документации `man vprintf` и `man stdarg`.

Программа 26.2: Функция для журналирования.

```
#include <stdarg.h>
/* ... */
void _log(int log_level, const char *msg, ...) {
    va_list v;
    va_start(v, msg);
    if(log_level <= LOG_LEVEL) {
        switch ( log_level ) {
            case 0: printf("CRITICAL: ");break;
            case 1: printf("ERROR   : ");break;
            case 2: printf("WARNING : ");break;
            case 3: printf("INFO    : ");break;
            default: printf("BTW     : ");break;
        }
        vfprintf(stderr, msg, v);
    }
}
```

Надёжность программных средств

Давайте «доведём до ума» наш исполнитель. Есть ряд причин, которые мешают назвать код 26.1 полноценной «промышленной» реализацией виртуального исполнителя BF. Всякое программное средство (ПС) должно удовлетворять следующим критериям

- ПС должно **работать правильно** для всех входных данных. А в исключительных ситуациях ПС должно работать логично, в частности выводить соответствующее информационные сообщения, стараться «выжить», или по меньшей мере ничего не испортить, сохранить данные, сделать всё возможное для того, чтобы пользователь мог после анализа сообщений об ошибке выяснить причину возникшей проблемы.

- ПС должно быть **легко поддерживаемым**, то есть легко развиваемым, расширяемым, дополняемым. Необходимо, чтобы ПС легко было тестировать, легко проводить анализ сбоев и устранять ошибки. Указанные операции объединяют термином **поддержка кода** (англ. *code maintenance*).

Первый критерий касается *поведения программы* с точки зрения внешнего пользователя, второй — *внутреннего устройства программы*. Он требует, чтобы у программы была продуманная структура и качественно написанный код, чтобы любой сторонний программист мог за короткое время разобраться в ней и, при надобности, расширить её функционал. Эта важнейшее свойство, которое косвенным образом определяет, какими усилиями может быть достигнуто первое.

Наш виртуальный исполнитель не удовлетворяет ни первому, ни второму критерию. В частности, если во время выполнения указатель `data_pt` (`stack_pt`) выйдет за пределы массива `data` (`stack`), или программа имеет размер более `N` символов, исполнитель поведёт себя непредсказуемым образом.

Не так просто определить, что значить «правильно работать»[3]. В нашем случае есть формальное описание языка BF. Но в этом описании не указано множество моментов:

- Какие допустимые размеры у массивов `data`, `code`, `stack`?
- Что делать, если указанный пользователем файл программы не существует?
- Что делать, если файл содержит некорректную программу (присутствуют недопустимые символы, квадратные скобки несбалансированны)?
- Что делать, если закончится память?
- Что делать, если поток ввода (вывода) был закрыт пользователем, а исполнителю необходимо выполнить команду ввода (вывода) символа?

Программисту необходимо самостоятельно на них ответить и соответствующим образом запрограммировать исполнитель BF. Ответы более менее очевидны — размеры хранилищ `data`, `stack` (не обязательно реализовывать их как массивы) необходимо расширять по мере надобности. Размер `code` должен быть равным размеру программы. В случае возникновения проблем, необходимо выводить исчерпывающее информационное сообщение (см. журналирование).

[3]Фраза «работать правильно» может иметь несколько интерпретаций:

- так, как представляет себе это заказчик;
- так, как понял и сформулировал менеджер проекта;
- так, как записал в техническом задании постановщих задания;
- так, как понял это задание программист;
- так, как решил сделать программист, который (иногда заслуженно) считает себя умнее постановщика задания, менеджера и, тем более, заказчика.

Ясно, что содержание данных пунктов может отличаться. А полученное в результате ПС может отличаться от всех этих пунктов. Разработка полного непротиворечивого формального описания логики работы ПС — отдельная тема.

Какой бы подробной не была спецификация логики работы ПС, она просто не может включить (описать), каким должно быть поведение ПС в экстремальных условиях. Например, к серверу может прийти слишком много запросов в секунду, может случится ошибка на жестком диске, ошибка может случится в каких-то внешних по отношению к ПС подсистемах.

Итак, ПС должно работать правильно для всех входных данных. Но для большинства ПС такая абсолютная правильность — очень серьезное требование, которое практически невозможно удовлетворить (исполнитель BF слишком прост и поэтому является исключением).

Понятие «надежность» (по сравнению с «абсолютной правильностью») более адекватно и точно отражает то, что в реальности можно требовать от ПС:

Определение 26.2. **Надёжность** *ПС (анг.* reliability*) — это высокая вероятность правильности работы ПС как при нормальных, так и при экстремальных условиях. Надёжность достигается за счет учёта возможных сбоев в различных подсистемах, от которых зависит работа ПС, и многоуровневой защиты от сбоев как внутренних, так и внешних подсистем.*

В экстремальных ситуациях словосочетание «правильность работы» имеет смысл заменить на «логичность поведения ПС». Даже в экстремальных ситуациях ПС должно делать попытки «выжить» и обеспечить сохранность данных, а также предоставить возможность восстановления работоспособности после «перезагрузки».

Надёжность в значительной степени определяется отсутствием ошибок в коде и и достигается с помощью массового и многопланового тестирования ПС.

Также существуют специализированные инструменты, осуществляющие анализ кода и проверяющие выполнение необходимых условий. Эти условия указываются прямо в коде, и непосредственно связаны с требованиями, предъявляемыми к ПС в спецификации логики работы. Например, это могут быть условия, которым должны удовлетворять данные в момент вызова некоторой функции и в момент завершения выполнения функции (post- и prerequisites). С помощью анализа логических условий, расставленный в коде, задачу тестирования можно трансформировать в задачу доказательства ряда утверждений (программа никогда не зависает, выходные данные удовлетворяют заданной спецификации и др.) Задача доказательства правильности работы программ довольно сложна, но есть прецеденты создания программ, для которых с помощью специальных инструментов доказано, что они *всегда* работают правильно.

Степень приближения к выделенному слову «всегда» соответствует надёжности.

Неправильно думать, что надёжность достигается на этапах кодирования и тестирования. Думать о надёжности ПС следует начинать на этапе проектирования. Надёжность обеспечивается архитектурой и внутренней логикой работы компонент. В частности, уменьшение числа зависимостей между компонентами за счёт тщательно продуманной архитектуры и исключения лишних зависимостей[4] позволяет уменьшить количество источников проблем и, тем самым, облегчить задачу достижения надёжности на этапах кодирования и тестирования. Важным способом достижения надёжности является разбиение на модули, и определение интерфейсов базовых библиотек, которые ставили бы программистов в такие рамки, в которых сложно совершить ошибку.

Часто излишнее стремление к надёжности приводит к слишком жёсткой архитектуре и слишком медленному коду. Например, в коде в самых различных местах можно вставлять кусочки, проверяющие корректность данных и состояний устройств, с которыми будет связано следующее действие. Здесь, как и везде в информационных технологиях, важно чувствовать баланс и не доводить стремление к надежности до крайности. Важно распределить задачу достижения надёжности по различным этапам разработки ПС, на каждом этапе делать то, что может быть сделано на этом этапе меньшими усилиями, чем на других. Шаги, предпри-

[4]См. [**?**], а также «Ортогональность в проектировании и принцип DRY» в Викиучебнике [17] и «Don't repeat yourself» в Википедии [18].

нимаемые для достижения надёжности, не должны сильно усложнять разработку.

Качество кода

Итак, вы решили разработать некоторое ПС и надеетесь, что оно будет использоваться. Чтобы ваши надежды оправдались, ПС должно обладать целым рядом свойств, который коротко можно обозначить словосочетаниями «товарный вид», «промышленный уровень» и т.п.

Конечно, нужно стремится достигнуть этих качеств как можно скорее и как можно меньшими усилиями. Но менеджерам проекта и программистам следует не забывать известные поговорки «тише едешь — дальше будешь» и «лучше день потратить, а потом за пять минут долететь». Во всяком проекте полезно иногда рассуждать на метауровене и не просто напролом идти к намеченной цели, а поработать над средствами, обеспечивающими удобный и надежный процесс движения к цели.

В программистских проектах показателем уверенного движения к намеченной цели является **качество кода** и удобство программистов при работе с разработанными частями ПС. Что нужно программистам, чтобы процесс разработки был удобным и эффективным? Проблема сложная и многогранная. Обозначим самые важные моменты, отражающие качество кода:

- выбранное разбиение ПС на модули действительно удобно, каждый модуль представляет простой целостный логический блок, с чётко определённым, понятным функционалом;
- код документирован, то есть имеется документация для разработчиков, описывающее назначение отдельных компонент, функций, и др. программных единиц, которая позволяет программистам понять, что уже сделано и как этим пользоваться;
- код не нуждается по пояснениях;
- код позволяет легко осуществлять отладку и поиск ошибок.

Первый пункт самый важный. Для его достижения нужен талант и опыт архитектора, а также хорошее логическое мышление. Три оставшихся пункта являются важными, но относятся сегодня уже к «чисто техническим» вопросам. Обобщить их можно так: код должен быть «самостоятельным». У этой самостоятельности есть три аспекта:

- понятность;
- самодокументированность;
- самоверифицируемость.

Понятность кода. Комментарии

Код должен легко пониматься как его автором (даже через год после написания), так и сторонним программистом.

Необходимо, чтобы за короткое время в любой странице написанного кода можно было разобраться и при необходимости осуществлять его поддержку (развивать, устранять ошибки, осуществлять отладку, анализировать).

Это очень серьёзное требование. Заметьте, что аналогичное требование к художественному тексту покажется нелепым. Если открыть страницу из середины книжки, то скорее всего упоминаемые герои и контекст ситуации окажутся неизвестными. Требовать от автора понятности содержания при чтении с любого места не имеет смысла. Тем не менее, в случае программного кода подобное требование до некоторой степени имеет право на существование.

Во-первых, нужно писать комментарии, а ещё лучше — писать такой код, который понятен без комментариев или требует небольших комментариев. Кроме того, контекст любого кусочка кода должен легко восстанавливаться. Должны быть средства (это уже требование к среде разработки), которые позволяют легко получить информацию по объектам, используемых в коде и быстро перейти к их определению.

Ясность кода достигается с помощью

- модульности программы и правильного именования модулей;
- разделения выполняемых задач на функции, с простой семантикой, явно выраженной в имени функции и именах переменных;
- структуризации тела функций;
- объявления переменных в правильных местах и правильного именования переменных.

Комментарии должны быть умеренными. Их следует писать в следующих местах:

- перед функциями описывать их семантику — что делают, что получают на вход, что возвращают в качестве результата;
- рядом с объявлением переменных указывать их «физический смысл»;
- рядом с «подпорками» описывать назначение подпорки;
- в местах, где присутствуют «призраки», указывать их наличие.

Напомним, что «подпоркой» в программистском жаргоне обозначают кусок кода, посвященый обработке крайнего случая входных данных, который не может быть обработан согласно общей разработанной схеме. В частности, в алгоритме сортировки quick sort можно было бы написать подпорку на случай, если входные данные упорядочены по возрастанию или убыванию, чтобы избежать квадратичного времени работы для случая, когда входной массив упорядочен по возрастанию или убыванию.

«Призрак» — это сущность, дуальная к «подпорке». «Призраки» — это информация, сыгравшая существенную роль при написании кода, но которая в этот код не попала. Часто «призраками» являются математические теоремы, которые легли в основу кода, или инварианты цикла.

Инвариант цикла — это функция от данных программы, значение которой не меняется (или монотонно убывает) от итерации к итерации. Рассмотрим для примера код, вычисляющий a^n:

```
f = 1; b = a;
while ( n ) {
    if ( n % 2 == 1 ) {
```

```
        f *= b;
    }
    b *= b;
    n /= 2;
}
```

Этот код станет яснее, если добавить комментарий

```
// на каждой итерации  f * (b ^ n) равно искомому числу
```

В данном случае инвариант цикла равен $\mathtt{f} \cdot (\mathtt{b}^{\mathtt{n}})$ и совпадает с искомым числом. Это выражение не меняется, так как на каждой итерации переменная `b` возводится в квадрат, а переменная `n` делиться на два. Ситуация, когда `n` делится на 2 с остатком обрабатывается с помощью условного оператора.

Когда `n` уменьшается до 0, переменная `f` становится равной искомому числу.

Обобщить указанные дуальные понятия можно так: «призрак» — элемент теории, не попавший в код, а «подпорка» — элемент кода, никак не представленный в теории (опущенный в теории из-за своей незначительности или специфичности).

Подпорки и призраки следует комментировать, если вы хотите чтобы ваш код был поддерживаемым и вы сами могли в нём разобраться по прошествии некоторого времени.

Самодокументированность кода

Самодокументированность кода (англ. *code self-documentation*) — это метод документирования кода с помощью комментариев прямо в коде. Специальная программа автодокументирования обрабатывает исходные коды, анализирует их, сканирует комментарии и создает документацию, удобную для разработчиков. При этом фиксируются некоторые правила комментирования функций и других единиц кода. Эти правила включают правила форматирования кусочков текста (смена стиля шрифта, форматирование заголовков различных уровней, форматирование ссылок на различные компоненты). Результатом работы программы автодокументирвания обычно является набор связных гипертекстовых документов, иногда со схемами (см. doxygen).

Метод самодокументироваия облегчает программистам задачу поддержки согласованности документации и кода (исправляя код, несложно в пяти строчках выше исправить соответствующий кусочек документации)[5].

Самоверифицируемость кода

Самоверифицируемость кода (или самопроверяемость кода) (англ. *code self-verification*) — это метод добавления в различные части кода проверок на отсутствия ошибок в имеющихся в программе данных. Данные должны быть корректными (например в языке Си часто проверяется, что указатель не ноль) и логически целостными. В значительной степени самоверифицируемость предназначена для самоконтроля программиста — в различных местах кода он

[5]Кроме самодокументированности кода важна также *самодокументированность программы*. Во многие программы встраивается документация пользователя, которая может быть вызвана из графического интерфейса пользователя или (если это консольное приложение) с помощью ввода специальной команды или указания специальной опции.

пишет проверки на то, что данные имеют тот «вид», который он ожидает. Данные должны быть правильными, потому что они были вычислены с помощью функций, которые должны работать правильно. Но вдруг в этих функциях допущена ошибка, или вдруг используемые внешние модули работают со сбоями.

Самопроверкой не следует злоупотреблять. Необходимо писать проверяющий код в тех местах, где действительна велика вероятность сбоя. После обнаружения некорректности данных проверяющий код должен написать соответствующее сообщения в журнал и закончить выполнение программы. После того как программа отлажена и ошибки не обнаруживаются, можно исключить проверяющий код. Обычно это делается с помощью директив препроцессора:

```
#ifdef _DEBUG
   // код, проверяющий консистентность данных
#endif /* _DEBUG */
```

Заметим, что словосочетание «самоверифицируемый код» звучит пожалуй слишком громко. Речь идёт не о достижении уверенности в абсолютной правильности кода, а лишь о некотором вспомогательном методе простого обнаружения и устранения ошибок. Его следует использовать в комбинации с массовым и многоплановым тестированием, коллективным анализом кода и пошаговой отладкой программы.

Задачи на программирование и модификацию исполнителя BF

Задача Л26.1. Напишите программу, которая выводит 7 раз букву `A`.

Задача Л26.2. Напишите программу, которая 50 раз выводит латинский алфавит. Используйте конструкцию цикл в цикле.

Задача Л26.3. Модифицируйте исполнитель BF так, чтобы при воде и вывода латинская буква `A` соответствовала числу `0`. То есть выполняя команды `.` при `*data_pt == 0` исполнитель должен выводить букву `A`. А команда `,` при введённой букве `A` должна привести к обнулению ячейки `*data_pt`. Добавьте возможность, чтобы смещение ASCII кодов можно было задавать при запуске в опции `-0`, например, чтобы символ `A` соответствовал 0 необходимо запускать программу следующим образом:

```
> ./bf -0A a.bf
```

По умолчанию смещение должно быть равно 0.

Задача Л26.4. Напишите программу, которая считывает слово из потока ввода до первой встретившейся буквы `A`, а затем выводит это слово 10 раз.

Задача Л26.5. Модифицируйте логику выполнения циклов исполнителя BF так, как указано в таблице эквивалентов на языке Си:

команда BF	**соответствующая команда языка Си**
`[`	`while ( *data_pt ) {`
`]`	`};`

Задача Л26.6. Сделайте так, чтобы память под `data`, `stack`, `code` выделялась динамически. Добавьте возможность увеличения размеров `data` и `stack` во время работы виртуального ис-

полнителя. Это можно сделать двумя способами —- используя функцию `realloc` или моделируя массивы `data` и `stack` с помощью списков ячеек. При этом во втором случае лучше сделать следующую оптимизацию: запрашивать память не под каждую ячейку, а под группы соседних ячеек и представлять ленты `stack` и `data` как список групп (массивов фиксированной длины) ячеек.

Задача Л26.7. Сделайте виртуальный исполнитель BF более надёжным. Осуществляйте проверку успешности выполнения команд, связанных с открытием файлов, вводом/выводом, выделением памяти. Проверяйте, что стек не пуст перед выполнением команда `pop`, и что он не заполнен до конца перед выполнением команды `push`.

Задача Л26.8. (самоверифицируемость исполнителя BF) Добавьте в код виртуального исполнителя BF элементы самопроверяемости. Перед тем, как работать с переменными, указывающими на ячейки стека, кода или данных, проверяйте их корректность.

Задача Л26.9. (самодокументированность исполнителя BF) Добавьте в код виртуального исполнителя BF элементы самодокументированности. Напишите комментарии в тех местах, где они по вашему мнению нужны. Добавьте вывод описания синтаксиса аргументов виртуального исполнителя BF при запуске с опцией `-h` и при некорректных аргументах командной строки.

Задача Л26.10. (расширение языка конструкцией «арифметический цикл») Добавьте в язык BF возможность использовать арифметические циклы, а именно, введите команды `(` и `)`, обозначающие начало и конец арифметического цикла. Перед открывающейся скобкой `(` может стоять целое неотрицательное число — число итераций цикла. Например, команда `10(>+)` означает 10 раз выполнить команды `>+`, то есть увеличить на единицу значение десяти ячеек правее активной ячейки. Для удобства круглые скобки можно не ставить, если тело цикла состоит одной команды или одного цикла типа `while` с квадратными скобками. Примеры:

`10(++)`	увеличить значение ячейки на 20
`15>`	сместиться вправо на 15 ячеек
`100[>.<-]`	цикл `while` внутри арифметического цикла из 100 итераций

Подсказки:

- Необходимо изменить тип единицы, помещаемой в стек. Назовем этот тип `frame_t` («стек фрейм»). Он будет состоять из трех полей: тип цикла, адрес начала цикла, число оставшихся итераций. В случае цикла с квадратными скобками третье поля не используется.
- Имеет смысл сделать препроцессинг программы BF. Указанные в условии умолчания относительно цикла из одного оператора и цикла включающего цикл с квадратными скобками сильно усложняют код обработки. Удобно перед исполнением обработать код и добавить в него опущенные круглые скобки.

Задача Л26.11. (сохранение образов процессов – process hibernation) Добавьте возможность сохранения образа процесса (выполняемой программы BF) в файл, а именно, пусть в любой момент можно остановить вычисления, сохранить состояние процесса в файл, а также загрузить существующий образ процесса в виртуальную машину BF и продолжить когда-то прерванные вычисления. Переход в режим зарузки/сохранения образа должен осуществляться при получении сигнала (см. сигналы в стандарте POSIX, `man signal` и сигнал SIGTERM). Обратите внимание на сложности, возникающие при сохранении состояния потоков ввода и вывода, связанные с их буферизацией. Как их можно преодолеть?

Какие бывают архитектуры

Исполнитель BF является императивным исполнителем. Программы на этом языке представляют собой список действий. Индикатором «императивности» также служит конструкция `switch` в основном цикле обработки команд. У исполнителя BF есть глобальное состояние — совокупность значений данных в массивах `code`, `data`, `stack` и указателей `code_pt`, `data_pt`, `stack_pt`.

Парадигмы программирования являются довольно абстрактными понятиями. На практике эти понятия заменяются на конкретные архитектуры, для которых уже сложно сказать, какая именно парадигма программирования заложена в их основу.

Архитектуры исполнителей различаются по следующим признакам:

- **принцип организации входных и выходных данных и внутреннего хранилища данных (интерфейсы взаимодействия с различными запоминающими устройствами и интерфейсы работы с входными/выходными данными))**

  Возможные значения: потоки данных (концы двух «труб»: из одной берём, в другую кладём), последовательность ячеек с возможностью произвольного доступа по номеру ячейки, двусвязный список, стек, ... Возможны комбинации различных типов структур данных.

- **тип основной логической единицы программы, базовый набор доступных единиц (команд, базисных функций, ...)**

  Возможные значения: инструкция, правило, определение, диаграмма перехода, различные комбинации этих объектов. Существует множество различных типов инструкций, правил, определений. Бывают высоко- и низкоуровневые инструкции. Например, бывают элементарные правила — правила-подставновки, а бывают правила, в которых левая часть — шаблон, а правая — алгоритм.

- **принцип осуществления модульности программ, принцип взаимодействия модулей**

  Возможные значения: сообщения, разделяемая память, потоки, общая шина данных, и др. Этот пункт касается высокоуровневых исполнителей (по большей части виртуальных), многопроцессорных и многоядерных архитектур.

Указанные признаки лишь частично характеризуют архитектуру. Полной информацией является схема (алгоритм) работы исполнителя и детальное описание всех его компонент.

В нашей реализации исполнителя BF мы явно разделили данные, код и стек — их адресные пространства не пересекаются. В реальных архитектурах адресное пространство общее, что даёт возможность программам во время исполнения менять собственный код. Самомодификация кода является важным инструментом современного программного обеспечения. Простым частным примером является загрузка в в оперативную память исполняемого кода по мере надобности. Но в то же время общее адресное

пространство является причиной возникновения труднообнаружимых ошибок в программах (программа может «затереть» кусочки собственного кода, а когда это проявится, сложно будет добраться до причины), источником уязвимостей операционных систем. Дело в том, что многие компьютерные вирусы работают по следующему принципу: вызывают переполнения буферов памяти, помещают необходимые байты в некоторый участок памяти и затем добиваются передачи управления выполнения на этот участок памяти. Описанное было бы невозможно, если пространства кода и данных были изолированными.

Но единое адресное пространства несёт в себе важную идею: во время выполнения программы можно дописывать (изменять) логику программы. Это используется в скриптовых языках, где во время исполнения можно создавать (удалять) переменные, функции, классы и другие программные единицы. Методология использования этой возможности с пользой получила название *метапрограммирование*.

Формат представления программы

В современных многозадачных операционных системах каждая выполняемая программа представляется в виде отдельного *процесса*. Процесс — это экземпляр виртуального исполнителя, по набору команд идентичного реальному процессору. Каждому процессу операционная система предоставляет адресное пространство (множество непосредственно адресуемых ячеек памяти), но ограниченного размера. Размер адресного пространства обычно равен степени двойки. Эта степень часто называется разрядностью архитектуры или размером машинного слова, хотя это не совсем точно[6]. В компьютерах с 32-битной архитектурой размер машинного слова равен 32 бита, и, соответственно, размер адресного пространства процессов равен $2^{32} = 4$ Gb (гигабайт) $\approx 4 \cdot 10^9$ байт. Поскольку процесс — это экземпляр *виртуального* исполнителя, то и память, предоставляемая процессу операционной системы, тоже *виртуальна*. Об этом будет отдельный разговор ниже.

В этом адресном пространстве находятся как данные, так и программа виртуального исполнителя (машинный код). Программа исполняется непосредственно *процессором*, но операционная система (ОС) предоставляет процессорное время процессам по очереди небольшими *квантами*, эмулируя тем самым, *многозадачность*.

[6]Размер машинного слова — это характеристика (измеряемая в битах) компьютерной архитектуры, отражающая различные её свойства:

- разрядность большей части регистров процессора;
- разрядность шины данных (типичный размер единицы данных, при передаче данных между памятью, процессором и другими компонентами архитектуры);
- максимальное значение беззнакового целого типа, напрямую поддерживаемого процессором;
- максимальный объём оперативной памяти, непосредственно адресуемой процессором (в современных ОС процесс может иметь доступ к сколь угодно большому объему памяти, но она не будет непосредственно адресуемой).

Процессу разрешается писать и читать данные из адресного пространства, но с некоторым ограничением. А именно, адресное пространство процесса поделено на *сегменты*. Различают пять основных типов сегментов:

- сегмент данных — глобальные переменные программы;
- сегмент кода — код программы;
- сегмент констант — данные, которые не могут быть изменены в процессе исполнения программы;
- сегмент стека — область памяти, которая используется процессором для эмуляции стека;
- куча — область памяти, которая выделяется в процессе выполнения программы, по мере необходимости.

Каждый сегмент имеет набор прав доступа. А именно:

- Право на запись — позволяет процессору записывать данные в эту область.
- Право на чтение — позволяет процессору обращаться к области памяти на чтение.
- Право на исполнение — дает право процессору помещать указатель текущей команды в эту область.

Существуют также дополнительные *атрибуты* сегмента: выгружаемый/невыгружаемый, инизиализируемый/неинициализируемый, комментарии, разделяемый/неразделяемый, выравнивание, и другие, но их мы рассматривать не будем.

Вот, какие права имеют различные сегменты памяти:

сегмент данных	чтение, запись
сегмент кода	чтение, исполнение
сегмент констант	чтение
сегмент стека	чтение, запись
куча	чтение, запись

Все сегменты памяти процесса имеют ограниченный размер и располагаются в адресном пространстве определенным образом. Способ описания сегментов и их расположения в памяти называют *форматом исполняемого файла*. Для каждой операционной системы характерен свой набор форматов исполняемых файлов, которые могут быть запущены в виде процессов. Но в целом, они имеют общие принципы организации. Для примера рассмотрим расположение сегментов исполняемого файла в формате ELF, который используется в ОС Linux:

0x00000000	Служебная область
0x08048000	Сегмент кода
0x0XXXXXXX	Сегмент констант
0xYYYYYYYY	Сегмент данных
0xZZZZZZZZ	Куча
	...
	граница кучи
	верхняя граница стека
	...
0xbfffffff	дно стека

Если процесс обращается к области памяти, не принадлежащей ни одному сегменту, либо нарушает права доступа к сегменту, то ОС инициирует ошибку выполнения процесса «Segmentation fault» (ошибка сегментации). С другой стороны, процесс может запросить у ОС увеличить область динамической памяти (отодвинуть вниз границу кучи), либо увеличить размер стека (сдвинуть вверх границу стека). ОС может отказать процессу в перемещении границ этих областей, выдав ошибку «No memory» (нет памяти) или «Stack overflow» (переполнение стека) соответственно.

Виртуальная память программы

Мы упомянули, что адресное пространство процесса виртуально (в современных многозадачных ОС). Что это значит и зачем это нужно?

ОС разделяет как физическую память, так и адресные пространства всех процессов на страницы памяти. Страницы обычно имеют фиксированный размер. Только некоторым страницам виртуального адресного пространства ОС сопоставляет страницы физической памяти. Таким образом, процессу доступно не всё адресное пространство, адресуемое машинным словом. При обращении к ячейке памяти, которой не сопоставлена физическая ячейка памяти, возникает ошибка «Segmentation fault».

Такая виртуализация адресного пространства процессов позволяет решить две задачи:

- **изоляция адресных пространств процессов**:

  Действительно, каждый процесс имеет свои 4Gb памяти и может быть уверенным, что они никак не пересекаются с адресными пространствами других процессов (но есть возможность обобществлять кусочки памяти между несколькими процессами).

- **экономия физической памяти**:

  Реальные компьютеры часто имеют сильно ограниченные ресурсы физической памяти, и не всегда процессу нужно всё адресное пространство целиком. Ради экономии, каждому процессу в начале выполнения выделяется только минимально необходимый объем.

- **предоставление удобного механизма работы с памятью:** Виртуальная память — это прежде всего абстрактный интерфейс взаимодействия с некоторым запоминающим устройством. Программисту не хотелось бы углубляться в детали устройства различных запоминающих устройств. Ему, по большому счёту, всё равно, где и как расположены байты, которыми он оперирует.

ОС для каждого процесса хранит карту, отображающую страницы виртуальной памяти на страницы физической памяти. Операционная система может создавать страницы физической памяти не только в ОЗУ, но и на диске, тем самым рассширив эффективный размер физической памяти. При этом если какой-либо процесс обращается к некоторой странице памяти, которая в данный момент находится на диске, ОС загружает эту страницу в ОЗУ, а наиболее старую страницу, которая давно не использовалась каким-либо процессом, выгружает на диск. Эта операция обмена страницами памяти между быстрым (ОЗУ) и медленным (диск) устройствами называется **свопингом** (англ. *swap* — обмен). А ситуация обращения к назагруженой в ОЗУ странице называется *page fault*. Программа, затребовавшая у ОС много памяти

может начать работать медленно, из за того, что часть адресного пространства была отображена на медленный дисковый накопитель, появилось множество ошибок page faults и начался активный свопинг.

Практические задачи

Рассмотрим следующую простую программу, которая печатает адреса переменных различного уровня видимости и типа хранения:

Изучение структуры адресного пространства.

```
#include <stdio.h>
#include <malloc.h>

static int f = 32;
int l;
const g = 3;
int x() {
        int j;
        register int m;
        printf("local (second) @0x%08x\n", &j);
        return 0;
}
int main() {
        int j;
        int *d;
        printf("local (first)  @0x%08x\n", &j);
        x();
        printf("function body  @0x%08x\n", x);
        printf("const          @0x%08x\n", &g);
        printf("static         @0x%08x\n", &f);
        printf("global         @0x%08x\n", &l);
        d = (int *)malloc(10000000);
        printf("dynamic        @0x%08x\n", d);
        free(d);
        return 0;
}
```

Задача Л26.12. Скомпилируйте и выполните программу 26 на различных компьютерах под различными операционными системами. Сравните результаты и сделайте выводы о взаимном расположении различных сегментов.

Задача Л26.13. Напишите программу, которая с помощью рекурсивных вызовов вызывает ошибку «Stack overflow». Во время выполнения осуществляйте трассировку значения границы стека.

Задача Л26.14. Напишите программу, которая с помощью вызовов !malloc(10001)! вызывает ошибку «No memory». Изучите, как ОС располагает выделяемые кусочки памяти в адресном

пространстве.

Задача Л26.15. Рассмотрим следующую программу:

```
#include <stdio.h>
main(){ int i; printf("%d", &i); while (1); }
```

Эта программа зависает. Если запустить два экземпляра данной программы, одинаковые или разные значения будут выведены?

Литература

[1] Кормен Томас Х., Лейзерсон Чарльз И., Ривест Рональд Л., Штайн Клиффорд. Алгоритмы: построение и анализ, 2-е издание. 2006. – 1296 стр. Пер. с анг. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to algorithms, Second Edition.

[2] Керниган Б.В., Пайк Р. Практика программирования.: Пер. с англ. – СПб.:Невский Диалект, 2001. – 381 стр.

[3] Керниган Б.В., Ритчи Д.М. Язык программирования С (Си), 2-е издание, 2007,

[4] Левитин А.В. Алгоритмы: введение в разработку и анализ.: Пер. с англ. – М.:Издательский дом "Вильямс", 2006. – 576 стр.

[5] Борисенко В.В. Основы программирования, – М.:Интернет-университет информационных технологий, 2005. – 328 стр.

[6] Computing Curricula 2005, The Joint Task Force on Computing Curricula. IEEE Computer Society, Assotiation for Computing Machinery, 2005.
`http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf`

[7] Дейкстра Э. Дисциплина программирования. М.:Мир, 1978, – 278 стр.

[8] Минский М.Л. Вычисления и автоматы. – М.: Мир, 1971.

[9] Успенский В.А., Семёнов А.Л. Теория алгоритмов: основные открытия и приложения. – М.:Наука, 1987.

[10] Вирт Н. Алгоритмы+структура данных=программа. - М.:Мир, 1985.

[11] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. Перевод с английского А.О. Слисенко под редакцией Ю.В. Матиясевича. Мир, 1979. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[12] Верещагин Н.К., Шень А.Х. Лекции по математической логике и теории алгоритмов, Часть 3. Вычислимые функции, М.:МЦНМО, 1999.

[13] Верещагин Н.К., Шень А. Лекции по математической логике и теории алгоритмов, Часть 2. Языки и исчисления, М.:МЦНМО, 2000.

[14] Зыков С.В. Введение в теорию программирования, М.:Интернет-университет информационных технологий, 2004. – 400 стр.

[15] Hunt, Andrew, and David Thomas. The pragmatic programmer: from journeyman to master. Reading, Mass: Addison-Wesley. 2000.

[16] Computational geometry, Springer, 2001.

[17] Викиучебник — коллективно создаваемый ресурс учебных материалов в Интернет: `http://ru.wikibooks.org`.

[18] Википедия — коллективно создаваемая энциклопедия в Интернет: `http://ru.wikibooks.org`.

[19] Непейвода Н.Н. Прикладная логика. Учебное пособие. Ижевск: Изд-во Удмуртского ун-та, 1997.

[20] Непейвода Н.Н. Основания программирования, Москва-Ижевск: Институт компьютерных исследований, 2003, – 868 стр.