

Марина Данииловна Князева – выпускница кафедры "Вычислительная техника" Пензенского государственного университета, кандидат технических наук, доцент ВАК.

Преподает программирование, IT- и AI-технологии в Московском государственном университете технологий и управления им. К. Г. Разумовского.

Область научно-педагогических интересов: языки программирования, IT- и AI-технологии, геоинформатика, автоматизированная обработка аэрокосмической информации, инновационные процессы в образовании.



Надежда Сергеевна Карамышева – выпускница кафедр "Математическое обеспечение и применение ЭВМ" (специалитет) и "Вычислительная техника" (магистратура, аспирантура) Пензенского государственного университета, кандидат технических наук, доцент ВАК. Преподает программирование, IT- и AI-технологии в ПГУ.

Область научно-педагогических интересов: языки программирования, IT- и AI-технологии, front-end и back-end программирование, обработка информации в интеллектуальных системах, инновационные процессы в образовании.



Дарья Дмитриевна Григорьева – выпускница кафедры "Вычислительная техника" Пензенского государственного университета, имеет опыт профессиональной работы в области программирования.

Область научно-педагогических интересов: IT- и AI-технологии, в том числе автоматизированное тестирование, front-end программирование, интеллектуальные системы, машинное обучение глубоких нейронных сетей.

9 785907 752634

ПензГУ

М. Д. Князева,
Н. С. Карамышева,
Д. Д. Григорьева

ОТ АЛГОРИТМА К ПРОГРАММЕ И ИСКУССТВЕННОМУ ИНТЕЛЛЕКТУ

С примерами на языках Python и Prolog

М. Д. Князева, Н. С. Карамышева, Д. Д. Григорьева

ОТ АЛГОРИТМА К ПРОГРАММЕ И ИСКУССТВЕННОМУ ИНТЕЛЛЕКТУ



С примерами на языках Python и Prolog

- Основные принципы алгоритмизации
- Реализация алгоритмов
- Алгоритмы поиска и сортировки данных
- Анализ экологических проблем города
- Сетевой суперкомпьютер – это просто!
- Программирование интеллектуальных систем с событиями
- Концептуальные модели, основанные на знаниях и правилах
- Интеллектуализация роботизированного производства

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Пензенский государственный университет» (ПГУ)

М. Д. Князева, Н. С. Карамышева, Д. Д. Григорьева

ОТ АЛГОРИТМА К ПРОГРАММЕ И ИСКУССТВЕННОМУ ИНТЕЛЛЕКТУ

С примерами на языках Python и Prolog

Учебное пособие

Под редакцией
доктора технических наук, профессора С. А. Зинкина

Пенза
Издательство ПГУ
2024

УДК 004.8
К54

Р е ц е н з е н т ы :

доктор технических наук, профессор, заведующий кафедрой «Электронные вычислительные машины» Рязанского государственного радиотехнического университета имени В. Ф. Уткина *Б. В. Костров*;

доктор технических наук, профессор, научный консультант Пензенского научно-исследовательского электротехнического института *А. И. Иванов*

Князева, Марина Данииловна.

К54 От алгоритма к программе и искусственому интеллекту. С примерами на языках Python и Prolog : учеб. пособие / М. Д. Князева, Н. С. Карамышева, Д. Д. Григорьева ; под ред. д-ра техн. наук, проф. С. А. Зинкина. – Пенза : Изд-во ПГУ, 2024. – 412 с.

ISBN 978-5-907752-63-4

Изложены основы разработки схем алгоритмов и особенности формирования алгоритмического подхода к решению задач. Приводится описание основных алгоритмических конструкций, позволяющих составлять алгоритмы решения прикладных задач. Содержится большое число примеров алгоритмов типовых задач, таких как поиск и сортировка данных, приближенное решение уравнений. Часть решенных задач имеет оригинальный характер: приведены алгоритмы и программы, связанные с экологией окружающей среды, сетевыми грид-вычислениями и интеллектуализацией некоторых областей человеческой деятельности на основе логического подхода к искусственному интеллекту. Предложены варианты реализации алгоритмов на языке программирования Python. Логические основы программирования заданы концептуальными графами и языком Prolog.

Издание подготовлено на кафедре «Вычислительная техника» ПГУ и предназначено обучающимся по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.03 «Прикладная информатика» для формирования начальных профессиональных компетенций. Кроме того, книга рекомендуется широкому кругу читателей для самостоятельного изучения основ построения алгоритмов и составления программ для различных предметных областей.

УДК 004.8

ISBN 978-5-907752-63-4

© Пензенский государственный университет, 2024



Предисловие

Программированию обычно
учат на примерах.
Н. Вирт

Алгоритмы, представленные в этой книге, предназначены для решения относительно простых задач. Многие из этих задач достаточно легко сформулировать и смоделировать. Как правило, они содержат совсем немного параметров, и их можно решить аналитически. Алгоритмы решения этих задач отличаются значительной компактностью.

Несомненно, задачи, рассмотренные в этой книге, весьма отличаются от задач, описывающих большие системы, например, для принятия административных решений, управления производством, решения финансовых задач и т.д. Но каждая большая программная система состоит обыкновенно из многих небольших программ, реализующих известные или достаточно простые алгоритмы, представленные блок-схемами.

Блок-схема – это схематичное представление процесса, системы или компьютерного алгоритма. Блок-схемы часто применяются в разных сферах деятельности, чтобы документировать, изучать, планировать, совершенствовать и объяснять сложные процессы с помощью простых диаграмм. Блок-схемы варьируются от простых, нарисованных вручную, до подробных, составленных на компьютере диаграмм со множеством шагов и процессов. Если учесть все возможные вариации, блок-схемы можно признать одними из самых распространенных видов схем во всем мире.

В качестве примеров в настоящем пособии приведено построение блок-схем очень простых алгоритмов, при этом акцент сделан на

различные реализации ветвлений и циклов, поскольку считается, что начинающие программисты наибольшее число ошибок допускают именно в этой части. В то же время многие программисты утверждают, что нужно рисовать лишь те схемы, которые помогают в работе и не требуют больших усилий для поддержания в актуальном состоянии. На начальных этапах работы программиста составление блок-схем программ помогает избежать появления ошибок. Использование псевдокодов и программ на языке программирования высокого уровня Python для записи одних и тех же алгоритмов помогает освоить программирование за короткое время. Произведено множество приложений для выполнения этих самых схем – Dia, MS Visio, yEd и др. Вместо блок-схем иногда применяют диаграммы деятельности UML, однако они удобнее в основном при изображении параллельных алгоритмов.

Все темы в книге рассматриваются по мере усложнения, но это не означает, что читатель должен изучать все в предложенном порядке. Основополагающими следует назвать первую и вторую главы. При решении задач использован в основном язык Python.

Часть I «Алгоритмическое мышление и программирование» содержит восемь глав с примерами на языке Python.

В *первой* главе излагаются основные правила, определения и конструкции для проектирования алгоритмов. Во *второй* главе предложены типичные примеры задач с реализацией линейных, разветвленных и циклических алгоритмов.

Главы *третья* и *четвертая* содержат известные алгоритмы поиска и сортировки данных. Подробно описаны схемы алгоритмов и дано их иллюстративное сопровождение.

В *пятой* и *шестой* главах рассматриваются алгоритмы численного решения алгебраических и дифференциальных уравнений, а также численного интегрирования, построенные на основе известных методов.

Седьмая глава содержит материал по актуальной на сегодняшний день теме экологии города, в том числе предложены алгоритмы решения соответствующих задач.

Восьмая глава посвящена созданию сетевого грид-приложения под названием «Сетевой суперкомпьютер». Читатель, желающий ознакомиться с методами разработки клиент-серверных приложений, а также родственных им приложений в архитектуре «Мастер-исполнители», найдет здесь материал по сетевому программированию на языке Python в TCP/IP сетях, как в локальных, так и в глобальных, в том числе и в сети Интернет.

Часть II «Сетевые и логические модели искусственного интеллекта, основанные на знаниях и правилах» включает три главы.

Девятая глава посвящена программированию задач на языках Python и Prolog на основе логического подхода к искусственному интеллекту. Изложение основано на примерах построения интегрированных декларативно-процедурных и концептуальных моделей для предметных областей человеческой деятельности.

В *десятой* главе приведено решение задачи построения семантической сети и концептуального графа для роботизированного производства, работающего под управлением беспроводной сети.

Заключительная, *одиннадцатая* глава посвящена методике построения и программирования сетевых концептуальных и логических моделей искусственного интеллекта, основанных на знаниях и правилах.

Приложение 1 содержит краткую справочную информацию, необходимую для понимания примеров на языке Python.

Приложение 2 содержит краткую справочную информацию о логических свойствах и использовании языка Prolog.

Библиографические списки разбиты на части по группам глав и содержат источники, рекомендуемые к дальнейшему изучению принципов алгоритмизации и программирования различных задач. Кроме того, в библиографический список к девятой, десятой и одиннадцатой главам включена использованная авторами литература как учебного, так и научного характера, цитируемая по тексту этих глав.

Дополнительный список литературы содержит современные литературные источники по машинному обучению, искусственным нейронным сетям и программным средствам их обучения.

Большинство алгоритмов в этой книге сопровождается соответствующими фрагментами программ на алгоритмическом языке высокого уровня Python. Для решения некоторых задач использован язык программирования Prolog, хорошо интегрирующийся с языком Python. Преимуществом языка Prolog является простота выражения сущностей и отношений из самых различных областей человеческой деятельности, что позволяет решать задачи, выходящие за границы традиционных вычислений. На языке Prolog написано огромное число программ, связанных с логическим подходом к искусственному интеллекту. Изложенные в настоящем пособии методы позволяют использовать как логическое представление знаний, так и сетевое – в форме семантичесих сетей и концептуальных графов. Рассмотрены также новые модели искусственного интеллекта, основанные на интеграции концептуальных графов с сетями Петри.

Приведены также контрольные примеры или даны рекомендации по типу и виду использованных исходных данных. Читатель, изучивший приведенные в пособии примеры программ, получит навыки программирования и сможет самостоятельно разрабатывать программы на тему любой из рассмотренных задач. Вставки и программы, написанные на языке Python, отмечены следующим значком:



Вставки и программы, написанные на языке Prolog, отмечены значком:



При изучении материалов пособия от читателя не требуется каких бы то ни было знаний по языкам программирования Python и Prolog, хотя наличие некоторого опыта в областях *front-end* и *back-end* современного программирования, в том числе Web-программирования на других языках, приветствуется. Книга предназначена для самостоятельного изучения основ построения алгоритмов и составления программ. Поэтому, чтобы научиться составлять алгоритмы, необходимо, прежде всего, желание делать это самостоятельно. Для начала внимательно прочитайте условие предложенной задачи и попытайтесь его понять. Затем необходимо ответить на

вопрос, как ее решить математически, а затем посмотреть, какое решение предлагается в книге.

Примечание. Основные примеры в настоящем учебном пособии реализованы на языке Python. При иллюстрации логического подхода к искусственному интеллекту некоторые примеры реализованы на языке Prolog. Python и Prolog – два разных языка программирования, каждый со своими особенностями и областями применения. Python – это мультипарадигмальный язык программирования, который может использоваться для различных целей, включая веб-разработку, научные вычисления, применение искусственных нейронных сетей, анализ данных и многое другое. Prolog – это логический язык программирования, разработанный для применения в области искусственного интеллекта и компьютерной лингвистики. Ниже даны некоторые сведения по интеграции языков Python и Prolog друг с другом и с другими системами программирования.

Python – это удобный язык для написания проектов, которые могут включать несколько языков программирования. Python может взаимодействовать с языками C/C++, Java, JavaScript, Go, Prolog и др. Разработан специальный вариант Python – Jython, предназначенный для работы на виртуальной машине Java (JVM), т.е. интерпретатор Python и программа на Python работают в той же среде, что и программы Java. Более подробные сведения об интеграции языка Python с другими языками можно найти по следующей ссылке:

http://www.python.org/moin/IntegratingPythonWithOtherLanguages

SWI-Prolog – это свободно распространяемая (открытая) реализация языка программирования Prolog, часто используемая для преподавания и работы с приложениями Semantic Web. Созданный пакет Janus реализует двунаправленный интерфейс между языками Python и SWI-Prolog с использованием переносимых низкоуровневых примитивов. Цель создания данного интерфейса состоит в том, чтобы сделать Python доступным для SWI-Prolog и, наоборот, SWI-Prolog доступным для Python, с минимальными усилиями по установке, обеспечивая при этом двунаправленный интерфейс высокого уровня с хорошей производительностью. Имеется пакет PyLog для работы с SWI-Prolog из Python. Реализованы интерфейсы SWI-Prolog

с языками Python, C/C++, Java, C#, Go, SQL и др. Обеспечивается поддержка Web-технологий, в том числе технологии Pengines распределенного логического программирования в Интернете. Более подробные сведения об интеграции языка SWI-Prolog с другими языками можно найти по следующей ссылке: http://www.swi-prolog.org.

Все предложенные в книге программы являются реально работающими. Работа между соавторами по подготовке книги распределялась следующим образом: главы 1–7 написаны М. Д. Князевой; программы на языке Prolog, главы 8–11 и приложение 2 написаны Н. С. Карамышевой; программы на языке Python и описания к ним, а также приложение 1 составлены Д. Д. Григорьевой.

ЧАСТЬ I
АЛГОРИТМИЧЕСКОЕ МЫШЛЕНИЕ И ПРОГРАММИРОВАНИЕ



Глава 1

Основные принципы алгоритмизации

Язык (программирования) должен отражать
фундаментальные и наиболее важные концепции
алгоритмов в очевидной и легко
воспринимаемой форме…
Н. Вирт

Способность выполнять и составлять алгоритмы занимает центральное место в компьютерной грамотности и играет важную роль при обработке информации и решении[1] задач. Алгоритм представляет собой необходимый инструмент при решении задач.

Найти решение задачи – это означает установить связь между заранее дифференцированными объектами или идеями (объектами, которые имеются, и объектами, которые необходимо найти, данными и неизвестными). Задача предполагает необходимость сознательного поиска соответствующего средства для достижения ясно видимой, но непосредственно недоступной цели. Процесс решения задачи – результат наших умственных усилий, т.е. последовательность в достаточной мере согласованных логических операций или шагов, начинающихся с условия (предпосылки) и заканчивающихся значением, к которому мы стремились. Каждый шаг приводит к некоторому новому положению.

Задача может быть теоретической или практической. Чтобы решить ее, необходимо составить хорошо продуманную, согласованную схему операций (логических, математических или материально

[1] Решение-объект (Solving object, Losungsgegenstand) – найти решение, значит найти объект; решение-процесс (Solving procedure, Losungsgang) – схема операций нахождения неизвестных.

обеспечивающих), ведущую от данных к неизвестному, от объектов, находящихся в нашем распоряжении, к объектам, которые необходимо достичь. Нематематические задачи тоже можно представить в аналогичном аспекте.

Различают конкретное и формальное мышление. Большинство людей так и не овладевают формальным мышлением. Алгоритмизация мыслительного процесса позволяет абстрагировать или индивидуализировать формальное мышление. Знание, которое было достигнуто только через понимание формальных операций, свойственных алгоритмам и программам, может теперь конкретизироваться и объединять все элементы, необходимые для владения средствами формального мышления.

Можно извлечь пользу из возможности сознательно мыслить так же, как «думает» компьютер, по аналогии с компьютерной программой или алгоритмом, где все происходит последовательно шаг за шагом, буквально механически. Имеется множество ситуаций, когда такой стиль мышления может оказаться полезным. Наличие компьютера значительно обогащает мыслительную деятельность не только технически, но и концептуально, влияя на мышление людей. Совет: думать, как компьютер открывает новый способ подхода к мышлению.

Развитие навыков алгоритмизации необходимо вне зависимости от того, используется ли компьютер или калькулятор, карандаш и бумага или другое «вспомогательное» средство для решения задачи. Алгоритмизация – это общая совокупность последовательных действий, которые необходимо выполнить для построения алгоритма решения задачи. Сущность алгоритмизации можно выразить следующими положениями:

- выделение конкретных действий (шагов) алгоритмического процесса;
- определение вида формальной записи для каждого действия (шага);
- установление определенного порядка выполнения каждого действия (шага);
- реализация выбранного метода вычисления;
- анализ результата исполнения полученной последовательности на соответствие к исходной задаче.

Результаты алгоритмизации формализуются в виде определенной схемы, которая задает алгоритм решения поставленной задачи. Разработка алгоритма требует определенной квалификации исполнителя, понимания содержания задачи и, конечно, больших временных затрат. Процесс подготовки решения задачи и ее непосредственная реализация на компьютере происходят за некоторое количество самостоятельных этапов (рис. 1.1).



Рис. 1.1. Этапы решения задачи на компьютере

Решение задачи в соответствии с готовым алгоритмом требует от исполнителя строгого следования заданным предписаниям. Исполнитель алгоритма не вникает в смысл того, что он выполняет, а действует формально. Поэтому при составлении алгоритма необходимо осознать в целом весь процесс решения задачи. Культура применения компьютера предполагает наличие алгоритмической культуры мышления и необходимых навыков использования компьютера.

Алгоритм и его свойства

Понятие алгоритма можно отнести к числу основных понятий как современной математики, так и информатики. Создание алгорит-

мов решения различных классов задач является одной из целей математики. Эти классы охватывают область от искусственного интеллекта до компьютерных игр.

Термин «алгоритм» происходит от имени узбекского математика Мухаммеда ибн Муса аль-Хорезми, который в IX в. предложил правила выполнения арифметических действий над целыми числами и простыми дробями в десятичной системе счисления. В средневековой Европе «алгоризмом» называлась десятичная позиционная система счисления и искусство счета в ней. В латинском переводе книги аль-Хорезми правила начинались словами «Алгоризми сказал», которые со временем трансформировались в «Алгоритм гласит» и стали обозначать любые точные правила действий.

Алгоритм – *algorithm* – представляет собой строгую систему правил, определенную последовательность действий над некоторыми объектами. Следуя такой системе правил, как инструкции, различные исполнители алгоритма будут действовать одинаково и получать, соответственно, одинаковые результаты.

> *Алгоритмом решения задачи называется путь решения задачи, определенная последовательность действий, которую необходимо выполнить для достижения результата*.

Под алгоритмом понимается *точное предписание* правил, которое задает вычислительный процесс для совокупности исходных данных и направляет на получение результата.

Основной целью вычислительного процесса является исполнение алгоритма с заданными исходными данными и получение результата. Процесс заключается в исполнении отдельных шагов в соответствии с определенным алгоритмом. Таким образом, алгоритмический процесс должен состоять из отдельных шагов, каждый из которых обеспечивает перемещение данных из одного состояния в другое. Построение систематизированной последовательности операций, которые необходимо выполнить для решения задачи, осуществимо с учетом следующих основных свойств алгоритмов:

1. *Дискретность.* Алгоритм представлен в виде конечной последовательности шагов: решение задачи алгоритм сводит к решению отдельных более простых задач; каждое действие, предусмотренное алгоритмом, исполняется только после того, как закончилось исполнение предыдущего.

2. *Определенность или детерминированность*. Каждый шаг алгоритма для решения должен быть четко и недвусмысленно определен, не должен допускать произвольной интерпретации. Это обеспечивает однозначность результата при заданных исходных данных.

3. *Результативность*. Алгоритм имеет определенное число входных величин – аргументов. Цель выполнения алгоритма – получение конкретного результата, имеющего вполне определенное отношение к исходным данным.

4. *Конечность*. Если действовать в соответствии с алгоритмом, то за конечное число шагов обязательно получается решение задачи. Строится бесконечный, сходящийся к искомому решению процесс. Он обрывается на некотором шаге, и полученное значение принимается за приближенное решение рассматриваемой задачи. Точность приближения зависит от числа шагов. Успешное завершение алгоритма необходимо даже при некорректности исходных данных (в этом случае следует предусмотреть сообщение об ошибке).

5. *Массовость*. Можно применять один и тот же алгоритм для решения множества однотипных задач, различающихся только исходными данными.

6. *Эффективность*. Алгоритм может быть выполнен не просто за конечное, а за разумно конечное время.

7. *Компактность*. Это свойство предполагает лаконичность изложения алгоритма. Как только компактность потеряна, алгоритм в значительной мере теряет право на существование.

Может случиться, что алгоритм уже существует для конкретной задачи, но его трудно и иногда невозможно описать в некоторой заданной форме. Человечество разработало немало эффективных алгоритмов, таких, например, как завязывание галстука, заплетание косы, завязывание шнурков на туфлях, порядок одевания и других, словесное описание которых достаточно сложно изложить без соответствующих иллюстраций. Очевидно, для точного определения алгоритма необходимо указывать (или учитывать) возможности исполнителя и для каких целей предназначен алгоритм.

Постановка задачи

Прежде чем понять задачу, следует уточнить ее основные характеристики, сформулировать цель решения задачи, подробно

описать ее содержание, провести анализ характера и сущности всех известных и неизвестных данных, определить условия, при которых задача может быть решена. Может оказаться полезным подразделить сложное условие на несколько частей или пунктов. Следует точно установить в самом начале решения задачи количество неизвестных. Во многих случаях бывает несложно устранить различие между задачей со многими неизвестными и задачей с одним неизвестным.

Обычно процесс точной формулировки задачи сводится к постановке правильных вопросов. Перечислим некоторые полезные вопросы:

- Понятна ли терминология используемой предметной области?
- Что дано?
- Что требуется найти?
- Что представляют собой неизвестные и сколько их?
- Какие нужны данные, чтобы найти неизвестные?
- Как определить решение?
- Какие данные необходимо вводить?
- Каких данных не хватает и все ли они необходимы?
- Являются ли какие-то имеющиеся данные бесполезными?
- Какие следует сделать допущения?
- Какие требования к точности решения?
- Какие ограничения на время решения задачи?
- И т.д.

Иногда задача поставлена так, что допускает много толкований и неточностей. В этом случае целесообразно уделить внимание тому, чтобы сделать условие более четким. Способы задания условия могут существенно повлиять на построение алгоритма. Для некоторых задач основная трудность заключаются в том, чтобы сделать условие простым и точным. *Условие необходимо для того, чтобы изложить, что должен выполнить алгоритм*, а абстракция этой записи лишь для упрощения.

Работа по уточнению различных закономерностей, существующих в предметной области задачи, относится к системному анализу, но этим приходится заниматься именно при разработке алгоритма. Задача, предназначенная для решения с помощью компьютера, сначала должна быть задана в математической постановке – постановка задачи должна быть формализована, т.е. выражена в терминах понятий, обладающих точно определенными свойствами и находя-

щихся между собой в строго определенных отношениях. Предварительно составляется строгая последовательность элементарных операций, выполняемых на каждом отдельном шаге. Такая последовательность должна реализовать выбранный метод решения.

Если сказано, что задача поставлена, то предполагается, что известны не только характеристики существующей ситуации, но и требования к решению задачи. Формализация постановки задачи позволяет наметить путь, который ведет от постановки решения задачи к ее решению, т.е. провести логический анализ формальной постановки задачи в результате выявления свойств и отношений между понятиями. Конкретность постановки задачи является важным моментом, так как от этого зависит успех решения задачи.

Построение модели

Всякое человеческое познание начинается
с созерцаний, переходит от них
к понятиям и заканчивает идеями…
И. Кант

Построение модели – это формализация описания задачи с использованием математических, статистических, эвристических (логических) и других методов, при которых существующие соотношения между величинами, определяющими результат, можно выразить с использованием математических формул и логических соотношений.

Если задача четко поставлена, то для нее несложно разработать математическую модель. Выбор модели существенно влияет на остальные этапы в процессе решения. Математическая формулировка и выбранный метод вычисления являются основой для определения последовательности действий, приводящих к получению искомого результата. Таким образом, создается модель явления с определенными допущениями, предположениями, точностью вычислений и т.д. В зависимости от содержания задачи построение ее модели может быть областью исследований таких дисциплин, как исследование операций, методы оптимизации, математическая статистика, численный анализ, теория информации и другие.

Большинство задач требует индивидуального рассмотрения. Тем не менее существуют полезные руководящие принципы [8]:

1. Выбор модели – дело искусства.

2. Изучение удачных моделей – наилучший способ приобрести опыт в моделировании.

   Приступая к разработке модели, следует задать, по крайней мере, два вопроса:

   - Какие известные математические или другие методы формализации (типовые структуры) более всего подходят для решения поставленной задачи, если в постановке задачи не указаны те, которые следует использовать?
   - Существует ли опыт решения аналогичных задач?

   Сначала следует рассмотреть необходимость математического описания того, что мы знаем и что необходимо найти. На выбор соответствующей структуры будут оказывать влияние такие факторы, как удобство представления, простота вычислений, ограниченность наших знаний, а также:

   - Вся ли важная информация для решения задачи хорошо описана математически?
   - Существует ли математическая величина, соотносимая с результатом?
   - Выявлены ли полезные соотношения между характеристиками модели?
   - Удобно ли работать с моделью?

Метод решения задачи следует из принятой модели. С учетом особенностей с помощью конкретных методов решение задачи должно быть доведено до реализации. В основе алгоритма может быть только конкретный метод вычисления, который должен привести к упорядоченным арифметическим и логическим действиям.

Если одна и та же задача может быть решена с помощью различных методов, то выбирается тот из них, который наилучшим образом удовлетворяет ее требованиям. Определяющими факторами в этом случае являются также время решения задачи, объем внешней и оперативной памяти.

Разработка алгоритма

После того, как построена модель решения задачи, можно приступить к разработке алгоритма ее решения. Выбор метода решения в большой мере зависит от выбора модели. На стадии разработки алгоритма требуется тщательное обдумывание.

Наиболее распространенными методами разработки алгоритмов являются метод частных целей, метод подъема и эвристический алгоритм. Для разработки алгоритма *методом частных целей* необходимо определить варианты возможностей решения задачи:

- Можно ли решить хотя бы часть задачи, игнорируя некоторые условия?
- Можно ли решить задачу для частных случаев?
- Есть ли что-то, что нам недостаточно понятно?
- Встречались ли мы с похожей задачей? Можно ли видоизменять ее для решения данной задачи?

Частные цели для разработки алгоритма могут быть установлены, если будут найдены ответы на поставленные вопросы.

Для второго метода, известного как *метод подъема*, необходимо в начале сделать предположение, что имеется некоторое начальное состояние решения задачи. Затем следует определить, насколько возможно движение дальше «наверх» – от начального к наилучшему решению. Когда алгоритм достигает этапа, с которого невозможно продвижение дальше, то это значит, что получено решение, и алгоритм заканчивается. К сожалению, нет гарантии, что последнее решение есть наилучшее решение.

При разработке *эвристического алгоритма* необходимо помнить, что такой алгоритм обычно помогает найти хорошие, но не обязательно оптимальные решения. Нередко эвристический алгоритм имеет быструю и простую реализацию по сравнению с известным точным алгоритмом, который гарантирует оптимальное решение.

Не существует универсальной структуры, способной описать эвристические алгоритмы, многие из этих алгоритмов основаны на методе частных целей или на методе подъема. Общий подход к построению эвристических алгоритмов заключается в перечислении всех требований к точному решению с указанием, для каких из них возможен компромисс, а какие непременно должны быть выполнены.

Современные методы программирования, основанные на структурном подходе, предусматривают использование специальных приемов, например, применение пошаговой детализации. На первом этапе разработки алгоритм рассматривается в общем виде как некоторая совокупность действий для преобразования исходной информации, все последующие этапы – это выявление все более частных особенностей.

При записи действий на любом этапе можно учитывать, что исполнителем алгоритма будет компьютер, который умеет совершать только вполне определенные действия, а именно: присваивание, ветвление, циклическое повторение. Поэтому любая детализация должна приводить к реализации таких конструкций. В процессе формализации необходимо обратить внимание на исполнение алгоритма. Пошаговое выполнение последовательности операторов позволяет получить ожидаемый результат, если четко определены начальные условия.

Пример

Найти объем ***V*** *правильной четырехугольной усеченной пирамиды*[2]*, если даны ее высота* ***h****, сторона* ***a*** *верхнего основания и сторона* ***b*** *нижнего основания (рис. 1.2). Составить алгоритм решения.*

Итак, что требуется найти? Что дано?

Даны a, b, h.

Неизвестен *V* – объем усеченной пирамиды.

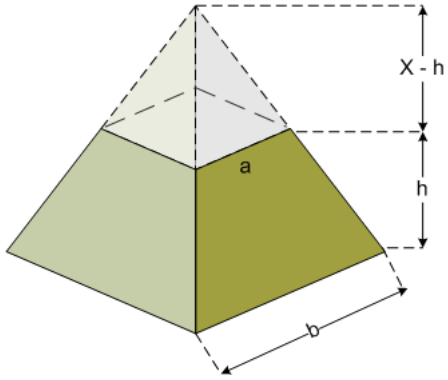


Рис. 1.2. Пояснение к алгоритму о пирамиде

Если нельзя решить предложенную задачу, то можно попытаться решить более легкую. Таким образом, мы можем найти объем полной пирамиды и объем той части полной пирамиды, которую можно «отсечь», чтобы получить усеченную пирамиду. В результате получился следующий алгоритм:

[2] *Пирамида с квадратным основанием будет правильной*, если основание ее высоты совпадает с центром квадрата. Усеченной пирамидой называется часть полной пирамиды, заключенная между ее основанием (ее нижнее основание) и плоскостью (ее верхнее основание), параллельной основанию. Высотой усеченной пирамиды называется расстояние между ее основаниями.

1. Вычислить объем полной пирамиды V_0.
2. Вычислить объем «отсеченной» (полной) пирамиды V_m.
3. Вычислить объем усеченной пирамиды $V = V_0 - V_m$.

Здесь $$V_0 = \frac{a^2 \times X}{3} \quad \text{и} \quad Vm = \frac{b^2 \times (X + h)}{3},$$

где X – высота полной пирамиды.

X можно определить из соотношения сторон $\frac{X}{X + h} = \frac{a}{b}$.

Из этого равенства следует $X = \frac{a \times h}{b - a}$.

На втором этапе можно составить окончательный алгоритм:

1. Ввод a, b, h.
2. Вычисление X = ah/(b – a).
3. Вычисление $V = a^2X/3$.
4. Конец.

Подобная детализация типична для большинства задач. Строгость конструирования алгоритма позволяет получить алгоритм с четко выраженной структурой. Процесс разработки алгоритма должен быть направлен на получение в итоге определенной последовательности алгоритмических конструкций.

Величины в алгоритмах

Данные в алгоритме – это факты и идеи, которые должны быть представлены в формализованном виде, что позволит передавать или обрабатывать их с помощью некоторого процесса или действия. В алгоритмах для данных используются специальные символические обозначения. Эти обозначения аналогичны обозначениям в обычной математике и представляют собой имена величин или их идентификаторы, которые при исполнении алгоритма заменяются конкретными для данной задачи числовыми значениями. В зависимости от характера решаемой задачи используются данные различных типов.

В качестве данных могут выступать целые, вещественные, логические, текстовые и другие значения.

> Величины, значения которых не изменяются в процессе выполнения алгоритма, называются *константами*.
> При написании алгоритмов используются понятия простых и индексных переменных (переменная с индексом). *Простая переменная* – это переменная, значение которой изменяется в процессе выполнения алгоритма.
> *Переменная с индексом* – это элемент некоторой заданной последовательности значений, называемой массивом.

В формулах, используемых при написании алгоритмов, знак «=» является *символом присваивания*, который означает, что вычисленное значение выражения, стоящего справа от этого символа, присваивается переменной, стоящей слева от символа присваивания.

Например, выражение **a = a + 0.005** означает, что значение **a** увеличивается на величину **0.005**, и полученный результат присваивается переменной **a**. При этом старое значение теряется.

Способы записи алгоритмов

Любой способ записи алгоритма подразумевает, что всякий описываемый с его помощью объект задается как конкретный представитель нередко бесконечного класса объектов, которые можно описывать данным способом. Средства, используемые для записи алгоритмов, в значительной мере определяются тем, кто будет исполнителем этих действий.

Если исполнителем будет человек, то запись может быть не полностью формализована, – на первое место выдвигаются понятность и наглядность. В данном случае для записи могут быть использованы словесная запись, псевдокод и схемы алгоритмов. Для записи алгоритмов, предназначенных для исполнителей – автоматов, в качестве которых часто выступает компьютер, необходима формализация. Поэтому в таких случаях применяют специальные формальные языки.

«Преимущество формального способа записи состоит в том, что он дает нам возможность изучать алгоритмы как математические объекты; при этом формальное описание алгоритма служит основой, позволяющей интеллектуально охватить этот алгоритм» [10].

Наконец, такой способ записи позволяет описать алгоритм настолько точно, что если будет задан такой алгоритм и заданы значения аргументов (вход), то не будет никаких сомнений относительно того, каким должен быть результат (выход).

Словесная запись алгоритма

Такой способ записи алгоритма содержит тщательно отобранный набор фраз, который не допускает лишних слов повторений, неоднозначностей. Имеются определенные соглашения о форме записи. Команды алгоритма следует пронумеровать, чтобы иметь возможность на них ссылаться. Допускается использование математической символики. В качестве примера можно привести словесную запись классического алгоритма Евклида для нахождения наименьшего общего делителя (НОД) двух натуральных чисел:

1. Обозревая два числа A и B, переходи к следующему пункту.
2. Сравни обозреваемые числа ((A равно B), A меньше, больше B) и переходи к следующему пункту.
3. Если A и B равны, то прекрати вычисление и переходи к п.7: каждое из чисел дает искомый результат.
4. Если числа не равны, то переходи к следующему пункту.
5. Если первое число меньше второго, то переставь их местами и переходи к следующему пункту.
6. Вычитай второе число из первого, обозревай два числа: вычитаемое и остаток; переходи к п. 2.
7. Конец алгоритма.

Команды такого алгоритма выполняются в естественной последовательности. Вид команд не формализован. Такой вольный тип записи не всегда удобен, так как в нем отсутствует, прежде всего, наглядность.

Псевдокод

Псевдокод – это язык записи структурированных алгоритмов, состоит из смеси языка высокого уровня и фраз языка – родного языка исполнителя (например, русский, английский и др.). Как правило, стандартов на псевдокод не существует.

Псевдокод существует как средство облегчения разработки программ. Тексты псевдокода обычно компактны и не требуют слишком много времени. По сравнению со словесным алгоритмом псевдокод ближе к специфическим программным конструкциям, так как программа псевдокод обладает всеми достоинствами структурированной записи. Поэтому алгоритм, написанный на псевдокоде, можно легко преобразовать в программный код и достаточно просто модифицировать.

Пример псевдокода для нахождения наименьшего общего делителя двух натуральных чисел, словесный алгоритм которого был приведен выше:

Начало
Ввод A, B
Если A ≠ B *то*
Если A > B *то* A = A – B
Иначе B = B – A
Конец Если
Конец Если
Вывод A, B
Конец

Для циклических структур при составлении псевдокода характерными являются конструкции вида:

для цикла с предусловием
Пока <условие> *цикл*
<тело цикла>
Конец цикла
и для цикла с параметром
Начало цикла <счетчик> =<начцикла> *до* <концикла>
<тело цикла>
Конец цикла

Основным достоинством псевдокода является то, что он позволяет пользователю легко разобраться в самом длинном и сложном алгоритме, поэтому чаще всего псевдокод используется для документирования программ. Большинство зарубежных фирм, занимающихся разработкой программ, применяют псевдокод в качестве средства документирования программ.

Схемы алгоритмов

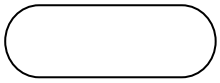
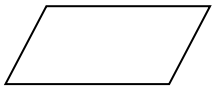
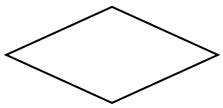
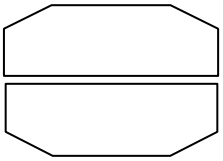
Схема алгоритма – графическое представление метода решения задачи, в котором используются символы для отображения операций и данных. Конфигурацию, перечень и размеры условных изображений, а также правила построения схем алгоритмов устанавливает ГОСТ 19.701–90 «Схемы алгоритмов, программ, данных и систем».

Для каждой группы операций определен соответствующий символ. Такое представление алгоритма более наглядно. Алгоритм выражен последовательностью графических символов, выполняющих определенные функции, и наличием связей между ними.

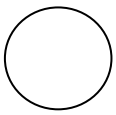
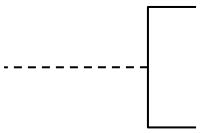
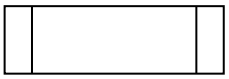
Основные символы, используемые для описания алгоритмов, приведены в табл. 1.1.

Таблица 1.1.

Таблица описания символов в схемах алгоритмов

Наименование	Символ	Назначение символа
1	2	3
Терминатор		Выход во внешнюю среду и вход из внешней среды. Начало и прерывание процесса обработки информации. Начало или конец алгоритма
Данные		Ввод – вывод данных. Носитель не определен
Процесс		Обработка данных любого вида. Выполнение определенной операции или группы операций, приводящее к изменению значения, формы или размещения информации
Решение		Функция переключательного типа. Имеет один вход и ряд альтернативных выходов, только один из которых может быть активизирован после вычисления условий, определенных внутри этого символа
Подготовка		Модификация команды или группы команд с целью воздействия на некоторую последующую функцию (инициализация программы, модификация индекса и др.). Для циклов с параметром
Границы цикла: **Начало цикла** **Конец цикла**		Символ состоит из двух частей. Обе части символа имеют один и тот же идентификатор. Циклы с предварительным условием и с последующим условием

Окончание табл. 1.2

1	2	3
Соединитель		Символ отображает выход в часть схемы и вход из другой части этой схемы и используется для обрыва линии и продолжения ее в другом месте. Соответствующие символы должны содержать одно и то же уникальное обозначение
Комментарий		Используется для добавления описательных комментариев или пояснительных записей в целях объяснения или примечаний. Пунктирные линии связаны с соответствующим символом или могут обводить группу символов. Текст помещается около ограничивающей фигуры
Документ		Отображает итоговые данные в удобочитаемой форме. Вывод информации/ результата на принтер. Список данных для вывода
Предопределенный процесс		Процесс состоит из одной или нескольких операций или шагов программы, которые определены в другом месте – в подпрограмме или модуле. Вызов процедуры или функции

Каждый символ предназначен для графической идентификации функции или действия, которые он отображает независимо от текста внутри этого символа. Символы в схеме располагаются равномерно. Последовательность действий в схеме устанавливается путем упорядоченного размещения символов по схеме и объединения их линиями (или, если необходимо, стрелками), которые называются потоками информации. Основными направлениями для линии потока информации являются направления сверху вниз и слева направо. Если направление перехода от одного символа к другому совпадает с основным, то стрелки можно не ставить.

Символы в схеме алгоритма должны быть одного размера, не должны изменяться углы и другие параметры. Минимальное количество текста помещается внутри каждого символа. Если объем текста превышает размер символа, то следует использовать символ комментария.

Алгоритм начинается и заканчивается символами «Начало» и «Конец». Для записи внутри символов команд используется естественный язык с элементами математической символики. Каждый графический символ схемы имеет один вход и один выход. Исключение составляют символ «Решение», который имеет один вход и ряд альтернативных выходов.

Схема алгоритма является наиболее наглядным способом представления алгоритма. При этом нет никаких ограничений на степень детализации, но отсутствие наглядности схемы не позволяет полноценно использовать ее в качестве инструкции к действию.

Структуры алгоритмов

Всякий раз, когда с задачей удается связать граф, обсуждение резко упрощается, и большие фрагменты словесного описания заменяются манипуляциями с картинками.
Ю. И. Манин

Различают последовательности действий, соответствующих структурам алгоритмов: линейные, разветвленные, циклические.

Линейный алгоритм

Линейная структура алгоритма предполагает, что для получения результата необходимо выполнить определенные шаги – действия, следующие друг за другом.

Пример

Для приготовления известкового цементного состава необходимо A % известкового теста, B % белого цемента, C % поваренной соли, D % пигмента, остальное составляет вода. Составить алгоритм расчета необходимого количества воды (в кг) при приготовлении Q состава, заданного в кг.

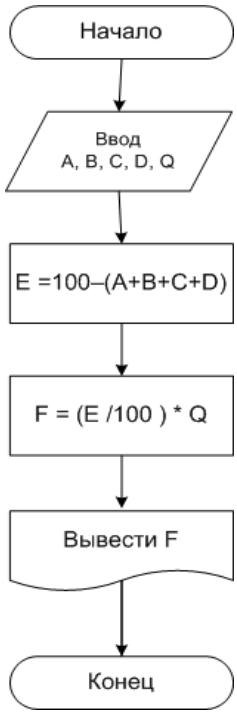


Рис. 1.3. Графическое изображение линейного алгоритма

Рассмотрим подробно условие задачи. Известно, что весь приготовленный цементный состав соответствует 100 % и определяется суммой (A + B + C + D + E) %, где E – процент воды. Чтобы определить количество воды в составе, обозначим его как F и воспользуемся следующим соотношением: F/Q = (E/100) %. Отсюда легко определить F = E×Q/100. Словесное описание линейного алгоритма будет иметь следующий вид:

1. Ввести A, B, C, D, Q.
2. Вычислить E = 100 – (A + B + C + D).
3. Вычислить F = (E/100) × Q.
4. Вывести F.
5. Конец.

Алгоритм имеет линейную структуру при любых исходных данных – каждое последующее действие следует из предыдущего. Одинаково наглядна как словесная запись, так и схема алгоритма на рис. 1.3.

Разветвленный алгоритм

Разветвленная структура алгоритма предполагает (рис. 1.4), что последовательность выполнения шагов алгоритма изменяется в зависимости от некоторых условий. Осуществляется выбор одного из двух / нескольких возможных вариантов. Словесно эта конструкция записывается так:

ЕСЛИ условие справедливо, ТО выполнить *действия*1,
ИНАЧЕ выполнить *действия*2.

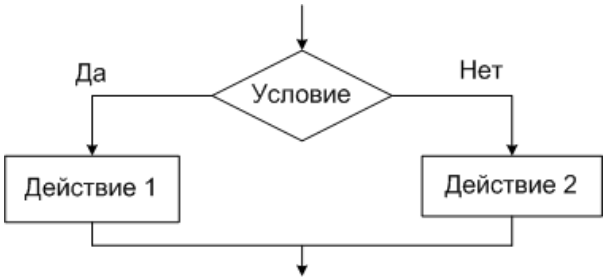


Рис. 1.4. Полная альтернатива

Разветвленный алгоритм содержит блок проверки некоторого условия, и в зависимости от результата проверки выполняется та или иная последовательность шагов (действий). Если есть «действия1» и «действия2», то говорят о *полной альтернативе* (рис. 1.4). Если же в качестве «действия 2» имеет место формулировка «перейти к п. N», то такая форма записи называется *неполной альтернативой* (рис. 1.5).

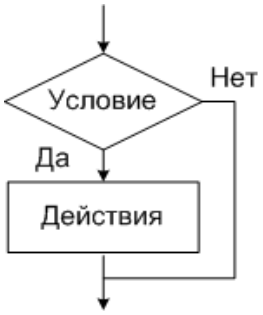


Рис. 1.5. Неполная альтернатива

> ***Условие*** *– это логическое выражение, которое может принимать два значения – «да», если условие верно, и «нет», если условие не выполняется.*

Часто условие к задаче задается на естественном языке, иногда используются общепринятые обозначения, иногда из незнакомой области приложения. Поэтому при формировании условия для алгоритмической записи следует помнить о принципе однозначности. Если имеется выбор нескольких альтернативных решений (более двух), то следует предусмотреть несколько выходов (рис. 1.6). Это можно исполнить: либо несколькими линиями от данного символа к другим символам, либо одной линией, которая имеет количество разветвлений, соответствующее количеству выходов.

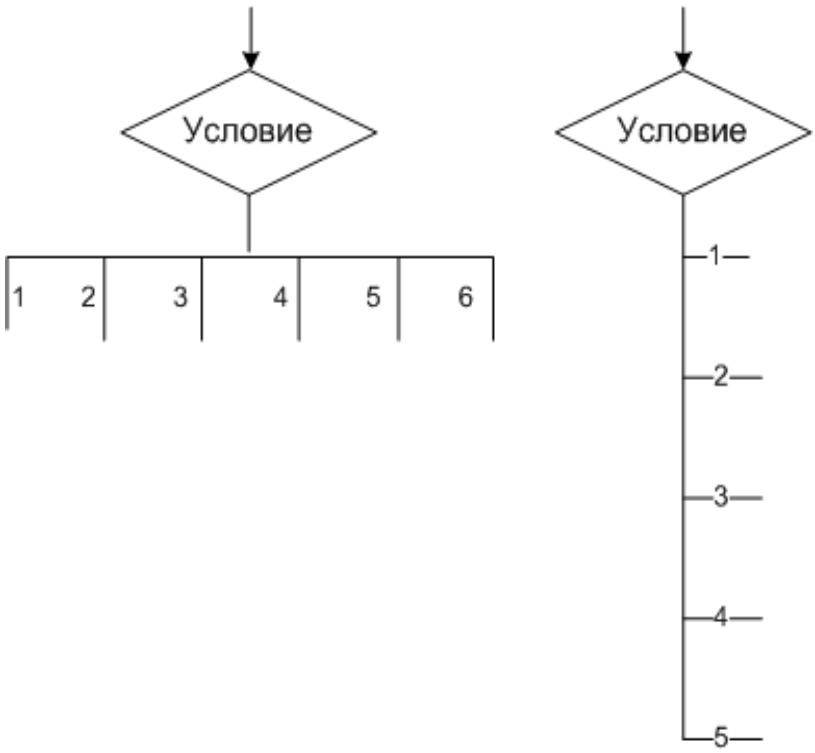


Рис. 1.6. Несколько альтернативных решений

При исполнении алгоритма один из выходов будет активизирован после вычисления условий, определенных внутри символа «решение». Соответствующие результаты вычисления могут быть записаны по соседству с линиями, отображающими эти пути. Каждый выход из данного символа должен сопровождаться соответствующими значениями условий, чтобы показать логический путь. Необходимо, чтобы все ссылки и условия были идентифицированы.

Пример

Самолет летит из пункта А в пункт В со средней скоростью V. Составить алгоритм для нахождения времени полета t_1, если есть встречный ветер, скорость которого V_1 и времени t_2, если ветра нет. Расстояние между пунктами А и В равно S.

Проанализируем условия задачи и выясним, какие данные мы имеем для составления алгоритма ее решения:

Исходные данные, которые надо ввести:
- скорость самолета;
- скорость встречного ветра;
- расстояние, которое пролетает самолет.

Условия:
- если нет встречного ветра – $v_1 = 0$;
- если встречный ветер имеется – $V_1 \neq 0$.

Формулы:
- вычисление времени, если есть встречный ветер $t_1 = s / (V – V_1)$;
- вычисление времени, если нет встречного ветра $t_2 = S / V_1$.

Так как имеются два альтернативных решения, алгоритм будет иметь разветвленную структуру. Псевдокод имеет следующий вид:

Начало
Ввести V, V_1, S
  Если $V_1 = 0$ *то* $t_2 = S / V_1$
    Вывести t_2
    Иначе $t_1 = S / (V – V_1)$ {если $V_1 \neq 0$ – комментарий*}
      Вывести t_1
  Конец Если
Конец

На рис. 1.7 показан разветвленный алгоритм с полной альтернативой. Схема алгоритма детально изображает особенности разработанного алгоритма.

Из двух вариантов решений может быть записано одно общее условие для блока «Решение», которое отражает «Да», если $V_1 = 0$ и «Нет», если $V_1 \neq 0$. Для каждого решения предусмотрен свой вывод результата.

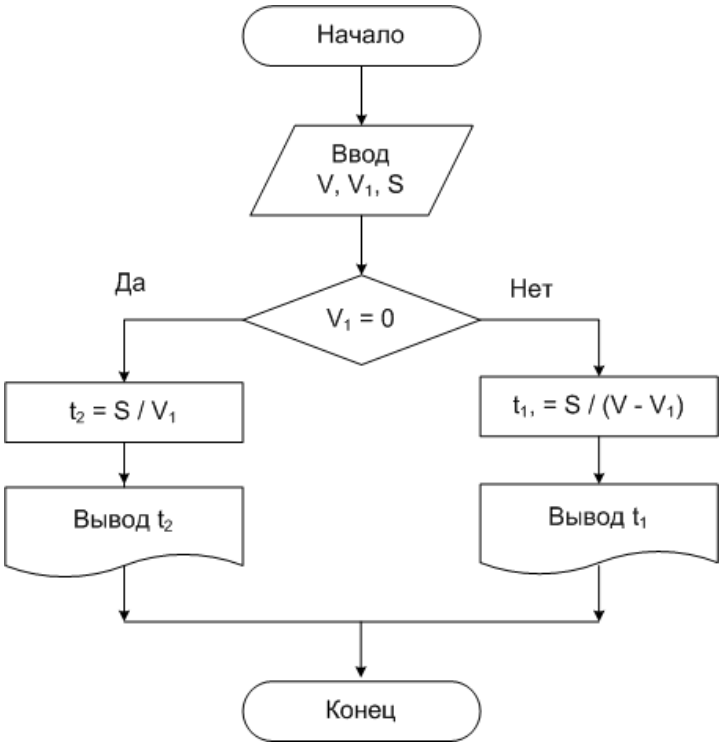


Рис. 1.7. Графическое изображение разветвленного алгоритма

На рис. 1.8 приведена схема алгоритма Евклида (вычисление НОД), словесная запись которого была приведена выше. Для алгоритма Евклида также используется разветвленный алгоритм с полной альтернативой, но вывод результата имеет смысл сделать общим. По сравнению со словесным алгоритмом схема алгоритма более наглядна.

Циклический алгоритм

Для обозначения многократно повторяющихся действий используются специальные циклические структуры. Такая структура содержит условие, которое необходимо для определения количества повторений для некоторой последовательности действий.

Основной блок цикла – ***тело цикла*** – производит требуемые вычисления с помощью операторов, расположенных внутри цикла и повторяющихся многократно в зависимости от условия работы цикла. Вспомогательные блоки цикла организуют циклический процесс: устанавливают начальное значение и новые значения данных, проверяют условие окончания циклического процесса.

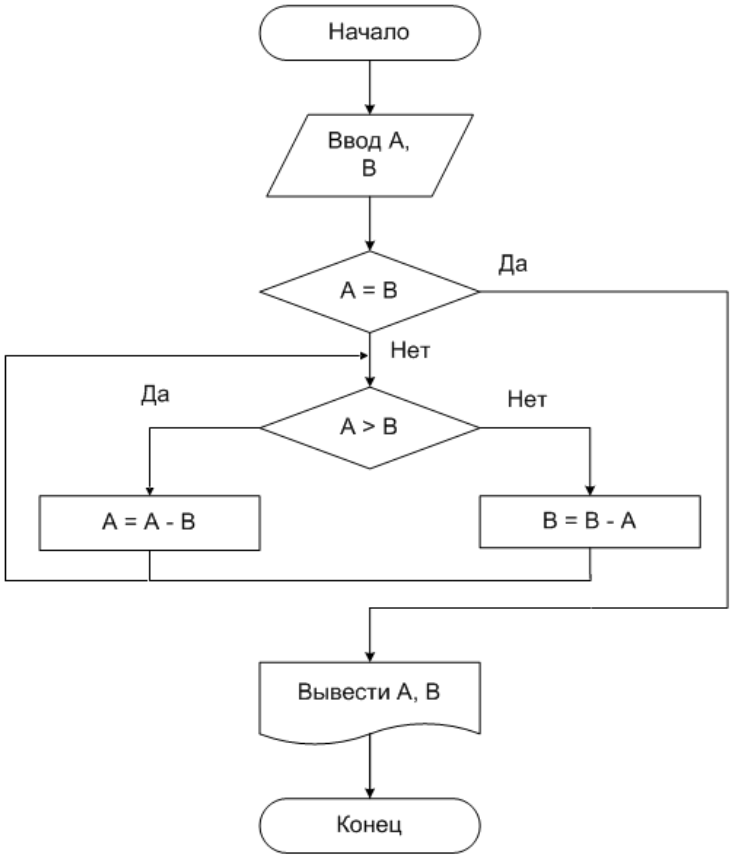


Рис. 1.8. Схема алгоритма Евклида – определение наибольшего общего делителя

Циклический алгоритм позволяет компактно описать большое число одинаковых вычислений над разными данными для получения необходимого результата. Различают следующие циклические структуры:

1) с предварительным условием (рис. 1.9) – циклы с предусловием (тело цикла может не выполняться ни одного раза);
2) с последующим условием (рис. 1.10) – циклы с постусловием (тело цикла выполнится хотя бы один раз).

Условие записывается в виде логического выражения. Операторы «тела цикла» для первой структуры повторяются, пока условие истинно. Если при входе в цикл условие невозможно выполнить, то будет осуществлен выход из цикла. Такой цикл может не выполниться ни разу.

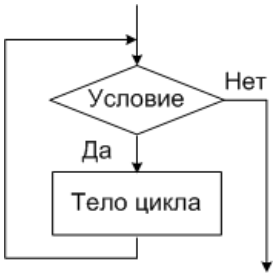




Рис. 1.9. Цикл с предусловием **Рис. 1.10. Цикл с постусловием**

Для циклов с постусловием условие проверяется на выходе. В этом случае операторы тела цикла,в отличие от предыдущего будут реализованы хотя бы один раз и до тех пор, пока не станет возможным условие выхода из цикла.

Для организации любого цикла необходимо выполнение следующих действий:

1) задать начальные значения параметра цикла перед его началом;

2) изменять параметры цикла перед каждым новым повторением цикла;

3) проверять условие повторения или окончания цикла;

4) переходить к началу цикла, если он не закончен, или выйти из цикла, если условие выхода из цикла выполняется;

5) чтобы цикл не повторялся бесконечное число раз, т.е. чтобы не произошло «зацикливание», необходимо правильно оформить изменение параметра цикла.

По способу определения числа повторений различают также *циклы с неизвестным числом* повторений и циклы с явно заданным числом повторений – *циклы с параметром*. Примером цикла с неизвестным числом повторений является итерационный цикл.

Выход из итерационного цикла осуществляется не после того, как цикл повторится заданное число раз, а при выполнении условия, связанного с проверкой значения монотонно изменяющейся в цикле величины. Часто эта величина характеризует точность, достигнутую на очередном шаге итерационного процесса, реализуемого алгоритмом. Для циклов с неизвестным количеством повторений возможен выход также по дополнительному условию, так называемые поисковые циклы.

Для изображения циклов с предусловием и постусловием можно использовать и другие символы (см. табл. 1.1, рис. 1.11, 1.12). В этом случае условия для инициализации, приращения, завершения и т.д. помещаются внутри символа – в начале или в конце – в зависимости от расположения операции, проверяющей условие.

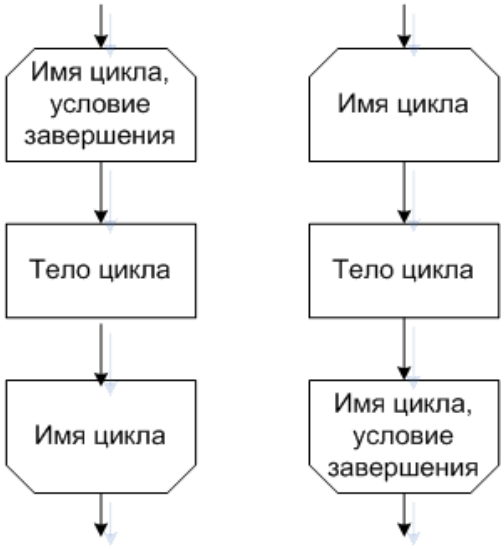


Рис. 1.11. Изображение цикла с помощью символа «Граница цикла»

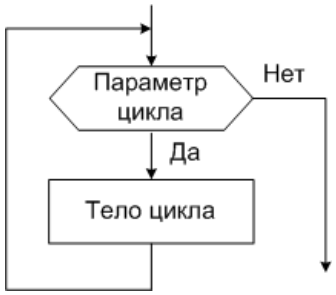


Рис. 1.12. Цикл с параметром

Цикл с параметром

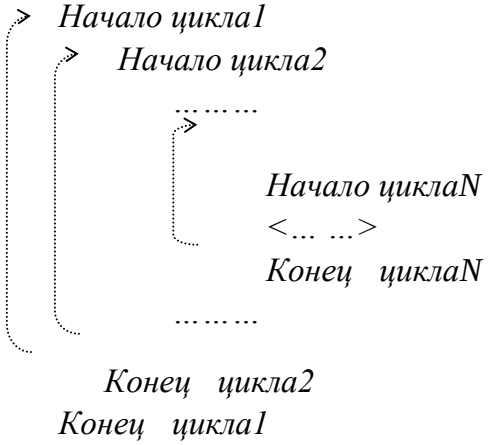
Определенная последовательность шагов выполняется несколько раз в зависимости от заданной величины. Эта величина называется параметром цикла. Цикл выполняется, пока параметр цикла принимает значения в заданном диапазоне с заданным шагом. Параметр цикла называется также счетчиком цикла. Это локальная переменная.

Параметр цикла включает имя переменной, его начальное и конечное (последнее) значения и шаг. ***Шаг*** – это величина изменения параметра цикла. Начальное, конечное значение и шаг должны иметь тот же тип, что и параметр цикла. Если начальное и конечное значения равны, то цикл не выполняется. Параметр цикла не следует изменять внутри цикла, не следует изменять внутри и конечное значение, так как эти значения устанавливаются в самом начале работы цикла и далее не вычисляются.

Тело цикла выполняется столько раз, сколько разных значений может принять параметр в заданных пределах. *Если тело цикла не содержит операторов, то цикл называется пустым.*

Цикл заканчивается, когда параметр цикла принимает конечное значение. Данный цикл выполняется в случае, если конечное значение параметра цикла больше начального. Параметр цикла после каждого выполнения тела цикла изменяется на величину шага. Если цикл не заканчивается и вызывает «зацикливание» алгоритма, то такой *цикл называется бесконечным*.

Алгоритмы решения сложных задач могут включать все перечисленные выше структуры. Например, внутри алгоритма циклической структуры может быть помещен другой цикл – *вложенный цикл*, при этом область действия внутреннего цикла должна полностью лежать в области внешнего цикла.



При вложенности циклы не должны «пересекаться». Для этого параметр каждого вложенного цикла должен иметь свое уникальное имя. Глубина вложенности определяется, прежде всего, условием задачи и возможностями компьютера.

Вычисление факториала

*Составить таблицу значений **1!, 2!, 3!, . . . , M!***

Для получения значений 1!, 2!, 3!, . . . , M! необходимо составить общую схему вычисления факториал **M! = 1 * 2 * 3 * (M – 1) * M**. Нам известно из математики, что факториал вычисляется как произведение от 1 до M, а именно:

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)!, \text{ для } \quad n \geq 1 \end{cases}$$

Примем за начальное значение переменной «ФАКТ» = 1. Если это значение задать равным 0, то в результате вычисления произведения все последующие значения будут нулевыми.

В схеме алгоритма рис. 1.13 используется цикл с параметром для k (символ «подготовка»), где k изменяется от 1 до M.

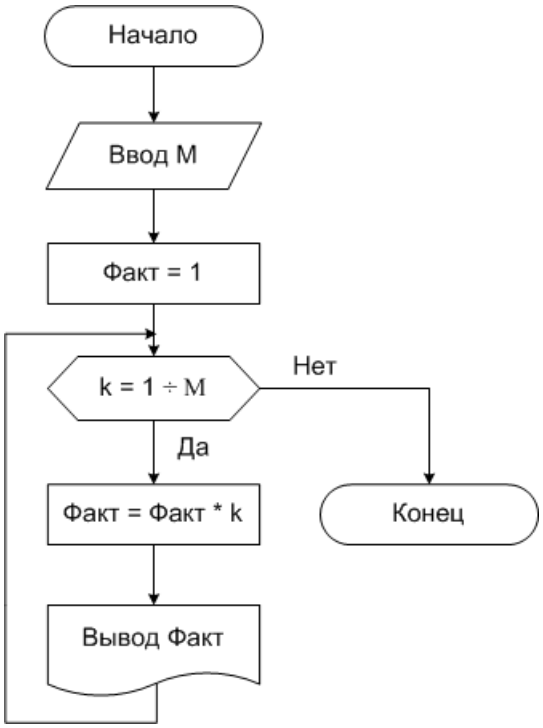


Рис. 1.13. Схема алгоритма для вычисления факториала

Словесное описание алгоритма для данной задачи имеет следующий вид:

1. Ввести M.
2. Факт = 1.
3. k = 1.
4. Факт = Факт * k.
5. k = k + 1.
6. Вывести «Факт».
7. Если k ≤ M, то перейти к п. 4, иначе перейти к п. 7.
8. Конец.

При составлении таблицы факториалов используется результат предыдущего действия. Вычисления будут выполнены за m шагов. Переменная «Факт» последовательно принимает следующие значения: 1; 1 × 2; 1 × 2 × 3; 1 × 2 × 3 × 4; и т.д.

Переменные с индексом. Массивы

Многие задачи, которые решаются с использованием компьютера, связаны с обработкой больших объемов информации, представляющей совокупность данных, объединенных единым математическим содержанием или близких между собой по смыслу. Примером являются координаты, задающие положение точки в пространстве, коэффициенты системы линейных уравнений, значение некоторой функции в произвольных точках, коэффициенты многочлена и т.д. Такие данные удобно представлять в виде линейных или прямоугольных таблиц.

В математике табличные величины называются векторами или матрицами. *В алгоритме и программе для представления табличных данных используется понятие* ***массив***. Массив состоит из компонент одного типа, поэтому структура массива однородна. Индекс или порядковый номер элемента массива имеет значение целого или порядкового типа. С помощью индекса можно выделить любую отдельную компоненту любого массива. Компоненты массива в связи с этим называют переменными с индексом.

> ***Массивом*** *называется совокупность однотипных данных, связанных общим именем.*

Переменная с индексом позволяет представить большое количество величин или компонент с одним общим именем. Каждая отдельная величина определяется индексом/ индексами в скобках после наименования переменной. Полная система таких величин называется массивом, а каждая отдельная величина – компонентом или элементом массива. Переменная с индексами может иметь один, два или три индекса и выше. Массив может быть одномерным (вектор), двумерным (матрица), трехмерным (тензор) и т.д. Термин «одномерный» определяет количество индексов (один), а не количество переменных. Число индексов определяет размерность массива.

Первая компонента одномерного массива – это элемент с номером 1, вторая – элемент с номером 2 и т.д. до тех пор, пока не будут пронумерованы все элементы. В обычных математических обозначениях возможна запись X_1, X_2, … , X_9, X_{10}. При программировании запись более формализована X(1), X(2), … , X(9), X(10).

Двумерный массив представляет собой таблицу или матрицу из горизонтальных строк и вертикальных столбцов. При этом первый из двух индексов определяет номер строки. Номер строки изменяется от 1 до M, где M – полное количество строк. Второй номер столбца, изменяющийся от 1 до N, где N – полное число столбцов. В обычной математической записи двумерный массив может быть представлен как

$$\begin{matrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{matrix}$$

При формализации задания он будет записан как A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), A(2,3). Индексы отделяются запятыми. Задавая соответствующие значения индексов, можно выполнить прямой доступ к любой компоненте массива. Текущие значения индексов не должны выходить за пределы заданного диапазона. В противном случае переменная с индексом не может быть определена.

Обычный прием работы с массивами, в особенности с большими массивами – выборочное изменение отдельных его компонент. При этом переменная массив рассматривается как массив составляющих переменных, и возможно присваивание отдельным компонентам, например, M[i] = 912.245. Выборочное изменение компоненты может привести к коррекции только одной компоненты массива. Но это может рассматриваться и как изменение всего составного значения элемента массива, если оно было представлено в виде формулы.

Так как индекс массива должен иметь определенный тип, то это позволяет вычислять индекс. На месте индекса может быть выражение, соответствующее значению индекса, которое идентифицирует соответствующую компоненту массива. Такое вычисляемое значение следует непременно контролировать, чтобы оно не выходило за пределы интервала значений индекса, заданного для этого массива.

Итак, основными характеристиками массива являются:

- имя массива;
- тип компонентов массива;
- размерность, равная количеству индексных позиций (измерений) массива;
- порядок нумерации компонент в последовательности (значения верхней и нижней границ для каждого индекса);
- количество компонент – длина массива или размер массива.

Имя массива образуется по общему правилу образования имен, т.е. представляет собой идентификатор. Однако оно не должно совпадать с именем ни одной простой переменной, используемой в этом же алгоритме.

Обработка одномерных массивов представляет значительный класс задач, которые связаны с последовательной обработкой элементов массива:

- задачи ввода и вывода;
- изменение порядка и значений (удаление и вставка элементов, сдвиг элементов массива, изменение длины массива и др.);
- вычисление суммы, произведения, среднего геометрического, среднего арифметического элементов массива;
- поиск данных в массиве по заданному признаку, поиск максимального и минимального элемента массива;
- сортировка данных в массиве.

Последовательную обработку массивов позволяет обеспечить цикл с параметром, который осуществляет прямой доступ к каждому элементу массива.

Для обработки двумерных массивов используются те же приемы, что и для одномерных массивов.

Вычисление суммы

Вычислить сумму $= \sum_{i=1}^{N} X_i \times Y_i$.

Другими словами, необходимо вычислить сумму S, которая представляет собой произведение вводимых чисел X_i, Y_i. Переменные выступают как два одномерных массива одной длины N. Размер массивов либо следует предварительно ввести, либо при исполнении алгоритма назначить конкретное число N. Символ $\sum$ (сигма) в математике используется для обозначения определения суммы одноименных данных, индексы которых изменяются; в данном случае i изменяется от 1 до N.

Алгоритмы вычисления суммы являются стандартными циклическими алгоритмами. Переменная, которая определяет сумму, выступает как накопитель.

Перед первым выполнением цикла необходимо задать начальное значение суммы (количество, разность, произведение и т.д.), так как не было произведено никаких действий для вычисления суммы, то ее начальное значение, если не было дополнительных условий, всегда равно нулю.

Начинаем вычисления. Шаг 1. Номер выбираемого элемента равен 1. В цикле вводим переменные X_1 и Y_1, элементы одномерных массивов.

Далее вычисляем сумму, значение которой равно предыдущему значению суммы (на первом шаге оно равно 0) + произведение X_1 *и* Y_1.

Переходим к следующим элементам, для этого *увеличиваем значение номера элемента* (или номера шага).

Проверяем, не превышает ли номер элемента количество элементов в массиве. В данном случае сравнение с N – заданный размер массива. Если не превышает N, то переходим к шагу 2 и вычисляем следующее значение суммы. Значение суммы равно сумме, полученной на шаге 1, + произведение X_2 и Y_2.

И так далее, пока не будут просуммированы все ($X_i \times Y_i$).

Словесное описание циклического алгоритма примет следующий вид:

1. Начальное значение суммы равно 0.
2. Начало с первого элемента, i = 1.
3. Ввести Xi , Yi.
4. Вычислить сумму: *сумма* = *сумма* + Xi *Yi.
5. Продолжить суммирование – перейти к следующему элементу i = i + 1.
6. Если i ≤ n, то перейти к п. 3, иначе перейти к п. 7.
7. Вывести значение «*сумма*».
8. Конец.

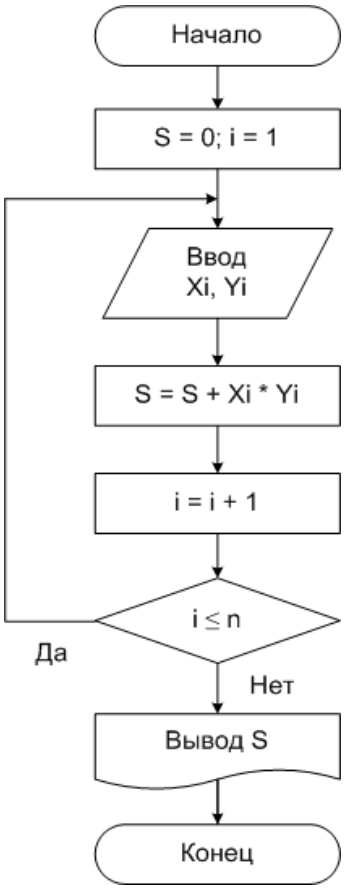


Рис. 1.14. Графическое изображение циклического алгоритма

Схема алгоритма изображена на рис. 1.14. Перед первым выполнением цикла необходимо задать начальное значение «сумма» = 0 и затем n раз вычислять «сумму» S при различных значениях Xi и Yi, где i принимает последовательно значения 1, …, n и используется цикл с постусловием, который можно заменить циклом с параметром.

В схеме алгоритма заменим слово «сумма» буквой S. Исходные данные – любые числовые значения – необходимо задавать на каждом шаге выполнения алгоритма. Для реализации такой алгоритм является достаточно простым.

Исполнение (тестирование) алгоритма

Алгоритм описывает правила и действия, реализуя которые можно получить решение задачи, если использовать исходные данные. Чтобы убедиться в правильности алгоритма, необходимо проводить его тестирование, не дожидаясь рационального окончательного алгоритма. В этом случае для фиксирования исполнения алгоритма указываются шаги алгоритма, аргументы, промежуточные величины, результаты на данном шаге и проверяемые условия.

Шаги исполнения дают возможность показать, сколько шагов исполнено на данный момент, за какое число шагов весь алгоритм будет исполнен. Нумерация шагов алгоритма упрощает ориентацию в описании алгоритма и в процессе его исполнения. При этом в каждый момент видно, какой элемент (оператор/ команда) алгоритма выполняется. Это помогает устранять ошибки.

Самый простой вариант тестирования – это решение контрольного примера, когда при известных данных известен результат.

Пример

Составить алгоритм для нахождения квадратного корня из любого натурального числа x.

Используем для составления алгоритма следующие вспомогательные переменные:

a и b – для определения интервала от 1 до x;

a1 – деление интервала [a, b] пополам;

x1 – переменная для промежуточных вычислений.

В алгоритме можно организовать цикл с предусловием, так как неизвестно количество шагов, за которое будет найдено значение корня a. Составим алгоритм в виде псевдокода:

Начало

Ввести x

a = 1

b = x + 1

```
Пока b – a > 1 выполнять цикл          {цикл с предусловием}
    a1 = ( a + b ) \ 2                 {целочисленное деление}
    x1 = x \ a1                        {целочисленное деление}
    Если x1 ≥ a1 то b = a1
                иначе a = a1
    Конец Если
Конец цикла
Вывести «Квадратный корень из x = », a
Конец
```

Теперь построим таблицу (табл. 1.2), каждый столбец которой соответствует изменениям значений переменных a, b, a1, x1, используемых в данном алгоритме.

Выполним тестирование для x = 4.

Таблица 1.2

Тестирование алгоритма

Шаг	**Проверка условий**	**a**	**B**	**a1**	**x1**	$\sqrt{x}$
Шаг 1		1	4+1=5			
	Проверка условия b – a >1? «да» 5 – 1 > 1			(1+5)\2=3	4\3=1	
	Проверка условия x1 ≥ a1? «нет» 3 > 1	1	=a1=3			
Шаг 2	Проверка условия b – a >1? «да» 3 – 1 > 1			(1+3)\2=2	4\2=2	
	Проверка условия x1 ≥ a1? «нет» 2 = 2	2	3			
	Проверка условия b – a >1? «нет» 3 – 2 = 1					**2**

Как видно из таблицы, квадратный корень из 4 равен 2. Алгоритм выполнен за два шага. Если в качестве исходных данных использовать x = 9, то искомый результат ($\sqrt{9}$ = 3) будет получен аналогично, но за три шага. Особенность алгоритма состоит в том, что он выполняется формально по заранее составленному плану.



Вопросы для самопроверки

1. Что называют алгоритмом?
2. Назовите основные этапы решения задачи на компьютере?
3. В чем заключаются особенности выбора метода решения?
4. Какие вы знаете способы записи алгоритма?
5. Какие основные виды структур алгоритма?
6. В чем отличие линейной структуры от разветвленной?
7. Что такое полная альтернатива и неполная альтернатива для разветвленного алгоритма?
8. В чем состоит рациональность применения в алгоритмах циклических структур?
9. Для решения каких задач используются циклические структуры?
10. Что такое «тело цикла»?
11. Может ли «тело цикла» не выполнится ни разу и в каких случаях?
12. В каких случаях используется цикл с параметром?
13. Чем отличается цикл «с предусловием» от цикла «с постусловием»?
14. В чем состоит проблема «зацикливания»?
15. В каких случаях используются вложенные циклы?
16. Какова допустимая глубина вложенных циклов?
17. Внешний цикл образован оператором цикла с параметром. Можно ли в качестве вложенного цикла использовать цикл с условием?
18. Можно ли в качестве параметра цикла во внешнем и внутреннем цикле использовать одну и ту же переменную?
19. Что такое массив?

20. В чем отличие одномерного массива от двумерного?
21. В чем заключается прямой допуск к элементам массива?
22. Как заполняется массив исходными данными?
23. Сколько индексов характеризует двумерный массив, и что они обозначают?
24. Какой индекс изменяется быстрее в двумерном массиве?
25. В чем заключается метод пошаговой детализации?

Глава 2

Реализация алгоритмов

После того как алгоритм разработан и мы убедились в том, что он работает верно, его необходимо реализовать посредством языка программирования, т.е. теперь можно приступить к написанию программы. Часто в техническом задании указывается, на каком языке программирования должен быть реализован алгоритм. Иначе его придется выбирать, исходя из возможностей реализации данного алгоритма на каком-то конкретном языке программирования.

> *Система программирования, которая входит в состав любого современного компьютера, является совокупностью средств, которые обеспечивают автоматизацию разработки и отладки программ. Система программирования включает также языки программирования, трансляторы с этих языков, библиотеки программ, компоновщики, отладчики программ и другие средства, необходимые для реализации программ.*

Язык программирования является искусственным языком, в котором строго предопределены синтаксис и семантика, поэтому в отличие от естественного языка здесь не допустимы никакие произвольные интерпретации. По близости к машинному языку языки программирования подразделяют на *языки низкого уровня* и *языки высокого уровня*. К первой группе относятся язык Ассемблер и машинные языки, языки, элементами которого являются команды компьютера, каждый оператор заменяется цифровым кодом этой команды.

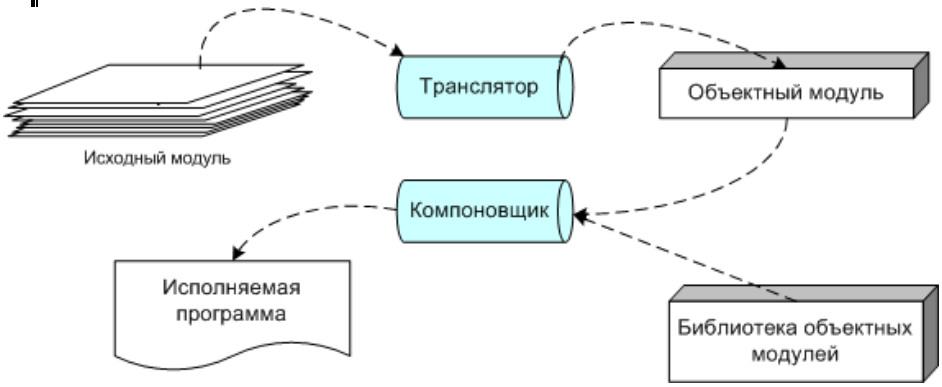
Языки высокого уровня позволяют отображать большинство алгоритмов в удобной форме, близкой к привычной математической записи. Существуют языки программирования, предназначенные для определенных областей применения. Языки программирования бывают также процедурно-ориентированные, проблемно-ориентиро-

ванные, машинно-ориентированные. Сегодня актуальными являются языки объектно-ориентированные, такие как C++, C#, Java, Python и др.

Программа обычно состоит из нескольких обособленных частей – программных модулей.

Специализированная среда программирования, как правило, имеет свой текстовый редактор, в котором пользователь обычно создает исходный текст программы на соответствующем языке программирования. Далее текст программы пользователь преобразует в исходный файл (или модуль).

> *Исходный модуль – исходный текст или программа на языке программирования.*



Чтобы программу выполнить, ее необходимо преобразовать в форму, которую может воспринимать компьютер. Это преобразование называется *трансляцией*, а программа, выполняющая это преобразование, – *транслятором*.

> *Транслятор – программа, преобразующая программу, написанную на одном языке, в программу, которая представлена на другом языке.*

В зависимости от порядка следования этапов трансляции трансляторы принято делить на ассемблеры, интерпретаторы и компиляторы.

Ассемблер преобразует исходный модуль в объектный, который является разновидностью объектной программы. Объектный модуль содержит машинные команды. *Компилятор* транслирует всю

программу сразу и выполнение программы идет далее без компилятора. Например, программа на языке Си преобразуется с помощью компилятора в программу на языке Ассемблера.

Транслятор, который последовательно анализирует предложения языка и немедленно их выполняет, а не компилирует, называют *интерпретатором*, т.е. этапы трансляции и выполнения реализуются поочередно.

Если в процессе трансляции не было обнаружено ошибок в исходном модуле, то будет создан объектный модуль. После трансляции программу сложно изменить. Готовую откомпилированную программу можно сразу запустить, но нельзя просмотреть исходный программный код. Такая система безопасности позволяет защитить программный код в целостности.

> *Объектный модуль содержит последовательность машинных команд и дополнительную информацию для связи с другими объектными модулями*.

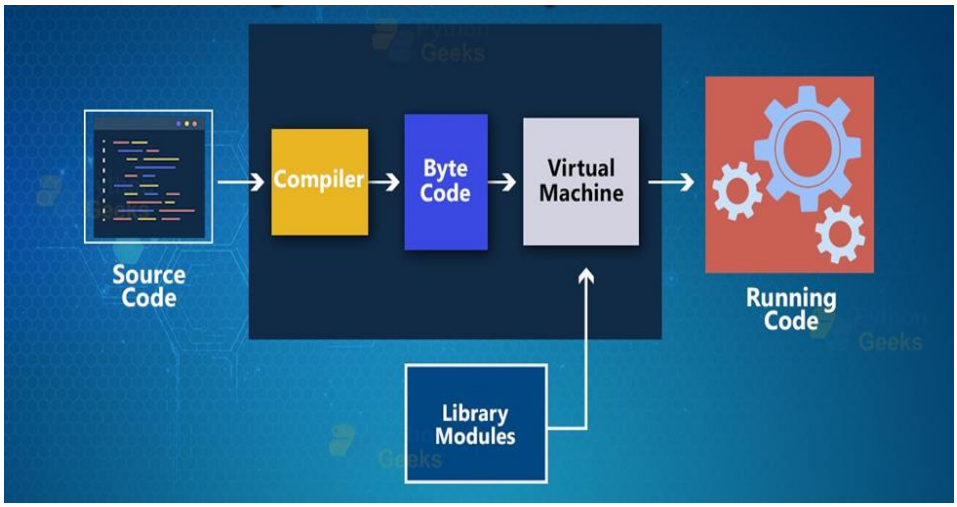
В формируемую программу транслятор включает обращение к объектным модулям, которые вычисляют значения математических функций, операции ввода-вывода и т.д. Эти модули в библиотеках подпрограмм автоматически извлекаются компоновщиком. Компоновщик объединяет несколько объектных модулей в один загрузочный модуль, который сформирован и готов для непосредственного выполнения на компьютере.

> *Загрузочный модуль, или выполнимый модуль, содержит программу на машинном языке, либо уже настроенный на конкретные адреса памяти (абсолютный модуль), либо с дополнительной информацией для настройки на адреса памяти (переместительный модуль) при загрузке модуля в память. Такая программа готова к выполнению. Обычно это файл с расширением *.exe.*

Чтобы ускорить и упростить поиск и исправление ошибок в программах, в системах программирования предусмотрены отладчики программ, которые упрощают процесс поиска ошибок, позволяют проверить отдельные фрагменты программ и проследить за изменением отдельных переменных.



> *Python – это интерпретируемый язык программирования. Он не конвертирует свой код в машинный, который понимается аппаратурой компьютера (в отличие от C и C++). Вместо этого Python-интерпретатор переводит код программы в байт-код, который запускается на виртуальной машине Python (PVM):*



Интерпретатор – это программа, которая конвертирует инструкции, написанные на Python, в байт-код и выполняет их. Интерпретатор – это программный слой между исходным кодом и аппаратурой компьютера. Интерпретатор компилирующего типа содержит компилятор и собственно интерпретатор. Компилятор переводит исходный код программы в промежуточное представление (байт-код), а интерпретатор (виртуальная машина) выполняет этот байт-код.

CPython интерпретатор компилирующего типа (благодаря этому достигается большее быстродействие выполнения программ), считается эталонной реализацией языка Python. Написан на C и этот исходный код CPython находится в открытом доступе. Его разработка проведена группой разработчиков под руководством Гвидо ван Россума – создателя Python. Данный интерпретатор может работать в режиме диалога (REPL – read-eval-print loop). Интерпретатор считывает законченную конструкцию языка, выполняет ее, выводит результаты и переходит к ожиданию ввода пользователем следующей конструкции.

Существуют и другие реализации, созданные для специфических целей и задач. Например, PyPy – это интерпретатор Python, написанный на RPython (ограниченный, Restricted Python). Особенностью PyPy является использование трассирующего JIT-компилятора (just-in-time), который «на лету» транслирует некоторые элементы в машинный код. Благодаря этому при выполнении некоторых операций PyPy обгоняет CPython в несколько раз. Но плата за такую производительность – более высокое потребление памяти.

От простых алгоритмов к сложным...

Никто не сможет научиться хорошо составлять большие программы, пока он не научится составлять малые....
Д. Грис

Учимся составлять алгоритмы на примерах. Рассмотрим примеры наиболее распространенных алгоритмов для решения задач. Прежде чем попытаться построить алгоритм, убедитесь, что для вас понятен смысл задачи, уточните, в чем состоит условие задачи, какие данные известны, а какие нет. Задачи в этом разделе расположены в порядке возрастания сложности.

___Пример 2.1. Время, заданное в минутах

Составить алгоритм для перевода значения интервала времени, заданного в минутах, в интервал, выраженный в часах и минутах.

Чтобы найти величину интервала, заданного в минутах *Тмин*, неплохо вспомнить, что 60 минут составляет интервал в 1 час. Из этого соотношения легко определить, сколько *Тчас* и *Тмин* в интервале, заданном в минутах, разделить исходное значение *Тмин* на 60. Результат деления = *Тчас*, а остаток от деления = *Тмин*.

Словесный алгоритм имеет следующий вид:

1. Ввести время в минутах *Тмин*.
2. *Тчас* = целое *Тмин* разделить на целое 60.
3. *Тмин* = остаток от деления целого *Тмин* на целое 60.

4. Вывод результата *Тчас*, *Тмин*.
5. Конец.

Алгоритм имеет линейную структуру. Как видно из алгоритма, исходные данные – *Тмин* должно быть целое число, результат – *Тчас* – также имеет целый тип. Для реализации целочисленного деления и вычисления остатка от деления в современных языках программирования имеются необходимые средства, поэтому программа будет иметь следующий вид (рис. 2.1):



```
tmin = int(input("Введите время в минутах: "))
thour = tmin//60
tmin %= 60
print("Время: {0} часов, {1} минут".format(thour, tmin))
```

```
Введите время в минутах: 90
Время: 1 часов, 30 минут
```

Рис. 2.1. Пример нахождения времени, заданного в минутах

___Пример 2.2. Наибольшее из трех чисел

Даны три числа a, b, c. Составить алгоритм для нахождения наибольшего.

Введем обозначения для «наибольшего» как max. В результате выполнения сравнений: a и b, a и c, далее b и a, b и c и, наконец, c и a, c и b, мы получим всего три альтернативных варианта выбора решения, в каждом из которых присутствует одна из трех исходных величин.

В словесном алгоритме для каждого из списка a, b, c запишем одно условие:

1. Ввести a, b, c.
2. Если a > b и a > c, то max = a.
3. Если b > a и b > c, то max = b.
4. Если c > a и c > b, то max = c.
5. Вывод max.
6. Конец.

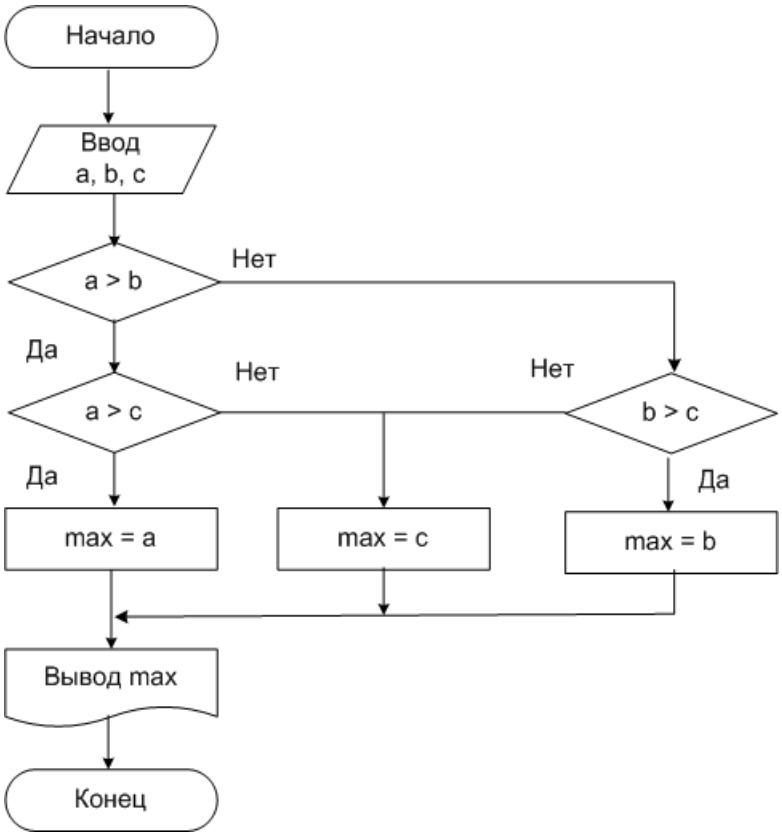


Рис. 2.2. Нахождение максимального из трех величин

В схеме алгоритма символ «Решение» содержит только одно условие, так как при этом имеется возможность использовать две альтернативы. Например, a > b, если «да», то a больше b, если «нет», то a меньше b или b больше a. С точки зрения реализации схема алгоритма (рис. 2.2) более наглядна. В качестве исходных данных могут выступать любые числовые значения.

___Пример 2.3. Сумма цифр

Дано некоторое целое число. Найти сумму цифр заданного числа. Составить алгоритм.

Введем обозначения для числа N, которое является исходным. Сумму цифр числа N обозначим через S. Ее необходимо подсчитать, поэтому на начальном этапе эта сумма должна быть равной 0. Переменная Nост выполняет вспомогательные функции и обозначает остаток от деления целого числа N на 10.

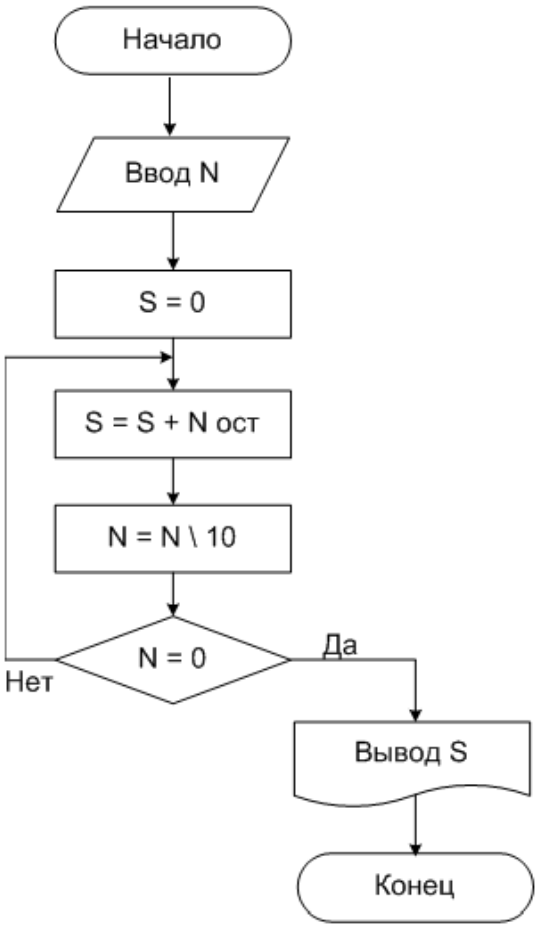


Рис. 2.3. Схема алгоритма – сумма цифр числа

Если известно, сколько знаков в числе, то цифры числа можно найти, используя операцию целочисленного деления. Например, для трехзначного числа N его цифры X, Y и Z:

X = N//100
Y = N//10%10
Z = N%10

Здесь «%» – стандартная операция для целых чисел – нахождение остатка от деления. Данная операция используется во многих языках программирования. В схеме алгоритма (рис. 2.3) «%» обозначено как Nост. Составить число N можно в виде суммы: N = 100*X + + 10*Y + Z.

Фрагмент программы для реализации вычисления суммы цифр любого числа имеет вид (рис. 2.4)

```
n = int(input("Введите число: "))
    s = 0
    while n!=0:
        s += n%10
        n //= 10

    print("Сумма цифр числа: {0}".format(s))
```

```
Введите число: 54783
Сумма цифр числа: 27
```

Рис. 2.4. Пример нахождения суммы цифр числа

___Пример 2.4. Ввести и изменить значения двух целочисленных переменных в зависимости от результата сравнения

Даны целые числа M и N, меньшее из этих чисел заменить их полусуммой, а большее – удвоенным произведением этих чисел. Составить алгоритм.

Схема алгоритма (рис. 2.5) должна реализовать оба возможных варианта для каждой из двух переменных в зависимости от их значений.

Реализующая алгоритм программа имеет следующий вид (рис. 2.6):

```
    m = int(input("Введите первое число: "))
    n = int(input("Введите второе число: "))

    res_m = 0
    res_n = 0

    if m > n:
      res_m = m * n * 2
      res_n = int((n + m) / 2)
    elif n > m:
```

```
    res_n = m * n * 2
    res_m = int((n + m) / 2)

print("m = {0}, n = {1}".format(res_m,  res_n))
```

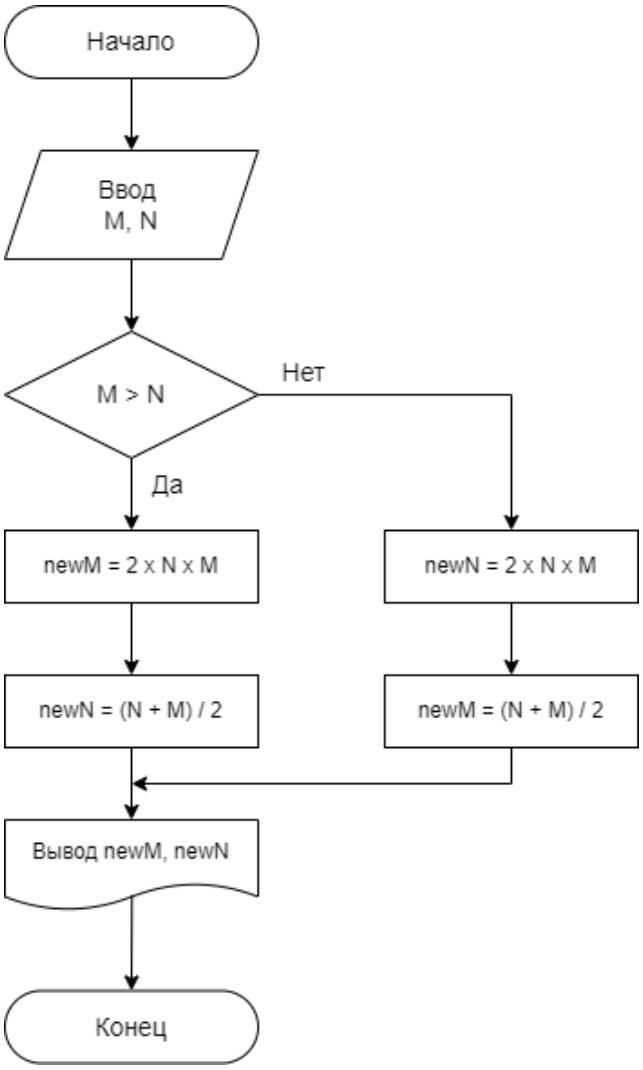


Рис. 2.5. Схема алгоритма нахождения меньшего из двух чисел

```
Введите первое число: 8
Введите второе число: 2
m = 32, n = 5
```

Рис. 2.6. Пример нахождения новых значений двух переменных

___Пример 2.5. Значение функции

Вычислить значение функции $Z = \dfrac{p + \dfrac{1}{x}}{px + 3} + \dfrac{23}{\sin x}$ *при известных значениях x и p. Составить алгоритм.*

Для вычисления значения Z необходимо провести анализ исходных данных, при которых можно реально обеспечить решение. В выражении присутствуют три дроби. Величины, которые составляют знаменатели этих дробей, не определены однозначно. Поэтому необходимо определить условия, при которых решение этого выражения является возможным.

Если значения для вычисления px и sinx можно определить, то и все исходное выражение, в целом, будет иметь решение. Для этого необходимо выполнить условия проверки исходных данных, такие как $x \neq 0$ и $px \neq -3$ (рис. 2.7).

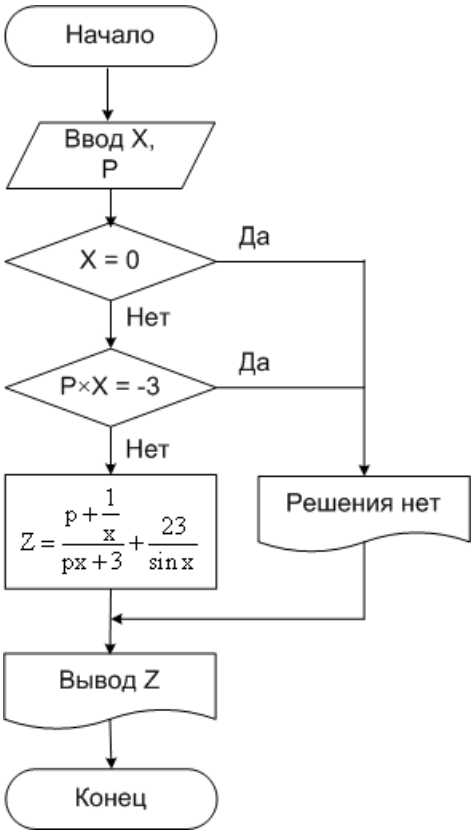


Рис. 2.7. Схема алгоритма для вычисления функции Z

Схема алгоритма (рис. 2.7) имеет разветвленную структуру при возможности альтернативного выбора из двух решений, хотя на самом деле выполняется выбор из четырех вариантов возможных решений. Первое условие «переходит» во второе, которое является одной из альтернатив первого условия, при этом варианты решения совпадают для первого и второго условия.

При реализации алгоритма необходимо помнить, что исходные данные и результат должны иметь вещественный тип (рис. 2.8, 2.9) .



```
import math
p = float(input("Введите p: "))
x = float(input("Введите x: "))
z = 0
if (x==0) or (p*x==-3):
    print("Решения нет")
else:
    z = (p+1/x)/(p*x+3)+23/math.sin(x)
 print("z = {0}".format(z))
```

```
Введите p: 12
Введите x: 1.57
z = 23.578621883923564
```

Рис. 2.8. Пример вычисления функции Z

```
Введите p: 3
Введите x: 0
Решения нет
z = 0
```

Рис. 2.9. Пример вычисления функции Z = 0

___Пример 2.6. Три условия

Составить алгоритм для вычисления функции при известном значении X:

$$Z = \begin{cases} X - 0{,}5 \text{ при } X < 0, \\ X \text{ при } 0 \le X < 4 \\ \sin X + \cos X \text{ при } X \ge 4 \end{cases}$$

Для составления алгоритма не требуется дополнительного анализа данных. Условие $0 \le X < 4$ при составлении схемы алгоритма следует разбить на два простых условия. Из возможных четырех условий получаем три в соответствии с условием задачи.

Схема алгоритма (рис. 2.10) имеет разветвленную структуру, в которой три альтернативных решения, причем второе условие является одной из альтернатив первого условия. Вывод результата должен быть организован как общий – один. Исходные данные и результат имеют вещественный тип.

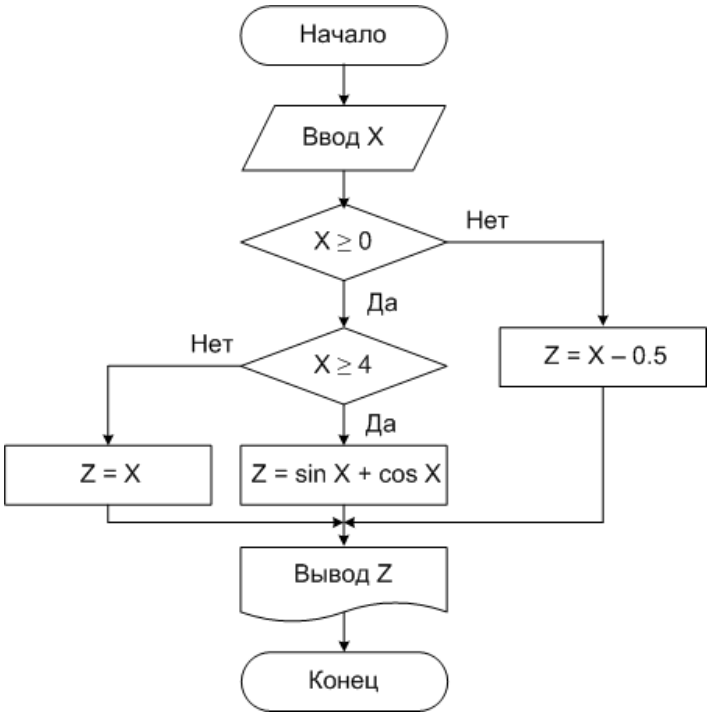


Рис. 2.10. Схема алгоритма для вычисления функции Z

Фрагмент программы имеет следующий вид (рис. 2.11):



```
import math
x = float(input("Введите x: "))
z = 0

if (x < 0):
   z = x - 0.5
elif (x >= 0) and (x < 4):
   z = x
elif x >= 4:
   z = math.sin(x) + math.cos(x)
print("z = {0}".format(z))
```

```
Введите x: -0.314
z = -0.8140000000000001
```

Рис. 2.11. Пример вычисления функции



Вопросы для самопроверки

Составить алгоритмы для решения следующих задач:

1. Определить время, через которое встретятся два тела, которые движутся навстречу друг к другу с равным ускорением, если известны их начальные скорости, ускорение, начальное расстояние.
2. Уравнение движения материальной точки имеет вид $x = -0.2 R^2$. Определить координаты точки через t секунд и путь s, пройденный за это время.
3. Колебательное движение точки описывается уравнением $x = 0.5 \cos \pi t$. Определить значения координаты, скорости ускорения через t секунд после момента t_0.

4. Материальная точка совершает прямолинейное движение по закону $s(t) = at + bt^2 – ct^3/3$, где $s(t)$ – путь; t – время. Определить наибольшее значение скорости.
5. Определить высоту фундамента, которая определяется формулой $h = \frac{b - b_0}{2\mathrm{tg}\alpha}$, где b и b_0 – параметры ширины фундамента, a – угол, который зависит от соотношений h, b, b_0.
6. Смешано v_1 литров воды температуры t_1 с v_2 литрами воды температуры t_2. Вычислить объем и температуру образованной смеси.
7. Будем считать, что Земля – идеальная сфера, радиус которой $R = 6350$ км. Определить расстояние до линии горизонта от точки с заданной высотой над Землей.
8. Известны количество жителей в государстве N и площадь его территории. Определить плотность населения в этом государстве.
9. При использовании дюймовой системы площадь измеряют в квадратных футах и квадратных ярдах[3]. Найти сумму двух площадей, выраженных в дюймовой системе, и перевести ее в метры.
10. В Великобритании жидкости измеряют в галлонах и бушелях[4]. Перевести объем жидкости, который задан в английских единицах, в литры.
11. Даны целые числа m, n ($0 < m \le 12$, $0 \le n < 60$), которые указывают конкретное время: «m часов, n минут». Определить наименьшее время – число полных минут, которое пройдет до того момента, когда:

а) часовая и минутная стрелки совпадут,

б) часовая и минутная стрелки расположатся перпендикулярно друг другу.

12. Известен год, номер месяца и число дней рождения человека, а также текущая дата: год, месяц и число. Определить возраст человека.

---

[3] *1 дюйм ≈ 2,54 см 12 дюймов составляют 1 фут; 3 фута равны 1 ярду;*
[4] *1 галлон = 4,54609 л; 8 галлонов составляют 1 бушель.*

13. Известны даты рождения двух человек (год, номер месяца и число) и текущая дата. Определить, кто из них моложе.
14. Даны натуральные числа a, b, c, d. Определить, можно ли из данных четырех отрезков составить прямоугольник.
15. Даны вещественные положительные числа a, b, c. Определить, существует ли треугольник со сторонами a, b, c.
16. Треугольник ABC задан длинами своих сторон. Найти длину высоты, опущенной из вершины A.
17. Определить высоту треугольника, если площадь треугольника равна s, а основание больше высоты на величину a.
18. Вычислить значение функции F(x), если

    а)

$$F(x) = \begin{cases} 0, & \text{при } x \le 0, \\ x^2 - x, & \text{при } 0 < x \le 1, \\ x^2 - \sin \pi x^3, & \text{в остальных случаях.} \end{cases}$$

    б)

$$F(x) = \begin{cases} kx, & \text{при } k < x, \\ kx + x, & \text{при } k \ge x, \end{cases}$$

где

$$k = \begin{cases} x^2, & \text{если } \sin x < 0, \\ |x|, & \text{если } \sin x \ge 0. \end{cases}$$

19. Определить значение $x = \lg y$, если $y = at^2 + bt + c$. Величины a, b и c известны.
20. Проверить, являются ли числа по обе стороны от заданного четного числа близнецами. *Два нечетных простых числа, разница между которыми равна двум, называются близнецами*.
21. Определить, являются ли значения целочисленных переменных M и N кратными 3. Если оба кратны 3, то вычислить их сумму, иначе вычислить их произведение.
22. Даны действительные числа a, b, c. Удвоить эти числа, если $a \ge b \ge c$, и заменить их абсолютными значениями, если это не так.
23. Даны числа a, b, c, d, не равные 0. Составить алгоритм вычитания и деления двух простых дробей.
24. Если сумма трех попарно различных действительных чисел a, b, c меньше единицы, то наименьшее из этих чисел заменить

полусуммой двух других, в противном случае заменить меньшее из a и b удвоенным произведением двух других значений.

25. Даны действительные числа a и b. Если a и b – отрицательные числа, то каждое значение заменить его модулем. Если отрицательное только одно, оба значения уменьшить в 2 раза. Если оба значения больше 0 и не одно из них не принадлежит интервалу значений [0.25–1.25], то оба значения увеличить на 0.75, в остальных случаях a и b оставить без изменения.
26. Даны целые числа M и N и действительные числа R, S, T. При $M < N^2$, $M = N^2$, $M > N^2$ заменить модулем соответственно значения R, S, T, а два других значения уменьшить в 2 раза.

Рекурсивные алгоритмы

> *Рекурсия – это способ организации процесса вычисления, когда алгоритм (функция или процедура программы) обращается сам к себе.*

Рекурсия в математике является значимым средством, так как принцип рекурсии позволяет решить сложную задачу путем последовательного решения более простых подзадач. Некоторые задачи поддаются рекурсивному решению вполне естественным способом, например, вычислением факториала, определением натуральных чисел и некоторых функций.

Как правило, рекурсия необходима в тех случаях, когда требуется перебрать слишком много вариантов. Рекурсию принято считать одной из разновидностей циклического алгоритма. Рекурсивная форма организации позволяет придать алгоритму более компактный вид. Таким образом решается проблема от сложного к простого – содержание рекурсивного алгоритма отражает более сложный объект через более простой объект такого же типа. Обычно рекурсивный алгоритм содержит следующие основные части:

- условие для завершения цикла;
- тело рекурсии, которое содержит действия – операторы, предназначенные для выполнения на каждой итерации;
- шаг рекурсии, на котором рекурсивный алгоритм вызывает сам себя.

Различают прямую и косвенную рекурсию. В первом случае алгоритм содержит функцию, которая сама себя вызывает. Если функция вызывает другую функцию, которая в свою очередь вызывает первую, то такая функция называется косвенно рекурсивной.

Основное требование к рекурсивным алгоритмам – процесс обращения не должен быть бесконечным. Другими словами, должна быть реализована проверка завершения вызова, или в рекурсивном определении должно присутствовать ограничение или граничное условие, при котором дальнейшая инициализация рекурсии прекращается. Достаточно часто в рекурсивных алгоритмах используют некоторую управляющую переменную, при достижении определенного значения которой алгоритм прекращает свою работу.

Обращение к рекурсивному алгоритму, реализованному в виде функции или процедуры, ничем не отличается от обычной подпрограммы. При каждом новом рекурсивном обращении к памяти создается новая копия подпрограммы со всеми локальными переменными, увеличение которых будет до тех пор, пока не действует ограничение. Максимально возможное количество копий рекурсивной подпрограммы, которое может находиться в памяти компьютера, называется глубиной рекурсии.

Примером классической рекурсивной функции является возведение некоторого действительного числа X в степень N или X^N (рис. 2.12) :



```
def stepn (n, x):
    if n==0:
        result = 1
    else:
        result = x * stepn(n-1, x)
    return result

n = int(input("Введите степень: "))
x = int(input("Введите число: "))
print("{0} в степени {1} = ".format(x, n), stepn(n, x))
```

```
Введите степень: 3
Введите число: 4
4 в степени 3 =  64
```

Рис. 2.12. Пример вычисления числа в степени

Если n = 0, то любое число в степени 0 равно 1, иначе число в степени n ≠ 0 равно предыдущему значению рекурсивной функции stepn (n – 1, x), умноженной на величину x. Обращение к функции stepn для вычисления имеет вид

y = stepn (n, y).

В левой части должна быть переменная, которой присваивается результат вычисления. В правой части содержится идентификатор функции, в скобках – фактические параметры, которые по типу должны совпадать с формальными параметрами – степень (если это не целое, то, соответственно, в stepn следует изменить тип) и само число, которое возводится в степень.

Другим наиболее распространенным примером рекурсивной функции является вычисление суммы элементов массива (рис. 2.13):



```
mas = [1, 3, 5, 7, 10]
k = int(len(mas))
k -= 1

def xsum(k, x):
   if x == []:
       result = 0
   elif k >= 1:
       result = x[k] + xsum(k - 1, x)
   else:
       result = x[k]
   return result

print("Сумма элементов массива [1, 3, 5, 7, 10] = ",
xsum(k, mas))
```

```
Сумма элементов массива [1, 3, 5, 7, 10] =  26
```

Рис. 2.13. Пример вычисления суммы элементов массива

Если количество элементов равно 0, то сумма элементов равна 0. В остальных случаях, если количество не равно 0, сумма равна предыдущему значению плюс последний (следующий по порядку) элемент.

___Пример 2.7. Сумма факториалов

Вычислить сумму факториалов S = 1! + 2! + 3! + … + N!.

Для вычисления факториала используем соотношение, рассмотренное выше:

$$n! = \begin{cases} 1, \text{ если } n = 0, \\ n \times (n-1)!, \text{ если } n \geq 1. \end{cases}$$

Известно, (n – 1)! = 1 × 2 × 3 ×… × (n – 1), где n – целое положительное число. Каждое новое значение факториала зависит от предыдущего. Рекурсивная функция примет следующий вид:



```
def fact (k):
    if k == 0:
        result = 1
    else:
        result = k * fact(k-1)
    return result
```

Для вычисления суммы факториала в качестве подпрограммы, вычисляющей факториал для разных значений N, используем эту рекурсивную функцию. Для вычисления суммы факториалов выполним типичный прием для вычисления подобных алгоритмов.

Схема алгоритма, которая изображена на рис. 2.14, реализует цикл с параметром k. Для организации накопления значений

переменной суммы SF до начала работы цикла зададим начальное значение, равное 0.

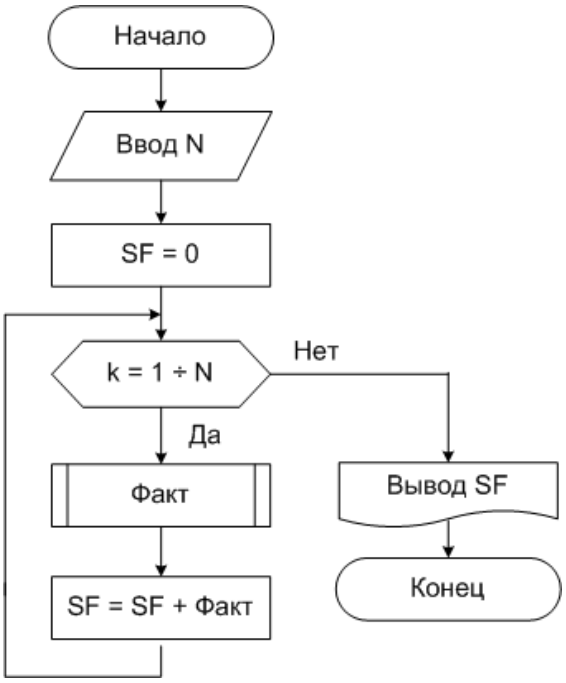


Рис. 2.14. Вычисление суммы факториалов

Все данные будут иметь целый тип. Простой циклический алгоритм можно реализовать следующим образом (рис. 2.15) :

```
def fact(k):
    if k == 0:
        result = 1
    else:
        result = k * fact(k - 1)
    return result

n = int(input("Введите число: "))
print("{0}! = ".format(n), fact(n))
```

```
Введите число: 5
5! =  120
```

Рис. 2.15. Пример вычисления факториала числа

Если и для суммы также использовать рекурсивную функцию, рассмотренную выше, то программный код для вычисления суммы факториала примет вид (рис. 2.16)



```
def fact (k):
    if k == 0:
        result = 1
    else:
        result = k * fact(k-1)
    return result
n = int(input("Введите число: "))
sf = 0
for i in range(1, n+1):
    sf = sf + fact(i)

print("Сумма факториалов = ", sf)
```

```
Введите число: 4
Сумма факториалов =  33
```

Рис. 2.16. Пример вычисления суммы факториалов

Примеры циклических алгоритмов

В данном разделе реализованы алгоритмы, в которых циклический процесс исполняется либо с помощью цикла с параметром, либо в виде итерационного цикла. В итерационных циклах используется понятие погрешность, которая необходима для проверки модуля разности между соседними итерациями.

Многие сложные вычислительные задачи можно свести к вычислению суммы чисел в числовом массиве или определению

количества чисел. Как было сказано выше, задача вычисления суммы последовательности для циклических алгоритмов является основной.

При работе с массивами удобно использовать массивы переменной размерности. При объявлении массивов с точно заданными границами следует выбирать разумные значения, а используемые границы изменения размеров массива вводить. В циклах программы в этом случае конечное значение параметра цикла может быть задано в виде переменной.

При удалении элементов из массива следует перемещать следующие за ним элементы (сдвигать) влево, при этом следует помнить, что в новом массиве может измениться количество элементов.

Рассмотрим наиболее типичные варианты решения таких задач.

___ Пример 2.8. Значение константы e

Многие из математических величин или значений функции могут быть выражены как суммы бесконечных последовательностей. Вычислить значение константы e[5], *заданной в виде степенного ряда:*

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

Обозначим сумму ряда как некоторое число Se. Так как количество повторений цикла определить невозможно, то следует организовать так называемый итерационный цикл с постусловием. При вычислении суммы бесконечного ряда будем учитывать только те слагаемые – члены ряда, значение которых больше по абсолютной величине некоторого заданного положительного числа Eps, где Eps – задаваемая величина погрешности.

Из математики известно, что суммой ряда называется предел, к которому стремится последовательность частичных сумм данного ряда. Ряд называется *сходящимся*, если такой предел существует. Иначе ряд называется *расходящимся*. Для получения требуемой точности результата следует накапливать сумму элементов ряда до тех пор, как следующий член ряда станет меньше заданной погрешности.

[5] Число e = 2,7182818284… – иррациональное число – основание натуральной логарифмической функции ln. Это число называют также *числом Эйлера*, неперовым числом.

В знаменателе каждой дроби имеем факториал, который выше уже был нами рассмотрен. Если использовать предыдущий результат вычислений, то чтобы получить новое значение, достаточно разделить на n, где n = 1, 2, 3, …. – задает номер шага.

Соответственно величина членов ряда определяется как следующая последовательность $X = 1, 1, \frac{1}{2}, \frac{1}{6}, \ldots$. При увеличении членов последовательности вычисляемая сумма приближается к значению e. Можно уменьшить значение погрешности Eps и сравнить ее со значением n-го члена ряда. Если значение X меньше, чем некоторое заданное Eps, то вычисляемая сумма удовлетворяет требованиям, и вычислительный процесс можно закончить.

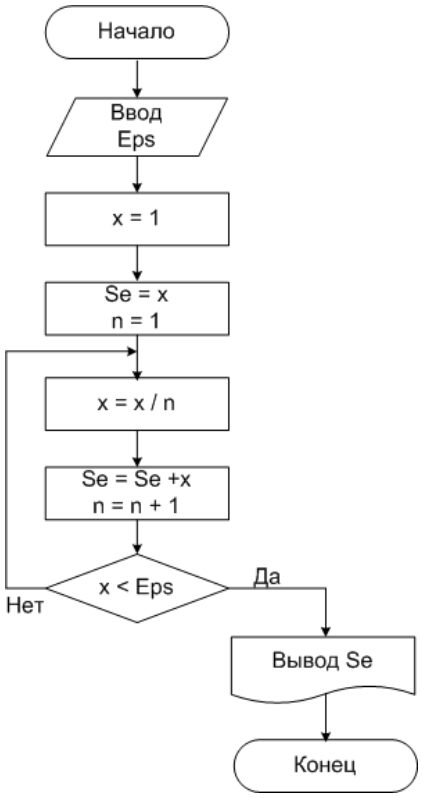


Рис. 2.17. Алгоритм вычисления величины e = 2.71828…

Типичный итерационный алгоритм (рис. 2.17) организован с циклической структурой, в которой использован цикл с постусловием. Так как X в данном примере – всегда положительное число, то в условии опущено определение X как числа по абсолютной величине:



```
import math
eps = float(input("Введите eps: "))
n = 1
x = 1
se = 1
while True:
    x /= n
    se += x
    n += 1
    if not ((abs(math.exp(1)) - se) > eps):
        break
print("eps = {0}, n = {1}, e = {2}".format(eps, n, se))
```

На рис. 2.18 показан результат работы программы. Для такого простого алгоритма нет смысла использовать рассмотренные выше рекурсивные функции для вычисления факториала и суммы, так как это может только усложнить программу.

```
Введите eps: 0.000000001
eps = 1e-09, n = 13, e = 2.7182818282861687
```

Рис. 2.18. Результат вычисления числа e

В результате выполнения программы чем меньше Eps, тем больше количество шагов и тем точнее получается результат – все большее количество совпадающих знаков в записи числа.

___Пример 2.9. Сумма ряда

Найти сумму элементов последовательности

$$S = \frac{2}{a_1} + \frac{4}{a_2 - 1} + \frac{6}{a_3 - 2} + \cdots$$

При построении схемы алгоритма (рис. 2.19) необходимо:

- составить формулу для вычисления каждого отдельного члена последовательности;
- определить область допустимых значений.

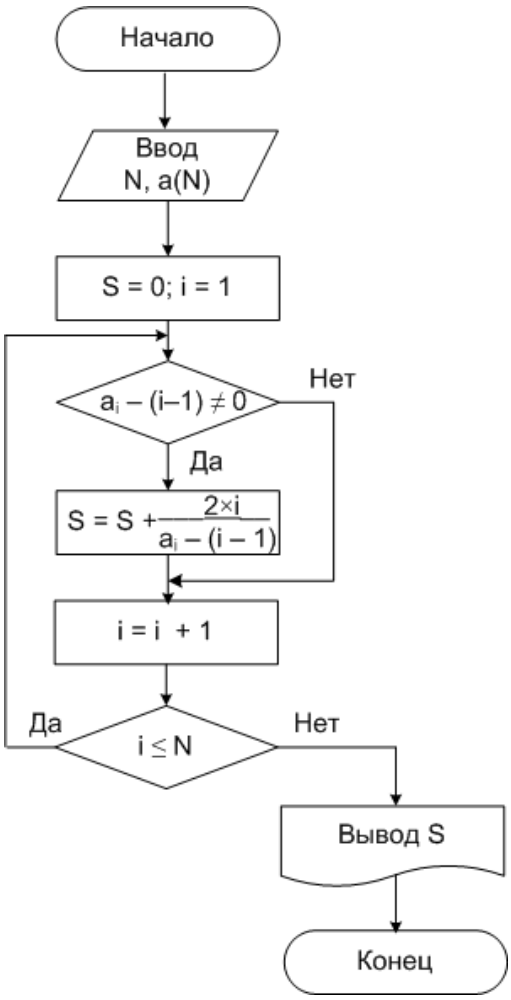


Рис. 2.19. Вычисление суммы последовательности

Последовательность имеет вид суммы некоторых выражений в виде дроби, знаменатель которой имеет вид, равный $a_i - (i + 1)$. Если для исходных данных справедливо равенство $a_i = (i + 1)$, то этот член последовательности вычислить нельзя (в знаменателе будет 0). Необходимо решить, как построить алгоритм: например, эти данные «обойти», т.е. просто пропустить без вывода сообщения о невозможности вычислений (разветвление с неполной альтернативой на рис. 2.20).

Цикл с постусловием в схеме алгоритма может быть реализован в программе как цикл с параметром. Реализованный алгоритм имеет вид



```
    s = float(0)
    a = []
    n = int(input("Введите длину последовательности: "))
    for i in range(n):
      a.append(int(input("Введите {0} элемент
последовательности ".format(i))))
    for i in range(n):
      if (a[i] - (i-1)) != 0:
        s += (2*i)/(a[i] - (i+1))
    print("Сумма последовательности = ", s)
```

```
Введите длину последовательности: 3
Введите 0 элемент последовательности 142
Введите 1 элемент последовательности 343
Введите 2 элемент последовательности 1232
Сумма последовательности =  0.009119781239784389
```

Рис. 2.20. Пример вычисления суммы элементов последовательности

___ Пример 2.10. Вычисление *sin x*

Вычислить функцию sin x, используя известный степенной ряд:

$$S = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Для вычисления этой суммы можно воспользоваться результатами примеров 2.8 и 2.9. Во-первых, необходимо оптимально описать формулу для вычисления члена последовательности с тем, что каждый шаг n должен максимально использовать результат предыдущих шагов. Поэтому в общей форме для члена последовательности не будем учитывать знак «+» или «–». Этот знак будем учитывать в формуле вычисления суммы ряда S (рис. 2.21).

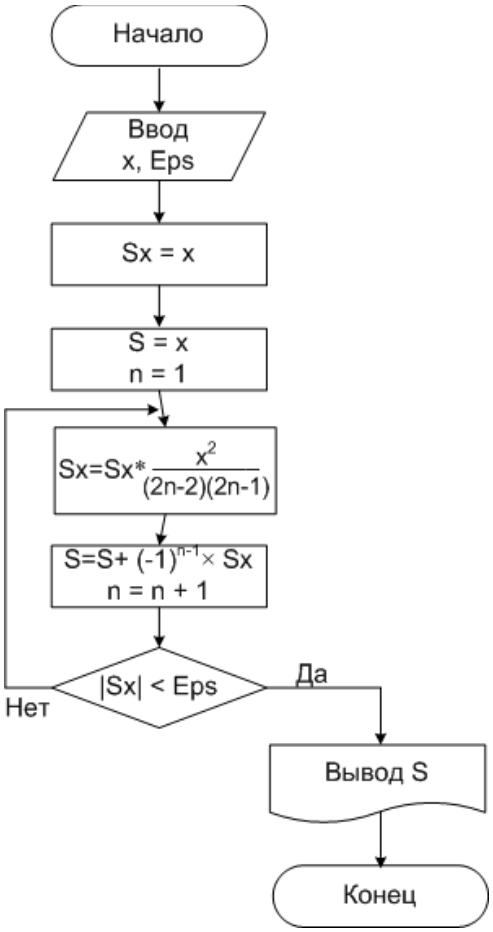


Рис. 2.21. Схема алгоритма вычисления sinX

Алгоритм реализуем в виде следующей программы (рис. 2.22):

```
x = float(input("Введите X в радианах: "))
eps = 10**(-4)
sx, s = x, x
n = 2
while True:
   sx = (sx*(x**2))/((2*n - 2)*(2*n - 1))
   s += sx*(-1)**(n-1)
   n += 1
   if abs(sx) < eps:
       break
print("x = ", round(x, 6), "sinX = ", round(s, 6))
```

```
Введите X в радианах: 3.141593
x =  3.141593 sinX =  -1e-06
```

Рис. 2.22. Результат вычисления sinX

В качестве контрольного примера используем данные, полученные в таблицах Excel для соответствующих значений Pi/n и sin(Pi/n). Результат реализации программы даже при Eps = 10^{-4} почти совпадает с приведенными ниже данными для sinx:

n	*Pi/n*	sin *x*
1	3,141593	0,000000
2	1,570796	1,000000
3	1,047198	0,866025
4	0,785398	0,707107
5	0,628319	0,587785
6	0,523599	0,500000

Чтобы увеличить точность результата, необходимо в программном коде уменьшить значение Eps.

___Пример 2.11. Сумма элементов матрицы

Вычислить сумму элементов двумерного массива.

Так как нет дополнительных условий, обозначим массив – матрицу как A(N, M), где N – количество строк, которые необходимо задать (рис. 2.23).

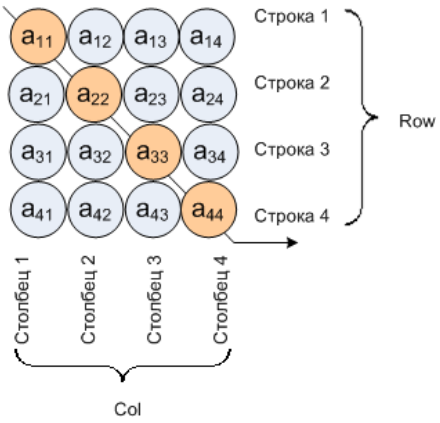


Рис. 2.23. Сумма элементов матрицы

В программе номер строки обозначен Row, M определяет количество столбцов, в программе и схеме алгоритма это – Col. Схема алгоритма (рис. 2.24) содержит два цикла, причем один вложен в другой. Верхний цикл по строкам – все номера строк изменяются строго по порядку 1, 2, …, N, внутренний цикл – аналогично по столбцам.

Когда номер строки равен 1, номер столбца изменяется 1, 2, …, N. Номер строки увеличивается и становится равным 2, номера столбцов изменяются снова: 1, 2, …, N. Процесс повторяется пока внешний цикл не «переберет» значения всех строк. Каждый раз во внутреннем цикле сумма увеличивается на величину A(Row, Col).

Алгоритм реализован в виде следующей программы (рис. 2.25):

```
n = int(input("Введите число строк: "))
m = int(input("Введите число столбцов: "))
mas = [0] * n
for i in range(n):
   mas[i] = [0] * m

for i in mas:
   print(i)
```

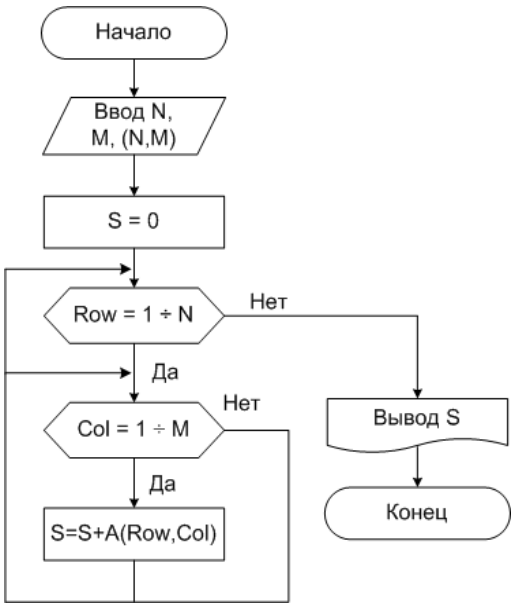


Рис. 2.24. Вычисление суммы элементов двумерного массива

```
Введите число строк: 5
Введите число столбцов: 6
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
```

Рис. 2.25. Результат создания двумерного массива



```
import random
n = int(input("Введите число строк: "))
m = int(input("Введите число столбцов: "))
s = 0
a = []

for i in range(n):
   a.append([])
   for j in range(m):
       a[i].append(random.randint(0,10))
print("Массив чисел: ")
for i in a:
   print(i)
for row in a:
   for col in row:
       s += col
print("Сумма элементов двумерного массива: ", s)
```

Результаты вычисления суммы элементов двумерного массива показаны на рис. 2.26.

```
Введите число строк: 4
Введите число столбцов: 5
Массив чисел:
[6, 1, 3, 10, 2]
[2, 6, 4, 4, 7]
[9, 5, 1, 10, 6]
[2, 4, 1, 4, 3]
Сумма элементов двумерного массива:  90
```

Рис. 2.26. Результат вычисления суммы элементов двумерного массива

___Пример 2.12. Объединение двух массивов

Составить алгоритм для объединения двух массивов в один чередованием элементов массива (рис. 2.27).

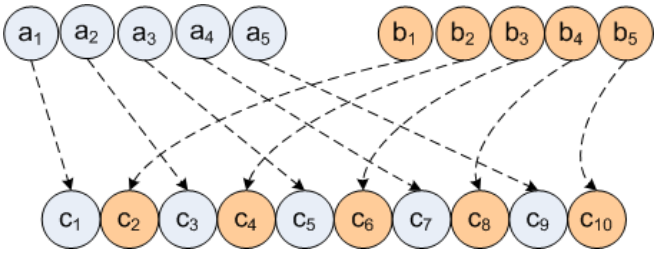


Рис. 2.27. Объединение двух массивов

Так как нет дополнительных условий, примем, что массивы имеют равное количество компонент.

Допустим, что имеется два массива длины N: массив a, в котором N элементов, и массив b, который также содержит N элементов. В этом случае массив, который образуется слиянием двух массивов длины N, будет иметь длину 2 × N.

Если массивы будут иметь разные длины, то общая длина нового массива будет определяться суммой количества компонент в каждом массиве: Na + Nb. Соединим массивы в один следующим образом:

- нечетные номера нового массива c – это компоненты из первого массива a;
- четные – элементы второго массива b.

Схема алгоритма (рис. 2.28) имеет циклическую структуру с постусловием. При реализации алгоритма в виде программы данный цикл можно представить в виде цикла с параметром.

Так как длина исходных массивов одинакова, то цикл можно организовать как для длины N, так и для 2 × N. В данном случае цикл для N, а индекс j для индексной переменной c_j должен увеличиваться на величину 2 на каждом шаге цикла. Тип числовых массивов не имеет значение, единственное, что надо учесть, это то, что все данные должны быть одного типа.

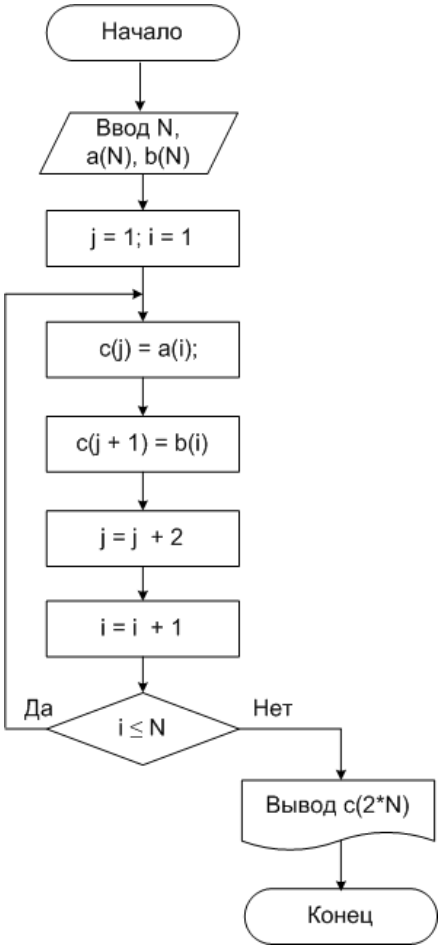


Рис. 2.28. Слияние двух линейных массивов чередованием элементов

Программа имеет следующий вид (рис. 2.29) :

```
import random
n = int(input("Введите длину массива: "))
a = []
b = []
```

```
c = []
for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел a: ", a)
for i in range(n):
    b.append(random.randint(0,10))
print("Массив чисел b: ", b)
for i in range(n):
    c.append(a[i])
    c.append(b[i])
print("Результат слияния элементов массива: ", c)
```

```
Введите длину массива: 5
Массив чисел a:  [5, 10, 5, 4, 1]
Массив чисел b:  [0, 8, 10, 6, 4]
Результат слияния элементов массива:  [5, 0, 10, 8, 5, 10, 4, 6, 1, 4]
```

Рис. 2.29. Результат слияния массивов

Для N = 5 величины в соответствии с реализацией алгоритма будут иметь значения:

N	a_i	b_i	c_j
1	5	0	5
2	10	8	0
3	5	10	10
4	4	6	8
5	1	4	5
6			10
7			4
8			6
9			1
10			4

Результатом работы алгоритма является массив C, который состоит из 10 компонент. Как видно из таблицы, в новом массиве чередуются элементы из двух исходных массивов.

____ Пример 2.13. Расположение элементов массива в обратном порядке

Расположить элементы заданного массива в обратном порядке.

Так как не имеем другой информации, назовем исходный массив как массив A(N), в котором N элементов (рис. 2.30).

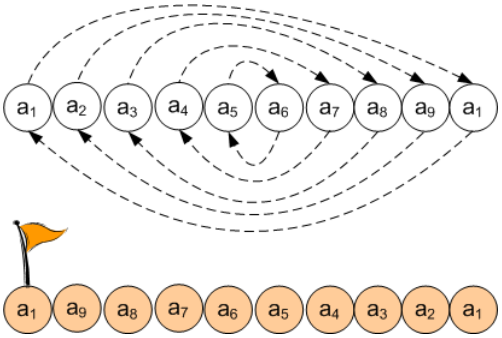


Рис. 2.30. Расположение элементов в обратном порядке

При составлении алгоритма можно использовать тот же исходный массив A и выполнять на каждом шаге «переприсвоение» элементов массива из верхней части элементам нижней части, и наоборот, т.е. осуществить некоторый двухсторонний обмен (рис. 2.31).

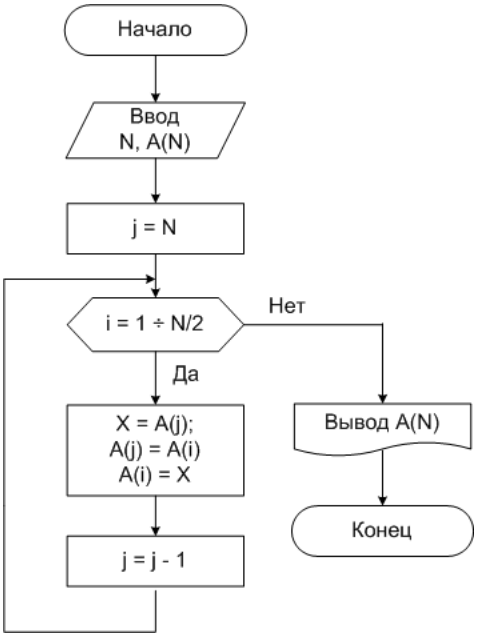


Рис. 2.31. Расположение заданного массива в обратном порядке

Фрагмент программы для реализации алгоритма на рис. 2.31 приведен ниже (рис. 2.32).



```
import random

n = int(input("Введите размерность массива: "))
j = n-1
a = []
x = 0
half = int(n/2)

for i in range(n):
   a.append(random.randint(0,100))
print("Массив чисел a: ", a)

for i in range(0, half):
   x = a[j]
   a[j] = a[i]
   a[i] = x
   j -= 1
print("Массив чисел a в обратном порядке: ", a)
```

```
Введите размерность массива: 5
Массив чисел a:  [97, 38, 13, 56, 58]
Массив чисел a в обратном порядке:  [58, 56, 13, 38, 97]
```

Рис. 2.32. Результат расположения массива в обратном порядке

Допустим, массив содержит четное число элементов, например, N = 10. Количество повторений для обмена равно 5. Некоторой переменной x присваиваем значение a(10), a(10) = a(1), и затем a(1) = x (=a(10)). Далее те же действия повторяем для a(9): x = a(9), a(9) = a(2), a(2) = x (= a(9)). Продолжаем выполнять эти действия до N/2 = 5. Если количество элементов в массиве нечетное, то средний элемент массива должен остаться на месте.



Вопросы для самопроверки

Составить алгоритмы для решения следующих задач:

1. Дано натуральное число n > 1. Получить наибольшее целое k, при котором $4^k < n$.
2. Дано натуральное число n. Можно ли представить n! в виде произведения трех последовательных целых чисел.
3. Дано некоторое натуральное число n (n > 0). Если оно четно, то разделить его на 2, иначе умножить на 3 и прибавить 1. Если полученное число не равно 1, то повторить все до тех пор, пока не получится 1. Для каких n это возможно?
4. Дано натуральное число m. Вычислить сумму:
   $1 \times 2 + 2 \times 3 \times 4 + \ldots + m \times (m + 1) \times \ldots \times 2m$.
5. Вычислить значения функции $y = 4x^3 - 2x^2 + 5$ для значений x, которые изменяются в интервале от –2,5 до 1,5, с шагом 0,25.
6. Последовательность чисел Фибоначчи $R_0, R_1, \ldots, R_n$ образуется по закону $R_0 = 0$; $R_1 = 1$; $R_i = R_{i-1} + R_{i-2}$, где i = 2, 3, 4,… Для натурального числа n > 1 получить последовательность $R_0, R_1, \ldots, R_n$.
7. Даны натуральное число n, действительные числа $a_0, a_1, \ldots, a_n$. Вычислить последовательности:
   а) $a_1 + 1!, \quad a_2 + 2!, \ldots, a_n + n!$;
   б) $a_1, -a_1 a_2, a_1 a_2 a_3, \ldots, (-1)^{n+1} \quad a_1 a_2 \ldots a_n$.
8. Определить, является ли заданное число M четным.
9. Даны целые числа a_1, a_2, a_3. Получить матрицу, элементы которой целые числа B_{ij} , где i, j = 1, 2, 3 и $B_{ij} = a_i - 3a_j$.
10. Дана матрица M (5, 6). Найти среднее арифметическое каждого из столбцов, а элементы второй строки уменьшить вдвое.
11. Дана матрица B (m, n). Найти среднее арифметическое каждого из столбцов и уменьшить каждый элемент в каждом столбце матрицы на соответствующую величину.
12. Дана матрица B (m, n), элементы которой имеют значения в интервале 1 до 10. Найти среднее геометрическое каждой строки. Уменьшить каждый элемент в каждой строке матрицы на соответствующую величину.

13. Дана матрица действительных чисел размера n × m. Найти суммы элементов матрицы, расположенных параллельно главной диагонали.
14. Дана матрица действительных чисел размера n × m. Найти среднее арифметическое элементов матрицы, расположенных на главной диагонали. Увеличить каждый элемент матрицы на соответствующую величину.
15. Для массива M(k) выполнить сдвиг элементов на d позиций влево и получить новый массив.
16. Дано натуральное число n. Найти сумму $n^2 + (n + 1)^2 + \ldots + (2n)^2$.
17. Одноклеточная амеба каждые 3 ч делится на 2 клетки. Определить, сколько клеток будет через 3, 6, 9, …, 24 ч, если первоначально была одна амеба.
18. Взаимное влияние некоторых двух конкурирующих видов на размер x_n, y_n их популяций в n-м году можно описать в виде системы

$$\begin{cases} x_{n+1} = 2\,x_n - y_n, \\ y_{n+1} = -x_n + 2\,y_n. \end{cases}$$

Составить алгоритм решения задачи.
19. Пусть $x_0 = a$, $y_0 = b$ ($a \neq b$), где a и b – данные числа. Найти численности обоих видов за все годы, которые предшествуют полному вымиранию одного из них.
20. В процессе лечебного голодания вес пациента за 30 дней снизился с 105 до 65 кг. Было установлено, что ежедневные потери веса пропорциональны весу тела. Вычислить, чему был равен вес пациента через k дней после начала голодания для k = 1, 2, 3, …, 29.
21. В ведомости указана зарплата, выплачиваемая каждому сотруднику некоторой фирмы за один месяц. Определить общую сумму денег, которые выплачены по ведомости.
22. Известно сопротивление каждого элемента в электрической цепи. Определить общее сопротивление цепи, если соединение элементов:
    а) последовательное,
    б) параллельное.
23. Счет в банке открыт 15 июня, первоначальная сумма вклада составила 1500 руб. Известно, что через каждый месяц размер вклада увеличивается на k % от имеющейся суммы. Требуется определить через сколько месяцев:

а) величина ежемесячного увеличения вклада превысит 100 руб.,
б) размер вклада превысит 3000 руб.

24. Известен годовой процент по вкладу и первоначальная сумма вклада. Установить, через сколько лет первоначальный вклад станет больше примерно вдвое.
25. Дано натуральное число *v*. Найти различные прямоугольные параллелепипеды, объем которых равен *v*, а ребра выражены натуральными числами. *Параллелепипеды, которые получены один из другого, если поменять ребро местами, считаются одинаковыми*.
26. Найти все натуральные решения уравнения $i^3 + j^3 + k^3 = z^3$ в интервале [1; 20]. Решения, которые должны получиться, если поменять местами i, j, k , следует отбросить.
27. Вычислить значения функции $z = \sin(x_i\, y_i) / (x_i - y_i)$, если y_i заданы массивом ($y_1, y_2, \ldots, y_{30}$), x изменяется от 0 с шагом 0.25.
28. Вычислить значения функции $w = (ax_i + by_i + cz_i)/3$, если значения x изменять от 0 с шагом 0.1; y изменять от y_0 с шагом 0.2; z_i заданы массивом ($z_1, z_2, \ldots, z_{11}$).

Глава 3

Алгоритмы поиска данных



Освободите вопрос от всех излишних представлений и сведите его к простейшим элементам…
Р. Декарт

Поиск – одно из наиболее часто встречающихся в программировании действий. Алгоритмы информационного поиска образуют самостоятельный раздел алгоритмов, которые допускают разнообразные решения и позволяют испытывать различные структуры данных. Для разработки подобных алгоритмов можно использовать методы пошаговой детализации.

Задача поиска заключается в отыскании в заданной последовательности элемента или нескольких элементов с заданными свойствами. Существует несколько основных вариантов организации поиска данных:

A) поиск номера элемента последовательности с заданным значением;

B) поиск максимального или минимального элемента и его номера в заданной последовательности.

Задачу A можно назвать основной задачей, а задачу B – производной от A.

Последовательность для поиска может быть как упорядоченная, так и неупорядоченная.

Для начала разработки алгоритма примем следующее допущение: пусть задана определенная последовательность данных $\{Xi\} = X_1, X_2, \dots, X_N$, где i изменяется от 1 до N, длина N которой является фиксированной величиной. Все элементы имеют разные значения. Необходимо найти некоторый заданный элемент.

Алгоритм поиска имеет аргумент P и заключается в нахождении записи, для которой P служит некоторым признаком поиска или ключом P. Результатом поиска может быть одно из двух: либо поиск

завершился успешно и запись, которая содержит P, найдена, либо поиск оказался неудачным, и запись не найдена.

Поиск номера элемента последовательности с заданным значением

По условию поиска для некоторой исходной последовательности необходимо определить номер элемента, значение которого равно значению заданной переменной P.

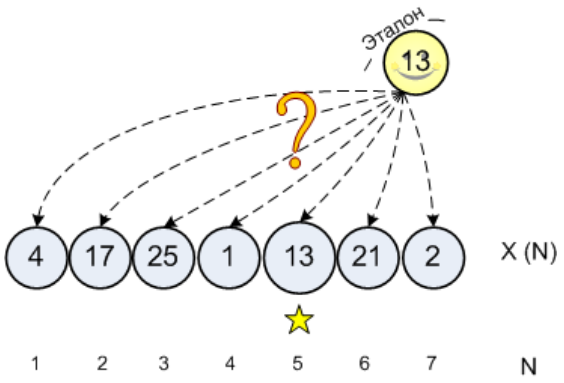


Рис. 3.1. Поиск номера элемента

Другой информацией об элементе, который следует найти, мы не располагаем. Поэтому очевиден простой линейный просмотр или последовательный перебор элементов массива. В этом случае поиск повторяется до тех пор, пока не будет найден нужный элемент последовательности, равный некоторому *эталону* P.

Особенностями такого поиска являются:

– независимость поиска;

– цикличность поиска – количество повторений не превышает количества элементов в заданной последовательности;

– простота реализации алгоритма.

Условиями окончания процесса поиска являются:

– элемент найден, т.е. P = XI;

– элемент не найден, хотя массив просмотрен от начала до конца, но при этом совпадений не было.

Если элемент будет найден, то индекс найденного элемента будет минимально возможным. Но в заданной последовательности может не оказаться элемента со значением P.

Если предполагается, что в исходной последовательности может быть только один элемент, равный эталону, то после того как этот элемент найден, поиск можно закончить. Если таких элементов, равных эталону P, имеется несколько, то можно задать дополнительное условие: какой попорядку найденных необходим элемент. В последнем случае задается дополнительная переменная, которая будет подсчитывать количество этих самых равных P элементов.

Иначе список просматривается до конца, и последний, из равных P, элемент мы будем считать результатом поиска.

Основной алгоритм поиска примет вид:

1) сравнить очередной элемент с переменной P;
2) перейти к следующему элементу;
3) если не все элементы просмотрены, то повторить начиная с п. 1.

Алгоритм должен однозначно реагировать на ситуацию, если искомого значения в последовательности нет. Для этой цели введем в алгоритм переменную k, начальное значение которой равно 0. Результат k > 0 – номер найденного элемента. Если элемент со значением P не найден, то результат k остается нулевым. Алгоритм (рис. 3.2) реализован в виде следующей программы (рис. 3.3):



```
import random
n = int(input("Введите размерность массива: "))
a = []
count = 0
for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел a: ", a)
p = int(input("Введите искомое значение: "))
for i in a:
    if i == p:
        count += 1
if count != 0:
    print("Количество найденных искомых элементов: ",
count)
else:
    print("Искомый элемент не найден")
```

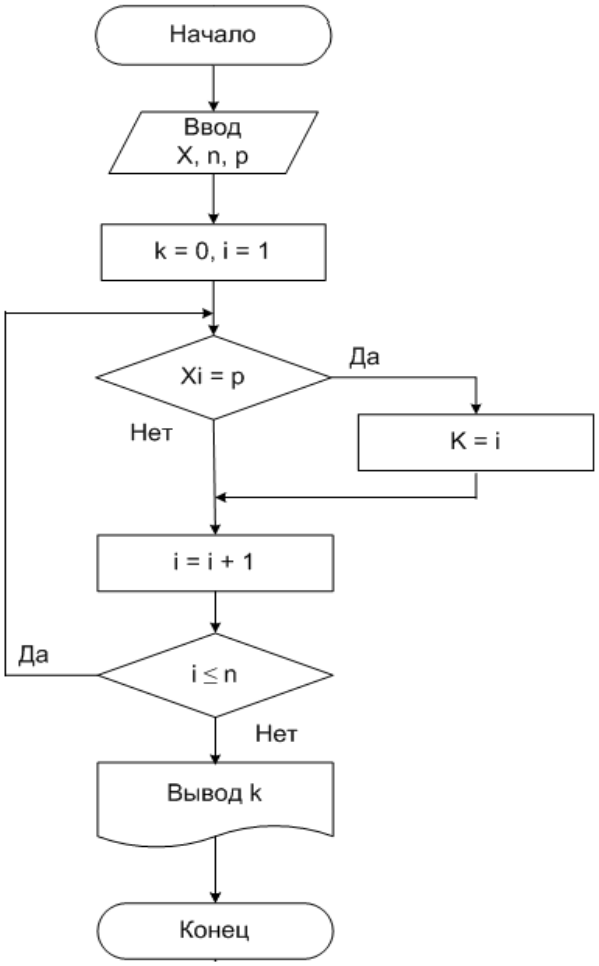


Рис. 3.2. Схема алгоритма поиска заданного элемента

```
Введите размерность массива: 6
Массив чисел a:  [4, 5, 0, 0, 10, 1]
Введите искомое значение: 0
Количество найденных искомых элементов:  2
```

Рис. 3.3. Результат поиска элемента в массиве

Реализация алгоритма достаточно проста: в цикле с параметром для изменения порядкового номера последовательности выполняется операция простого сравнения каждого элемента последовательности x(i) с заданным «эталоном» P. Эффективность поиска прямо пропорциональна количеству вычислений.

Алгоритм поиска в упорядоченной последовательности

Если последовательность упорядочена, т.е. значения элементов последовательности возрастают или убывают с увеличением порядковых номеров (индексов) элементов, то к ней есть возможность применить другой алгоритм поиска. Стандартным методом поиска в упорядоченном массиве является метод деления отрезка пополам.

Алгоритм, в котором последовательно уменьшается промежуток исследуемых данных в два раза, называется алгоритмом дихотомии – ***деление отрезка пополам****.*

В процессе такого поиска границы промежутка можно сдвинуть друг к другу. После каждого сравнения сдвигается либо верхняя, либо нижняя граница промежутка. Промежутком данных или отрезком является отрезок индексов от 1 до N.

Пусть значение наименьшего индекса в исходной последовательности равно A, а значение наибольшего индекса – B. Если разделить рассматриваемый интервал [A, B] пополам, то получится некоторое значение C. Сравниваем X[C] с заданным значением P. Если X[C] > P, то элемент следует искать для B = C – 1, если X[C] < P, то для A = C + 1. Следует повторять эти действия до тех пор, пока A и B не совпадут. Проверить условие X[A] = P, и по результатам проверки может быть сформирован результат.

Чтобы сбалансировать количество возможных вычислений в обоих случаях, переменную C выбирают так, что отрезки значений A – C и B – C должны быть близки.

Словесный алгоритм поиска имеет следующий вид:

1. Вычислить C.
2. Сравнить X[C] с переменной P.
3. Определить A и B.
4. Если A и B не совпадают, то повторить с п. 1.

Проверить совпадение выделенного значения с переменной P. Перед выполнением алгоритма необходимо задать начальные значения для A и B.

Схема алгоритма дихотомического поиска в упорядоченном массиве представлена на рис. 3.4. Эффективность данного метода по сравнению с методом перебора возрастает с ростом N (количества шагов) экспоненциально.

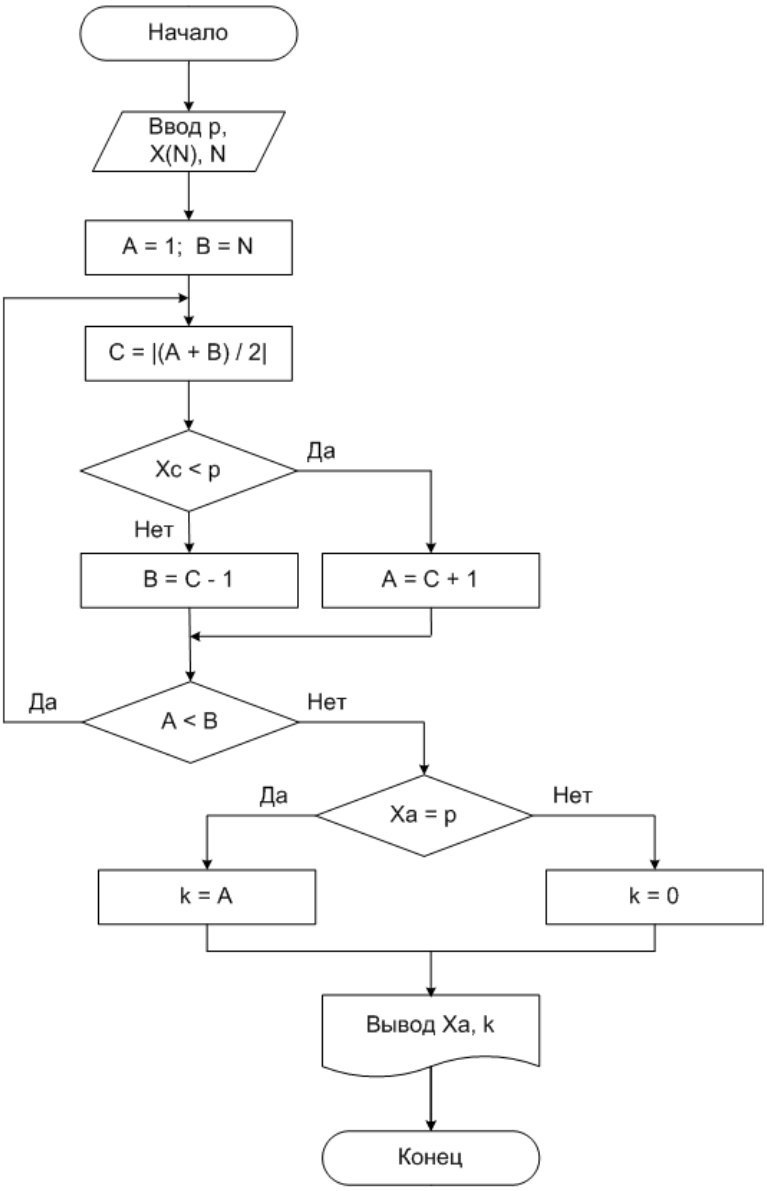


Рис. 3.4. Схема алгоритма дихотомического поиска

Поиск минимального и максимального элемента в заданной последовательности

Для поиска минимального элемента необходимо назначить некоторый *эталон* – переменную, которой заранее присваивается

значение, например, первого элемента. Поиск проводится путем последовательного сравнения всех элементов с эталоном min.

После того, как эталонная переменная определена, сравниваем с ней первый элемент. И если он меньше эталона, то изменить эталон. Для этого присвоить эталону значение первого элемента и продолжить сравнение теперь со вторым элементом. Таким образом, можно сравнить все элементы последовательности.

Все действия выполняются одинаково для всех элементов исходной последовательности:

1) сравнить эталон с очередным элементом последовательности;
2) перейти к следующему элементу;
3) если не все элементы последовательности просмотрены, то повторить с п. 1.

В результате анализа ситуации можно определить следующие обязательные действия для разработки алгоритма:

- задать номер элемента последовательности, с которого начинается сравнение;
- присвоить значение этого элемента переменной, которая отмечена как эталон;
- текущий номер элемента не должен превышать общую длину последовательности;
- переход к следующему элементу заключается в увеличении на единицу индекса.

Обозначим исследуемую последовательность X_1, X_2, ..., X_N. Начинаем поиск с первого элемента и его принимаем за эталонное значение min = X(1), переменной Nom присвоим начальное значение, равное 1, это соответствует номеру элемента, который назначен эталоном для минимального элемента. Для определения номера минимального элемента алгоритм будет иметь аналогичную структуру. Процесс отыскания максимума и его номера можно совместить в одном цикле.

Полученный алгоритм можно преобразовать в алгоритм определения номера максимального элемента. Для этого в алгоритме (рис. 3.5) максимальный элемент заданной последовательности будет иметь имя max.

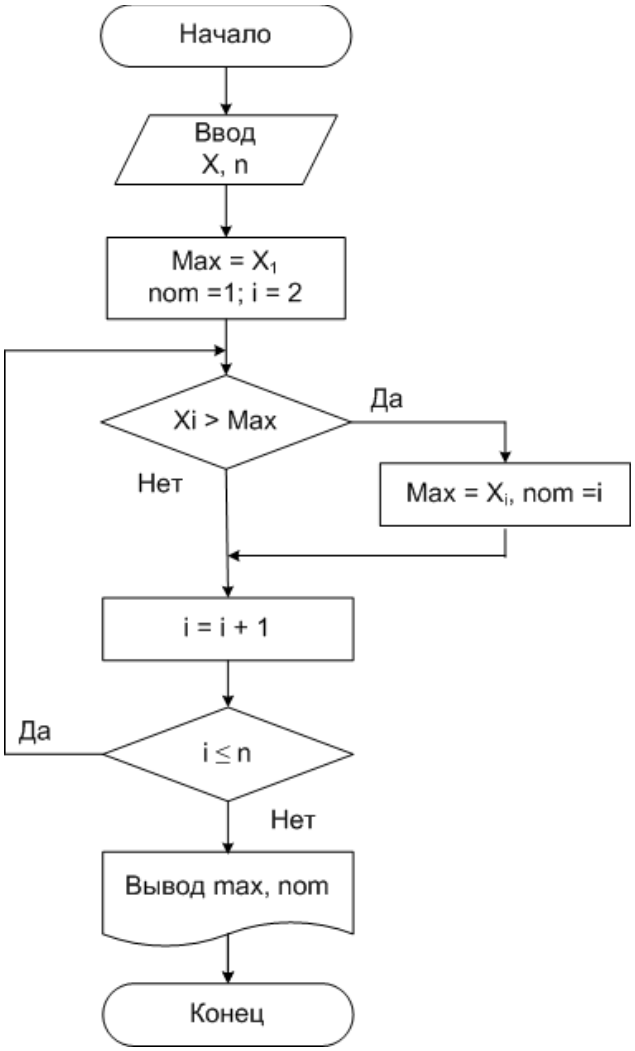


Рис. 3.5. Схема поиска номера максимального элемента

Если необходимо найти максимальный элемент и его номер в последовательности с совпадающими значениями (если есть несколько одинаковых элементов), то необходимо уточнить, какой – первый или последний из одинаковых элементов – считать результатом сравнения. Структура алгоритма остается неизменной. Отличие только в сравнении эталонной переменной с текущим элементом массива.

Приведенная ниже программа позволяет найти и минимальное, и максимальное значение в заданной последовательности (рис. 3.6):



```
import random
n = int(input("Введите размерность массива: "))
a = []
count = 0
for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел a: ", a)
xmax = a[0]
xmin = a[0]
for i in a:
    if i > xmax:
        xmax = i
    elif i < xmin:
        xmin = i
print("max = {0}, min = {1} ".format(xmax, xmin))
```

```
Введите размерность массива: 6
Массив чисел a:  [9, 1, 10, 9, 1, 7]
max = 10, min = 1
```

Рис. 3.6. Результат поиска максимального элемента в массиве

Если необходимо найти первый по порядку следования минимальный элемент и его номер, то предыдущий алгоритм остается практически без изменения.

Примеры алгоритмов – поиск данных

В этом разделе приводятся задачи, для которых реализуются поисковые алгоритмы. Большая часть задач использует поиск данных в некоторой заданной последовательности в связи с какой-то другой вычислительной задачей. Данные представляют собой массив или матрицу, поэтому все алгоритмы, представленные в данном разделе, можно отнести к рассмотренным алгоритмам с циклической структурой.

___Пример 3.1. Сумма положительных чисел

Вычислить сумму положительных чисел в таблице Tabl(5,5) (рис. 3.7).

Рассмотрим пример как типичную задачу – поиск, в данном случае следует найти числа, значения которых больше нуля, а вернее найти их сумму.

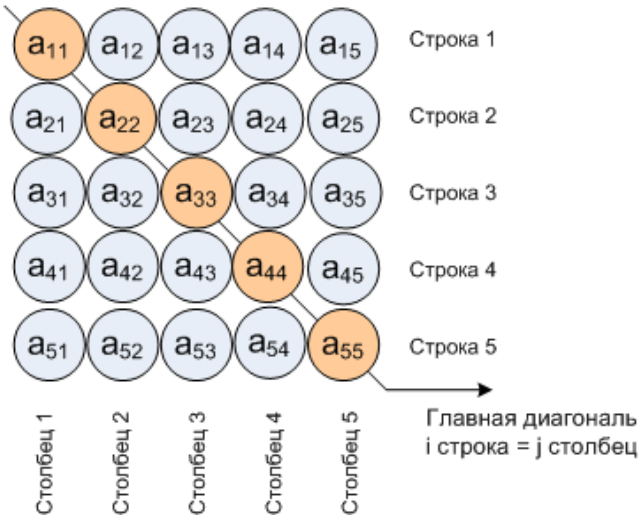


Рис. 3.7. Вычисление суммы положительных чисел

По условию задачи имеем таблицу, в которой 5 строк и 5 столбцов. Всего $5 \times 5 = 25$ элементов. Положительным является число, величина которого больше 0. Необходимо выполнить сравнение «Если Tabl(i, j) > 0» для 25 вводимых чисел. Иначе – сравнение для 5 строк по 5 столбцов. Имеем два вложенных цикла. Первым будет цикл по строкам – по i, вложенным по отношению к нему является цикл по столбцам – по j.

Величина суммы вычисляется «накоплением» от начального задания СумП = 0. Далее сумма увеличивается на величину, равную следующему по порядку элементу матрицы – всего 25 увеличений.

Словесное описание алгоритма вычисления суммы положительных элементов в таблице имеет вид:

1. СумП = 0.
2. Номер строки i = 1.
3. Номер столбца j = 1.

4. Ввести элемент таблицы Tabl(i, j).
5. Если Tabl(i, j) > 0, то СумП = СумП + Tabl(i, j).
   следующий элемент столбца j = j + 1.
6. Все элементы столбца просмотрены?
   Если j ≤ 5 , то перейти к пункту 4.
   следующий элемент строки i = i + 1.
7. Все элементы строки просмотрены.
   Если i ≤ 5, то перейти к пункту 3.
8. Вывести «СумП».
9. Конец.

Возможны случаи, когда циклические вычисления надо производить в зависимости от дополнительных условий. В этом случае алгоритм, кроме циклической структуры, содержит также разветвленную структуру с дополнительными блоками сравнения (рис. 3.8).

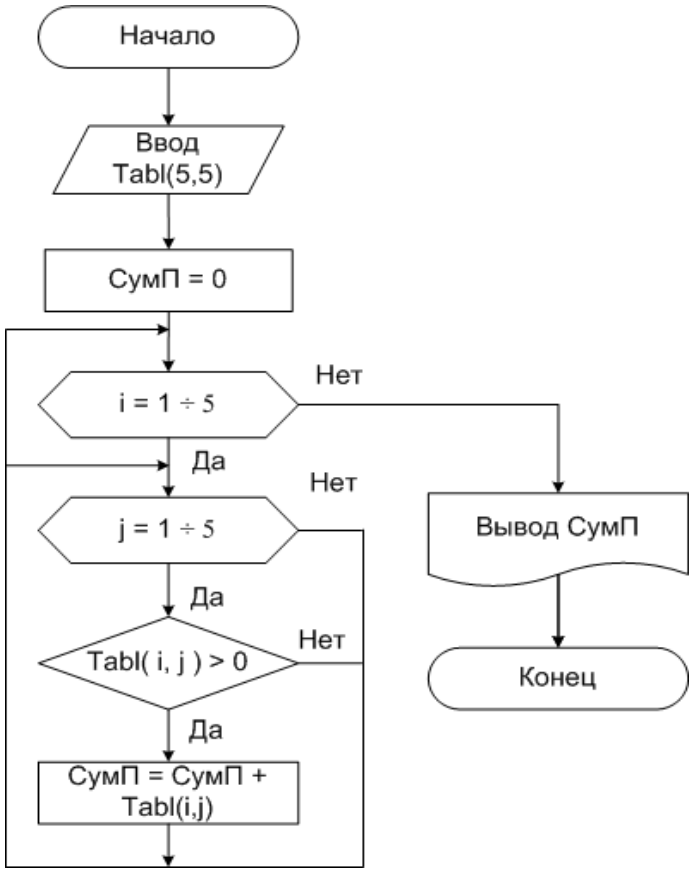


Рис. 3.8. Схема алгоритма вычисления суммы положительных элементов в двумерной таблице

Программа для нахождения суммы положительных элементов в таблице чисел (рис. 3.9):



```
import random
n = int(input("Введите число строк: "))
m = int(input("Введите число столбцов: "))
s = 0
a = []
for i in range(n):
    a.append([])
    for j in range(m):
        a[i].append(random.randint(-10,10))
print("Массив чисел: ", a)
for row in a:
    for col in row:
        if col > 0:
            s += col
print("Сумма положительных элементов двумерного массива: ", s)
```

```
Введите число строк: 4
Введите число столбцов: 5
Массив чисел:
[-3, 9, 9, 5, -5]
[6, 2, -5, 9, 4]
[10, -9, -10, -1, 10]
[6, -8, 8, 10, 7]
Сумма положительных элементов двумерного массива:  95
```

Рис. 3.9. Вычисление суммы положительных элементов двумерного массива

Изменение величины «СумП» происходит в зависимости от условия положительности введенного Tabl(i, j) на каждом шаге данного алгоритма. Величина суммы изменяется, если условие истинно, на величину элемента таблицы Tabl(i, j).

Изменим условие задачи. Допустим, нам необходимо дополнительно найти количество отрицательных элементов главной диагонали исходной матрицы Tabl(5, 5). В этом случае в словесный алгоритм можно добавить:

n.1. – ***k*** = 0 – начальное количество отрицательных элементов главной диагонали;

n.5a. – если номер строки i равен номеру столбца j и Tabl(col, row) < 0, увеличить k = k + 1.

Соответствующий фрагмент программы имеет следующий вид:



```
for row in a:
    for col in row:
        if col < 0:
            s += col
print("Сумма отрицательных элементов двумерного мас-
сива: ", s)
```

Как видно из алгоритма и программы, исходными данными могут быть любые числовые данные, необходимо лишь помнить, что индекс массива – величина всегда целая. Поэтому, несмотря на то, что элементы массива Tabl(5, 5) объявлены в программе как вещественные, но переменные row и col (строки и столбцы), а также переменная k имеют в программе целый тип.

ПРИМЕР 3.2. КОЛИЧЕСТВО ЧИСЕЛ В МАССИВЕ

Дана таблица A, в которой 100 целых чисел. Требуется найти все числа, значение которых равно 25. Составить алгоритм.

Найти все числа означает найти их количество. Обозначим количество – k, начальное значение k = 0. Эта величина в отличие от суммы всегда должна иметь целый тип и всегда одно и то же правило изменения, если не задано дополнительных условий. Если сравнение на совпадение A(i) и числа 25 истинно, то количество должно увеличиться на 1 (рис. 3.10).

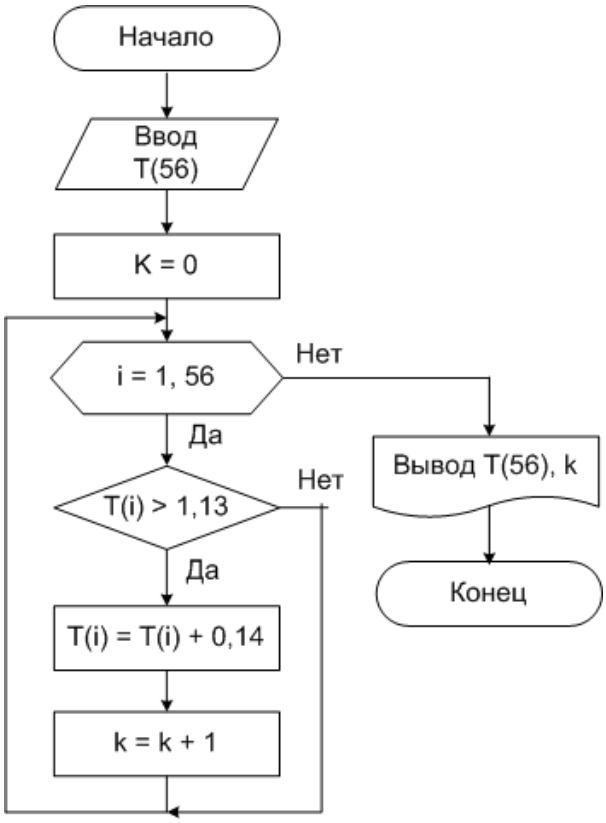


Рис. 3.10. Схема алгоритма поиска чисел, равных заданному числу

Фрагмент программы для нахождения k представлен ниже. Как видно из программы, реализация алгоритма достаточно проста:

```
k = 0
for row in a:
    for col in row:
        k +=1
print("Количество элементов двумерного массива: ", k)
```

При нахождении количества по условию происходит перебор всех возможных значений массива, при этом параметр цикла пробегает все значения от 1 до 100, и если элемент соответствует заданному эталону «25», то количество увеличивается на 1.

___Пример 3.3. Поиск чисел по условию

Дана таблица T, состоящая из 56 чисел. Составить алгоритм нахождения количества чисел, значение которых больше некоторой величины, равной 1,13. Каждое из найденных чисел увеличить на 0,14.

Данная задача частично напоминает предыдущий пример – в части, определения количества. Поэтому приведенный выше фрагмент программы успешно может быть использован и здесь. При истинности проверяемого условия выполняются два действия:

1) изменяется значение переменной t(i) = t(i) + 0.14;
2) k увеличивается на 1.

Поскольку в условии задачи мы имеем дело с вещественными числами, то при задании элементов исходного массива T следует помнить, что тип чисел вещественный. Как в алгоритме (рис. 3.11), так и в приведенной ниже программе использован простой цикл с параметром.

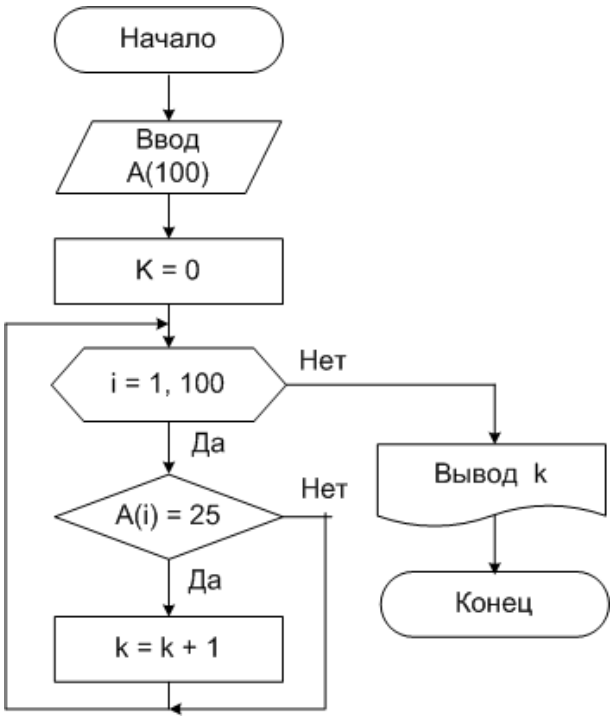


Рис. 3.11. Схема алгоритма поиска чисел больших заданного

Программа имеет вид (рис. 3.12):



```
import random
n = int(input("Введите число строк: "))
m = int(input("Введите число столбцов: "))
s = 0
a = []

for i in range(n):
   a.append([])
   for j in range(m):
       a[i].append(random.randint(-10,10))
print("Массив чисел: ")
for i in a:
   print(i)
for row in a:
   for col in row:
       if col > 0:
           s += col
print("Сумма положительных элементов двумерного
массива: ", s)
```

```
Введите число строк: 4
Введите число столбцов: 5
Массив чисел:
[4, 5, -7, -2, -7]
[-2, 2, 0, 8, 4]
[-3, -7, -5, 3, 9]
[-9, 6, 8, 1, 4]
Сумма положительных элементов двумерного массива:  54
```

Рис. 3.12. Вычисление суммы положительных элементов двумерного массива

При запуске программы можно убедиться, что алгоритм работает для любого количества различных чисел (рис. 3.12).

Пример 3.4. Изменение элементов в зависимости от условия

Дана таблица WR, в которой записано 35 чисел. Составить алгоритм для нахождения суммы положительных элементов массива, отрицательные значения элементов в таблице WR увеличить в 2 раза.

Для нахождения суммы введем обозначение суммы S. В условии задачи нет дополнительных условий на тип исходных данных. Известно лишь, что они могут быть и отрицательные (< 0), и положительные (> 0). Сложить положительные числа означает найти S только для тех, значение которых больше 0 в заданном массиве.

Если значение рассматриваемого числа WR(i) меньше 0, то его величину надо умножить на 2. Таким образом реализуем заданное условие в виде классического разветвления. Выбор одного из двух решений «да» или «нет» – из условия «WR(i) > 0» (рис. 3.13, 3.14).

```
Введите число строк: 4
Введите число столбцов: 5
Массив чисел:
[-2, 8, 8, -8, 0]
[8, -5, -1, 9, 4]
[6, 2, 6, 4, 9]
[-7, 3, 2, -6, -3]
Сумма положительных чисел:  69
Изменённый массив чисел:
[-4, 8, 8, -16, 0]
[8, -10, -2, 9, 4]
[6, 2, 6, 4, 9]
[-14, 3, 2, -12, -6]
```

Рис. 3.13. Результат изменения массива

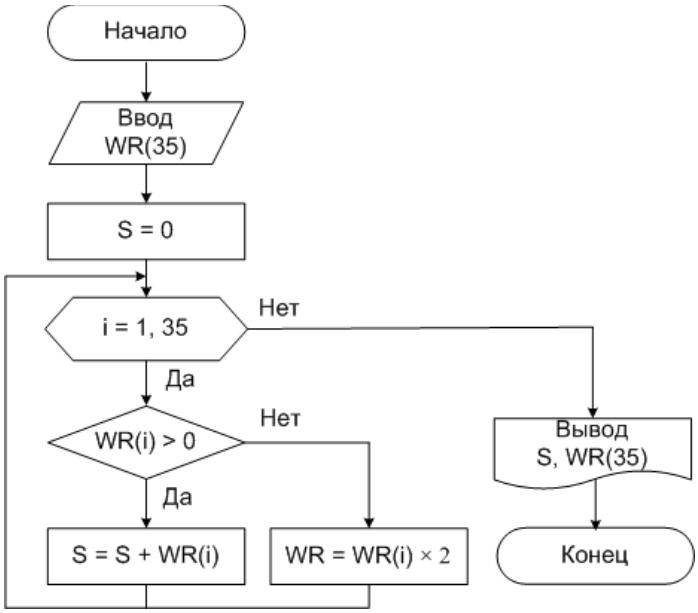


Рис. 3.14. Поиск данных в массиве и их изменение

Реализованный алгоритм в виде программы:

```
import random
n = int(input("Введите число строк: "))
m = int(input("Введите число столбцов: "))
a = []
for i in range(n):
    a.append([])
    for j in range(m):
        a[i].append(random.randint(-10,10))
print("Массив чисел: ", a)

sum = 0
i = 0

for i in range(n):
    for j in range(m):
        if a[i][j] > 0:
            sum += a[i][j]
        elif a[i][j] < 0:
            a[i][j] *= 2
```

```
print("Сумма положительных чисел: ", sum)
print("Измененный массив чисел: ", a)
```

При выполнении программа успешно работает при любых исходных числах в таблице. Количество чисел определяется только возможностью компьютера.

Пример 3.5. Поиск в тексте заданного символа

Задан текст. Найти, содержится ли в данном тексте некоторый заданный символ, и сколько раз он повторяется. Составить алгоритм.

Имеем текст или некоторую последовательность символов, которая является исходной. Назовем эту последовательность Slovo, каждый символ этой последовательности равен i, вернее, это номер позиции, в которой находится символ. Также имеется заданный символ или строка символов, с которым будем сравнивать исходную последовательность. Символ обозначим S1. Длину символьной последовательности можно определить как некоторое вычисляемое N – средства языка программирования позволяют это сделать.

Строка фактически является массивом символов, в котором с каждым символом строки ассоциируется уникальное значение символа. Обратиться к отдельному элементу строки внутри символьной переменной можно с помощью выражения <*имя строки* [i]>, где i – индекс символа, на который ссылается алгоритм. Значение индекса первого элемента строки равно 1, второго 2 и т.д.

Для определения количества найденных символов, равных S1, введем переменную kl. Переменная k необходима для обозначения номера позиции, в которой найден символ S1. В схеме алгоритма (рис. 3.15, 3.16) реализована циклическая структура с предусловием – пока номер позиции символа в рассматриваемой последовательности меньше длины всей последовательности (в программе это text).

Если найдено совпадение в какой-то позиции символа с S1, то этот найденный номер позиции увеличивается на 1 (т.е. переходим к рассмотрению следующего символа позиции). Программа, которая реализует приведенный выше алгоритм, имеет вид:

```
      text = str(input("Введите строку: "))
symbol = str(input("Введите символ: "))
count = 0

for i in text:
    if i == symbol:
        count += 1

if count != 0:
    print("Количество найденных символов: ", count)
else:
    print('Символ не найден')
```

```
Введите строку: Поиск в тексте заданного символа
Введите символ: о
Количество найденных символов:  4
```

Рис. 3.15. Результат поиска символа в тексте

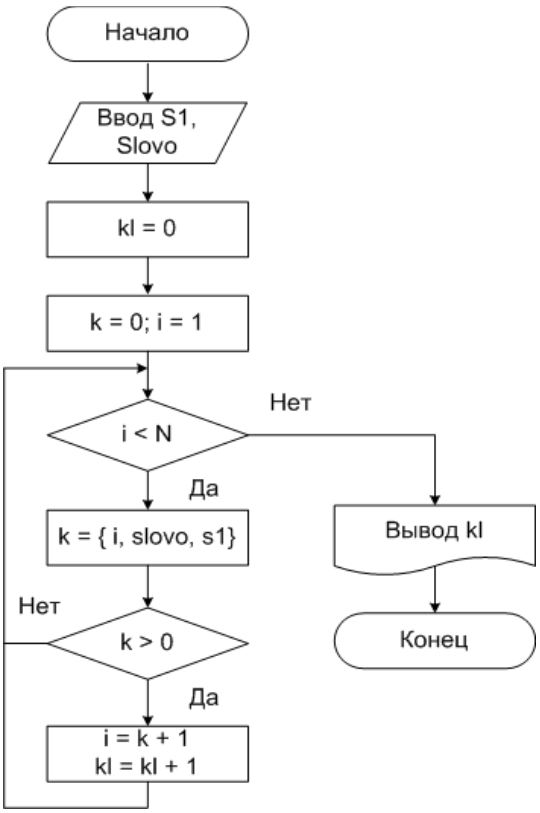


Рис. 3.16. Поиск заданного символа в заданном тексте

Например, в слове «абракадабра» буква «а» содержится 5 раз, в позициях 1, 4, 6, 8, 11. Аналогично можно ввести любой текст или слово.

___ Пример 3.6. Среднее арифметическое последовательности

Вычислить среднее арифметическое для элементов массива кратных 5. Составить алгоритм.

Для вычисления среднего арифметического следует вспомнить, что это такое. Среднее арифметическое некоторой числовой последовательности равно сумме всех элементов этой последовательности, деленное на общее количество ее членов.

Поэтому в данной задаче необходимо выполнить следующие шаги:

1) найти в заданной последовательности A(i) элементы, которые кратны некоторому заданному числу;
2) найти сумму S найденных элементов (кратных заданному числу);
3) найти количество k найденных чисел;
4) вычислить среднее арифметическое как частное от деления накопленной суммы S (п. 2) на количество k (п. 3).

Для реализации п. 1–3 используем цикл с параметром (рис. 3.17), в котором определяем кратные 5 элементы заданного массива. Чтобы реализовать шаг 4, следует проверить, были такие числа найдены или нет. Если k = 0 (деление на 0 в формуле S = S/k), то таких чисел не было найдено и, соответственно, найти среднее арифметическое нет возможности, так как при этом и сумма таких чисел тоже будет равна 0.

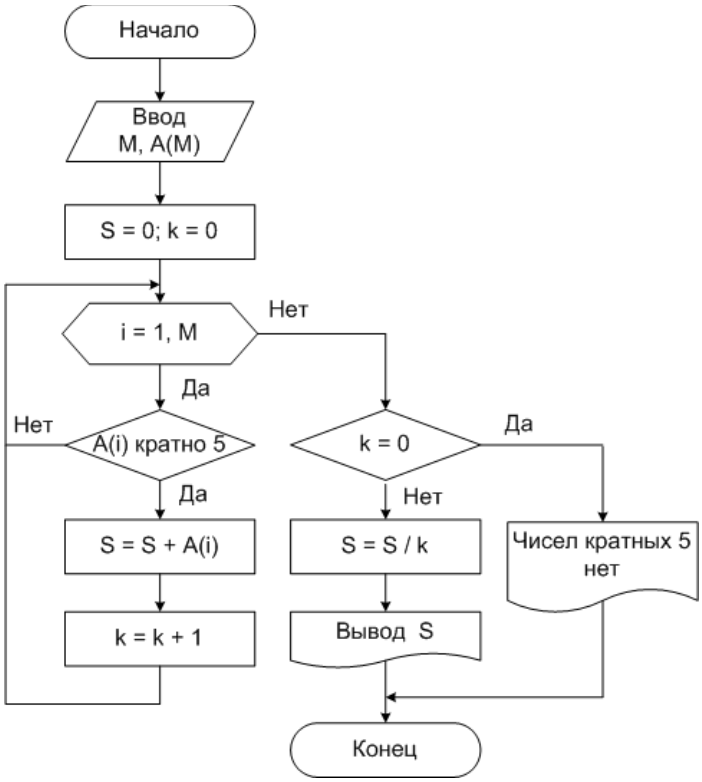


Рис. 3.17. Схема алгоритма вычисления среднего арифметического

Программа для нахождения среднего арифметического заданной последовательности A(M) приведена ниже (рис. 3.18):

```
import random
size = int(input("Введите размер последовательности:
"))
sum = 0
count = 0
a = []

for i in range(size):
   a.append(random.randint(0,10))
print("Массив чисел: ", a)

for i in a:
   sum += i
```

```
        if (i % 5 == 0) and (i != 0):
            count += 1
    print("Среднее арифметическое элементов двумерного
    массива: ", sum/size)
    if count != 0:
        print("Количество чисел, кратных 5: ", count)
    else:
        print("Нет чисел, кратных 5: ")
```

```
Введите размер последовательности: 5
Массив чисел:  [7, 5, 3, 1, 7]
Среднее арифметическое элементов двумерного массива:  4.6
Количество чисел, кратных 5:  1
```

Рис. 3.18. Вычисление среднего арифметического элементов массива

В программе числа генерируются случайным образом на каждом шаге при каждом новом i. Вводимые числа должны быть целого типа, так как кратность определяется с помощью стандартной функции: A(i) % 5 = 0. Иначе кратность числа следует определять другими средствами.

Пример 3.7. Самое длинное слово в тексте

В заданном тексте найти самое длинное слово.

Допустим, имеется некоторый текст Frag, переменная типа строка String неизвестной длины. Чтобы выделить слова в тексте, необходимо выделить пробелы между словами – количество пробелов равно количеству слов минус 1. Обозначим массив слов, как Slovo, тип – строка String. Введем символическое ограничение не более 10 знаков в слове, в тексте не более 100 слов или не более 256 знаков. Определим словесный алгоритм:

1. Ввод текста.
2. Выделить слова в тексте в массив слов.
3. Найти самое длинное слово.

Для последнего пункта можно использовать стандартный алгоритм поиска в заданной последовательности. В данном случае

последовательность будет состоять из массива Slovo(1), Slovo(2), …, который можно определить, если решить следующие задачи:

1) выделить пробел n;
2) определить количество выделенных слов m;
3) определить отдельное слово в массив слов Slovo(m);
4) повторить с п. 1, пока не достигли конца текста.

Реализованный алгоритм представим в виде следующей программы (рис. 3.19):



```
text = str(input("Введите строку: ")).split()
max_word = text[0]
for i in text:
    if len(i) > len(max_word):
        max_word = i
print("Самое длинное слово в строке: ", max_word)
```

```
Введите строку: Москва - столица России, многонациональный город на Москве-реке в западной части страны
Самое длинное слово в строке:  многонациональный
```

Рис. 3.19. Результат поиска самого длинного слова в тексте

Для работы этой программы следует предусмотреть ввод текста – переменная Frag. Для успешной работы программы смысл текста не играет роли, а вот количество символов в исходном тексте для данного случая должно быть не больше 256, так как при описании типа не указано определенное количество символов.

___ Пример 3.8. Удалить элементы из числового массива

Дан целочисленный массив M (N). Изменить массив, удалив из него отрицательные элементы, а значение положительных чисел увеличить на 1. Определить количество элементов в новом массиве (рис. 3.20).

Переформируем массив за счет удаления отрицательных элементов и далее выполним сдвиг элементов влево на «освобожденные» места. В результате получим новый массив на основе старого, поскольку часть данных будет удалена, и оставшиеся исходные данные будут перемещены, т.е. изменится их порядковый номер.

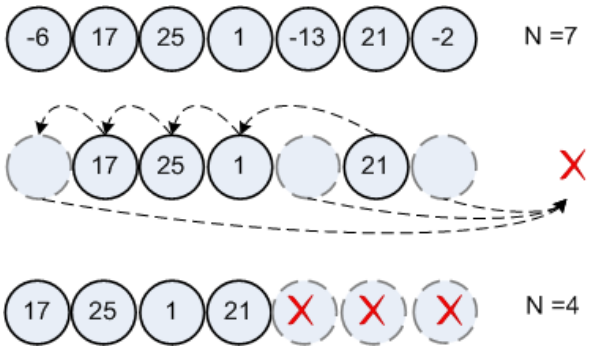


Рис. 3.20. Удалить элементы из числового массива

Для составления алгоритма установим начальное значение для индекса обновленного массива как j = 0. Эта величина также позволит определить количество элементов в переформированном массиве M.

Чтобы найти отрицательные элементы, используем обычный линейный просмотр массива M(n), в данном случае количество элементов не является проблемой при составлении алгоритма. Главное в этом случае быстро посмотреть результат и убедиться в работоспособности алгоритма. Схема алгоритма предложена на рис. 3.21.

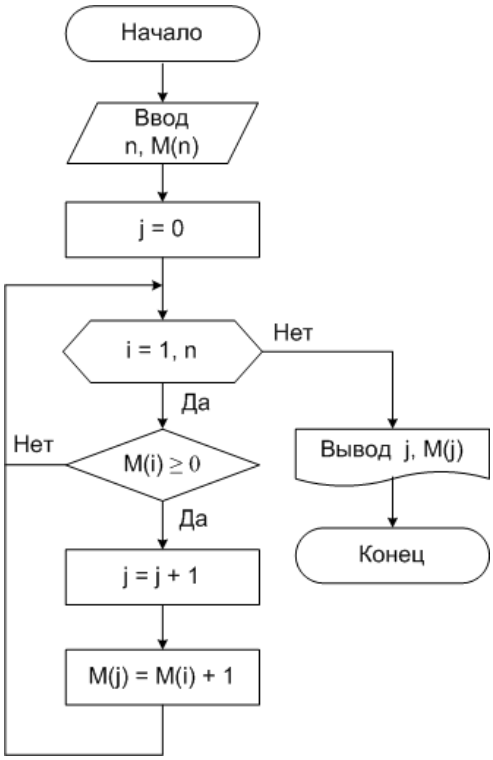


Рис. 3.21. Удаление элементов из числовой последовательности

Ниже представлена реализация алгоритма в виде программы (рис. 3.22):



```
import random
n = int(input("Введите длину массива: "))
a = []
for i in range(n):
    a.append(random.randint(-10,10))
print("Массив чисел: ", a)
for i in range(n-1, -1, -1):
    if a[i] >= 0:
        a[i] += 1
    elif a[i] < 0:
        del a[i]
print("Измененный массив: ", a)
print("Длина измененного массива: ", len(a))
```

```
Введите длину массива: 6
Массив чисел:  [0, 3, 6, 3, -8, -5]
Изменённый массив:  [1, 4, 7, 4]
Длина изменённого массива:  4
```

Рис. 3.22. Результат удаления элементов массива

Просмотр выполняем в цикле с параметром, каждый элемент массива сравниваем с 0. Если встретилось отрицательное число, то его можно «пропустить», и переходим к следующему числу. Если число M(i) положительное, то значение нового индекса для старого элемента массива увеличиваем на 1, и этот элемент мы переписываем на новое место, с новым номером и с новым значением. Новое значение этого элемента будет на 1 больше, согласно условию задачи. В результате просмотра элементов всего массива будет сформирован новый массив. Тип исходных данных – целый или вещественный – роли не играет.

Пример 3.9. Пифагоровы числа

Составить алгоритм вывода пифагоровых чисел.

Известно, что сумма квадратов длин катетов прямоугольного треугольника равна квадрату длины гипотенузы $a^2 + b^2 = c^2$. *Числа a, b, c, если они удовлетворяют этому условию, называются пифагоровыми числами.* Например, основная «тройка» пифагоровых чисел 3, 4, 5:

$3^2 + 4^2 = 5^2$, 9 + 16 = 25.

Производной от основной, которая получена умножением на 2 основной «тройки», являются «тройка» чисел 6, 8, 10:

$6^2 + 8^2 = 10^2$, 36 + 64 = 100 и т.д.

Составим алгоритм нахождения пифагоровых чисел. Если использовать готовую процедуру нахождения квадратного корня из c^2, то алгоритм будет работать быстрее. В алгоритме использован вложенный цикл – цикл в цикле. Все величины, используемые в алгоритме (схема алгоритма на рис. 3.10), имеют целый тип, – это необходимо учесть при составлении программы, фрагмент которой представлен ниже. Переменная c1 используется для вычисления суммы $a^2 + b^2$.

При реализации алгоритма (рис. 3.23) в программе следует предусмотреть два цикла с параметром для перебора значений a и b, а также цикл с предусловием, в котором будет проверяться условие существования пифагоровых чисел $a^2 + b^2 = c^2$.



```
n = int(input("Введите n: "))

for a in range(1, n):
    for b in range(1, n):
        c1 = a**2 + b**2
        c = 1
        while (c1 > c*c) and (c < n):
            c += 1
            if (c1 == c**2):
                print(a, b, c)
```

При реализации алгоритма выводятся тройки соответствующих пифагоровых чисел (рис. 3.24). Тип чисел a, b, c – целый.

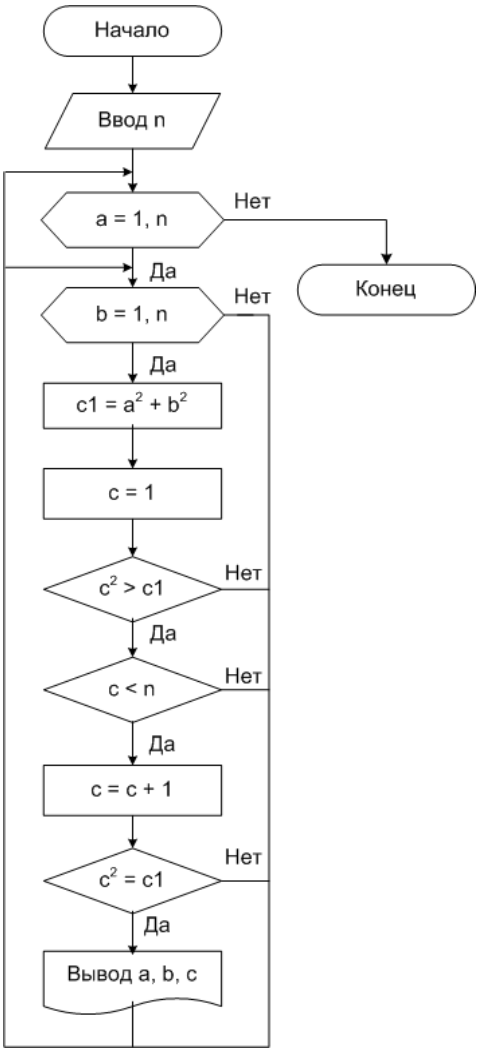


Рис. 3.23. Схема алгоритма для нахождения пифагоровых чисел

```
Введите n: 13
3 4 5
4 3 5
5 12 13
6 8 10
8 6 10
12 5 13
```

Рис. 3.24. Результат вычисления пифагоровых чисел

Пример 3.10. Простые множители

Разложить число M на простые множители.

Известно, что простые числа – это натуральные числа, которые больше 1 и имеют только два делителя: единицу и само это число. Чтобы проверить, является ли данное число простым или нет, следует определить, имеет ли данное число n (n> = 2) делители в интервале [2; n – 1]. Все четные числа (те, что делятся на 2 без остатка) также являются составными.

Пусть M – заданное число, которое надо разложить на простые множители. Необходимо:

1) выяснить, имеет ли число м делители;
2) просмотреть все возможные делители числа M и установить для каждого делителя, является ли оно простым числом.

В качестве исходных данных имеем целые числа, поэтому выполняем целочисленное деление и получаем целое число. Словесный алгоритм будет следующим:

1. Ввести *Целое Число M*.
2. n = M.
3. Первый возможный делитель.
4. Пока n ≠ 1, выполнять цикл {*цикл с предусловием*}, иначе перейти к п. 9.
5. Вычислить *частное* = n \ k {*целочисленное деление*}.
6. Если *частное* ≠ n, то увеличить k = k + 1 и перейти к п. 3.
7. Иначе вывести k.
8. n = *частное*.
9. Перейти к п. 3.
10. Конец (рис. 3.25).

Для реализации алгоритма в программе используем цикл предусловием:

```
m = int(input("Введите число: "))
n = m
k = 2
print("Простые множители числа:")
```

```
while (n != 1):
    h = int(n/k)
    if h*k != n:
        k += 1
    else:
        print(k)
        n = h
```

```
Введите число: 153
Простые множители числа:
3
3
17
```

Рис. 3.25. Результат нахождения простых множителей числа

Задавая различные значения для M, получаем список простых делителей.

Например, для M = 210 простыми делителями являются 2, 3, 5, 7;
для M = 872 простыми делителями являются 2, 2, 2, 109;
для M = 870 простыми делителями являются 2, 3, 5, 29.



Вопросы для самопроверки

Составить алгоритмы для решения следующих задач:

1. Даны натуральное число n, действительные числа a_1, … , a_n. В последовательности a_1, … , a_n все отрицательные элементы, которые не принадлежат интервалу [1,5–2,5], заменить на 2. Определить количество отрицательных элементов. Определить количество элементов, которые принадлежат интервалу [1,5–2,5].
2. *Числа, которые можно одинаково прочитать слева направо и справа налево, называются палиндромами*, например, 21312, 52325, 4224. Проверить, является ли заданное число палиндромом.

3. В заданном интервале найти все натуральные числа, равные кубу суммы своих цифр.
4. Натуральное число называется *совершенным*, если оно равно сумме всех своих простых делителей, например, 6 = 1 + 2 + 3. Найти все совершенные числа в заданном интервале.
5. Известно, что разность любого натурального числа и суммы его цифр кратна 9. Проверить этот факт для чисел в заданном интервале.
6. Число делится на 11, если разность между суммой цифр, стоящих на нечетных местах, кратна 11. Проверить этот признак для всех натуральных чисел, значение которых не превосходит заданного m, и показать числа, кратные 11.
7. Определить, сколько различных букв содержится в заданном слове.
8. Найти два наименьших числа, которые начинаются на пять и из которых, перенеся первую цифру в конец, можно получить новое число, в пять раз меньшее, чем искомое.
9. Проверить, можно ли заданное натуральное число представить в виде:
   а) произведения двух простых чисел;
   б) произведения трех простых чисел;
   в) квадрата какого-либо простого числа;
   г) куба какого-либо простого числа.
10. Даны натуральные числа n, числа n, b_0, … , b_n. Вычислить $f(b_0) + f(b_1) + \ldots + f(b_n)$, где

$$F(x) = \begin{cases} x^2, \text{ если } x \text{ кратно } 5; \\ x + 1, \text{ если } x \text{ – четное число}; \\ 0{,}5 - 5x, \text{ в остальных случаях.} \end{cases}$$

11. Даны натуральные числа m, n, действительные числа a_1, …, a_n ($m < n$). Найти среднее арифметическое всех чисел данной последовательности, кроме a_m.
12. Даны целые числа p, q, a_1, …, a_{55} ($0 \le q < p$). В исходной последовательности a_1, …, a_{55} заменить нулями элементы последовательности, модуль которых при делении на p дает в остатке q.
13. Дан текст, в начале которого имеются пробелы и в котором имеются цифры. Найти порядковый номер максимальной цифры. Если максимальных цифр несколько, то следует определить номер первой из них.

14. Дано слово. Если длина его нечетная, то удалить среднюю букву, в противном случае – две средних буквы.
15. Дано слово. Удалить из него все повторяющиеся буквы, оставив их первые вхождения: в слове должны остаться только различные буквы.
16. Дано слово. Переставить его первую букву на место последней буквы. При этом вторую, третью, …, n-ю буквы сдвинуть влево на одну позицию.
17. Дан текст. Если в нем нет малых латинских букв, то оставить его без изменения, иначе каждый из символов, которые следуют за первой группой малых латинских букв, заменить точкой.
18. Определить, есть среди чисел $t^3 – 17in^2 + n^3$ (i = 1, 2,…, n) хотя бы одно число, которое кратно a и не кратно b.
19. Дано натуральное число n. Получить сумму тех чисел вида $t^3 – 17in^2 + n$ (i = 1, 2,…, n) , которые являются утроенными нечетными числами.
20. Дана последовательность $a_1, \dots, a_{1000}$. Требуется определить, сколько членов последовательности с номерами 1, 2, 4, 8, 16, … имеют значение меньше, чем 0,75, при этом считать, что
    $a_k = \sin^2(3k + 5) – \cos(k^2 – 15)$, k = 1, 2, …, 1000;
    $a_1, \dots, a_{1000}$ – заданные действительные числа;
    $a_1 = 0.01$; $a_k = \sin(k – a_{k–1})$, k = 2, …, 1000.
21. Дано натуральное число n, действительные числа $x_1, \dots, x_n$. Получить (1 + r)/(1 + s), где r – сумма всех элементов последовательности $x_1, \dots, x_n$, которые не превосходят 1, а s – сумма элементов, больших 1.
22. Найти наименьший элемент в массиве X(10) и его порядковый номер. Переставить местами найденный и первый элементы.
23. В массиве A(10) найти элемент, минимальный по абсолютному значению и не равный нулю. Образовать новый одномерный массив так, чтобы его элементы определялись как целое частное от деления соответствующего элемента массива A на найденный элемент.
24. Заполнить матрицу заданного размера числами 1, 2, 3, 4 так, чтобы по горизонтали, вертикали и диагонали не было одинаковых рядом стоящих чисел.
25. Найти среднее арифметическое положительных элементов каждого столбца матрицы A(n, m) , заданной целыми числами.

26. Определить количество положительных элементов каждого столбца матрицы A(n, m), заданной целыми числами.
27. Дана действительная матрица размера n × m, в которой не все элементы равны нулю. Получить новую матрицу путем деления всех элементов данной матрицы на ее наибольший по модулю элемент.
28. Дана действительная матрица размера 6 × 7. Найти среднее арифметическое наибольшего и наименьшего значений ее элементов.
29. В исходной действительной матрице размера n × m поменять местами строку, в которой находится элемент с максимальным значением, со строкой, в которой находится элемент с минимальным значением.
30. Двумерный массив задан случайным образом. Найти:
    а) количество нулевых элементов главной диагонали;
    б) элементы массива, значение которых меньше 1, и присвоить им значение $\sqrt{ai + j^2 b}$;
    в) суммы элементов строк и столбцов.
31. Дана матрица размера n × m, элементы которой заданы следующим образом:

$$a(i, j) = \begin{cases} (41 + j)^2 - |i - j^2|^{1/3}, \text{ если } i > j; \\ (j^3 + 7 - i)^{1/2}, \text{ в остальных случаях.} \end{cases}$$

    Определить в главной диагонали матрицы число элементов, значение которых превышает величину ci – bj, где c и b – случайные числа из интервала 0.5 ÷ 95.5. Найти номера строк, которым принадлежат эти элементы.
32. Дана символьная матрица размера n × m. Получить последовательно все строки матрицы, исключая те, для которых есть равные среди строк с меньшими номерами.

Глава 4

Сортировка данных

Методы сортировки служат
великолепной иллюстрацией
базовых концепций
анализа алгоритмов…
Д. Кнут



Сортировка – важная часть вычислительной математики и отнимает значительную часть времени работы компьютера. Сортировка относится к алгоритмам обработки таблиц (массивов) любого типа.

> *Сортировка данных – это обработка информации, в результате которой элементы ее (записи) располагаются в определенной последовательности в зависимости от значения некоторых признаков элементов этой информации.*

Наиболее распространенным видом сортировки данных является *упорядочение массива* – расположение записей сортируемого массива данных в порядке монотонного изменения некоторого ключевого признака. Сортировка данных позволяет сократить во много раз продолжительность решения задач, которые связаны с обработкой больших массивов информации. С отсортированными данными легче работать. Когда элементы отсортированы, как в телефонном справочнике, их проще найти, обновить, исключить, легче отыскать, какие элементы пропущены.

Смысл любой сортировки заключается в перестановке элементов таблицы в определенном заданном порядке. Упорядочение осуществляется в процессе многократного просмотра исходного массива. Обычно методы сортировки подразделяют на два класса:

- внутренняя сортировка, когда все записи хранятся в быстрой оперативной памяти;
- внешняя сортировка, когда все записи в ней не помещаются.

Методы внутренней сортировки обеспечивают большую гибкость при построении структур данных и доступа к ним. Для внутренней сортировки характерно, что число записей мало, и весь процесс можно провести в оперативной памяти компьютера с большим быстродействием.

Внешние методы обеспечивают достижение нужного результата в условиях ограниченных ресурсов, поэтому в существующих методах внешней сортировки время обращения к внешней памяти занимает значительную часть общего времени сортировки. Поэтому важной целью методов внешней сортировки данных является минимизация количества просмотров сортируемого массива, записанного во внешней памяти, используемой, как правило, в режиме последовательной выборки.

Невозможно назвать алгоритм сортировки – универсально наилучшим в любой ситуации. Имеется много наилучших способов, но только в случаях, когда известно, что сортируется, на каком компьютере и с какой целью. Эффективность алгоритма будет зависеть от множества факторов:

- сколько элементов участвует в сортировке;
- все ли элементы могут быть помещены в доступную область или в доступный интервал;
- в какой степени элементы уже отсортированы;
- какой диапазон и распределение значений сортируемых элементов;
- записаны ли элементы в файл или массив;
- предполагается ли, что элементы будут периодически исключаться или дополняться;
- можно ли сравнивать элементы параллельно.

Достаточно хороший общий алгоритм затрачивает на сортировку N-записей время, пропорциональное N logN, при этом требуется около logN «проходов» по данным.

Сортировать можно любые данные, важно, чтобы их можно было сравнить тем или иным способом. Сортировать можно значения скалярных типов, упакованные массивы, символы, строки и т.д. Отсортировать числовую таблицу – это означает переставить элементы

в ней так, чтобы они расположились в порядке убывания (возрастания) значений с возрастанием (убыванием) нового номера элемента (табл. 4.1).

Таблица 4.1

Исходные данные для сортировки

Номер элемента	1	2	3	4	5	6	7
Исходный числовой массив	4	17	25	1	21	13	2
Упорядочение по возрастанию	1	2	4	13	17	21	25
Упорядочение по убыванию	25	21	17	13	4	2	1

Сортировка символьной (или текстовой) информации заключается в упорядочении значений (текстовых строк) по алфавиту (табл. 4.2).

Таблица 4.2

Сортировка символов по алфавиту

Номер элемента	1	2	3	4	5	6	7
Исходный символьный массив	цикл	этап	байт	код	план	язык	схема
Упорядочение по алфавиту	байт	код	план	схема	цикл	этап	язык

Метод сортировки является устойчивым, если относительный порядок элементов с равными значениями не меняется после упорядочения. Сортировку можно рассматривать и как самостоятельную задачу (например, для получения упорядоченного по алфавиту списка сотрудников, какого-либо учреждения) и как вспомогательную – для облегчения последующего поиска элементов в упорядоченной таблице или массиве.

Анализ известных алгоритмов сортировки очень полезен, так как в них используются практически все универсальные приемы конструирования алгоритмов. Для оценки алгоритмов сортировки обычно используют функциональную зависимость времени от количества N сортируемых элементов. Обыкновенно алгоритм представляет интерес при больших значениях N. Но, как правило, на практике необходимы методы сортировки, позволяющие экономить оперативную память, не требующие дополнительных массивов. Рассмотрим далее такие методы.

Простой выбор

Выделяется наименьший (или наибольший) элемент массива и отделяется от остальных. Далее опять выделяется наименьший (или наибольший) из оставшихся и т.д. (рис. 4.1).

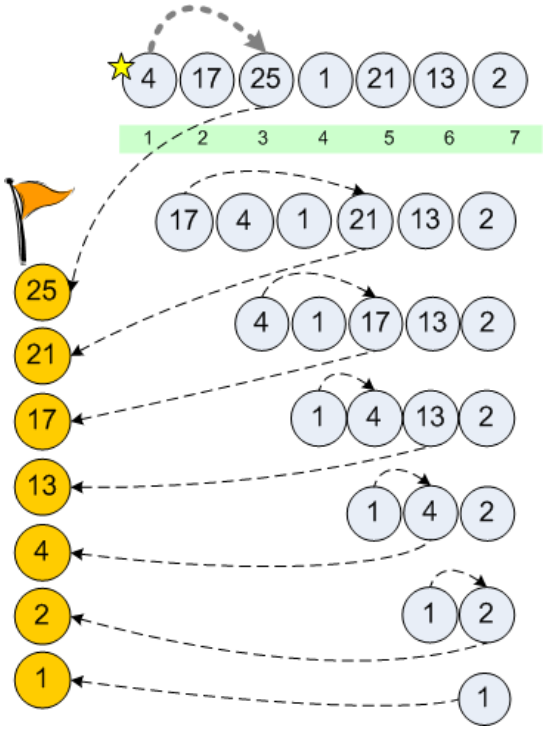


Рис. 4.1. Выделение элемента массива

Например, расположить элементы последовательности A_1, A_2, … , A_N в порядке убывания. Найти в массиве элемент с наибольшим значением и поменять его местами с первым элементом. Далее те же действия выполнить с остальными N–1 элементами массива, затем с N–2 элементами и т.д., пока не останется один элемент – последний, наименьший.

Допустим, что два первых элемента являются упорядоченными. Теперь необходимо отыскать максимальный элемент среди остальных. Если несколько элементов последовательности оказываются равными, то следует найти первый среди максимальных элементов. Таким образом, можно достичь устойчивости сортировки. Найденный элемент из двух первых и третий по списку элемент следует поменять местами. Для получения результата необходимо N–1 раз

найти максимальное значение в массиве, длина которого будет уменьшаться с каждым шагом на 1 (рис. 4.2).

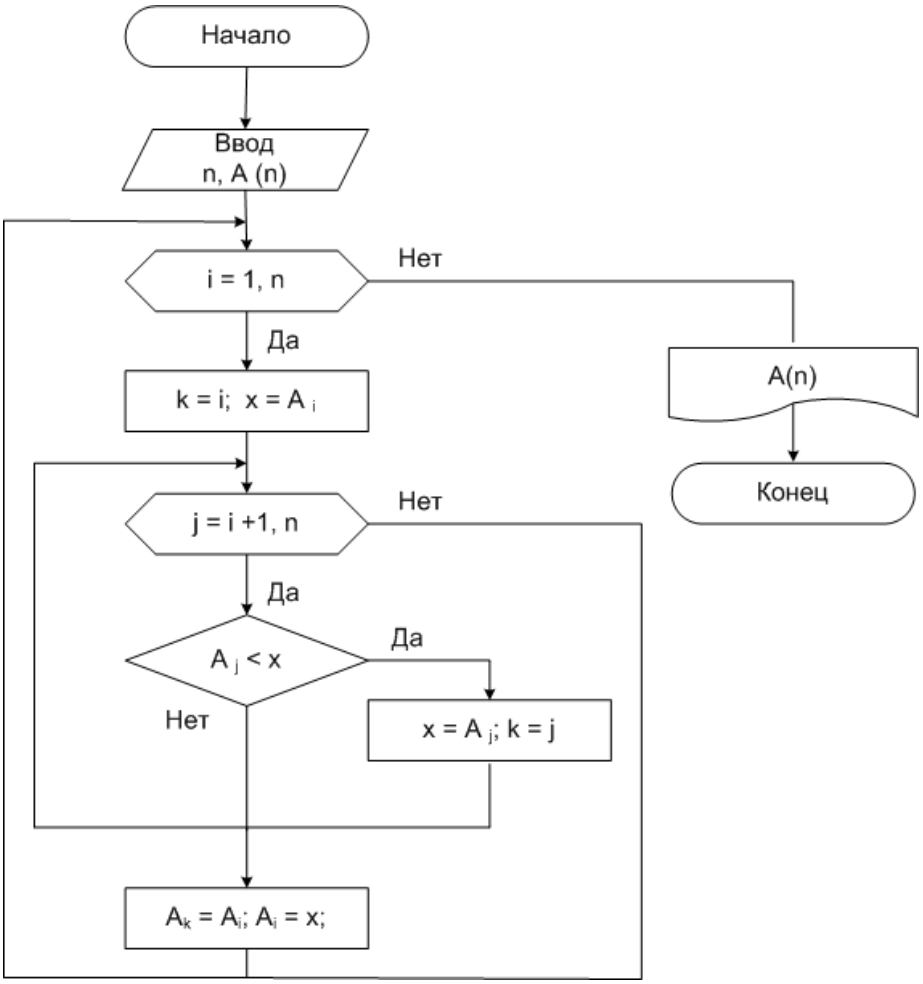


Рис. 4.2. Схема алгоритма сортировки простым выбором

Словесный алгоритм сортировки простым выбором будет следующим:

1) начинаем сортировку с первого элемента i =1;
2) найти максимальный элемент и его номер в массиве $A_1, A_2, \ldots, A_N$;
3) поменять местами A_i и максимальный элемент A_k;
4) перейти к следующему элементу i = i + 1;
5) если рассмотрены не все N–1 элементы, то повторить с п. 2.

Используем разработанный выше алгоритм поиска максимума и присвоим максимальной величине имя x. Чтобы выполнить сортировку по возрастанию, производим все те же действия, только находим на каждом шаге минимум. В связи с этим в алгоритме изменяется только условие $A_j > x$ на $A_i < x$. Для перестановки переменных в сортируемом массиве местами используется операция «переприсвоения» вида $A_k = A_i$; $A_i = x$, где k – номер максимального элемента (одного из двух) на данном этапе.

Программа для организации сортировки простым выбором:



```
# Сортировка простым выбором
import random
n = int(input("Введите длину массива: "))
a = []
j = n

for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел: ", a)

for i in range(n):
    k = i
    x = a[i]
    for j in range(i+1, n):
        if a[j] < x:
            x = a[j]
            k = j
    a[k] = a[i]
    a[i] = x
print("Отсортированный массив: ", a)
```

В программе реализован алгоритм сортировки по возрастанию.

В качестве контрольного примера возьмем случайную последовательность

```
52 35 19 8 27 12 56 7 2 19
для N =10 Простой выбор
N прохода
1       2 35 19 8 27 12 56 7 52 19
2       2 7 19 8 27 12 56 35 52 19
3       2 7 8 19 27 12 56 35 52 19
4       2 7 8 12 27 19 56 35 52 19
5       2 7 8 12 19 27 56 35 52 19
6       2 7 8 12 19 19 56 35 52 27
7       2 7 8 12 19 19 27 35 52 56
```

Чтобы упорядочить слова по алфавиту, тип сортируемых данных следует изменить на строковый (рис. 4.3).

```
Введите длину массива: 10
Массив чисел:  [4, 4, 5, 7, 0, 4, 10, 2, 10, 8]
Отсортированный массив:  [0, 2, 4, 4, 4, 5, 7, 8, 10, 10]
```

Рис. 4.3. Пример сортировки данных методом простого выбора

Достаточно простой для реализации алгоритм. Работает достаточно быстро, если последовательность частично упорядочена или при малых значениях N. Алгоритм сортировки выбором в среднем выполняет N(N–1)/2 сравнений и 3(N–1) присваиваний.

Простой обмен

Если два элемента расположены не по порядку, то они меняются местами. Этот процесс повторяется до тех пор, пока элементы не будут упорядочены….

Любой метод сортировки, так или иначе, связан с обменом – перестановкой двух элементов в памяти. Для обменной сортировки это основная характеристика процесса. Идея сортировки простым обменом заключается в многократном сравнении элементов массива, стоящих рядом, и перестановке этих элементов в заданном порядке. Таким образом, соседние элементы A_1 с A_2, A_2 и A_3, A_3 и A_4 … поочередно сравниваются. Если $A_{i-1} > A_i$, то элементы меняются местами.

Пусть исходные данные заданы табл. 4.1. Необходимо их расположить по возрастанию. Для этого следует организовать просмотр данных от конца к началу или от начала к концу. Например, сравнить значения элементов 7 и 6 и поменять их местами, так как они расположены по убыванию. Далее сравнить элементы 6 и 5 и также поменять местами и т.д.

В результате минимальный элемент переместится в массиве на первое место, и новая таблица после первого «прохода» примет новый вид (рис. 4.4).

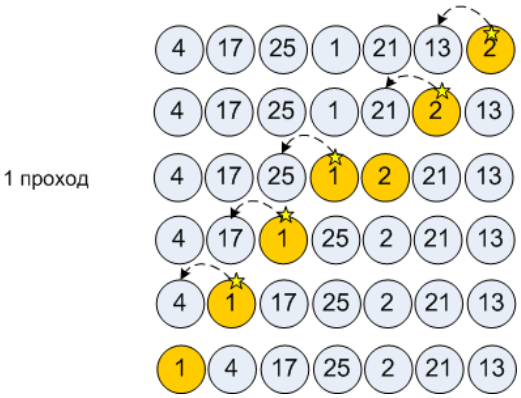


Рис. 4.4. Таблица после первого «прохода»

Все дальнейшие просмотры уже не включают просмотр первого элемента. Каждый последующий просмотр будет «устанавливать» очередной минимальный элемент в левую часть таблицы (рис. 4.5).

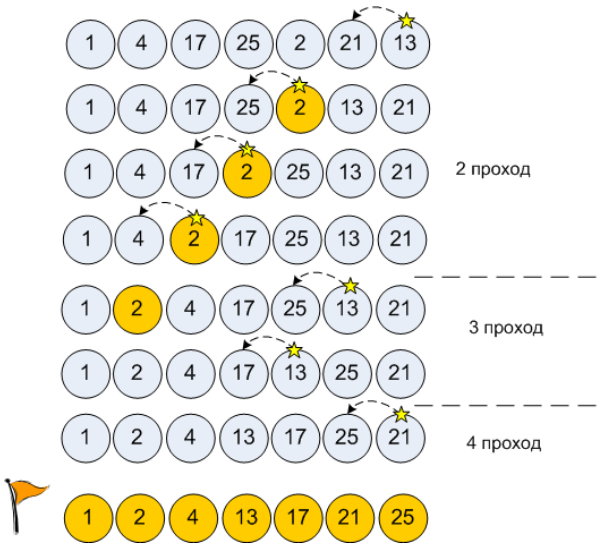


Рис. 4.5. «Устанавление» элементов в левую часть таблицы

Если представить, что элементы таблицы – это пузырьки в сосуде с водой, каждый с весом равным ему по значению, то каждый проход с обменом снизу вверх по таблице приводит к «всплыванию пузырька» на соответствующий его весу уровень. Благодаря такой аналогии сортировка простым обменом получила название *пузырьковой сортировки*.

Составим алгоритм для решения этой задачи (рис. 4.6). Основная часть алгоритма – цикл, который должен выполниться за N–1 раз. Выбор границ 1 и (N–1) или 2 и N влияет только на задание индексов сравниваемых элементов. Пусть это будут 2 и N.

Словесный алгоритм будет следующим:

1) i = 2;
2) сравнить попарно элементы A_n, A_{n-1}, . . . , A_i, A_{i-1};
3) i =i + 1;
4) если i ≤ N , то повторить с п. 2.

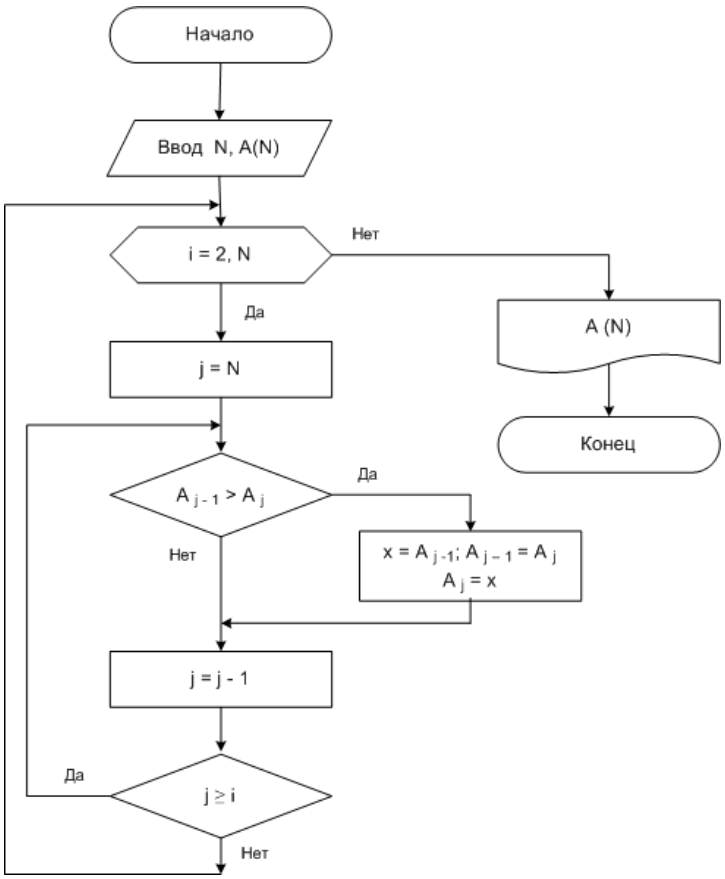


Рис. 4.6. Схема алгоритма простой обмен или пузырьковая сортировка

Для организации цикличности сравнения в схеме алгоритмов можно использовать цикл с параметром, который зависит от i, так как при каждом новом проходе по таблице длина ее будет уменьшаться.

Рассмотрим «попарное сравнение» более подробно:

1) j = N;
2) сравнить A_i и A_{i-1} и при необходимости поменять местами;
3) j = j – 1;
4) если j ≥ i , то повторить с п. 2.

Сравнение в п. 2 можно записать, используя операцию переприсвоения:

$$\text{Если } A_i, > A_{i-1}, \text{ то } X = A_{i-1};$$
$$A_{i-1} = A_i;$$
$$A_i = X.$$

Здесь X – вспомогательная переменная для хранения A_{i-1} , которое затем следует присвоить – перенести в A_i. На рис. 4.6 представлена схема алгоритма пузырьковой сортировки, для которой использован цикл с постусловием.

Программа для реализации этого алгоритма имеет вид:



```
# Сортировка простого обмена или пузырьковая сортировка
import random
n = int(input("Введите длину массива: "))
a = []

for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел: ", a)

for i in range(n):
    for j in range(n-i-1):
        if (a[j+1] < a[j]):
            x = a[j]
            a[j] = a[j+1]
            a[j+1] = x

print("Отсортированный массив: ", a)
```

Программа выполняет сортировку простым обменом по возрастанию для любых вещественных чисел.

В качестве контрольного примера возьмем случайную последовательность (рис. 4.7).

```
Введите длину массива: 10
Массив чисел:  [9, 2, 7, 10, 2, 2, 6, 8, 8, 0]
Отсортированный массив:  [0, 2, 2, 2, 6, 7, 8, 8, 9, 10]
```

Рис. 4.7. Пример сортировки данных методом простым обменом

Алгоритм обменной сортировки осуществляет максимально возможное количество сравнений, много раз приходится просматривать список и выполнять много перестановок. Это дает повод считать его неэффективным алгоритмом. Увеличить возможности алгоритма можно, например, чередованием проходов «вперед» и «назад».

Простые вставки

Элементы просматриваются по одному. Каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов.

Сортировки простым выбором и простым обменом относятся к сортировкам с убывающим шагом. Сортировка простыми вставками[6] работает со списком неупорядоченных целых чисел и сортирует их в порядке возрастания. Это делается примерно так же, как это делают игроки, они упорядочивают сданные им карты, поднимая каждый раз по одной карте.

Отсортированный список вначале пуст. На каждом шаге первое число из неотсортированного списка удаляется и помещается в отсортированный список на соответствующее место. Для этой цели отсортированный список должен каждый раз быть просмотрен, начиная с наименьшего числа и до тех пор, пока нужное место не будет найдено. При этом все реорганизации списка происходят в одном списке. Алгоритм заканчивает свою работу, когда элемент, стоящий до сих пор на последнем месте, будет вставлен на соответствующее место в отсортированном списке (рис. 4.8).

[6] В работах [5] и [8] этот метод рассмотрен как метод прямого включения и работает со списком положительных неупорядоченных целых чисел

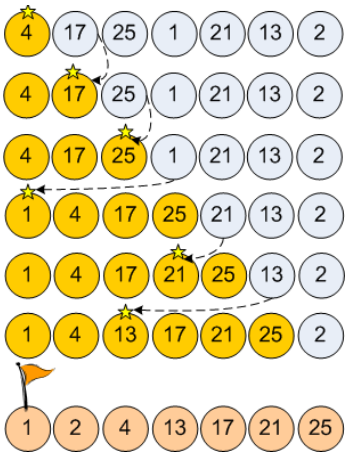


Рис. 4.8. Сортировка списка

Пусть в заданной последовательности A_1, A_2, … , A_N первые i–1 элементы уже отсортированы. Необходимо найти в этой части последовательности место для i-го элемента и сравнить его по порядку со всеми элементами, стоящими левее до тех пор, пока не окажется, что некоторый $A_k \in A_i$. Затем для части последовательности A_{k+1}, A_{k+2}, …, A_{i+1} выполнить сдвиг на один элемент вправо, чтобы освободить место элемента A_{k+2} для элемента A_i. В результате исходная последовательность будет отсортирована.

Если провести сравнение элемента со всеми другими элементами, стоящими слева, то может оказаться, что не найдется такого A_k, что $A_k \in A_i$. Это возможно при условии, если A_i меньше тех элементов, которые находятся слева. Необходимо закончить просмотр, выполнить сдвиг элементов A_1, A_2, …, A_i, и элемент A_i поставить на первое место.

Как в предыдущих случаях, основой алгоритма вставок является цикл, который должен определить, для какого элемента последовательности следует найти место в левой упорядоченной части.

Обобщенный алгоритм вставок следующий:

1) i = 2;
2) найти место A_k для A_i в упорядоченной последовательности, где уже находится A_{i-1};
3) выполнить сдвиг элементов A_{k+1}, A_{k+2}, … , A_{i+1} вправо;
4) поставить элемент A_i на нужное место (A_k = X, если X определено);
5) i = i + 1;
6) если $i \le N$, то повторить с п. 2.

В результате выполнения п. 2 вычисляется номер k – индекс элемента, за которым справа должен следовать A_i. Чтобы найти место

для A_i, необходимо сравнить его последовательно со всеми левыми соседями до элемента $A_k \in A_i$. Если такого элемента нет, то следует остановиться на первом элементе.

При детализации выполняется сдвиг последовательности вправо. В этом случае последовательность необходимо просматривать из конца в начало и последовательно сдвигать элементы:

1) $j = 1$;
2) $A_i = A_{i-1}$;
3) $j = j - 1$;
4) если $j > k$, то повторить с п. 2.

Окончательный алгоритм сортировки простыми вставками изображен на рис. 4.9.

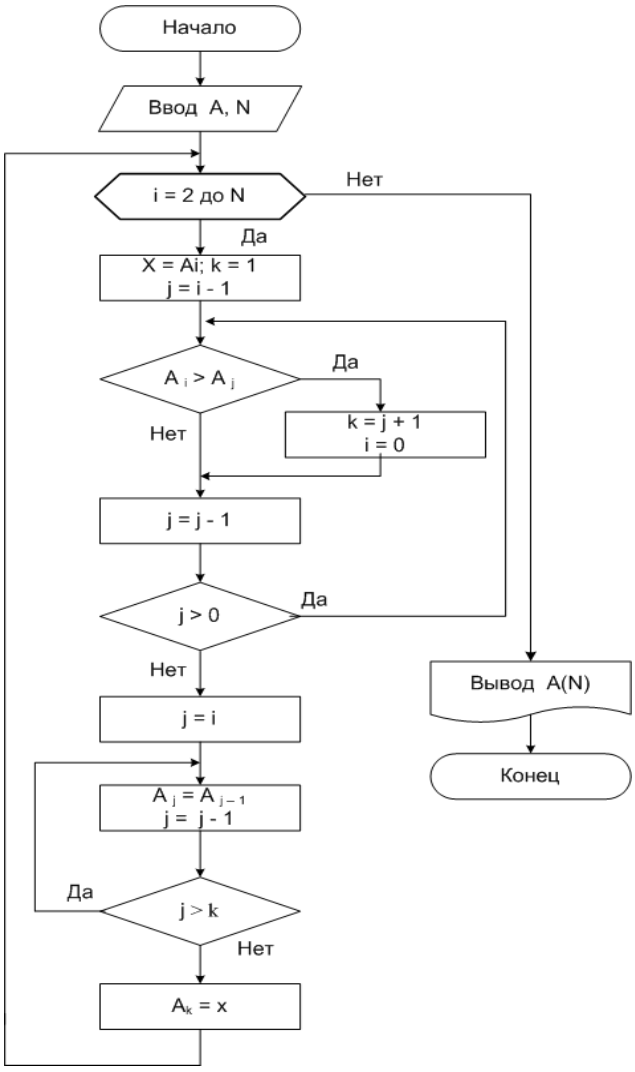


Рис. 4.9. Схема алгоритма сортировка простыми вставками

Программа имеет следующий вид (рис. 4.10):



```
# Сортировка простыми вставками
import random
n = int(input("Введите длину массива: "))
a = []

for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел: ", a)

for i in range(1, n):
    k = a[i]
    j = i - 1
    while (a[j] > k) and (j >= 0):
        a[j+1] = a[j]
        j -= 1
    a[j+1] = k

print("Отсортированный массив: ", a)
```

```
Введите длину массива: 10
Массив чисел:  [9, 0, 0, 8, 9, 10, 3, 1, 7, 0]
Отсортированный массив:  [0, 0, 0, 1, 3, 7, 8, 9, 9, 10]
```

Рис. 4.10. Пример сортировки простыми вставками

В качестве контрольного примера возьмем последовательность из 10 случайных чисел:

```
36 48 15 46 53 40 44 20 26 5 для N =10 Простые вставки
36 48 15 46 53 40 44 20 26 5
36 48 15 46 53 40 44 20 26 5
```

```
15 36 48 46 53 40 44 20 26 5
15 36 46 48 53 40 44 20 26 5
15 36 46 48 53 40 44 20 26 5
15 36 40 46 48 53 44 20 26 5
15 36 40 44 46 48 53 20 26 5
15 20 36 40 44 46 48 53 26 5
15 20 26 36 40 44 46 48 53 5
```

Сложность данного алгоритма зависит от того, какой алгоритм поиска используется для определения места выделенному элементу в новом массиве. Программа реализует алгоритм сортировки простыми вставками по возрастанию для любых вещественных чисел. Метод прост для программной реализации и является достаточно эффективным при малых значениях N (при $N \leq 25$). Не требует дополнительного объема памяти. При больших значениях N он становится невозможно медленным, если только данные не окажутся почти упорядоченными.

Метод Шелла

Сортировка Шелла является усовершенствованным методом простых вставок[7]. Метод предложен Д. Шеллом в 1959 г. и назван именем автора [16]. Характерным для данного метода является то, что сначала рассматриваются отдаленные, а затем – близкорасположенные записи. Каждый проход в этом случае характеризуется некоторым смещением h для сортируемых записей – интервал, который разделяет сравниваемые записи (рис. 4.11).

Другими словами, каждая запись отстоит от предыдущей записи на h позиций. Часто используемую последовательность смещений 8, 4, 2, 1 не следует считать постоянной. Применять можно любую последовательность, но последнее смещение всегда h = 1, например, 7, 5, 3, 1.

[7] Сортировка Шелла имеет и другое название – *сортировка включением с убывающим приращением.*

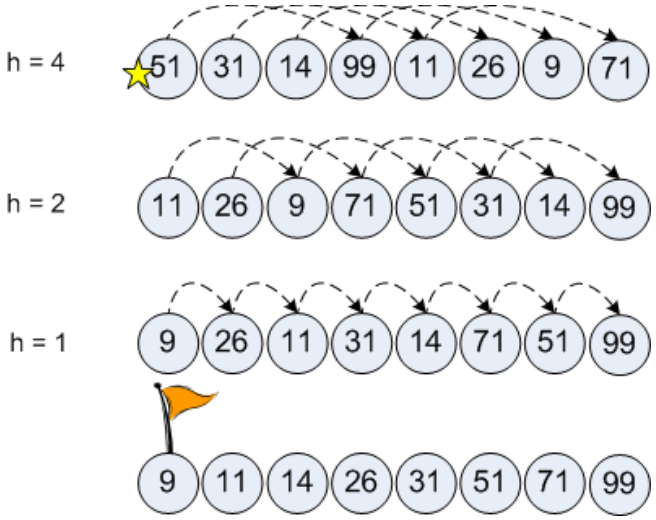


Рис. 4.11. Смещение h для сортируемых записей

Рассмотрим сортировку Шелла на примере из восьми элементов. В результате *первого прохода* должны быть сгруппированы и отсортированы все элементы со смещением h = 4, отстоящие друг от друга на 4 позиции.

После первого прохода элементы объединяются в группы с элементами, которые отстоят друг от друга на две позиции, и снова сортируются. Процесс завершается проходом, во время которого сортируются все элементы рассматриваемого массива в той последовательности, которая получена после последнего прохода, с помощью обычной сортировки.

В результате имеем упорядоченный массив. Каждый проход в данном методе будет использовать результаты предыдущего прохода.

В программе использован вариант, когда смещение задается четным числом, поэтому каждое следующее смещение определяется как деление на 2 предыдущего значения h. Уменьшение h производится до тех пор, пока не станет равным 0. Значение вспомогательной переменной hs характеризуется двумя значениями 0 или 1, если обмен состоялся, то hs = 1.

В алгоритме (рис. 4.12) смещение задано как h = N\2 (целочисленное деление). Чтобы изменить вид последовательности смещений, можно задать конкретную последовательность или формулу, вычисляющую h.

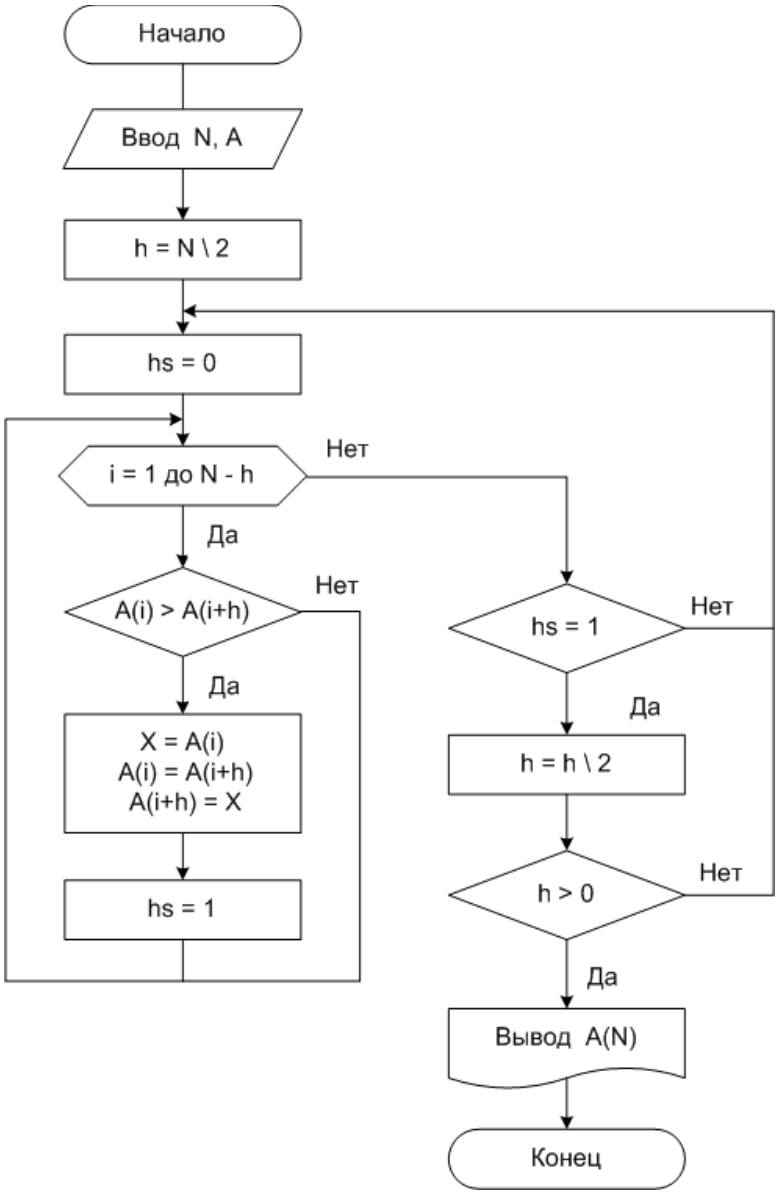


Рис. 4.12. Сортировка Шелла

Программа для сортировки методом Шелла имеет следующий вид (рис. 4.13):

```
# Сортировка методом Шелла
import random
n = int(input("Введите длину массива: "))
a = []
```

```
for i in range(n):
    a.append(random.randint(0,10))
print("Массив чисел: ", a)

h = n//2
while h > 0:
    for i in range(h, n):
        j = i
        x = a[i]
        while (a[j-h] > x) and (j >= x):
            a[j] = a[j-h]
            j -= h
            a[j] = x
    h //= 2

print("Отсортированный массив: ", a)
```

```
Введите длину массива: 10
Массив чисел:  [0, 10, 5, 8, 8, 8, 10, 4, 10, 3]
Отсортированный массив:  [0, 3, 4, 10, 5, 8, 10, 8, 8, 10]
```

Рис. 4.13. Пример сортировки данных методом Шелла

Данный метод достаточно легко программируется. Он эффективен при умеренно больших N ($N \le 1000$). В работе [16] рекомендована последовательность приращений 1, 4, 13, 40, 121, …, где $h_{k-1} = 3h_k + 1$, $h_t = 1$ и $t = [\log_3 n] - 1$ и последовательность 1, 3, 7, 15, 31, …, где $h_{k-1} = 2h_k + 1$, $h_t = 1$ и $t = [\log_2 n] - 1$. В последнем случае затраты, которые потребуются для сортировки N элементов методом Шелла, гарантированно пропорциональны N.



Вопросы для самопроверки

Составить алгоритмы для решения следующих задач:

1. Что называют алгоритмом?
1. Даны пять попарно различных целых числа a, b, c, d, e. Упорядочить их по возрастанию, используя для этого не более 7 сравнений.
2. Даны действительные числа $a_1, \ldots, a_n$. Получить попарно различные целые числа $j_1, j_2, \ldots, j_n$, такие, что $1 \le j_k \le \ldots \le n$, $k = 1, \ldots, n$ и $a_{j1} \ge a_{j2} \ge \ldots \ge a_{jn}$.
3. Дан массив действительных чисел $a_1, \ldots, a_n$. Необходимо переставить элементы этого массива так, чтобы после перестановки они были упорядочены по возрастанию. Найти элемент массива, имеющий наименьшее значение, переставить его с первым элементом, затем проделать то же самое, начав со второго и т.д.
4. Даны целые числа $a_1, \ldots, a_n$. Необходимо переставить элементы этого массива так, чтобы после перестановки они были упорядочены по возрастанию. Последовательным просмотром чисел $a_i, \ldots, a_n$ найти наименьшее i такое, что $a_i > a_{i+1}$. Поменять a_i и a_{i+1} местами, возобновить просмотр с элемента a_{i+1} и т.д. Тем самым наибольшее число передвинется на первое место. Следующие просмотры начинать опять сначала, уменьшая на одно количество просматриваемых элементов. Массив будет упорядочен после просмотра, в котором участвовали только первый и второй элементы.
5. Дан массив действительных чисел $a_1, \ldots, a_n$. Необходимо переставить элементы этого массива так, чтобы после перестановки они были упорядочены по возрастанию. Просматривать последовательно числа $a_2, \ldots, a_n$ и каждый новый элемент a_i вставлять на подходящее место в уже упорядоченную совокупность $a_i, \ldots, a_{i-1}$. Это место определяется последовательным сравнением a_i с упорядоченными элементами $a_i, \ldots, a_{i-1}$.
6. Даны действительные числа $a_1, \ldots, a_n$, p, натуральное число k ($a_1 \le a_2 \le \ldots \le a_n$, $k \le n$). Удалить из $a_1, \ldots, a_n$ элемент с номером k

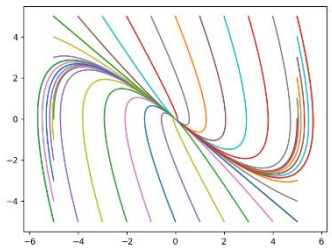
(т.е. a_k) и вставить элемент равный p так, чтобы не нарушалась последовательность.

7. Даны натуральное число n, целые числа a_1, …, a_n. Найти наибольшее значение, которое встречается в последовательности a_1, … , a_n после выбрасывания из нее:
   а) одного из членов со значением $\max(a_1, \ldots, a_n)$;
   б) всех членов со значением $\max(a_1, \ldots, a_n)$ (предполагается, что не все числа равны между собой).
8. Даны два упорядоченных по возрастанию массива A(n) и B(m). Получить слиянием новый массив C(n + m), в котором упорядочить элементы по убыванию.
9. Даны два неупорядоченных массива A(n) и B(m). Получить слиянием новый массив C(n + m), в котором необходимо упорядочить элементы по возрастанию. Совпадающие элементы оставить по одному. Определить новое количество в массиве C.
10. Упорядочить по возрастанию каждую строку матрицы B(M). Далее упорядочить всю матрицу по возрастанию элементов первого столбца перестановкой строк.
11. Даны два массива возрастающий X(n) и убывающий Y(m). Объединить оба массива в один Z, который упорядочить по возрастанию. Удалить из нового массива одинаковые элементы. Определить, сколько элементов в новом массиве.
12. Элементы массива QW(k) случайно перемешаны. В массиве M(k) хранятся номера массива QW(k) до перемешивания. Восстановить исходное состояние массива QW(k).
13. Перенести в начало массива S(k) все отрицательные элементы по возрастанию, далее нулевые и затем положительные также по возрастанию.
14. Дана матрица T(m), элементы которой – строковые переменные. Упорядочить строки матрицы по возрастанию.
15. Даны две матрицы X(n, m) и Y(n, m), строки которых упорядочены по алфавиту. Выполнить построчное слияние матриц.
16. Даны две матрицы X(n, m) и Y(k, m), строки которых упорядочены по алфавиту. Выполнить построчное слияние матриц.
17. В матрице QW(n, m) переставить элементы так, чтобы на главной диагонали и линиях, параллельных ей, стояли возрастающие последовательности, причем перестановка элементов возможна только в пределах этих линий.

18. Дана матрица натуральных чисел X(n, m). Упорядочить строки матрицы по возрастанию суммы цифр в десятичной системе счисления, используемых для записи элементов строки.
19. В матрице QW(n, m), где n = m, переставить строки и столбцы так, чтобы элементы главной диагонали образовали убывающую последовательность.
20. Дана матрица натуральных чисел Y(n, m), в которой столбцы упорядочены по убыванию. Переместить содержимое матрицы в массив S(k) по столбцам и упорядочить по убыванию, удалив одинаковые элементы. Определить новое количество элементов.
21. Дана матрица натуральных чисел Y(n, m), в которой столбцы упорядочены по убыванию. Вычислить сумму элементов каждого столбца и из полученных сумм получить новый массив S(k), который упорядочить по убыванию.
22. Дана матрица натуральных чисел Y(n, m), в которой строки упорядочены по возрастанию. Вычислить сумму элементов каждой строки и из полученных сумм получить новый массив S(k), который упорядочить по возрастанию.

Глава 5

Численное решение уравнений



Метод решения хорош, если с самого начала мы можем предвидеть – и далее подтвердить это, что следуя этому методу, мы достигнем цели…
Г. Лейбниц

В данном разделе будут рассмотрены алгоритмы нахождения корней нелинейных уравнений известными численными методами. Многие задачи моделирования можно свести к решению уравнений, которые обычно представляют самостоятельную задачу. Выбор подходящего алгоритма для решения уравнения, как правило, зависит от характера рассматриваемой задачи.

Обычно нелинейные уравнения делят на *трансцендентные* и *алгебраические*.

> *Уравнения, содержащие тригонометрические или другие специальные функции,* например, *lnx* или e^x*, называют трансцендентными.*

В общем случае нелинейное уравнение можно записать в виде

$$f(x) = 0, \qquad (5.1)$$

где f(x) – функция, которая определена и непрерывна на интервале [a, b].

Корнем уравнения f(x) = 0 называется всякое число x*, при котором f(x*) = 0.

Требуется найти корни уравнения (5.1), т.е. те значения x, которые обращают уравнение в тождество. В процессе приближенного отыскания корней уравнения обычно можно выделить два этапа:

- отделение корня;
- уточнение корня.

Отделение корня – нахождение узких промежутков, которые могут содержать только один корень. Одна из точек этого промежутка принимается за начальное приближение корня. Фактически на этом этапе определяется грубое значение корня. В общем случае этап отделения корня трудно алгоритмизировать.

Для второго этапа *уточнения корня* используются прямые и итерационные методы. При прямом методе корень уравнения может быть найден за конечное заранее известное число итераций, если известна формула для нахождения корней. Прямые методы позволяют решить некоторые простые алгебраические и тригонометрические уравнения. Примером прямого решения может служить решение квадратного уравнения.

В итерационных методах процедура решения задается в виде многократного применения некоторого алгоритма. Полученное решение всегда является приближенным, даже если будет близким к точному значению корня. Итерационные методы позволяют определить корень x как предел некоторой последовательности x_0, x_1, …, x_n, и решение не может быть достигнуто за конечное заранее известное число итераций. Основные методы – это итерационные: метод дихотомии, метод простой итерации, метод Ньютона. Основной характеристикой этих методов является скорость сходимости итерационного процесса. Алгоритм построения итерационной последовательности, который применяется для уточнения приближенного значения корня, должен быть простым и эффективным.

Графическая иллюстрация метода показана на рис. 5.1, схема алгоритма – на рис. 5.2.

Метод половинного деления (или дихотомии)

Пусть действительный корень уравнения f(x) = 0 отделен, и функция f(x) непрерывна на интервале [a, b] отделения корня. Построим процесс сужения интервала [a, b] так, чтобы искомый корень всегда находился внутри суженого интервала.

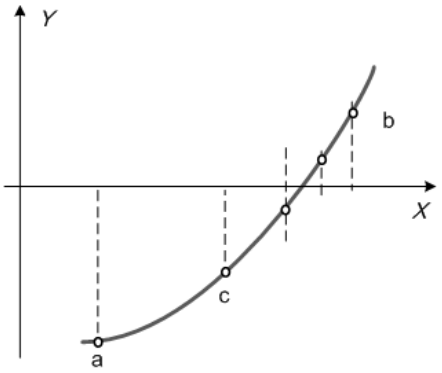


Рис. 5.1. Метод половинного деления

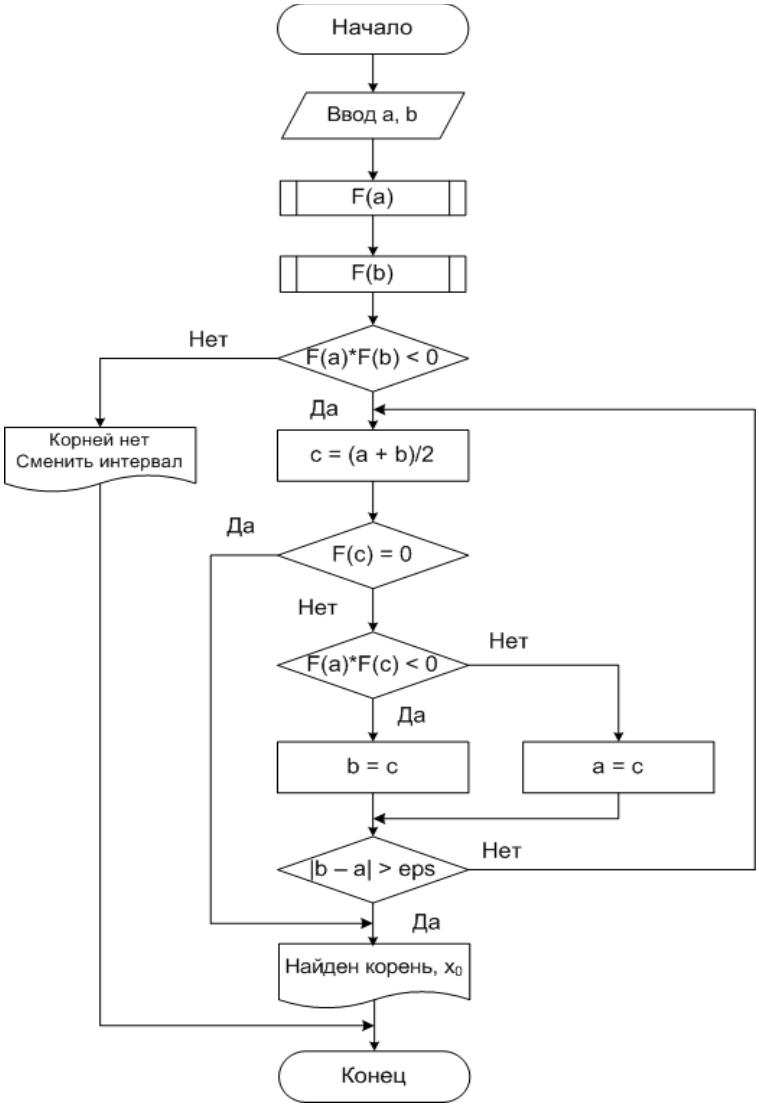


Рис. 5.2. Алгоритм метода половинного деления

Очевидно, что в этом случае погрешность приближенного значения корня не превышает $|b_k - a_k|$, где a_k, b_k – граничные точки интервала на k-й итерации. Найдем середину отрезка $x_1 = (a + b)/2$, вычислим в этой точке $f(a + b)/2$ и составим следующие произведения: $f(a) \times (f(a + b)/2)$ и $f(b) \times (f(a + b)/2)$. Из двух половин отрезков выберем тот, в котором произведение является отрицательной величиной (< 0), и обозначим новые границы отрезка через переменные a_1, b_1.

Далее новый отрезок разделим пополам, вновь составим аналогичные произведения и выберем тот из отрезков, в котором произведение – величина отрицательная. Погрешность метода определяется очевидным соотношением $\varepsilon_k = |x^* - x_k| \le |a_k - b_k| \le |b - a|/2^k$, которое указывает на скорость сходимости метода. С увеличением k погрешность стремится к 0.

Методом половинного деления (рис. 5.3) найти корень уравнения $F(x) = x4 + 2x - x - 1 = 0$, лежащий в интервале [0.5; 1.2].

Программный код метода половинного деления приведен ниже:



```
def f(x):
    f = (x**4)+2*(x**3)-x-1
    return f

a = float(input('Ввести начальное значение интервала a: '))
b = float(input('Ввести начальное значение интервала b: '))
eps = float(input('Ввести точность вычислений eps: '))

print('a = {0}, b = {1}, eps = {2}'.format(a,b,eps))

if f(a)*f(b) < 0:
    while abs(b-a) > eps:
        c = (a+b)/2
        if f(c) == 0:
            print('Точное значение корня: ', c)
        else:
            if f(a)*f(c) < 0:
                b = c
            else:
```

```
                a = c
            print('Результат вычислений: значение корня =
{0},\nТочность вычислений = {1}'.format(c,eps))
    else:
        print('Значение корня не найдено или решение не
единственное')
```

```
Ввести начальное значение интервала a: 0.5
Ввести начальное значение интервала b: 1.1
Ввести точность вычислений eps: 0.01
a = 0.5, b = 1.1, eps = 0.01
Результат вычислений: значение корня = 0.8,
Точность вычислений = 0.01
Результат вычислений: значение корня = 0.9500000000000001,
Точность вычислений = 0.01
Результат вычислений: значение корня = 0.875,
Точность вычислений = 0.01
Результат вычислений: значение корня = 0.8375,
Точность вычислений = 0.01
Результат вычислений: значение корня = 0.85625,
Точность вычислений = 0.01
Результат вычислений: значение корня = 0.865625,
Точность вычислений = 0.01
```

Рис. 5.3. Пример реализации алгоритма половинного деления

Хотя метод половинного деления не обладает высокой эффективностью, но с увеличением числа итераций он позволяет найти более точное приближение значение корня. Данный метод достаточно прост, но медленно сходится. После того, как найден интервал, в котором заключен корень, его ширина после N итераций убывает в 2^N раз.

Метод простой итерации

Метод простой итерации, или метод повторных подстановок, является общим методом решения уравнений. Как правило, хорошая сходимость позволяет применять его достаточно часто в практике

вычислений. При использовании метода простой итерации для уточнения корня уравнение (5.1) заменяется эквивалентным уравнением

$$x = \varphi(x). \qquad (5.2)$$

Соответствующая итерационная формула имеет вид: $x_{n+1} = \varphi(x_n)$. Сходимость метода итерации имеет линейный характер. Графически на этапе отделения корня уравнение $\varphi(x) = x$ можно представить в виде двух пересекающихся линий $y = \varphi(x)$ и $y = x$ (рис. 5.4).

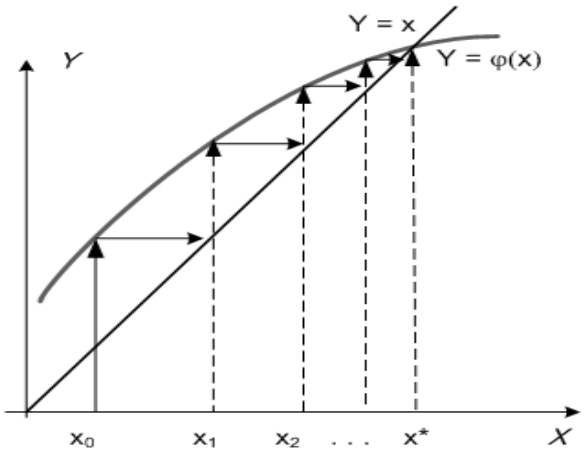


Рис. 5.4. Метод простой итерации

Каждый действительный корень *x** уравнения $\varphi(x) = x$ является абсциссой точки пересечения кривой $y = \varphi(x)$ с прямой $y = x$. Схема алгоритма представлена на рис. 5.5.

Приведенная ниже программа метода простой итерации (рис. 5.6) позволяет найти корни уравнения $f(x) = x + \ln x$ на интервале [0,5; 0,9] с точностью eps = 10^{-6}. В качестве первого приближения корня можно взять произвольное число x_1 из некоторой окрестности корня уравнения $x_2 = \varphi(x_1)$, затем $x_3 = \varphi(x_2)$, …., $x_n = \varphi(x_{n-1})$.

Для нахождения приближенного значения корня с погрешностью ε достаточно взять число итераций n, при котором будет выполняться неравенство $|x_n - x_{n-1}| < \frac{1-q}{q}\varepsilon$, если $q = \max|\varphi'(x)| < 1$. Прекращение процесса происходит по критерию $|x_n - x_{n-1}| < \varepsilon$, где ε – заданная точность вычислений, n = 0, 1, 2, ...

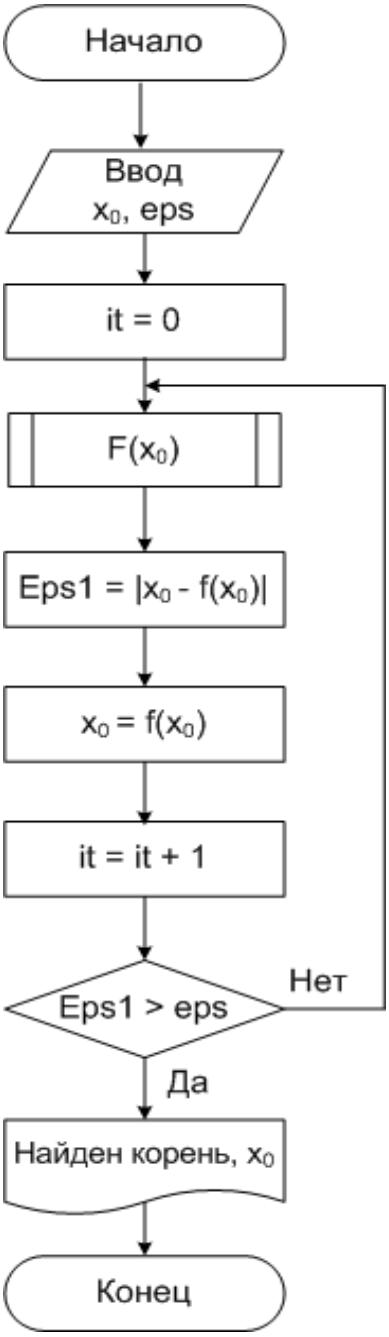


Рис. 5.5. Схема алгоритма простой итерации

Функцию $\varphi(x) = x$ найдем из соотношения $\varphi(x) = x - f(x)/q$, тогда $\varphi(x) = x - (x + \ln x)/q$, далее определяем q, как было сказано выше:

$$q = \max|f'(x)| = \max\left|1 + \frac{1}{x}\right| = 11.$$

В результате получаем формулу для φ(x) = x – (x + ln x)/11.



```
from math import log as ln
def f(x):
   fi = x-(x+ln(x))/11
   return fi

x0 = float(input('Ввести начальное значение интервала
x0: '))
eps = float(input('Ввести точность вычислений eps:
'))

print('x0 = {0}, eps = {1}'.format(x0, eps))
it = 0

while True:
   eps1 = abs(x0 - f(x0))
   x0 = f(x0)
   it += 1
   if (eps1 < eps):
       print('Значение x0 = {0}, число итераций =
{1}'.format(x0, it))
       break
```

```
Ввести начальное значение интервала x0: 0.5
Ввести точность вычислений eps: 0.0001
x0 = 0.5, eps = 0.0001
Значение x0 = 0.5668817871585079, число итераций = 19
```

Рис. 5.6. Результат реализации метода простой итерации

Простота метода простой итерации делает его привлекательным, но не следует забывать, что этот метод не всегда может обеспечить сходимость. Поэтому следует прекращать итерационный процесс, как только сходимость не обеспечивается.

Метод Ньютона

Метод Ньютона имеет широкое применение для построения итерационных алгоритмов. В отличие от предыдущих методов для определения интервала, в котором заключен корень, не требуется находить значение функции с противоположными знаками. Вместо интерполяции в методе Ньютона осуществляется экстраполяция с помощью касательной к кривой в данной точке. В связи с этим метод Ньютона называется также методом касательных. На рис. 5.7 показана графическая интерпретация метода, а на рис. 5.8 – схема алгоритма, в основе которого лежит разложение функции f(x) в ряд Тейлора:

$$f(x_n + h) = f(x_n) + hf'(x_n) + \frac{h^2}{2} f''(x_n) + \ldots \qquad (5.3)$$

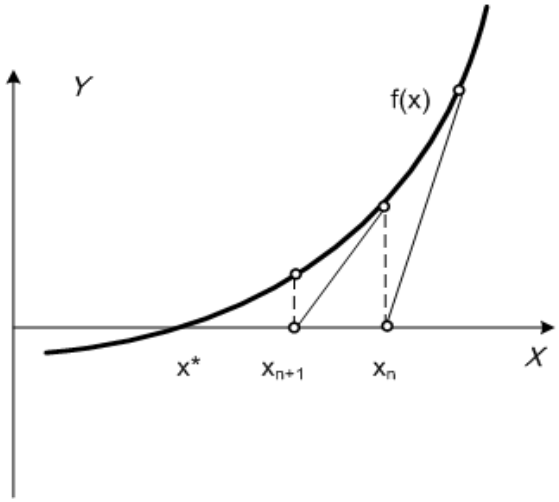


Рис. 5.7. Метод Ньютона

Члены, содержащие h во второй и в более высокой степени, отбрасываются; и при этом используется соотношение $x_n + h = x_{n+1}$. Предполагается, что переход от x_n к x_{n+1} приближает значение функции к нулю так, что $f(x_n + h) = 0$. Тогда имеем значение для x_{n+1} вида

$$x_{n+1} = x_n - f(x_n)/f'(x_n). \qquad (5.4)$$

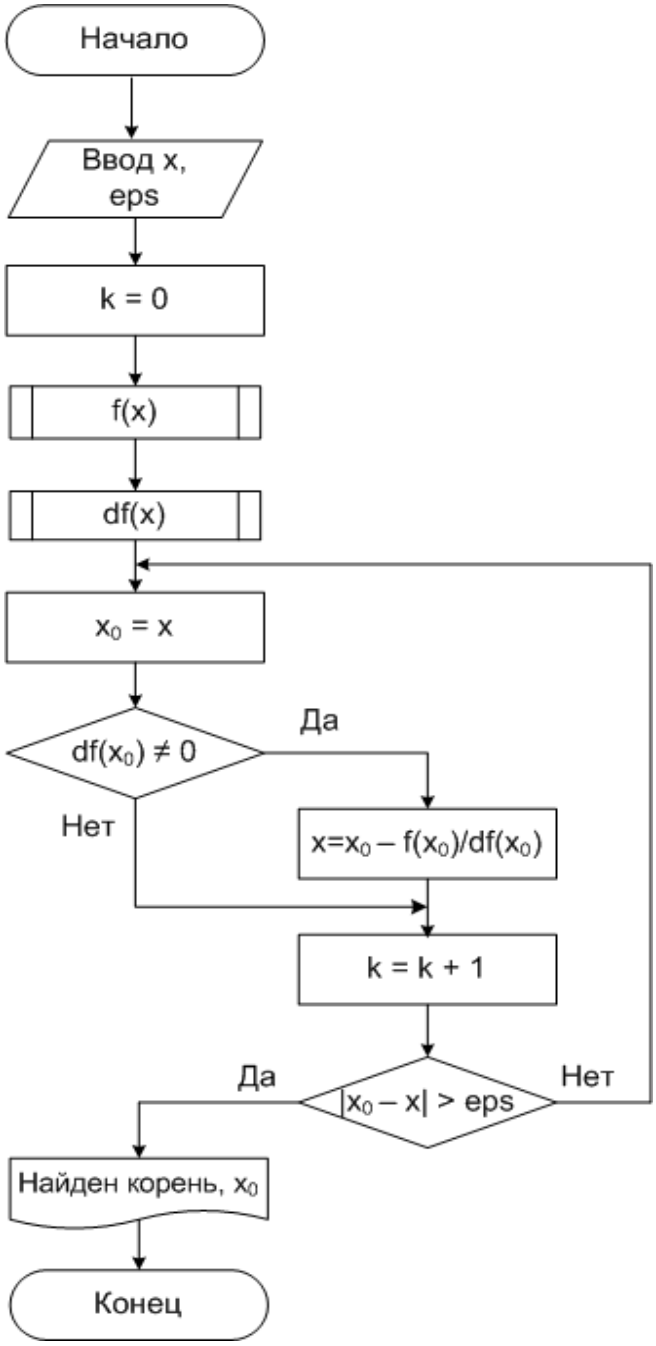


Рис. 5.8. Схема алгоритма метода Ньютона

Выражение (5.4) является основным соотношением метода Ньютона, так как позволяет построить последовательность приближений x_{n+1} к точному значению корня по заданному приближению x_0.

Геометрически процесс (рис. 5.7) означает замену на каждой итерации кривой y = f(x) на касательную к ней в точке $[x_n, f(x_n)]$ и определенного значения x_{n+1} как координат точки пересечения касательной и оси абсцисс. Другими словами, значение x_{n+1} соответствует точке, в которой касательная к кривой в точке x_n пересекает ось x. Кривая $f(x_n)$ отличается от прямой линии, поэтому значение $f(x_{n+1})$ не равно нулю. Далее процесс повторяется, последовательно приближаясь к x*.

Если вторая производная $f''(x)$ неизвестна, то поступают следующим образом. Задают x_0 как некоторое начальное приближение и вычисляют x_1 по формуле (5.4). Если выполняется условие $x_1 \le x^*$, то далее вычисляются приближения $x_2, x_3, \ldots, x_n$. Вычисления заканчиваются при достижении условия $|x_n - x_{n+1}| \le eps$, где x_n – корень уравнения.

Приведенная ниже программа находит корень уравнения методом касательных. Для ее успешной реализации необходимо вычисление функции f(x) и первой производной df. Это можно выполнить по аналогии с методом итерации, рассмотренным выше:



```
from math import log as ln

def f(x):
   fi = x-(x+ln(x))/11
   return fi

def df(x):
   df = (10*x-1)/10*x
   return df

x = float(input('Ввести начальное значение интервала
x: '))
eps = float(input('Ввести точность вычислений eps: '))

k = 0
print('k = {0}, x = {1}, eps = {2}'.format(k, x, eps))
```

```
while True:
   x0 = x
   if df(x0) != 0:
       x = x0 - f(x0)/df(x0)
       print('k = {0}, x = {1}, f(x) = {2}'.format(k,
x, f(x)))
   k += 1
   if abs(x0-x) > eps:
       print('Найден корень ', x0)
       break
```

Быстрота сходимости в большей мере зависит от выбора исходной точки. Достаточное условие сходимости метода Ньютона можно получить из соответствующего условия для метода простой итерации (рис. 5.9). Процесс сводится к определению точного значения корня, если начальное приближение выбрано так, что $f(x) \times f''(x) > 0$.

```
Ввести начальное значение интервала x: 15
Ввести точность вычислений eps: 0.0000013
k = 0, x = 15.0, eps = 1.3e-06
k = 0, x = 14.940088692373847, f(x) = 13.336076255225473
Найден корень  15.0
```

Рис. 5.9. Результат реализации метода Ньютона

Метод хорд

Один из недостатков метода Ньютона состоит в том, что, применяя его, приходится дифференцировать функцию f(x). Если нахождение производной вызывает затруднения, то можно ограничиться вычислением только значения f(x) и использовать некоторое приближение, что значительно упрощает алгоритм.

Метод хорд заключается в следующем. В отличие от предыдущего метода нелинейная функция f(x) на определенном интервале заменяется линейной функцией, в качестве которой берется хорда – прямая линия, которая соединяет концы данной функции. В точке пересечения этой прямой с осью абсцисс определяют значение функции. Если это значение меньше заданной погрешности функции,

то абсцисса точки пересечения принимается за корень функции f(x). На рис. 5.10 показана графическая интерпретация метода хорд.

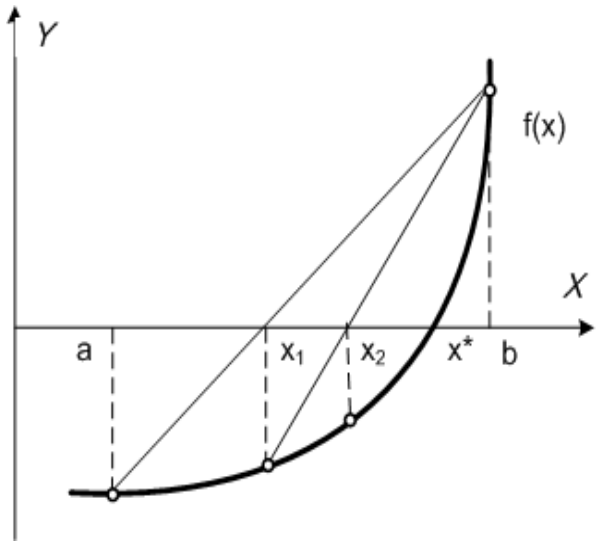


Рис. 5.10. Метод хорд

Заменим производную $f'(x)$, которая используется в методе Ньютона в выражении (5.4), разностью последовательных значений функции, отнесенной к разности значений аргумента $f'(x) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$. В результате получим итерационную формулу для метода хорд:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}. \qquad (5.5)$$

Пусть на некотором интервале [a, b] функция f(x) имеет корень. В этом случае будет справедливым следующее неравенство: f(a)f(b) < 0. Найдем значение x_1 из выражения $x_1 = a - \frac{(b-a)f(a)}{f(b) - f(a)}$.

Если справедливо условие $f(x_1)f(a) < 0$, то $b = x_1$, иначе $a = x_1$. В качестве x_1 принимается тот конец интервала [a, b], который остался после выбора. Каждое последующее x_k находится по формуле (5.5). Условие окончания процесса $|x_n - x^*| < \varepsilon$, где ε – заданная точность вычислений. Схема алгоритма рассматриваемого метода хорд показана на рис. 5.11.

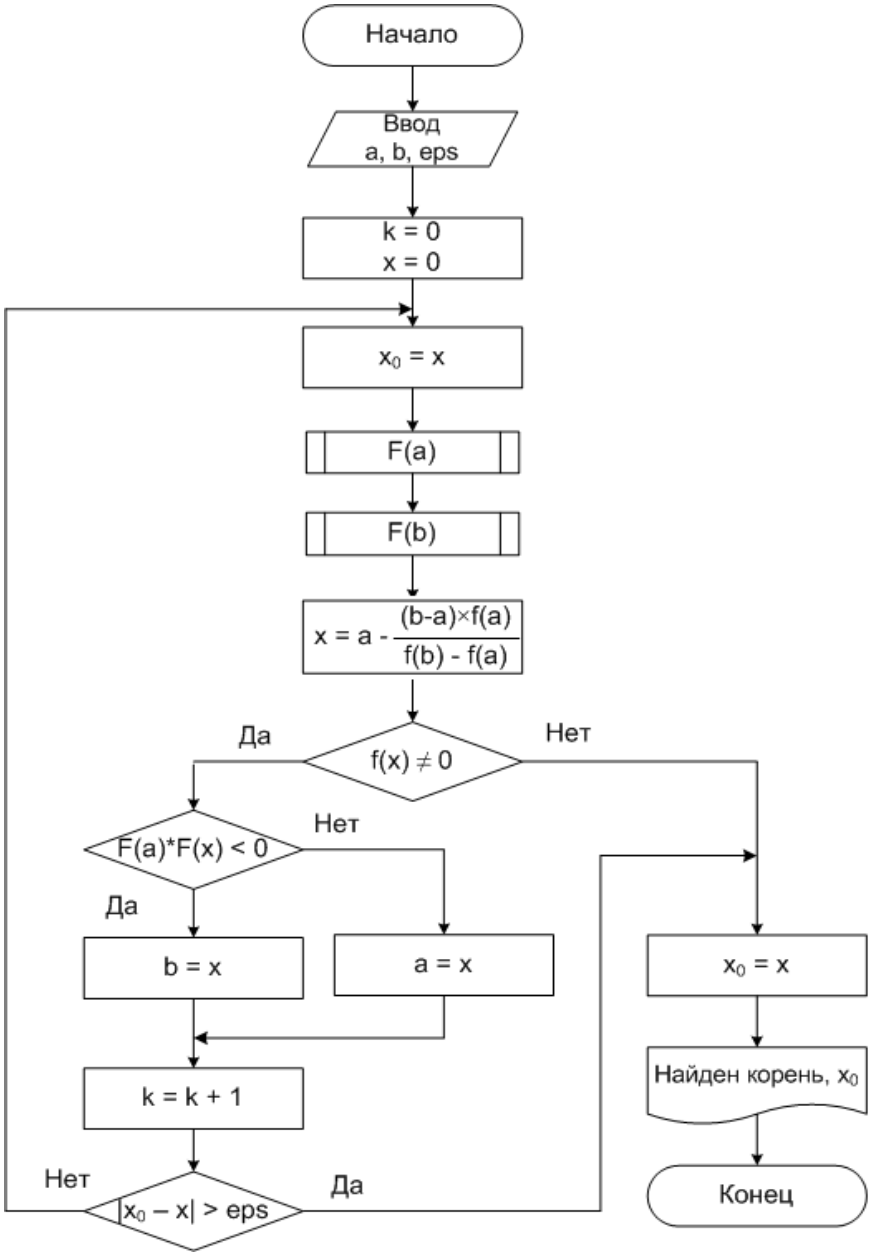


Рис. 5.11. Схема алгоритма метода хорд

Программа, реализующая алгоритм метода хорд, имеет следующий вид:

```
def f(x):
    f = x**4+2*x**3-x-1
    return f
```

```
a = float(input('Ввести начальное значение интервала
a: '))
b = float(input('Ввести конечное значение интервала
b: '))
eps = float(input('Ввести точность вычислений eps:
'))

k = 0
x = 0
x0 = x
print('a = {0}, b = {1}, eps = {2}'.format(a, b,
eps))

while abs(x0 - x) < eps:
   x0 = x
   x = a - ((b-a)-f(a))/(f(b)-f(a))
   if f(x) != 0:
       if f(a)*f(x) < 0:
           b = x
       else:
           a = x
       k += 1
   else:
       break
x0 = x
print("Найден корень уравнения ", x0)
print("Количество итераций  ", k)
```

```
Ввести начальное значение интервала a: 13
Ввести конечное значение интервала b: 2
Ввести точность вычислений eps: 0.001
a = 13.0, b = 2.0, eps = 0.001
Найден корень уравнения  11.998784637822071
Количество итераций   1
```

Рис. 5.12. Результат реализации метода хорд

Для реализации алгоритма (рис. 5.12) необходимо задать определенную функцию и оформить ее в виде дополнительной подпрограммы пользователя типа Function. В процессе работы программы должен быть задан интервал [a, b] для описанной функции и точность вычислений.



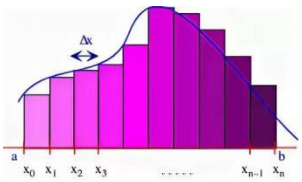
Вопросы для самопроверки

Известными методами найдите корень уравнения с погрешностью eps = 10^{-5} на отрезке [a, b]:

1.	$5x - 8\ln x = 8$	[3,5; 4,0]
2.	$3x - e^x = 0$	[0,5; 1,0]
3.	$e^{-x-2} + x = 0$	[0; 1,5]
4.	$e^x + e^{-3x} = 4$	[1,0; 1,5]
5.	$2^x - 4x = 0$	[0; 0,5]
6.	$2^x - 2{,}2x = 0$	[0,5; 1,0]
7.	$2x - \ln x - 7 = 0$	[4; 4,5]
8.	$x - \sin x = 0{,}25$	[1,0; 1,5]
9.	$2\sqrt{x} - \cos(\pi x/2) = 0$	[0; 0,5]
10.	$\sqrt{x} - \cos(0{,}387x) = 0$	[0,5; 1,0]
11.	$3x + \cos x + 1 = 0$	[–1,0; 0,0]
12.	$x + 0{,}5\sin x - 2 = 0$	[1,0; 1,5]
13.	$2{,}7x - \ln x = 4/8$	[2,0; 2,5]
14.	$x^3 + 2\sin x - 2 = 0$	[0,5; 1,0]
15.	$e^x - 6x - 3 = 0$	[–0,5; 0,0]
16.	$2x - 1{,}3^x = 0$	[0,5; 1,0]
17.	$\ln x + (x+1)^x = 0$	[0,1; 0,5]
18.	$x\,2^x = 1$	[0,5; 1,0]
19.	$\sqrt{x} + 1 = 1/x$	[0,5; 1,0]
20.	$x - \cos x = 0$	[0,5; 1,0]
21.	$3x - \cos x - 1 = 0$	[0,5; 1,0]
22.	$4\sqrt{x} = \sin(\pi x/6)$	[0,0; 0,5]
23.	$x - 0{,}21\sin(0{,}5 + x) = 0$	[0,0; 0,5]

24.	$x\ln x = 100$	[29,5; 30,0]
25.	$2 - x\ln x$	[1,5; 2,0]
26.	$(x - 10) - (1/2)\, e^x = 0$	[0,0; 0,5]
27.	$2 - x\, e^e = 0$	[0,5; 1,0]
28.	$2{,}2x + 2^x = 0$	[–0,5; 0,0]
29.	$x + \sin x = 0$	[–2,0; –1,5]
30.	$x - \cos x = 0$	[0,5; 1,0]

Глава 6



Алгоритмы для численного интегрирования и решения дифференциальных уравнений

На практике при моделировании различных технических задач не всегда удается найти нужную формулу для вычисления значения определенного интеграла или же она получается очень сложной. В этих случаях приходится прибегать к методам численного интегрирования.

Как известно, в основе численного интегрирования лежит приближенное вычисление площади под кривой, описываемой подынтегральной функцией.

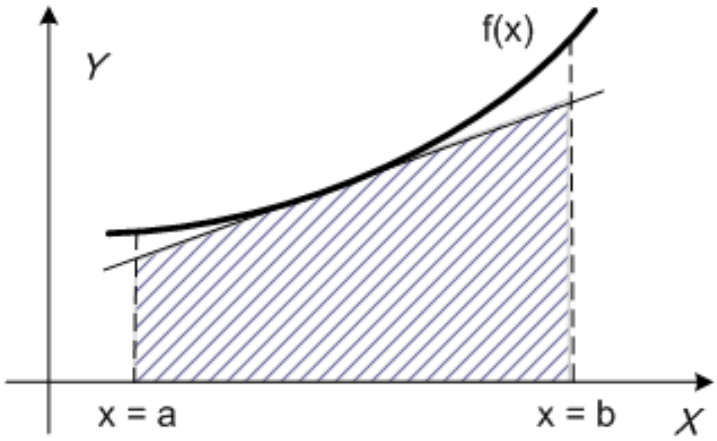


Рис. 6.1. Геометрическое представление определенного интеграла

Таким образом, вычисление определенного интеграла с геометрической точки зрения сводится к вычислению площади криволинейной трапеции, ограниченной графиком функции f(x), осью X и прямыми линиями X = a и X = b (рис. 6.1). Интеграл вида

$$S = \int_a^b f(x)\,dx \qquad (6.1)$$

на рис. 6.1 S представляет собой заштрихованную площадь, f(x) – непрерывная функция от x на всем интервале $a \le x \le b$.

Обычно при интегрировании подынтегральная функция заменяется более простой. Чем меньше интервалы интегрирования, тем точнее можно вычислить интеграл. Поэтому исходный интервал [a, b] делят на несколько более мелких, на каждом из них применяют соответствующую формулу интегрирования и затем суммируют результаты. Как правило, погрешность в таких случаях определяется путем двойного интегрирования и разницей вычислений значений интегралов.

Метод прямоугольников

Приближенное значение интеграла можно вычислить следующим образом. Функцию f(x) на отрезке [a, b] можно заменить константой $f(x) \approx f(c)$. Другими словами, площадь трапеции надо заменить площадью прямоугольника с основанием b – a и высотой f(c). Поэтому следующая формула называется формулой прямоугольников (рис. 6.2):

$$S = \int_a^b f(x)\,dx \approx (b - a)\, f(c). \qquad (6.2)$$

Если выбрать в качестве исходной точки c левый конец отрезка [a, b], т.е. c = a, выходит формула левых прямоугольников:

$$S = \int_a^b f(x)\,dx \approx (b - a)\, f(a). \qquad (6.3)$$

Если принять c = b, или (a + b)/2, то результатом будут формулы соответственно правых и срединных (средних или центральных) прямоугольников:

$$S \approx (b - a)\, f(c);$$
$$S \approx (b - a)\, f((a + b)/2). \qquad (6.4).$$

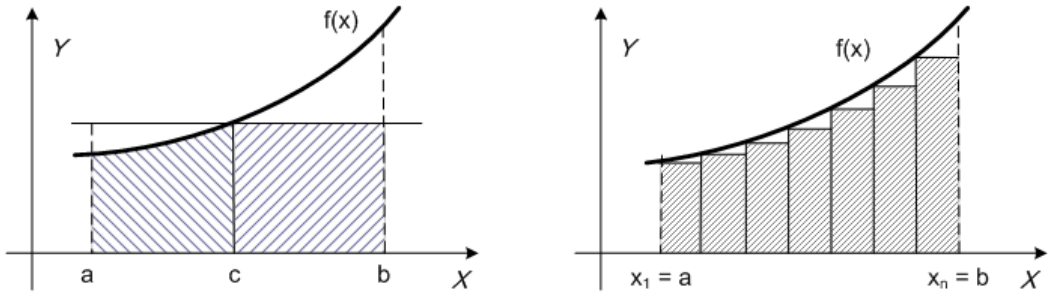


Рис. 6.2. Графическая интерпретация метода прямоугольников

Погрешность полученных формул достаточно велика. Для уменьшения погрешности интегрирования отрезок [a, b] делим на n равных частей длины h = (b – a) / n и применим к нему одну из полученных выше формул. Например, используя формулу левых прямоугольников (рис. 6.2) к отрезку $[X_i; X_{i+1}]$, где i = 0, 1, 2, 3, …, n – 1 получим

$$S \approx (X_1 - X_0)\, f(X_0) + (X_2 - X_1)\, f(X_1) + \ldots + (X_n - X_{n-1})\, f(X_{n-1}), \quad (6.5)$$

где R_n – погрешность метода прямоугольников. Если учесть, что $X_i - X_{i-1} = h$, то приближенное значение величины S можно преобразовать к виду

$$S \approx h\, f(X_0) + h\, f(X_1) + \ldots + h\, f(X_{n-1}) = h \sum_{i=0}^{n-1} f(X_i). \quad (6.6)$$

Эта формула называется общей формулой левых прямоугольников. Аналогичные формулы можно получить для правых и средних прямоугольников. Погрешность метода определяется величиной R_n:

Погрешность для левых прямоугольников $R_1 = \frac{b-a}{2} h\, |f'(x)|$,

для средних прямоугольников $R_2 \le \frac{b-a}{24} h^2\, |f''(x)|$, где $a \le x \le b$.

Как видно из приведенных формул, вычислить погрешность достаточно сложно – необходимо предварительно вычислить значение первой и второй производных. В связи с этим на практике, как правило, значение интеграла с заданной точностью можно достичь, если абсолютная величина разности двух последовательных приближений результата, которые были получены при разных значениях n, не превышает заданной погрешности.

Алгоритм для вычисления по формуле средних прямоугольников представлен на рис. 6.3. В алгоритме и программе использованы следующие обозначения:

n – количество отрезков/ интервалов интегрирования;

dx = (b – a) /n – отрезок (b – a) делится на n отрезков;

S1 – сумма площадей n прямоугольников с основанием dx/n;

S2 – сумма площадей 2n прямоугольников с основанием dx/(2n).

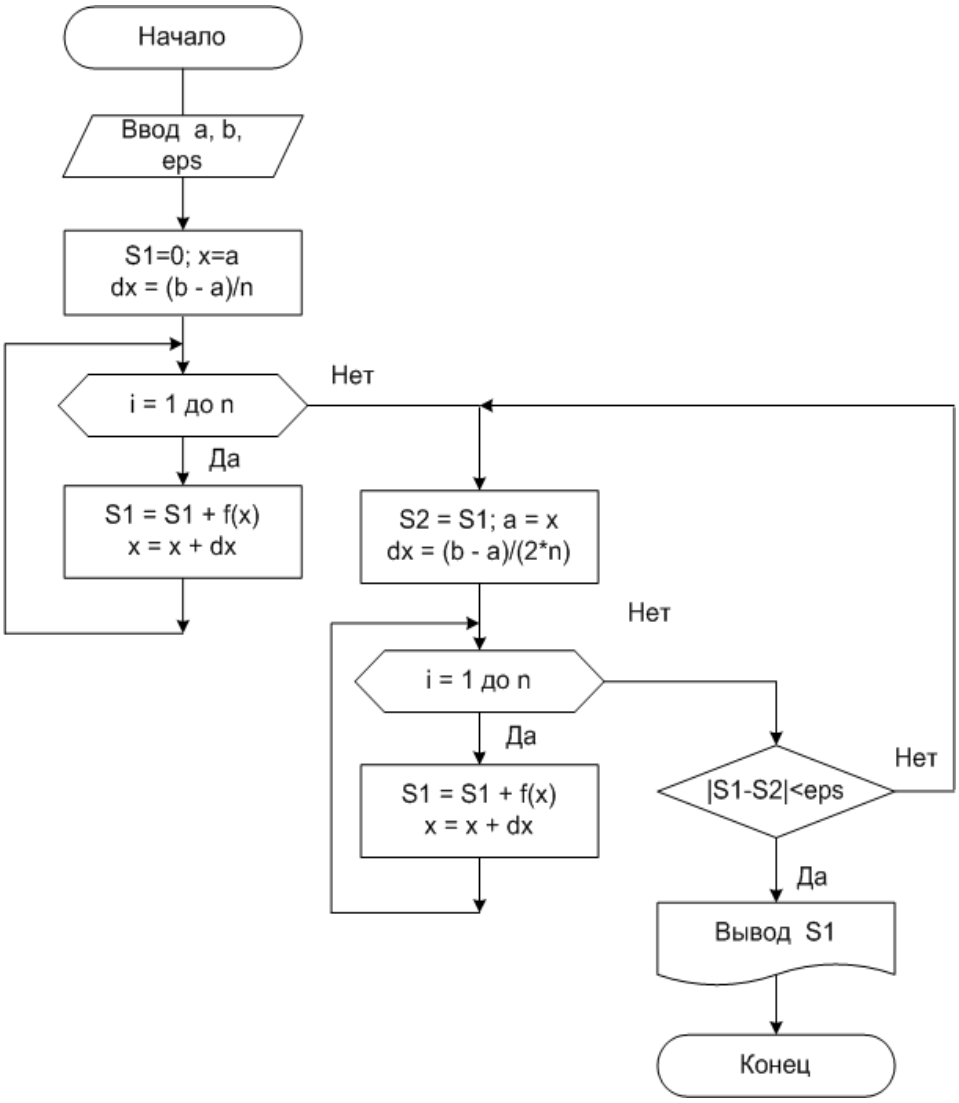


Рис. 6.3. Вычисление интеграла методом прямоугольника

Ниже приведена программа, которая вычисляет интеграл методом средних прямоугольников. В качестве тестового примера используется вычисление интеграла вида $\int_a^b \frac{dx}{x}$.



```
def f(x):
    if x != 0:
        f = 1/x
    else:
        print("Значение интервала для данной функции не
может быть равным 0")
    return f

a = float(input('Ввести начальное значение интервала
a: '))
b = float(input('Ввести конечное значение интервала b:
'))
eps = float(input('Ввести точность вычислений eps: '))
print('a = {0}, b = {1}, eps = {2}'.format(a, b, eps))

s1 = 0
x = a
n = 5
dx = (b-a)/n

for i in range(n):
    s1 += f(x)
    x += dx

s2 = s1
a = x
dx = (b-a)/(2*n)

while True:
    if abs(s1-s2) < eps:
```

```
            print("Результат вычислений ", s1)
            break
        for i in range(n):
            s1 += f(x)
            x += dx
```

Для реализации программы необходимо ввести значения a, b и значение точности вычислений – eps (рис. 6.4). Однако не стоит задавать слишком малое значение eps. Известно, что метод прямоугольников имеет значительную погрешность, поэтому при малых значениях eps процесс вычисления может затянуться или просто вызвать зацикливание.

```
Ввести начальное значение интервала a: 2
Ввести конечное значение интервала b: 5
Ввести точность вычислений eps: 0.00001
a = 2.0, b = 5.0, eps = 1e-05
Результат вычислений  1.687546006624954
```

Рис. 6.4. Результат вычисления интеграла методом прямоугольника

Метод трапеций

Теперь рассмотрим приближенное вычисление определенного интеграла $S = \int_a^b f(x)\,dx$, используя формулу трапеций. По сравнению с методом прямоугольников метод трапеций является более точным, так как трапеция более точно заменяет криволинейную трапецию, чем прямоугольник. Разобьем интервал интегрирования на n равных частей, каждый длиной h = (b – a)/n. Рассмотрим один из этих интервалов (рис. 6.5).

Площадь, лежащая под кривой y = f(x) между x_i и x_{i+1}, равна

$$S_i = \int_{x_i}^{x_{i+1}} f(x)\,dx \qquad (6.7).$$

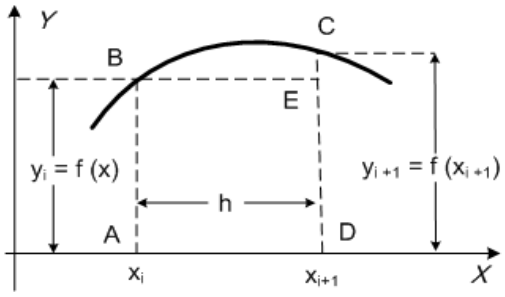


Рис. 6.5. Метод трапеций

Если h достаточно мало, то площадь без большой ошибки можно приравнять к площади трапеции ABCD (рис. 6.5).

Если $y_i = f(x)$, то площадь треугольника $S_{BEC} = (y_{i+1} - y_i)h/2$, то в результате имеем площадь трапеции равной $S_i = ½(y_{i+1} + y_i)h$. Но поскольку $\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx$, то получаем

$$S = \sum_{i=0}^{n-1} S_i, \qquad (6.8)$$

где $x_0 = a$ и $x_n = b$. Теперь подставим формулу (6.8) в (6.7) и получим формулу для вычисления определенного интеграла по формуле трапеций:

$$S = \int_a^b f(x)\,dx = h/2(y_0 + 2y_1 + \ldots + 2y_{n-1} + y_n) + R_n. \qquad (6.9)$$

Формула (6.9) называется квадратурной формулой трапеций. Геометрически (рис. 6.6) означает замену кривой y = f(x) на ломаную линию. При использовании формулы (6.9) необходимо правильно выбрать h, так как возникает ошибка, равная сумме площадей между кривой y = f(x) и хордой BC.

Погрешность $R_n = \int_a^b y\,dx - \frac{h}{2}[2\sum_{i=0}^{n} y_i + (y_0 + y_n)]$ – остаточный член формулы трапеций. Известно, что если функция y = f(x) имеет непрерывную вторую производную f″(x) на отрезке [a, b], то остаточный член равен $|R_n| = \max|f''(x)|(b-a)/12$, если $a \le x \le b$.

На рис. 6.7 показана схема алгоритма метода трапеций. Программная реализация алгоритма вычисления определенного интеграла методом трапеций представлена ниже (рис. 6.8).

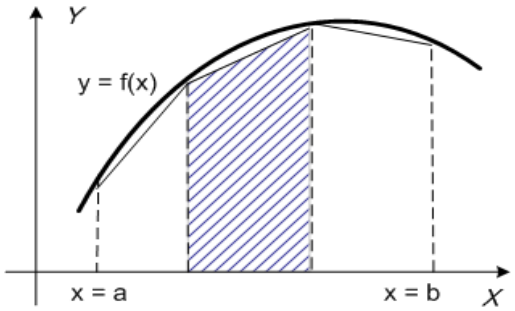


Рис. 6.6. Графическая интерпретация метода трапеций

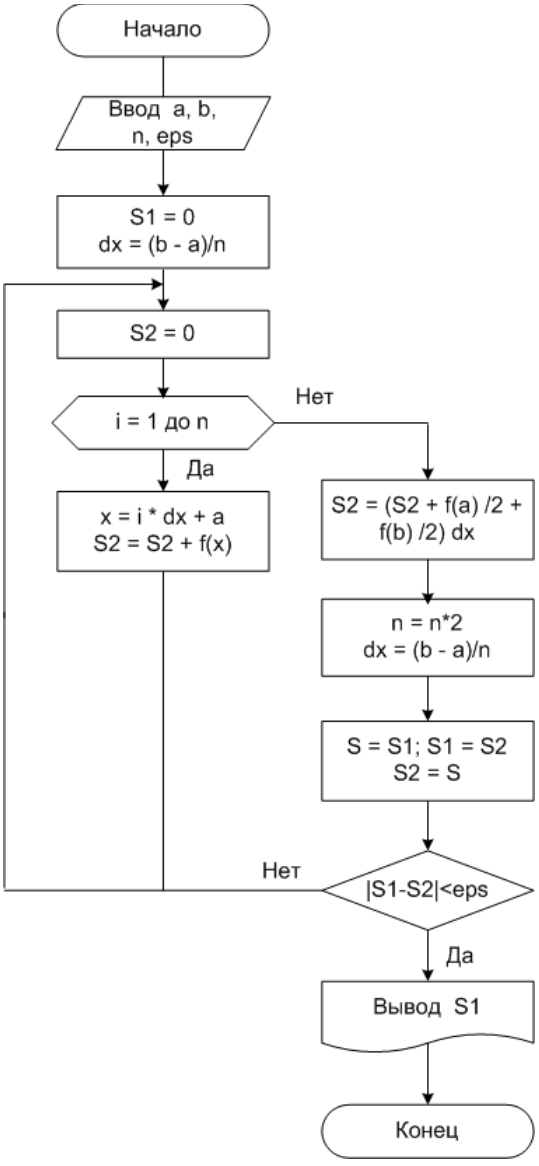


Рис. 6.7. Схема алгоритма метода трапеций



```
    def f(x):
        if x != 0:
            f = 1/x**2
        else:
            print("Значение интервала для данной функции не
может быть равным 0")
        return f

    a = float(input('Ввести начальное значение интервала
a: '))
    b = float(input('Ввести конечное значение интервала b:
'))
    n = float(input('Ввести n: '))
    eps = float(input('Ввести точность вычислений eps: '))
    print('a = {0}, b = {1}, n = {2}, eps = {3}'.format(a,
b, n, eps))

    s1 = 0
    dx = (b-a)/n

    while True:
        s2 = 0
        for i in range(int(n)):
            x = i*dx+a
            s2 += f(x)
        s2 = (s2+f(a)/2+f(b)/2)*dx
        n = n*2
        dx = (b-a)/n
        s = s1
        s1 = s2
        s2 = s
        if abs(s1-s2) < eps:
            print("Результат вычислений s1 = ", s1)
            break
```

```
Ввести начальное значение интервала a: 2
Ввести конечное значение интервала b: 5
Ввести n: 5
Ввести точность вычислений eps: 0.00001
a = 2.0, b = 5.0, n = 5.0, eps = 1e-05
Результат вычислений s1 =  0.3000091552995874
```

Рис. 6.8. Результат вычисления интеграла методом трапеций

Правило трапеций – один из простейших методов численного интегрирования. Он демонстрирует основной принцип приближенного численного интегрирования. Двойной просчет позволяет организовать подбор шага интегрирования (число разбиений n) для обеспечения заданной погрешности интегрирования. Для этого последовательно шаг удваивается, а погрешность контролируется.

Метод Симпсона

Более точную формулу получим, если профиль криволинейной полоски считать параболическим. Рассмотрим вертикальную полоску (рис. 6.9), которая ограничена непрерывной кривой y = f(x), осью x (y = 0) и двумя вертикальными линиями x = –h, x = h. Геометрический смысл метода Симпсона заключается в том, что при малом значении h кривую y = f(x) можно заменить параболой[8], которая проходит через точки A(–h, y_{-1}), B(0, y_0), C(h, y_1).

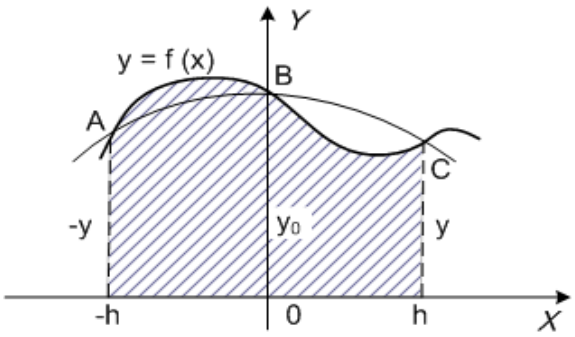


Рис. 6.9. Метод Симпсона

[8] Формула Симпсона называется также формулой парабол.

Подставляя последовательно значения x = –h, 0, h для интеграла $\int_{-h}^{h} y\,dx = S$, получим его приближенное значение в следующем виде:

$$\int_{-h}^{h} y\,dx \approx 1/3 \times h\,(y_{-1} - 2y_0 + y_1) + 2y_0 = h/3 \times (y_{-1} + 4y_0 + y_1), \quad (6.10)$$

где h = (b – a)/2.

Для увеличения точности вычисления определенного интеграла отрезок [a, b] разбиваем на n = 2n частичных промежутков $x_0 = a$; $x_1 = a + h$; …; $x_n = b$, где n – достаточно большое натуральное число. К каждому из них можно применить формулу Симпсона, полагая h = (b – a)/n. В силу свойств аддитивности данных определенных интеграл приближенно будет представлять сумму полученных таким образом результатов.

Вычисляются значения $y_i = f(x_i)$ функции в точках x_i и находится значение интеграла по формуле Симпсона:

$$S = S_n + R_n \text{ , где } S_n = h/3\,[y_0 + 4y_1 + 2\,y_2 + 4\,y_3 + \ldots + y_{2n}]$$

или

$$S = \int_a^b f(x)\,dx = h/3[y_0 + y_n + 4(y_1 + y_3 + y_{n-1}) + 2(y_2 + y_4 + \ldots + y_{n-2}] + R_n.$$

Затем количество точек удваивается и производится оценка точности вычислений $R_n = |S_{2n} – S_n|/15$. Если $R_n > \varepsilon$, то количество точек разбиения удваивается. Значение суммы $2(y_1 + y_2 + \ldots + y_{2n–1})$ сохраняется, поэтому для вычисления интеграла при удвоении количества точек требуется вычислять значения y_i лишь в новых точках.

На рис. 6.10 показана схема алгоритма метода Симпсона. Программная реализация алгоритма вычисления определенного интеграла методом Симпсона представлена ниже.

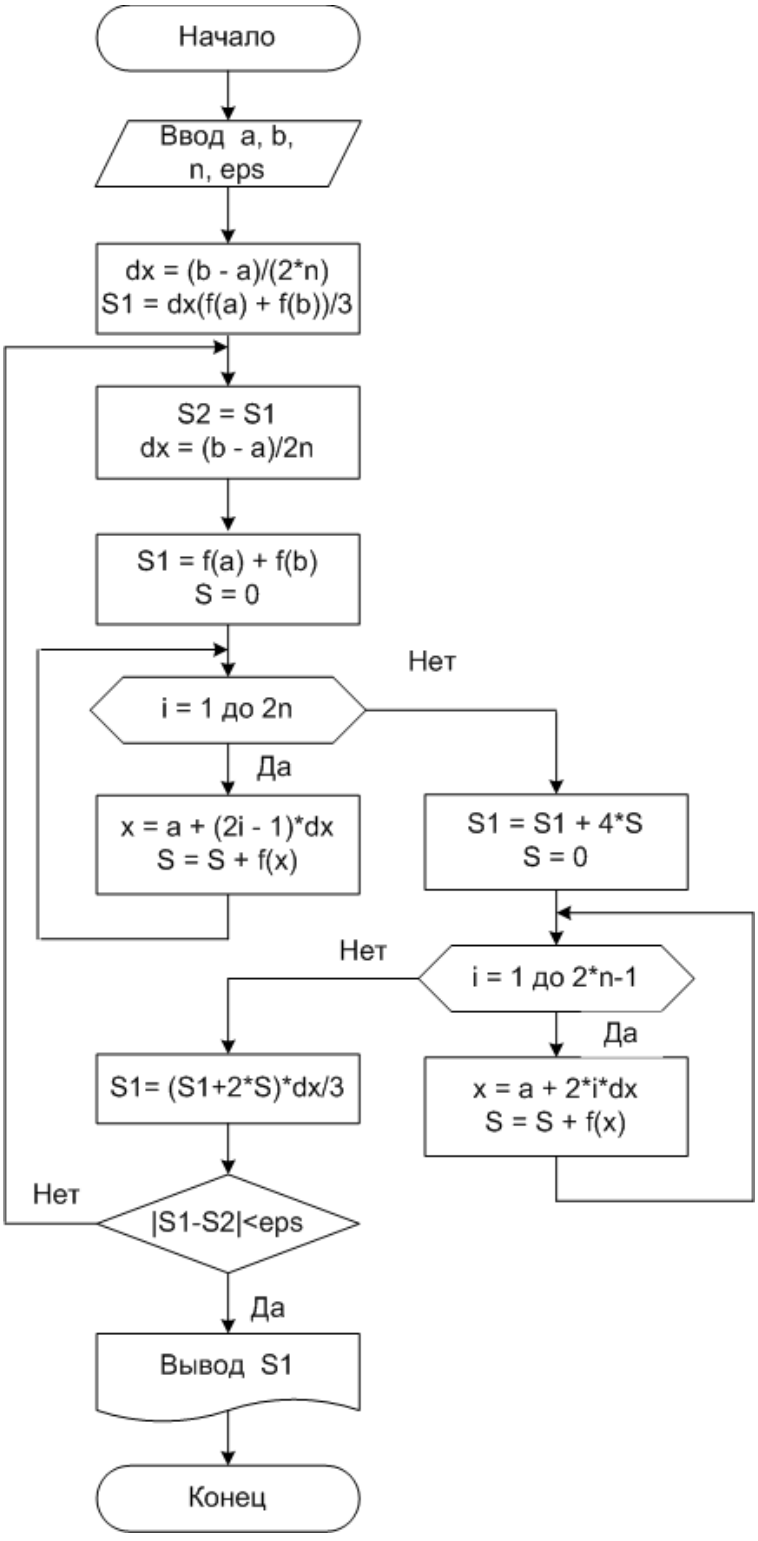


Рис. 6.10. Вычисление интеграла методом Симпсона



```
        def f(x):
            if x != 0:
            f = 1/x
        else:
            print("Значение интервала для данной функции не
может быть равным 0")
        return f

    a = float(input('Ввести начальное значение интервала
a: '))
    b = float(input('Ввести конечное значение интервала b:
'))
    n = float(input('Ввести n: '))
    eps = float(input('Ввести точность вычислений eps: '))
    print('a = {0}, b = {1}, n = {2}, eps = {3}'.format(a,
b, n, eps))

    dx = (b-a)/(2*n)
    s1 = dx*(f(a) + f(b))/3

    while True:
        s2 = s1
        dx = (b-a)/(2*n)
        s1 = f(a)-f(b)
        s = 0
        for i in range(2*int(n)):
            x = a+(2*i-1)*dx
            s += f(x)
        s1 += 4*s
        s = 0
        for i in range(2*int(n)-1):
            x = a+2*i*dx
            s += f(x)
        s1 = (s1+2*s)*dx/3
        if abs(s1-s2) < eps:
            print("Результат вычислений s1 = ", s1)
            break
```

```
Ввести начальное значение интервала a: 4
Ввести конечное значение интервала b: 13
Ввести n: 5
Ввести точность вычислений eps: 0.0001
a = 4.0, b = 13.0, n = 5.0, eps = 0.0001
Результат вычислений s1 =  2.118633337982265
```

Рис. 6.11. Результат вычисления интеграла методом Симпсона

По сравнению с методами прямоугольников и трапеции метод Симпсона является более точным (рис. 6.11). Для вычисления с помощью данной программы необходимо задать подынтегральную функцию для f(x), пределы интегрирования a и b и допустимую погрешность вычислений.

Решение обыкновенных дифференциальных уравнений

Любая задача проектирования или моделирования может быть сведена к решению дифференциальных уравнений. Результатом решения дифференциального уравнения является функция, поэтому решение дифференциального уравнения требует значительных временных затрат.

> ***Дифференциальным уравнением*** *называется уравнение, которое содержит неизвестную функцию одного или нескольких переменных, независимые переменные и производные* неизвестной функции по независимым переменным.

В зависимости от числа независимых переменных и типа входящих в них производных дифференциальные уравнения подразделяют на два вида:

1) *обыкновенные уравнения* содержат одну независимую переменную и производную по ней;
2) *уравнения в частных производных* содержат несколько независимых переменных и производных по ним.

Чтобы решить обыкновенное дифференциальное уравнение, необходимо знать значения зависимой переменной и (или) ее произ-

водных при некоторых значениях независимой переменной. Если эти дополнительные условия задаются при одном значении переменной, то такая задача называется задачей с начальными условиями, или задачей Коши.

Пусть дано дифференциальное уравнение $\frac{dy}{dx} = f(x, y)$ и начальные условия при $x = x_0$ $y(x_0) = y_0$. Необходимо найти функцию $y(x)$, которая будет удовлетворять и данному уравнению и начальным условиям. Для решения вычисляют сначала значение производной, а затем малое приращение x и переходят к новой точке $x_1 = x_0 + n$.

Положение новой точки определяется по наклонной кривой. Графическое представление численного решения представляет собой последовательность характерных прямолинейных отрезков, аппроксимирующих кривую $y = f(x)$. Численный метод должен определять порядок действий при переходе от одной точки кривой к другой.

Известные одношаговые методы позволяют достаточно просто находить решение для задачи Коши – находить следующую точку по кривой $y = f(x)$, при этом требуется информация лишь о предыдущем шаге.

Метод Эйлера

Рассмотрим дифференциальное уравнение первого порядка вида

$$y' = f(x, y), \qquad (6.11)$$

где $y' = \frac{dx}{dy}$, при начальном условии $y(x_0) = y_0$.

Метод Эйлера – один из самых простых методов, позволяющий интегрировать дифференциальное уравнение 1-го порядка. Точность метода невелика, поэтому на практике используется достаточно редко, но в учебных целях позволяет понять алгоритм других более эффективных методов. Метод основан на разложении y в ряд Тейлора в окрестности x_0.

$$y(x_0 + h) = y(x_0) + h\, y'(x_0) + 1/2\, h^2\, y''(x_0) + \ldots. \qquad (6.12)$$

Если h мало, то члены, содержащие h во 2-й и более степенях, являются достаточно малыми более высоких порядков и ими можно пренебречь. При численном интегрировании дифференциального

уравнения первого порядка (6.11) сначала следует выбрать фиксированное приращение аргумента $h = (x_n – x_0)/ n$, где x_n – конечная точка интегрирования. Предположим, что на небольшом участке интегральной кривой ее наклон является постоянной величиной и h = const:

$$y(x_0 + h) = y(x_0) + h\, y'(x_0).$$

Можно получить приближенное значение зависимой переменной при малом смещении h от начальной точки. Величину $y'(x_0)$ можно найти из дифференциального уравнения, если подставить в него начальные условия:

$$y'(x_0) = f(x_0, y_0).$$

Этот процесс можно продолжать при любом количестве шагов, используя соотношение $y_{n+1} = y_0 + h\, f(x_n, y_n)$, где n =1, 2, … Графически метод показан на рис. 6.12: уравнение (6.11) задает направление касательной к искомой точке с координатами (x_0, y_0). Двигаясь вдоль касательной, можно получить приближенное значение решения в точке x_1: $y_1 = y_0 + h\, f(x_0, y_0)$.

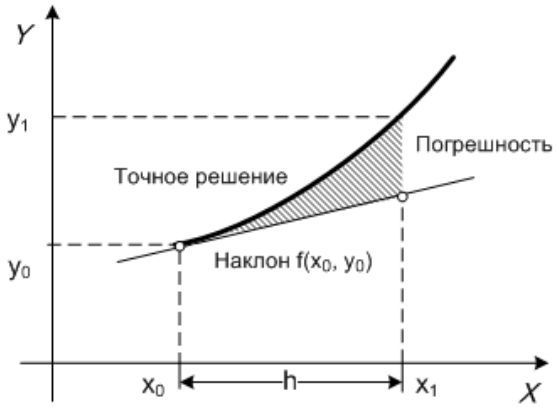


Рис. 6.12. Геометрическая интерпретация метода Эйлера

Продолжая этот процесс, можно построить систему равностоящих точек $x_i = x_0 + i\, h$, где i = 0, 1, 2, …, n. В результате вместо интегральной кривой в действительности получаем совокупность прямых или так называемую *ломаную Эйлера*. На рис. 6.13 показана схема алгоритма Эйлера.

Метод Эйлера – это простейший пошаговый метод. Начиная со второго шага погрешность метода Эйлера систематически возрастает и имеет порядок h^2, так как члены, имеющие h в более высоких степенях, как сказано выше, отбрасываются.

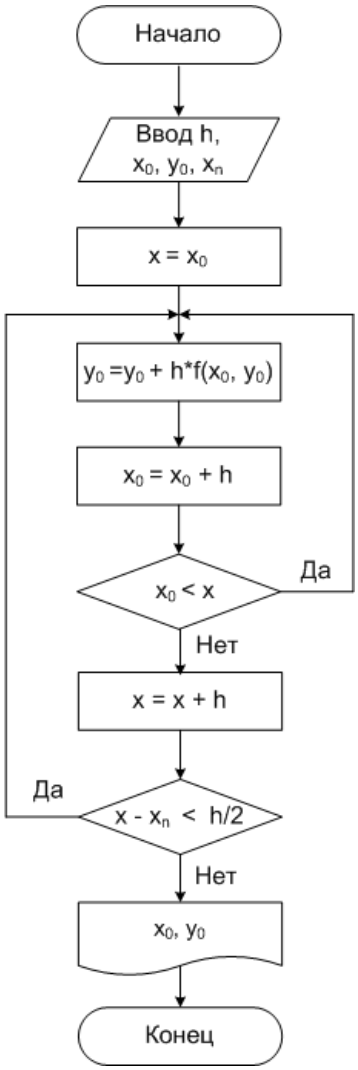


Рис. 6.13. Схема алгоритма метода Эйлера

Программная реализация алгоритма приближенного решения дифференциального уравнения первого порядка методом Эйлера представлена ниже. Как было сказано выше, наиболее часто используемым методом оценки точности является метод двойного прогона. Этот метод использован и в алгоритме метода Эйлера (рис. 6.14).

```
import math
h = int(input("Введите h: "))
x0 = int(input("Введите x0: "))
y0 = int(input("Введите y0: "))
```

```
xn = int(input("Введите xn: "))
def f(x0, y0):
   f = math.cos(y0) + 3*x0
   return f
x = x0
while ((x - xn) < h/2):
   y0 += h*f(x0,y0)
   x0 += h
   if x0 >= x:
       x += h
print("x0 = ", x0)
print("y0 = ", y0)
```

```
Введите h: 15
Введите x0: 3
Введите y0: 19
Введите xn: 2
x0 =  18
y0 =  168.83056927280006
```

Рис. 6.14. Результат вычисления интеграла методом Эйлера

Программа решает дифференциальное уравнение, которое должно быть определено в f(x, y). В процессе работы программы необходимо ввести данные для начальных условий x0 и y0, шаг интегрирования h и значение правой границы отрезка интегрирования xn. Вычисления выполняются с шагом h и с шагом h/2. Сопоставление полученных приближений показывает, что метод Эйлера имеет невысокую точность.



Вопросы для самопроверки

1. Вычислить приближенно интеграл $\int_0^1 \frac{dx}{1+x}$: а) по формуле трапеций; б) по формуле Симпсона, разделив отрезок [0; 1] на n = 10 равных частей. Результаты вычисления сравнить с точным значением интеграла, равным $\ln 2 \approx 0.69315$.

2. Вычислить значение интеграла $I = \int_0^2 f(x)\,dx$ по формуле трапеций и по формуле Симпсона при h = 1.0. В качестве f(x) взять следующие функции:
   а) $f(x) = 1 + x$;
   б) $f(x) = 1 + x^2$;
   в) $f(x) = 1 + x^3$;
   г) $f(x) = 1 + x^4$
   и сравнить результаты вычислений.

3. Вычислить приближенно интеграл $\int_{-1}^1 (1 + x + x^2 + x^3)\,dx$ при h = 1 известными методами.

4. Вычислить приближенно интеграл $\int_0^1 \sin x\,dx$ при h = 0.5 известными методами. Точное значение 0.45970. Чему равна точность, чтобы получить результат вычислений более близкий к точному решению?

5. Вычислить приближенно интеграл $\int_0^{10} e^{-x}\,dx = 0.999955$:

а) по формуле трапеций для n = 10;

б) по формуле Симпсона для n = 10;

в) по формуле прямоугольников для n = 10, 18.

Сравнить результаты вычислений.

6. Вычислить приближенно интеграл $I = \int_0^1 \frac{dx}{1+x^2} = 0.785398$:

а) по формуле трапеций для n = 10;

б) по формуле Симпсона для n = 10;

в) по формуле прямоугольников для n = 5.

Сравнить результаты вычислений.

7. Вычислить приближенно интегралы по формуле прямоугольников для n = 10:

$$\int_1^2 \frac{dx}{x} = 0.719; \quad \int_0^4 x^2\, dx = 18.24$$

8. Вычислить приближенно интегралы по формуле трапеций:

$$\int_0^{\pi/2} \sin x\, dx = 0.9981\ (n = 10); \quad \int_0^{\pi/4} \frac{\sin x}{x}\, dx = 0.758\ (n = 5);$$

$$\int_0^{1/2} \frac{dx}{\sqrt{1-x^2}} = 0.524\ (n = 5).$$

Вычислить приближенно интегралы по формуле Симпсона с точностью 0.001:

$$\int_0^{0.8} \cos x\, dx = 0.717\ (2n = 10); \quad \int_0^4 x^2\, dx = 21.13\ (2n = 10).$$

9. Методом Эйлера найти y(2), если $y' = x - y$, $y(1) = 0.370$, $h = 0.2$.

10. Найти приближенное решение уравнения $y' = xy/2$, в интервале [0; 1] при начальных данных $x_0 = 0$, $y_0 = 1$.

11. Дано уравнение $y' = 0.4y$, $y_0 = 1000$. Вычислить y_1 методом Эйлера для h = 1; h = 0.5; h = 0.25.

12. Дано уравнение $y' = 4 - 2x$, $y(0) = 2$. Решить уравнение, принимая h = 0.5 и продолжая до x = 5.

13. Дано уравнение $y' = -x/y$, $y(0) = 20$. Вычислить $y(24)$, используя метод Эйлера для $h = 2$.
14. Дано уравнение $y' = (6y + 2x + 1)/x$, $y(1) = -17/30$. Вычислить $y(4)$, повторить вычисления для $y(1) = -16/30$.
15. Дано уравнение $y' = xy^2 + 3xy$, $y(0) = -0.5$. Вычислить $x(3)$ при $h = 0.5$,
16. Дано уравнение $y' = \cos x - \sin y - y$. Найти решение до $x = 10$ при $h = 0.5$.
17. Дано уравнение $y' = 2y/x + x^2e^x$ и $y(1) = 0$. Вычислить до $x = 5$ при $h = 0.2$.

Глава 7

Экологические проблемы города



Новые формализмы всегда страшат,
и требуются значительные усилия, чтобы
убедить новичка в том, что формализм
не только полезан, но и необходим.
Э. Дейкстра

Любая дисциплина для описания сложных явлений обращается к математике, чтобы преодолеть сложности описания. И экология не является исключением.

В современном понимании экология[9] – научная стратегия выживания человечества. Экология как понятие за последние годы приобрела интегральный характер, обогатилась новыми знаниями и стала наукой, которая затрагивает все сферы экономической, социальной и духовной жизни человека и общества. Экологические проблемы сегодня все более ориентируются на злободневные проблемы по оздоровлению окружающей среды. Знание вопросов охраны окружающей среды и рационального природопользования, экологического права и экономического регулирования охраны окружающей среды, а также умения применять их при решении профессиональных задач стали обязательными для каждого специалиста.

Сегодня говорят о новой экологической среде с высокой концентрацией антропогенных факторов, таких как загрязнение атмосферы, высокий уровень шума, искусственное освещение, электромагнитные излучения и т.д.

[9] *Экология* (от греч. *ойкос* – дом, жилище + *логос* – наука, учение) – наука об отношениях сообществ растительных и животных организмов между собой и с окружающей средой. С середины XX века экология является основой рационального природопользования и охраны живых организмов.

Экология питания

Питание – экологический фактор. Правильное питание обеспечивает высокий уровень работоспособности и оптимальную продолжительность жизни человека. Для нормальной жизнедеятельности необходимо соблюдать правильный режим питания, т.е. регулярный прием пищи в установленное время с целесообразным распределением суточного рациона между отдельными приемами пищи.

При трехразовом питании для взрослого человека рекомендуется следующее распределение пищи:

30 % суточной калорийности – на завтрак;
45–50 % суточной калорийности – на обед;
20–25 % суточной калорийности – на ужин;
если 100 % – суточная калорийность.

Энергетическая ценность пищевого рациона служит объективным количественным показателем при оценке калорийности пищевого рациона и выражается в килокалориях 1ккал = 4,184 кДж. Нормы физиологической потребности в пищевых веществах и энергии распределены на пять групп отдельно для женщин и мужчин. Величина энергетической потребности колеблется в пределах от 1800 до 4200 ккал в зависимости от характера деятельности. Минимальное соотношение количества белков, жиров и углеводов в рационе человека принято как постоянное соотношение 1 : 1 : 4,6 (в граммах).

Пример 1

В пищевой рацион молодого специалиста 23 лет входит 50 г белка, 80 г жиров, 400 г углеводов. Суточная калорийность составляет 2700 ккал. Оцените качественный состав и калорийность пищевого рациона. Определите рациональный режим питания при трехкратном приеме пищи.

Пусть CK – суточная калорийность, тогда KS – калорийность завтрака = CK × 0,3; KO – калорийность обеда = CK × 0,45; KU – калорийность ужина = CK × 0,25. Качественный состав пищи определим из соотношения B/G = 1 и (U/B и U/G) = 4,6 (схема алгоритма рис. 7.1).

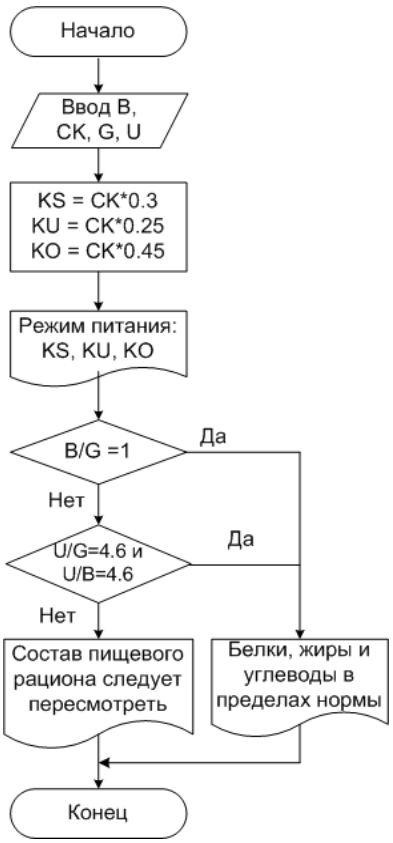


Рис. 7.1. Определение рационального режима питания

Освещение

Одним из важнейших условий планировки помещений является естественное и искусственное освещение. Под освещенностью понимают плотность светового потока на освещаемой поверхности. За единицу освещенности принят люкс (лк).

Для оценки естественного освещения помещения используется световой коэффициент:

$$CK = S1 / S,$$

где S1 – площадь одного застекленного окна данного помещения;

S – площадь данного помещения (площадь пола).

Чем больше величина СК, тем лучше естественное освещение. Величина СК для различных помещений имеет свою норму:

Для жилых помещений	0.166 ÷ 0.125
Для общественных и учебных помещений	0.25 ÷ 0.2
Для коридоров и лестниц	0.083 ÷ 0.077

Пример 2

Площадь учебного кабинета по информатике составляет 45 м^2, застекленная поверхность одного окна составляет 2,5 м^2. В кабинете 3 окна. Какая экологическая обстановка в кабинете, если в нем занимается 35 человек? Оценить естественную освещенность.

Точная оценка уровня естественного освещения возможна при использовании коэффициента естественной освещенности КЕО = (E1/E2) ×100 %, где E1 – освещенность внутри помещения в *люкс*, E2 – освещенность на горизонтальной плоскости под открытым небом, при условии ее освещенности рассеянным светом. В качестве источников искусственного света сегодня широко используются люминесцентные лампы и лампы накаливания. Люминесцентные лампы в 3–4 раза экономичнее, чем лампы накаливания, позволяют создать рассеянный свет, не дают резкой тени и обладают малой яркостью. Алгоритм определения естественной освещенности учебного помещения представлен на рис. 7.2.



Рис. 7.2. Определение естественной освещенности учебного помещения

Пример 3

Известна площадь учебной аудитории S, которая освещается N лампами накаливания, каждая мощностью P1 (или N люминесцентными лампами мощности P1). Определить искусственную освещенность в данном помещении.

Нормативы освещенности на рабочем месте:

Виды помещений	Лампы накаливания	Люминесцентные лампы
Жилые помещения	75 люкс	100 люкс
Общественные помещения	100 люкс	150 – 300 люкс

Общая мощность ламп равна произведению мощности одной лампы на количество ламп, или P = P1 × N, соответственно, удельная мощность PR = P/S.

Чтобы определить искусственную освещенность, необходимо найти произведение удельной мощности на коэффициент *e*. Коэффициент e зависит от размера площади, мощности лампы и напряжения сети:

Мощность лампы	Напряжение в сети 127 В	Напряжение в сети 220 В
До 110 Вт	2,4	2,0
Более 100 Вт	3,2	2,5

Допустим, в нашем примере напряжение в сети равно 220 В, тогда коэффициент e равен либо 2, либо 2,5 (схема алгоритма представлена на рис. 7.3).

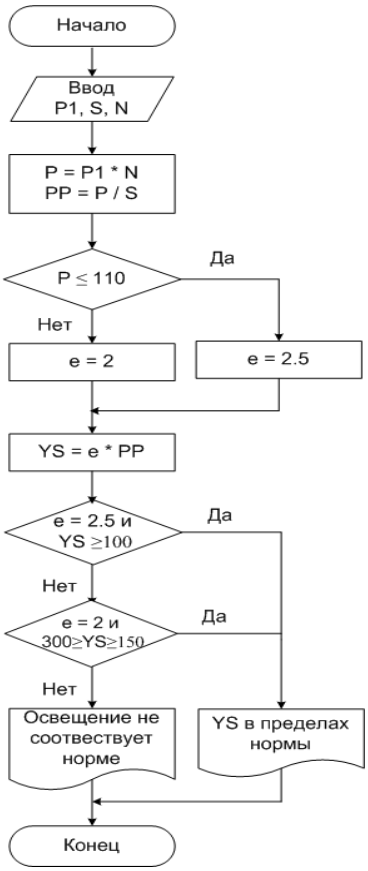


Рис. 7.3. Определение искусственной освещенности

Экология литосферы

Литосфера – это твердая оболочка Земли, или земная кора. Почва – это часть литосферы, особое природное образование. Она является основным элементом биосферы, поддерживающим жизнь на Земле. Естественный химический состав почвы оказывает определенное влияние на здоровье человека, иными словами, состояние почвы играет важную роль для человека. Почва образуется в результате различных факторов, из которых наиболее важным является климат, время года, рельеф местности и др. Большое влияние на состояние почвы оказывает развитие промышленности и сам человек. Поэтому можно сказать, что решающее значение в охране почвы имеет рационально организованная очистка от отходов производства и потребления.

Образующиеся в процессе жизнедеятельности человека отходы требуют специальной переработки – утилизации, хранения, захоронения и т.д. (Для образующихся в процессе жизнедеятельности человека жидких и твердых отходов требуется переработка). Для сбора твердых бытовых отходов (ТБО) необходимо иметь достаточное количество мусороприемников в домах, дворах и на улицах. Дворовые контейнеры имеют объем 0,6–0,8 м^2, которые рассчитаны на 350–500 жителей. Мусор из контейнеров вывозят на утилизацию.

Пример 4

В жилом доме проживает M человек. Мусоропровод, который имеется в жилом здании, имеет контейнер вместимостью N кг каждый. Мусор должны вывозить ежедневно. Сколько контейнеров необходимо, если от одного жителя в сутки образуется 2 кг твердых бытовых отходов.

Введем обозначения для количества контейнеров – k, n1 – вес ТБО на каждого жителя.

Определим общий ТБО объем как $V = n1 \times M$.

Если исходные данные были введены верно, то количество контейнеров определяется формулой $k = V / N$, причем значение k должно быть целой величиной. Для этого необходимо выполнить округление до большего целого.

Алгоритм для определения количества контейнеров представлен на рис. 7.4.

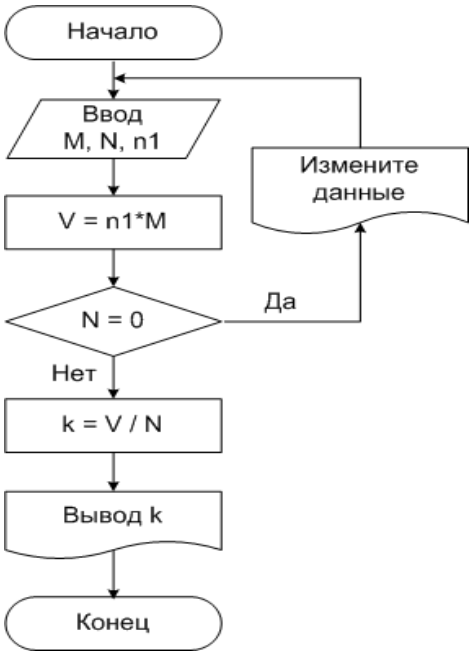


Рис. 7.4. Определение количества контейнеров

Пример 5

Завод по сжиганию мусора ТБО может принять в сутки 2000 т мусора. Получаемый пар может привести в действие генератор мощностью 60 000 кВт, что достаточно для обеспечения электроэнергией жителей 60 000 жилых домов. Сколько жилых домов можно обеспечить электроэнергией, если увеличить мощность генератора в 1,5 раза либо если уменьшить поступление мусора до 1000 т в сутки.

Введем обозначения:

S – общий вес мусора (стандарт); S1 – новые данные (S);

S2 = S / S1 – соотношение стандартного значения к новому;

W – мощность генератора;

n – количество изменений W;

k – количество жилых домов, обеспеченных электроэнергией.

Чтобы определить количество жилых домов, обеспеченных электроэнергией, следует воспользоваться предложенными в задаче соотношениями, которые реализуются в алгоритме, представленном на рис. 7.5.

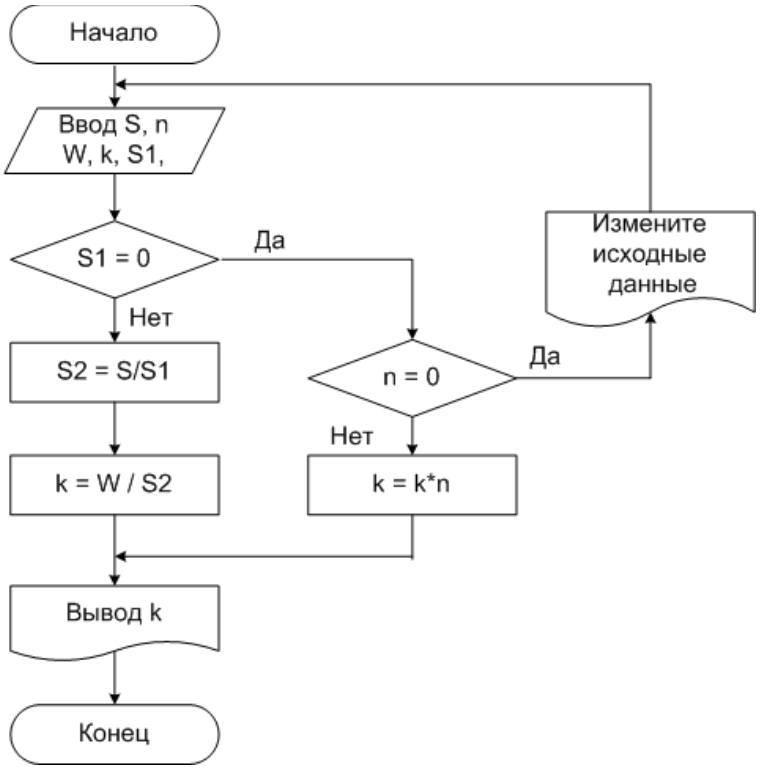


Рис. 7.5. Энергия при сжигании ТБО

Оценка степени экологической безопасности

Экологическая безопасность – процесс обеспечения защищенности важных интересов человека, общества, природы от потенциальных и реальных угроз, создаваемых антропогенным или естественным воздействием на окружающую среду.

Оценка степени экологической безопасности выполняется для определенной территории, которая, как правило, предназначена для экологической рекреации и экологического туризма. Для этого должны быть известны следующие параметры:

- базовый норматив платы за загрязнение окружающей среды (руб. · т) – BN;
- масса загрязняющих веществ, попадающих на рассматриваемую территорию в результате выбросов или при размещении отходов (т) – M;
- коэффициент экологической ситуации и экологической значимости региона – KE;

- коэффициент индекса платы, который ежегодно утверждается министерством природопользования России – KI;
- количество видов загрязняющих ингредиентов – N.

Если все эти величины известны, можно составить алгоритм (рис. 7.6).

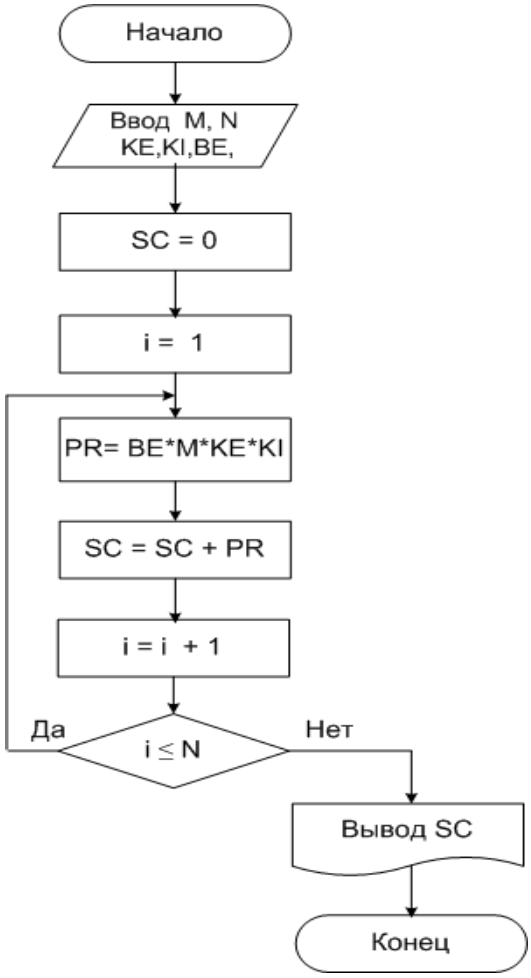


Рис. 7.6. Определение степени экологической безопасности

Коэффициент экологической значимости региона задается из следующей таблицы, в которой реально присутствуют данные о состоянии атмосферы (воздух и почва) для некоторых регионов РФ[10] (табл. 7.1).

[10] Для данного примера приведены значения коэффициента состояния экологической ситуации и экологической значимости состояния атмосферы воздуха и почвы лишь для части экономических регионов РФ.

Таблица 7.1

Данные о состоянии атмосферы (воздух и почва) для некоторых регионов РФ

Название региона	Состояние воздуха	Состояние почвы
Северный	1.4	1,4
Северо-Западный	1,5	1,3
Центральный	1,9	1,6
Волго-Вятский	1,1	1,5
Центрально-Черноземный	1,5	2,0
Поволжский	1,9	1,9

Определение ущерба от загрязнения атмосферы

Атмосфера – внешняя оболочка биосферы. Различные негативные изменения атмосферы Земли связаны, как правило, с изменением второстепенных компонентов атмосферного воздуха. Существует два главных источника загрязнения:

- естественные (вулканы, лесные пожары, выветривания и др.);
- антропогенные (топливно-энергетические комплексы, транспорт, машиностроительные предприятия).

Экономический ущерб, который наносится годовыми выбросами загрязнениями в атмосферу, можно вычислить по формуле

$$Y = G \times SG \times F \times M,$$

где GY – константа, значение которой равно 2,4, руб. / усл. т;

SG – показатель относительной опасности загрязнения – зависит от типа территории;

F – показатель учета характера примесей в атмосфере;

M – масса годового выброса загрязнений источника, усл. т.

Коэффициент SG зависит не только от типа территории, но и от ее однородности. Если территория активного загрязнения неоднородна, то значение SG складывается из суммы значений SG для разных территорий:

$$SG = \frac{1}{SAS} \sum_{i=1}^{N} S_i \times SG_i ,$$

где SAS – площадь всей загрязненной зоны, либо является величиной заданной, либо должны быть даны дополнительные рекомендации по ее вычислению;

N – общее количество территорий, которые относятся к загрязненной зоне.

Некоторые значения показателя относительной опасности загрязнения атмосферного воздуха над территориями различного типа приведены в следующей таблице (табл. 7.2).

Таблица 7.2

Показатели загрязнения атмосферного воздуха

Тип загрязненной территории	Значение SG
Пригородные зоны отдыха, садово-дачные участки	10
Территории промышленных предприятий	4
Леса	0,025–0,2
Пашни	0,1–0,25
Сады, виноградники	0,5
Пастбища	0,05

Если SG вычисляется как сумма, то, соответственно, это отображено в схеме алгоритма (рис. 7.7).

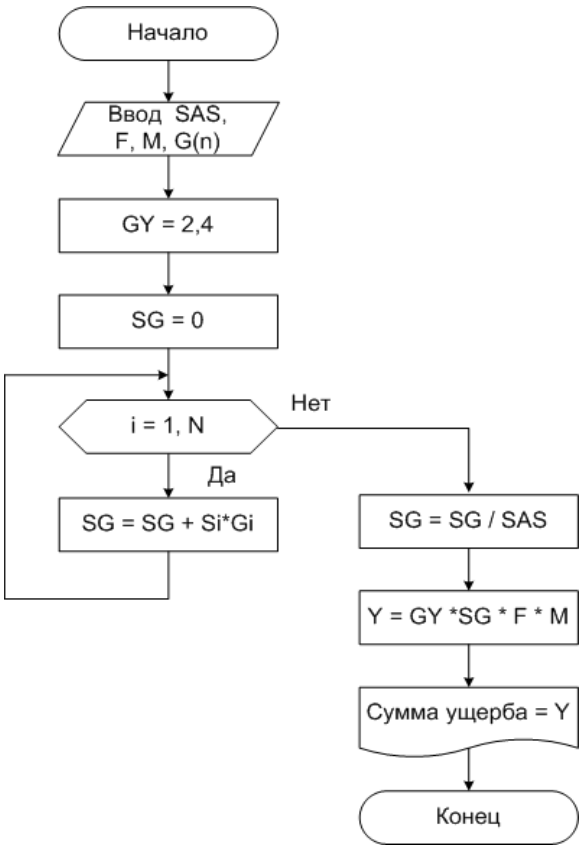


Рис. 7.7. Определение ущерба от загрязнений атмосферы

Плата за загрязнение атмосферы от передвижных источников

Загрязнение атмосферного воздуха отработанными газами автомобилей и других движущих средств может рассматриваться как допустимые выбросы и выбросы, превышающие допустимые. Рассмотрим первый вариант.

Удельная плата для различных видов топлива неодинакова и имеет следующие значения (табл. 7.3).

Таблица 7.3

Удельная плата для различных видов топлива

Название топлива	Марка	Плата (руб./т; руб./м^3)
Бензин этилированный	АИ93	38,0
	А76, 72	25,0
Бензин неэтилированный	АИ93	10,0
	А76, 72	11,0
Дизельное топливо		21,0
Сжатый природный газ		9,0
Сжиженный газ		11,0

При использовании различных видов одной тонны топлива удельная плата за допустимые выбросы k – вида топлива – будет равна *сумме произведения базового норматива* платы за выброс одной тонны i-го загрязняющего вещества на *массу* этого вещества, которое содержится в виде отработанного газа *технически исправного транспортного средства при использовании одной тонны его вида топлива*:

$$Y_k = \sum_{i=1}^{N} H_i \times M_i .$$

Основными нормируемыми загрязняющими веществами для транспортных средств являются оксиды углерода и азота, углеводороды, сажа, соединения свинца, диоксид серы.

Снизить уровень выбросов можно, если провести следующие мероприятия:

- заменить карбюраторы грузовых автомобилей дизельными;

- установить фильтры для выхлопных газов;
- ограничить интенсивность движения;
- совсем запретить движение автомобилей.

Плата за допустимые выбросы равна

$$П = \sum_{k=1}^{m} Y_k \times R_k ,$$

где R_k – количество вида топлива, которое израсходовано источником загрязнения, например, транспортным средством. Схема алгоритма показана на рис. 7.8.

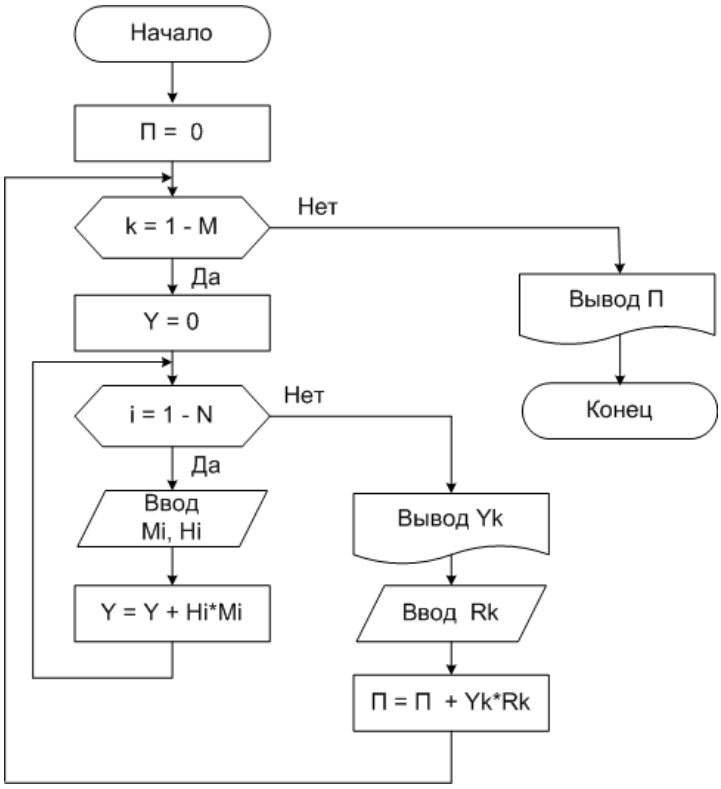


Рис. 7.8. Плата за выбросы в атмосферу от транспортных средств

Глава 8

Сетевой суперкомьютер – это просто!

Грид-технологии и вычисления

Цель работы над темой – создание грид-приложения типа «Master-Worker» (мастер-исполнитель) для параллельной и распределенной обработки информации на основе сетевых возможностей языка Python путем модификации клиент-серверной архитектуры.

Грид (англ. *grid* – решетка, сеть), по распространенному определению является географически распределенной инфраструктурой, объединяющей множество ресурсов разных типов (процессоры, долговременная и оперативная память, хранилища и базы данных, сети), доступ к которым пользователь может получить из любой точки, независимо от места их расположения (http://gridclub.ru/about).

Grid-технологии позволяют создать географически распределенные вычислительные инфраструктуры, которые объединяют разнородные ресурсы и реализуют возможность коллективного доступа к этим ресурсам. Принципиальной новизной этих технологий является объединение ресурсов путем создания компьютерной инфраструктуры нового типа, обеспечивающей глобальную интеграцию информационных и вычислительных ресурсов на основе сетевых технологий и специального программного обеспечения промежуточного уровня (*middleware*), а также набора стандартизованных сервисов (служб) для обеспечения надежного совместного доступа к географически распределенным информационным и вычислительным ресурсам: отдельным компьютерам, кластерам, хранилищам информации и сетям (https://ru.bmstu.wiki/Грид-система).

Грид-вычисления имеют много общего с облачными вычислениями. Под «облаком» здесь подразумевается Интернет. Облачная система предлагает, как правило, больше услуг, чем грид-вычисления: в Интернете из облака могут быть получены, например, веб-

хостинг, несколько операционных систем, поддержка баз данных и многое другое. Концептуально облачные системы являются развитием грид-систем, отличаясь способом виртуализации сетевых ресурсов и предоставляемыми долговременными сервисами.

При реализации грид-технологий отдельным пользователям предоставляются вычислительные ресурсы, такие как хранилища, приложения, данные, процессоры с ограниченным знанием того, где находятся указанные ресурсы. Для грид-вычислений осуществляется организация распределенных вычислений, которая включает в себя координацию и совместное использование вычислений, данных, приложений, хранилищ и других сетевых ресурсов в динамической и территориально распределенной среде, построенной на базе компьютерной сети.

Современный подход к организации распределенных вычислений предполагает выбор сетевой архитектуры Grid Computing and Networking, что дословно переводится как сетевой компьютерный грид (*grid* – решетка). Для использования практических преимуществ Grid Computing and Networking организациям необходимо обосновать переход на эту относительно новую вычислительную парадигму, современная реализация которой предполагает также взаимодействие грид-системы с клиентом при посредстве облачной среды, которой соответствует термин Cloud Computing – облачные вычисления.

Несмотря на наличие в названии термина «вычисления», собственно вычисления, как правило, не являются глобальными. При наличии спроса на вычисления, в том числе высокопроизводительные вычисления, через «облако» чаще всего организуется связь с отдельной грид-системой, кластером компьютеров или с суперкомпьютером.

Концепции, как и приложения грид-вычислений, не всегда являются ясными и понятными для нетехнических специалистов. При выборе распределенных грид-вычислений необходимо объяснить заказчику лежащий в основе грид сетевой механизм и ответить на важные для бизнеса вопросы.

В настоящей работе распределенная обработка данных связана с выполнением заданий многочисленных пользователей. На рис. 8.1 представлена структура вычислительной сети, в которой могут быть реализованы распределенные вычисления. Например, это может быть компьютерная TCP/IP сеть. Эта сеть содержит клиентские компьютеры, серверы, коммутаторы, маршрутизаторы и каналы связи.

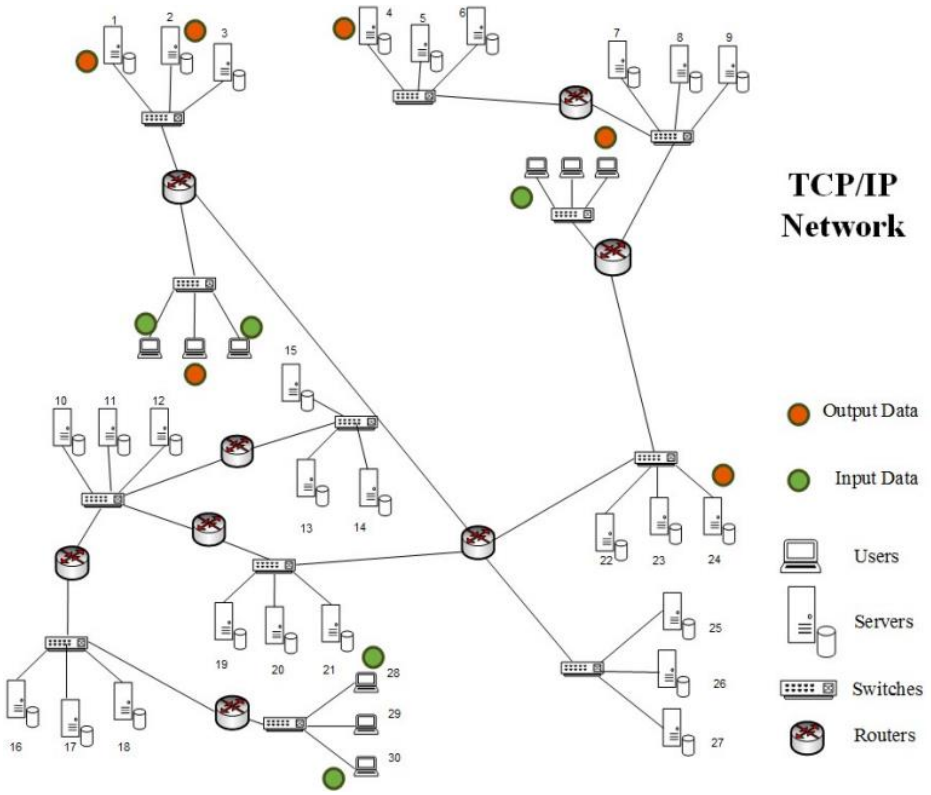


Рис. 8.1. Распределенная обработка данных в компьютерной сети

Зелеными кружками на рис. 8.1 обозначены источники данных или запросы к распределенной базе данных, вводимые с пользовательских компьютеров. Коричневыми кружками обозначены конечные результаты, сохраняемые в локальных базах данных или принимаемые клиентами.

Грид-вычисления являются развивающейся областью и связаны с несколькими другими инновационными вычислительными системами, некоторые из которых являются частными случаями грид-вычислений.

Совместно используемые вычисления обычно относятся к совокупности компьютеров, которые совместно используют вычислительную мощность для выполнения определенной задачи. В этой связи в облачных средах реализуется сервис «программное обеспечение как услуга» (SaaS), известная как служебные вычисления, в которой компания предлагает определенные услуги (например, хранение данных или увеличение мощности процессора) за определенную плату. Облачные вычисления – это система, в которой приложения и хранилище являются ресурсами сети, а не компьютера пользователя.

Реализации крупномасштабных вычислительных проектов

Возможны различные технологии разработки приложений подобного вида, из которых наиболее близкими являются технологии, положенные в основу создания программного комплекса BOINC (англ. *Berkeley Open Infrastructure for Network Computing*) или WLCG (англ. *Worldwide LHC Large Hadron Collider Computing Grid*).

Платформа BOINC, широко используемая система промежуточного программного обеспечения с открытым исходным кодом, позволяет избавиться от многих недостатков, присущих волонтерским вычислениям и связанным с их ненадежностью, неоднородностью и переменным составом ресурсов. Волонтерские вычисления VC (англ. *Volunteer Computing*) основаны на использовании пользовательских компьютеров и позволяют выполнять большой объем вычислительных работ. Потенциальная производительность VC огромна: в мире существует более 2 миллиардов пользовательских настольных компьютеров и ноутбуков. В настоящее время во многих учреждениях реализовано около 30 проектов VC для многих научных областей. Около 700 000 устройств активно участвуют в проектах VC. Эти устройства имеют около 4 миллионов ядер основных (центральных, CPU) процессоров 560 000 графических (GPU) процессоров, что в совокупности обеспечивает среднюю производительность 93 петафлопс. Используемые вычислительные устройства – в первую очередь, это современные компьютеры, имеющие среднюю производительность 16,5 гигафлопс. Большинство из них имеют графические процессоры и поддерживают вычисления общего назначения с использованием интерфейса прикладного программирования (API) для параллельных вычислений OpenCL или архитектуру параллельных вычислений CUDA.

Проект Worldwide LHC Computing Grid (WLCG) реализуется на основе глобального сотрудничества около 170 вычислительных центров, объединяющих национальные и международные грид-инфраструктуры, расположенные в более чем 40 странах. WLCG объединяет глобальные вычислительные ресурсы для хранения, распространения и анализа примерно 200 петабайт данных, ожидаемых каждый год от работы Большого адронного коллайдера (БАК) в ЦЕРНе – Европейском центре ядерных исследований (CERN – The European Center for Nuclear Research), крупнейшем в мире исследовательском центре физики элементарных частиц.

Глобальные вычислительные проекты, использующие архитектуру “Master-Worker”

В числе наиболее известных – вычислительные проекты, использующие ресурсы обычных компьютеров: MD@home, MoneyBee, RSAttack 576, SETI@home, Seventeen Or Bust, Folding@home, GIMPS, grid.org, Lifemapper, climateprediction.net, distributed.net, Distributed Folding, ECC2-109, Fermat Search, Find-a-Drug. Они предназначены для решения разнообразных задач, таких как моделирование теплового движения молекул, анализ биржевых котировок при помощи технологий нейронных сетей, тестовый «взлом» многобитного ключа для алгоритма шифрования, обработки данных, полученных с радиотелескопа, решение задач в области теории чисел, поиск новых лекарственных соединений, составление электронного атласа живой природы Земли, анализ климатических моделей, моделирование и определение жизнеспособности белковых структур, поиск делителей для чисел Ферма, поиск лекарств от рака, СПИДа, атипичной пневмонии и других заболеваний. (Источник: https://parallel.ru/meta/internet.html).

Сетевые приложения в подобных системах, как правило, построены на основе относительно простых архитектур типа «клиент-сервер» или «мастер-слуга». В первом случае приложения-клиенты передают данные на сервер для обработки, а во втором – приложение-мастер передает данные для вычисления на множестве приложений-слуг. В рамках развития распределенных систем и вычислений актуально решение проблем, связанных с организацией взаимодействий с облачными системами, а также с внешним или внутренним аудитом. На одном из первых мест находится проблема обеспечения безопасности вычислений и хранения данных. Многим пользователям может потребоваться и организация виртуальных топологий для распределенных параллельных вычислений, а не только простой иерархической или клиент-серверной архитектуры. Актуально использование мобильных беспроводных сетей и сети Internet в качестве инфраструктуры для распределенных вычислений.

Огромные вычислительные мощности компьютеров в приведенных в качестве примеров вычислительных платформах, их архитектура приспособлены для реализации концепции, выраженной слоганом

фирмы SUN Microsystems (ныне в составе фирмы Oracle) “*The network is the computer*” (“Сеть – это компьютер”). Обычно массовые параллельные вычисления в подобных вычислительных системах реализуются на основе программно-аппаратных реализаций архитектур «клиент-сервер», «мастер-работник» или на основе метода SPMD (англ. *Single Program, Multiple Data* – единая (тиражируемая) программа, множество данных), используемого для достижения параллелизма.

Прототипная архитектура «Клиент-сервер» для создания грид-приложения: некоторые понятия и определения

Большинство технологий распределенных вычислений основано на модели «Клиент-сервер». Клиент-сервер (англ. *Client-server*) – вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Физически клиент и сервер – это программное обеспечение. Обычно они взаимодействуют через компьютерную сеть посредством сетевых протоколов и находятся на разных вычислительных машинах, но могут выполняться также и на одной машине. Программы, расположенные на сервере, ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных или сервисных функций.

Преимущества архитектуры клиент-сервер:

1) отсутствие дублирования кода программы-сервера программами-клиентами;

2) так как все вычисления выполняются на сервере, то требования к компьютерам, на которых установлен клиент, снижаются;

3) все данные хранятся на сервере, который, как правило, защищен гораздо лучше большинства клиентов. На сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа.

Недостатки:

1) неработоспособность сервера может сделать неработоспособной всю вычислительную сеть. Неработоспособным сервером следует считать сервер, производительности которого не хватает на обслуживание всех клиентов, а также сервер, находящийся на ремонте, профилактике и т.д.;

2) поддержка работы данной системы требует отдельного специалиста – системного администратора;

3) высокая стоимость оборудования.

Одним из основных протоколов, предназначенных для управления передачей данных в сетях и подсетях TCP/IP, является TCP (*Transmission Control Protocol*).

Механизм TCP предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета. TCP осуществляет надежную передачу потока байтов от одной программы на некотором компьютере к другой программе на другом компьютере.

Альтернативой протоколу TCP является протокол UDP (*User Datagram Protocol*). UDP использует простую модель передачи, без неявных «рукопожатий» для обеспечения надежности, упорядочивания или целостности данных. Таким образом, UDP предоставляет ненадежный сервис, и дейтаграммы могут прийти не по порядку, дублироваться или не дойти совсем. UDP подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении.

Взаимодействие двух компьютеров при реализации сетевого приложения в архитектуре «клиент-сервер» иллюстрирует рис. 8.2.

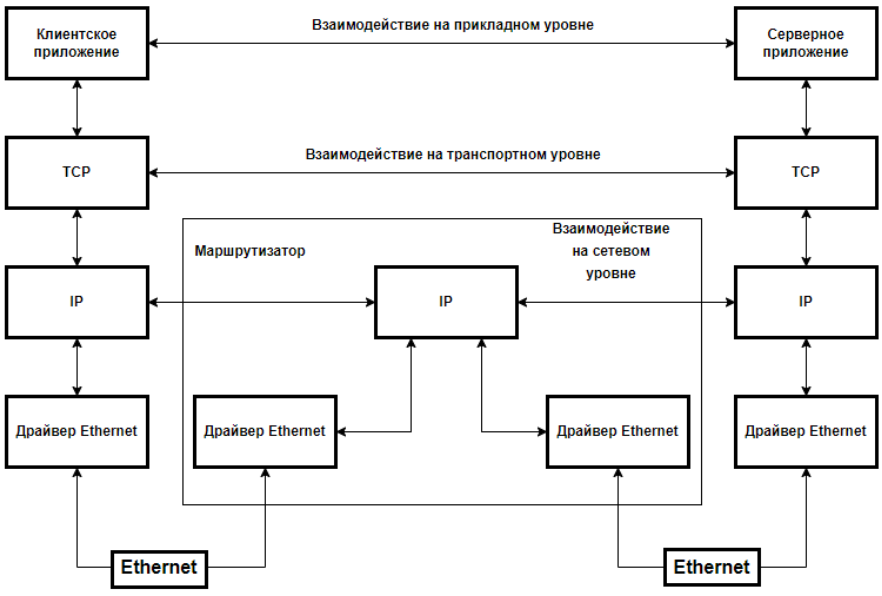


Рис. 8.2. Взаимодействие двух компьютеров при реализации сетевого приложения в архитектуре «клиент-сервер»

Протоколы TCP и UDP являются базовыми для организации протоколов прикладного уровня в сложных системах распределенной обработки и хранения для передачи данных между узлами сети.

Основной принцип технологии «клиент-сервер» заключается в разделении функций приложения на три группы:

– ввод и отображение данных (взаимодействие с пользователем);

– прикладные функции, характерные для данной предметной области;

– функции управления ресурсами (файловой системой, базой данных и т.д.).

Поэтому в любом приложении выделяются следующие компоненты: компонент представления данных, прикладной компонент, компонент управления ресурсом.

Связь между компонентами осуществляется по определенным правилам, которые называют «протокол взаимодействия».

Socket (гнездо, разъем) – абстрактное программное понятие, используемое для обозначения в прикладной программе конечной точки канала связи с коммуникационной средой, образованной вычислительной сетью. При использовании протоколов TCP/IP можно говорить, что socket является средством подключения прикладной программы к порту локального узла сети.

Взаимодействие между клиентом и сервером осуществляется с помощью сокетов – программных интерфейсов для обеспечения информационного обмена между процессами.

Виды сокетов:

– серверный сокет, принимающий сообщения;

– клиентский сокет, который отправляет сообщения.

Сокеты работают на транспортном уровне стека протоколов и соответственно бывают двух типов:

– потоковые (на основе TCP; в коде обозначаются SOCK_STREAM) сокеты с установленным соединением на основе протокола TCP, передают поток байтов, который может быть двунаправленным, т.е. приложение может и получать, и отправлять данные;

– дейтаграммные (на основе UDP; в коде обозначаются SOCK_DGRAM) сокеты, не требующие установления явного соединения между ними. Сообщение отправляется указанному сокету и, соответственно, может приниматься от указанного сокета. В разработанном приложении используется потоковый тип сокетов.

Сокет состоит из IP-адреса и порта.

IP-адрес – это уникальный сетевой адрес узла в компьютерной сети, построенной для протокола межсетевого уровня IP. В версии протокола IPv4 IP-адрес имеет длину 4 байта (например, 127.0.0.1), а в версии протокола IPv6 IP-адрес имеет длину 16 байт. IP-адрес должен быть уникальным.

Порт представляется натуральным числом, записываемым в заголовках протоколов транспортного уровня (TCP, UDP и др.). Порт используется для определения процесса-получателя пакета в пределах одного хоста.

В языке Python для работы с сокетами используется встроенная библиотека socket. Одной из основных функций модуля является функция socket(), которая возвращает объект типа сокет, обладающий соответствующими функциями для работы с соединением:

```
class socket.socket
sock = socket.socket().
```

Функции объекта socket:

`socket.bind(address)` – привязывает сокет к адресу address (инициализирует IP-адрес и порт); сокет не должен быть привязан заранее;

`socket.listen([backlog])` – переводит сервер в режим приема соединений; параметр backlog (int) задает количество соединений, которые будет принимать сервер;

`socket.accept()` – принимает соединение и блокирует приложение в ожидании сообщения от клиента и в результате возвращает следующий кортеж: conn: объект соединения (сокет), который можно использовать для отправки/получения данных; address: адрес клиента;

`socket.recv(bufsize[, flags])` – читает и возвращает данные в двоичном формате (набор байтов) из сокета; параметр bufsize (int) – максимальное количество байтов в одном сообщении;

`socket.send(bytes[, flags])` – отправляет данные клиенту и возвращает количество отправленных байт; параметр bytes (bytes) представляет двоичные данные;

`socket.close()` – закрывает сокет.

Работа с сокетом во многом схожа с работой с файловым объектом по принципу «Открыли соединение – считали данные – закрыли соединение».

О реализации «тонкого клиента».

Одним из распространенных подходов к созданию программ параллельной обработки данных с клиент-серверной архитектурой является технология, которая носит название «Тонкий клиент».

Тонкий клиент (англ. *thin client*) в компьютерных технологиях – бездисковый компьютер-клиент в сетях с клиент-сервеной или терминальной архитектурой, который переносит все или большую часть задач по обработке информации на сервер.

Необходимо отметить, что с развитием данной технологии понятие «Тонкий клиент» стало гораздо шире. В настоящее время в качестве тонкого клиента может выступать любой компьютер, который основную часть вычислений выполняет не самостоятельно, адресуя их другому компьютеру (серверу) по сети. При этом сам тонкий клиент реализует лишь пользовательский интерфейс, обеспечивая ввод/вывод обрабатываемых и обработанных данных пользователю.

В качестве сервера выступает вычислительная машина, оснащенная достаточно мощным аппаратным обеспечением, способным обработать множество запросов от пользователей (тонких клиентов) одновременно. Таким образом, параллелизм, который обеспечивается сервером, в данном случае направлен не на решение одной задачи, а на одновременное решение множества задач от большого количества клиентов.

Создание клиентского приложения

Как и любое сетевое приложение, выступающее в роли клиента, тонкий клиент должен реализовывать основные шаги алгоритма работы клиентского приложения:

1) создание сокета;
2) установка соединения с сервером;
3) передача данных для обработки на сервер;
4) прием результатов обработки от сервера;
5) закрытие соединения.

Помимо представленных этапов тонкий клиент должен реализовывать пользовательский интерфейс, однако так как в рамках лабораторных работ требуется консольный интерфейс, рассмотрим более подробно каждый из этапов, описанных выше.

Создание сокета

С точки зрения WinsockAPI сокет – это дескриптор, который может получать или отправлять данные. На практике все выглядит так: создаем сокет с определенными свойствами и используем его для подключения, приема/передачи данных и т.п. Создавая сокет, необходимо указать его параметры. Можно создать два основных типа сокетов, работающих по TCP/IP протоколу – SOCK_STREAM и SOCK_DGRAM. Разница в том, что для первого типа сокетов (их еще называют TCP или *connectimon-bamsed socket*), для отправки данных сокет должен постоянно поддерживать соединение с адресатом, при этом доставка пакета адресату гарантирована. Во втором случае наличие постоянного соединения не нужно, но информацию о том, дошел ли пакет или нет, получить невозможно (так называемые UDP или *connectionless sockets*).

И первый, и второй типы сокетов имеют свое практическое применение. Для данной работы необходимо использование TCP (*connection-based*) сокетов.

Установка соединения с сервером

После создания сокета клиентскому приложению необходимо осуществить соединение с сервером с помощью функции *connect*.

Структура sockaddr предназначена для хранения адреса, а в разных протоколах используется своя адресация. Поэтому и структура *sockaddr* может выглядеть по-разному. Для интернет-протоколов структура имеет имя sockaddr_in.

Передача данных для обработки на сервер

После установки соединения с сервером необходимо передать данные для обработки. Для передачи данных используется функция send.

Прием результатов обработки от сервера

После того, как данные были переданы для обработки на сервер, необходимо осуществить прием результатов обработки от сервера. Для приема данных используется функция recv.

Закрытие соединения

Для завершения сеанса сначала необходимо проинформировать партнера, с которым происходило соединение (в данном случае сервер), об окончании передачи данных. Для этого используется функция shutdown.

Создание серверного приложения

Выше был рассмотрен процесс создания клиентского приложения. Создание серверного приложения содержит многие аналогичные этапы, однако в то же время имеет и ряд существенных отличий. Алгоритм работы серверного приложения представлен ниже:

1. Создание сокета.
2. Получение локального адреса.
3. Получение запроса на соединение.
4. Создание нового потока и возврат к п. 3.

Дочерний поток при этом должен работать по следующему алгоритму:

1. Прием данных для обработки от клиента.
2. Обработка полученных данных.
3. Передача результатов обработки клиенту.
4. Закрытие соединения.

Алгоритм работы дочернего потока не содержит новых шагов, поэтому обратим внимание на алгоритм работы основного потока серверного приложения. Создание сокета в серверном приложении выполняется аналогично созданию клиентского сокета. Три последующих шага рассмотрим подробнее.

Получение локального адреса

Для того, чтобы все запросы на соединения, приходящие на данную ЭВМ и обращающиеся к указанному порту, были выполнены, операционная система направляет данному процессу сообщение о том, что необходимо получить локальный адрес. Данная операция производится посредством вызова функции bind.

Получение запроса на соединение

Для того, чтобы сокет мог в дальнейшем принимать входящие соединения, нужно использовать функцию listen, которая переводит сокет в «слушающий режим». Максимальное число соединений, поддерживаемых системой, задается константой SOMAXCONN.

После того, как сокет был помещен в слушающее состояние, он может принимать входящие соединения. Для приема входящих соединений используется функция accept, которая извлекает очередной запрос на соединение из очереди, автоматически создает новый сокет, выполняет связывание и возвращает его дескриптор.

Создание нового потока

Для параллельной работы с несколькими клиентами следует сразу же после извлечения запроса из очереди порождать новый поток (нить), передавая ему дескриптор созданного функцией accept сокета, затем вновь извлекать из очереди очередной запрос и т.д. В противном случае, пока не завершит работу один клиент, сервер не сможет обслуживать всех остальных. Для создания нового потока может быть использована функция _beginthread или иные функции для порождения потоков. Вся работа с клиентом должна осуществляться в порожденном потоке (см. алгоритм работы дочернего потока).

Сетевое приложение может быть выполнено в архитектуре «клиент-сервер» или ее модификации «клиент-распределенный мультисервер» и с возможной организацией взаимодействия с базой данных на объектно-ориентированном языке.

Уровни архитектуры: серверное приложение может быть реализовано в виде консольного приложения или GUI-приложения. Настройки сервера должны меняться без изменения исходного кода (аргументы командной строки, config-файлы и т.д.). Клиентское приложение: оконное приложение с использованием стандартных библиотек пользовательского интерфейса.

Ход выполнения работы над созданием грид-приложения с архитектурой “Master-Worker” (“Мастер-Исполнители”) на языке Python

Работа грид-системы должна заключаться в следующем. Узел M (Master, или Мастер) является источником однотипных заданий для узлов типа E (Worker, или Employee – Исполнитель). На рис. 8.3 представлена структура простой грид-системы «Master-Worker» с архитектурой, масштабируемой на уровне узлов-исполнителей E (далее – просто исполнителей).

После инициирования приложений каждый узел E регистрируется на узле M, который формирует список регистрации. Далее узел M готовит для каждого узла E данные для обработки и передает их. Каждый узел E выполняет одну и ту же программу, но с различными исходными данными. По завершении обработки данных узлы-исполнители E передают результаты узлу M, который формирует

окончательный результат. Например, при численном интегрировании некоторой функции по методу прямоугольников с постоянным шагом узлы-исполнители E вычисляют высоты прямоугольников как значения функции, а узел M складывает эти значения и умножает сумму на величину шага интегрирования. Такая вычислительная модель имеет название SPMD (Single Program, Multiple Data – одна программа, множество данных) и используется для достижения параллелизма при решении различных задач.

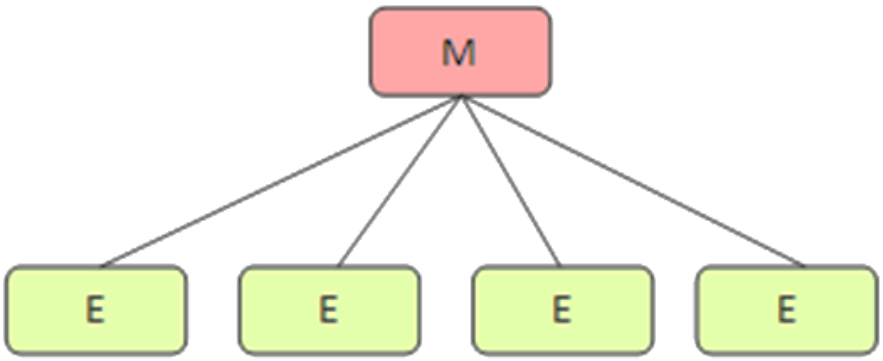


Рис. 8.3. Упрощенная структура приложения «Master-Worker»

Рассматривается пример создания грид-приложения типа «Master-Worker» («Мастер-Исполнители») для параллельной и распределенной обработки информации путем модификации простого клиент-серверного приложения на основе сетевых возможностей языка Python.

Задача для своего рода тестирования грид-приложения заключается в следующем: разработать программу для расчета определенного интеграла методом левых прямоугольников. Задан определенный интеграл:

$$\int_{0.5}^{1} e^{-x} dx.$$

Параллельная обработка должна быть реализована на уровне вычисления групп слагаемых с шагом интегрирования 0,0000001. В ходе выполнения работы необходимо на языке Python разработать собственную программу грид-приложения «Master-Worker», или «Мастер-Исполнители».

Разработка программы Master (Мастер)

Основные функции, выполняемые программой Master (далее – просто мастер):

– разбиение диапазона определенного интеграла в соответствии с числом подключаемых исполнителей;

– выделение нового потока для каждого подключаемого исполнителя;

– отправление каждому исполнителю диапазона определенного интеграла для расчета;

– выполнение суммирования результатов клиентских вычислений.

В текстах программ для соответствующего мастеру объекта сохранено имя server. Аналогично, в текстах программ для соответствующего исполнителю объекта сохранено имя client. Это связано с тем, что основная часть кода соответствует клиент-серверной архитектуре программы-прототипа.

Для того, чтобы процессы-исполнители могли связаться с мастером, в языке Python для работы с сокетами используется встроенная библиотека socket (при помощи сокетов реализуются программные интерфейсы для обеспечения информационного обмена между процессами). Мастер создает TCP/IP сокет для обмена данными при помощи метода socket, который принимает семейство адресов AF_INET (IPv4) и тип сокета SOCK_STREAM (TCP):



```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Данная часть программы является общей как для исполнительских, так и для мастерских сокетов. Теперь свяжем наш сокет с данными хостом и портом с помощью метода bind, которому передается кортеж, первый элемент которого – адрес хоста, а второй – адрес порта:



```
server.bind(("127.0.0.1", 1234))
```

Далее нам необходимо принимать соединения, введя предварительно количество подключаемых исполнителей, а затем при помощи метода listen, устанавливая для данного сокета режим прослушивания. Метод принимает один аргумент – максимальное количество подключений в очереди:



```
    numberClients = int(input("Enter the number of connec-
tions: "))
    server.listen(numberClients)
```

Далее мы можем принимать подключения с помощью метода accept, который возвращает кортеж с двумя элементами: новый сокет и адрес исполнителя. Именно этот сокет и будет использоваться для приема и посылки данных исполнителю:



```
    client_socket, address = server.accept()
```

Воспользовавшись библиотекой потоков threading, выделяем под исполнителя поток и далее передаем диапазон для вычисления определенного интеграла:



```
    listen_accepted_client = threading.Thread(target=lis-
ten_client, args=(client_socket, address, newIntegral))
    listen_accepted_client.start()
```

В самом потоке происходит отправка диапазона определенного интеграла исполнителю, мастер и исполнитель передают байты, и поэтому, чтобы передать сообщение, нужно воспользоваться командой encode("utf-8"). Исполнитель использует тот же стек протоколов, тот же адрес и порт. Исполнитель должен получить информацию в байтах, а затем декодировать сообщение decode("utf-8"). Аналогично происходит передача информации и в обратном направлении:



```
    client.send(str(integral['upper']).encode("utf-8"))
    client.recv(2048).decode("utf-8")
    client.send(str(integral['lower']).encode("utf-8"))
    client.recv(2048).decode("utf-8")
    client.send(str(integral['step']).encode("utf-8"))
    client.recv(2048).decode("utf-8")
```

Получив результат расчета определенного интеграла от исполнителя, мастер производит суммирование и закрывает соединение. Обратите внимание на примитивные объекты блокировки класса Lock() модуля threading, где Lock.acquire() устанавливает блокировку, а Lock.release() снимает блокировку. Эти методы синхронизации позволяют избежать состояния гонки потоков в борьбе за общий ресурс:



```
locker.acquire()
sumAreas += float(client.recv(2048).decode("utf-8"))
locker.release()
client.close()
```

Дождавшись завершения работы всех исполнителей, мастер выводит в консоль результат и время расчета интеграла:



```
print(f"...\nsumAreas: {sumAreas} in {time.time() - start}s")
```

Разработка программы Worker (Исполнитель)

Основные задачи исполнительной части:

– принять диапазон определенного интеграла для расчета;

– вычислить определенный интеграл согласно методу левых прямоугольников;

– отправить результат расчета мастеру.

Для того, чтобы установить связь с мастером, исполнительная программа должна создать сокет. Создание исполнительного сокета аналогично сокету мастера:



```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

При помощи метода connect подключаемся к сокету мастера:



```
client.connect(("127.0.0.1", 1234))
```

После подключения к мастеру, получаем диапазон определенного интеграла:



```
integral['upper'] = float(client.recv(2048).decode("utf-8"))
client.send(' '.encode("utf-8"))
integral['lower'] = float(client.recv(2048).decode("utf-8"))
client.send(' '.encode("utf-8"))
integral['step'] = float(client.recv(2048).decode("utf-8"))
client.send(' '.encode("utf-8"))
```

Исполнители выполняют вычисление определенного интеграла согласно методу левых прямоугольников; полученные частичные результаты отправляются мастеру и соединения закрываются:



```
while (integral['lower'] < integral['upper']):
sumAreas += integral['step']*math.exp(-integral['lower'])
integral['lower'] += integral['step']
client.send(str(sumAreas).encode("utf-8"))
client.close()
```

Исходный код программы мастера server.py:



```
import time
import socket
import threading

sumAreas = 0.0  # результат расчета интеграла
locker = threading.Lock()   # синхронизировать потоки при помощи
блокировок

def listen_client(client, address, integral):
   global sumAreas

   # Отправить данные для расчета исполнителю
   client.send(str(integral['upper']).encode("utf-8")) # отправить
верхнюю границу интеграла исполнителю
   client.recv(2048).decode("utf-8") # ждать ответа от исполнителя
```

```
   client.send(str(integral['lower']).encode("utf-8")) # отправить
нижнюю границу интеграла исполнителю
   client.recv(2048).decode("utf-8") # ждать ответа от исполнителя
   client.send(str(integral['step']).encode("utf-8"))  # отправить
шаг интеграла исполнителю
   client.recv(2048).decode("utf-8") # ждать ответа от исполнителя

   # Принять результат расчета интеграла исполнителем
   locker.acquire()    # установить блокировку
   print("Server: connected to ", address)
   print(integral)
   sumAreas += float(client.recv(2048).decode("utf-8"))    # сум-
мировать полученные от исполнителей результаты
   locker.release()    # снять блокировку

   client.close()  # Закрыть соединение

def main():
   # Инициализация мастера
   server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  #
создать сокет
   server.bind(("127.0.0.1", 1234))    # связать наш сокет с дан-
ным хостом и портом
   numberClients = int(input("Enter the number of connections: "))
# вести количество подключенных исполнителей
   server.listen(numberClients)    # запустить для данного сокета
режим прослушивания
   print("...")

   # Инициализация данных
   integral = {
       'upper': 1.0,
       'lower': 0.5,
       'step': 0.0000001
   }   # инициализация интеграла

   bias = (integral['upper'] - integral['lower']) / numberClients
# разбить диапазон на несколько интервалов

   # Ждать подключения N исполнителей
   for i in range(numberClients - 1):
```

```
      client_socket, address = server.accept()    # принять под-
ключение
      newIntegral = {
          'upper': integral['lower'] + bias,
          'lower': integral['lower'],
          'step': integral['step']
      }   # создать новые диапазоны для каждого исполнителя (см.
"разбить диапазон...")

      # создать новый поток:
      # [target = это функция, которая вы-полняется в потоке],
      # [args = это кортеж, в котором хранятся аргументы, переда-
ваемые в вызываемую функцию]
      listen_accepted_client = threading.Thread(target=lis-
ten_client,

args=(client_socket, address, newIntegral))

      listen_accepted_client.start()  # запустить созданный поток
      integral['lower'] += bias   # прибавить смещение (bias) к
нижней границе интеграла (см. "разбить диапазон...")

   # частный случай обработки последнего интеграла (верхняя гра-
ница последнего интеграла = верхней границе исходного интеграла
   #см. "разбить диапазон...")
   client_socket, address = server.accept()
   listen_accepted_client = threading.Thread(target=listen_client,
args=(client_socket, address, integral))
   listen_accepted_client.start()
   main_thread = threading.main_thread()   # получить экземпляр
основного потока

   # Ждать завершение работы N исполнителей
   start = time.time() # засечь время

   for t in threading.enumerate():
      # список "threading.enumerate()" включает в себя основной
поток,
      # так как присоединение основного потока самого к себе вы-
зывает взаимоблокировку, то его необходимо пропустить
      if t is main_thread:
```

```
            continue
        t.join()

    print(f"...\nsumAreas: {sumAreas} in {time.time() - start}s")
# вывести результат расчета интеграла + время расчета интеграла

if __name__ == "__main__":
    main()
```

Например, если разбить данный диапазон для четырех исполнителей, то получим: (1.0 – 0.5) / 4 = 0.125. Это смещение (bias = 0.125) нужно прибавить к нижней границе интеграла ('lower': 0.5) и так до предпоследнего клиента (numberClients – 1). ОБРАТИТЕ ВНИМАНИЕ, что верхняя граница последнего интеграла должна быть равна верхней границе исходного интеграла ('upper': 1.0).

В итоге получаем четыре промежуточных результата:



```
    '''
    - {'upper': 0.625, 'lower': 0.5,   'step': 1e-07}  |  1-й ис-
полнитель
    - {'upper': 0.75,  'lower': 0.625, 'step': 1e-07}  |  2-й ис-
полнитель
    - {'upper': 0.875, 'lower': 0.75,  'step': 1e-07}  |  3-й ис-
полнитель
    - {'upper': 1.0,   'lower': 0.875, 'step': 1e-07}  |  4-й ис-
полнитель
    '''
    bias = (integral['upper'] - integral['lower']) / numberCli-
ents
```

Исходный код программы исполнителя client.py:



```
import math
import socket

def main():
```

```
    # Инициализация исполнителя
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  #
создать сокет
    client.connect(("127.0.0.1", 1234)) # подключиться к мастеру

    # Инициализация данных
    integral = {'upper': 0.0, 'lower': 0.0, 'step': 0.0} # инициа-
лизация интеграла
    sumAreas = 0.0  # результат расчета интеграла

    integral['upper'] = float(client.recv(2048).decode("utf-8"))
# принять верхнюю границу интеграла от мастера
    client.send(' '.encode("utf-8"))    # отправить ответ мастеру
    integral['lower'] = float(client.recv(2048).decode("utf-8"))
# принять нижнюю границу интеграла от мастера
    client.send(' '.encode("utf-8"))    # отправить ответ мастеру
    integral['step'] = float(client.recv(2048).decode("utf-8")) #
принять шаг интеграла от мастера
    client.send(' '.encode("utf-8"))    # отправить ответ мастеру

    # Программа расчета интеграла (метод левых прямоугольников)
    while (integral['lower'] < integral['upper']):
        sumAreas += integral['step'] * math.exp(-integral['lower'])
        integral['lower'] += integral['step']

    client.send(str(sumAreas).encode("utf-8"))  # отправить резуль-
тат расчета интеграла мастеру
    client.close()  # Закрыть соединение

if __name__ == "__main__":
    main()
```

Результат работы программы

Программа-исполнитель:

Перед запуском программы-исполнителя убедитесь, что программа-мастер уже запущена и никаких действий более выполнять не требуется. Программа автоматически отработает и завершит свое выполнение.

Обратите внимание на то, что не рекомендуется запускать данную программу в IDLE, так как в ней потоки не заменяются в процессе выполнения. Клиент, запущенный несколько раз, будет реализован не различными потоками, а одним, что вызовет бесконечное ожидание.

Программа-мастер

После запуска программы-мастера необходимо ввести количество подключаемых исполнителей и программа автоматически перейдет в режим ожидания соединения от них. Никаких действий более выполнять не требуется. Для выхода из программы использовать комбинацию клавиш CTRL+C. После подключения исполнителя в окне программы отобразится сообщение об успешном подключении (рис. 8.4). (Обратите внимание на то, что многопоточная программа [Multithreading] выполняется в несколько раз быстрее, нежели программа, выполняемая в главном потоке [Main thread])

```
Enter the number of connections: 3
...
Server: connected to  ('127.0.0.1', 64702)
{'upper': 0.6666666666666666, 'lower': 0.5, 'step': 1e-07}
Server: connected to  ('127.0.0.1', 64703)
{'upper': 0.8333333333333333, 'lower': 0.6666666666666666, 'step': 1e-07}
Server: connected to  ('127.0.0.1', 64704)
{'upper': 1.0, 'lower': 0.8333333333333333, 'step': 1e-07}
...
sumAreas: 0.2386512743470955 in 0.7198209762573242s
```

Рис. 8.4. Запуск программы

Ответом является значение переменной sumAreas = 0,23865.

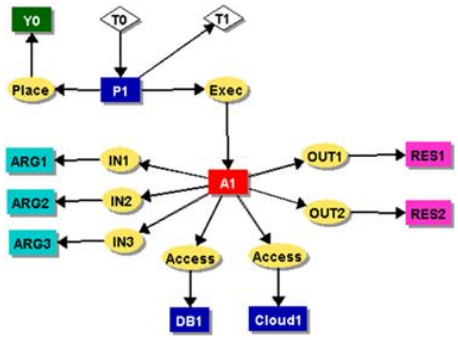
Для сравнения результатов при тестировании приложения данная задача численного интегрирования решалась и с помощью других технологий. Использование технологий параллельного программирования MPI (Message Passing Interface), OpenMP (Open Multi-Processing) и OpenCL (Open Computing Language) для вычисления значения интеграла по методу трапеций дало в точности такой же результат, как и использование распределенного приложения Master-Worker.

Программа Master-Worker может быть модифицирована и для других применений: поиск информации в распределенных базах данных, поиск информации в Интернете, организация виртуального вычислительного кластера, облачный сервис и др.

ЧАСТЬ II

СЕТЕВЫЕ И ЛОГИЧЕСКИЕ МОДЕЛИ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА, ОСНОВАННЫЕ НА ЗНАНИЯХ И ПРАВИЛАХ

Глава 9



Программирование интеллектуальных систем с событиями

Всякий раз, когда с задачей удается связать граф, обсуждение резко упрощается, и большие фрагменты словесного описания заменяются манипуляциями с картинками.
Ю. И. Манин

Декларативный и процедурный (императивный) подходы в программировании

В предыдущей, восьмой, главе были рассмотрены принципы создания распределенных сетевых приложений в архитектуре «Мастер – Исполнители», являющейся модификацией широко распространенной клиент-серверной сетевой архитектуры. Более сложные приложения для использования в асинхронных, параллельных и распределенных системах требуют поиска новых подходов в программировании и автоматическом синтезе программ. Один из таких подходов состоит в том, чтобы обеспечить поддержку необходимой семантики предметно

ориентированных сетевых приложений. Трудно даже перечислить эти предметные области, в которых используются методы информатики и программирования. Как высказался ряд известных ученых в области информатики и программирования (точное авторство до сих пор не установлено): «Информатика – не более наука о компьютерах, чем астрономия – наука о телескопах…». Предметные области приложений методов информатики и программирования включают исследование операций, моделирование мозга, исследование химических реакций, анализ военных конфликтов, политические системы, экономику и макроэкономические события, транспортные потоки на дорогах, биологические популяции, семантические сети для представления естественного языка, языкознание и литературоведение.

Среди средств поддержки многих предметных областей – экспертные системы имитационного моделирования, а также методы и средства, созданные в связи с развитием концепций Семантической паутины.

При программировании мы применяли до сих пор процедурный, или императивный, подход, в процессе которого вычисления описываются в терминах операторов, изменяющих состояние программы. Декларативное программирование, с другой стороны, отражает логику вычисления, не описывая способ его выполнения. Императивное программирование похоже на то, что делают компьютеры – последовательное выполнение инструкций. На основе декларативного подхода к решению некоторой задачи, сформулированной на естественном языке, можно описать фразу при помощи логического формализма – языка исчисления предикатов, обладающего ясной формальной семантикой, включающей правила вывода. С практической точки зрения нас больше интересуют концептуальные графы и семантические сети, представляющие графическую версию исчисления предикатов. При изучении материала настоящей главы от читателя требуется знание элементов алгебры высказываний, а также начальные знания о графах, предикатах, функциях и отношениях.

Главным итогом реализации декларативно-процедурного метода программирования в настоящей главе будет программа на языке Python. Некоторые примеры сопровождаются построением небольших баз знаний на языке Пролог. Однако никаких предварительных знаний читателя об этом языке не требуется. Такие примеры иллюстрируют декларативный подход к созданию приложений и пред-

назначены для расширения кругозора у начинающего программиста. Этот подход может пригодиться при изучении и использовании языка запросов к базе данных SQL – другого декларативного языка.

Концептуальные графы и реализуемые на их основе семантические сети с событиями содержат в явной форме как декларативную, так и процедурную, или императивную, составляющие модели представления знаний о предметной области. Ниже представлено название и гиперссылки на загрузки интеллектуального редактора концептуальных графов CharGer Conceptual Graph (CG) Editor 3.6, который использован при подготовке настоящей главы. Он предназначен для исследователей и практиков, которые хотят создавать и редактировать концептуальные графы для базы знаний. CharGer поддерживает большинство функций CG, включая узлы-концепты, узлы-отношения, акторы и контексты.

CharGer Conceptual Graph Editor 3.6

Интеллектуальный редактор концептуальных графов
https://charger-conceptual-graph-editor.soft112.com/
http://downloads.fyxm.net/CharGer-98186.html
https://github.com/delugach/Charger

Концептуальный граф содержит два типа узлов – узлы-концепты и узлы-отношения. Концепты представляют собой сущности предметной области, связанные отношениями. В логических выражениях концепты могут представляться предметными константами или предметными переменными, а также составными высказываниями. Отношения представляются дугами, связывающими концепты или, как в нашем случае, узлами с именами предикатов и изображаются овалами. Концепты являются аргументами предикатов и изображаются прямоугольниками. Семантические сети состоят из множества концептуальных графов, представляющих логические формулы. Следуя большинству источников, будем считать, что концептуальные графы – это подкласс семантических сетей, которым соответствует строгое описание в терминах логики предикатов первого порядка [1–6].

Далее нами при определении концептуальных графов будут использоваться бинарные и унарные отношения и одноименные им

предикаты. Унарный предикат выражает свойство концепта, а бинарный предикат – отношение между концептами.



___ Пример 9.1. Задача «Из жизни слонов»

Декларативное программирование ближе к естественному образу мышления, в котором у нас есть сущности и отношения между ними. Как, например, решить задачу об узнавании следующего объекта, анализируя следующую фразу на естественном языке:

«Объект является слоном в том случае, если он обладает свойством «быть слоном», или имеет свойство «быть родителем» и существует объект, связанный с родителем узами родства и обладающий свойством «быть слоном».

При решении этого примера нам потребуется знание некоторых самых простых элементов языка Prolog (для подготовки примера использовалась версия SWI Prolog, источник – https://www.swi-prolog.org [7]). Пролог благодаря своим особенностям используется в области искусственного интеллекта, компьютерной лингвистики и нечислового программирования в целом. В настоящей главе язык Пролог фрагментарно рассматривается как альтернатива обычным процедурным и объектно-ориентированным языкам. Один из таких языков, Python, был взят за основу в настоящем учебном пособии. Этот язык позволяет программировать любые логические конструкции, необходимые для реализации декларативно-процедурного подхода к созданию приложений в настоящей, девятой главе. Заметим, что к декларативному стилю написания программ относится также и стиль языка структурированных запросов SQL к реляционным базам данных.

Программа на Прологе состоит из фактов и правил, которые образуют базу знаний Пролог-программы, и запроса к этой базе, который задает цель поиска решений. Часто программу, написанную на Прологе, называют базой знаний. Предикат описывает отношение между объектами, которые являются его аргументами. Факты констатируют наличие заданного предикатом отношения между указанными объектами. Программа на языке Пролог строится из последовательности фактов и правил, затем формулируется утверждение

(запрос), которое Пролог пытается доказать с помощью правил. Язык сам ищет решение с помощью методов поиска и сопоставления, которые в нем заложены.



Пролог (англ. Prolog – Programming in Logic) является декларативным языком программирования: логика программы выражается в терминах отношений, представленных в виде фактов и правил.
https://www.swi-prolog.org/

Каждое слово, начинающееся с прописной буквы, является переменной.

Каждый элемент, записанный строчными буквами, есть предметная константа, представляющая объект. Предметные константы, имена которых начинаются с заглавной буквы, цифры или символа подчеркивания "_", должны быть заключены в апострофы – одиночные кавычки; только тогда такими именами будут представляться объекты.

Определим небольшую реляционную базу данных, которая содержит всего два кортежа – <*'Хати', слон* > и <*'Джамбо', 'Хати'* >, принадлежащие двум различным отношениям *конкр* и *родитель*. Переходя от понятий баз данных к понятиям логического программирования, рассмотрим два высказывания:



конкр('Хати', слон).
родитель('Джамбо', 'Хати').

где точка в конце каждого высказывания обязательна и является элементом синтаксиса в соответствии с правилами задания фактов в языке Пролог, *конкр* и *родитель* – бинарные предикатные имена (предикатные константы), а '*Хати*', *слон*, '*Джамбо*' – предметные константы. Первое высказывание означает тот факт, что *конкретный* (сокращенно *конкр*) объект с именем '*Хати*' является слоном, точнее, что

объект ‘*Хати*’ является объектом класса *слон*. Второе высказывание свидетельствует о том, что некий объект ‘*Джамбо*’ является родителем некоего объекта ‘*Хати*’. Данным высказываниям соответствуют следующие графические формы – концептуальные графы (здесь и далее используется распространенный в международной практике редактор CharGer – Conceptual Graph Editor 3.6) [8, 9] (рис. 9.1):



Рис. 9.1. Представление редяционной базы данных

Для того, чтобы узнавать, что некоторый объект *X* тоже является слоном, как и объект *Y*, введем следующее правило:



конкр(X, слон) :- родитель(X, Y), конкр(Y, слон).

То есть если некоторый объект, обозначенный предметной переменной *Y*, является конкретизацией класса *слон*, а некоторый объект, обозначенный предметной переменной *X*, является родителем объекта *Y*, то отсюда следует, что объект *X* тоже является конкретизацией класса *слон*.

Данное правило можно представить следующим концептуальным графом (рис. 9.2):

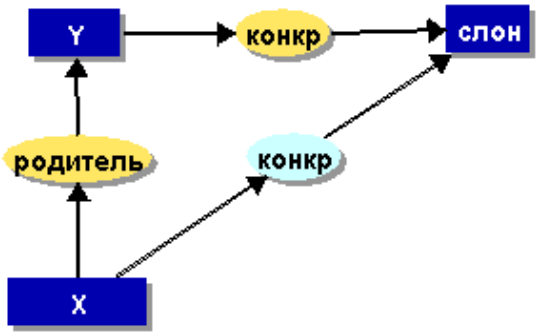


Рис. 9.2. Конкретизация класса *слон*

Дополнительная связь *конкр*, факты и правило в совокупности образуют базу знаний, или программу на языке Пролог:



конкр('Хати', слон).

родитель('Джамбо', 'Хати').

конкр(X, слон) :- родитель(X, Y), конкр(Y, слон).

Теперь, если мы хотим проверить факт, что *'Джамбо'* является слоном, то должны спросить:

?- конкр('Джамбо', слон).

Ответом будет слово *true*, так как фактически мы имеем дело с виртуальным концептуальным графом следующего вида (рис. 9.3).

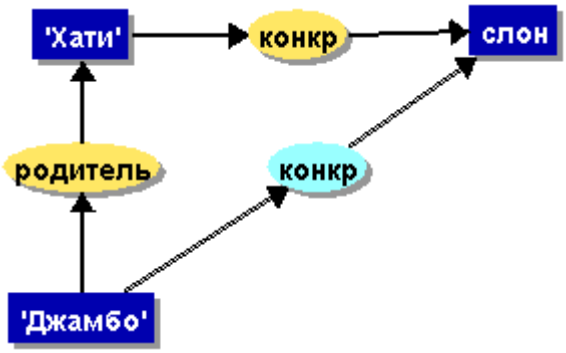


Рис. 9.3. Концептуальный граф

Далее ответом на запрос

?- конкр(X, слон).

будут два имени

X = 'Джамбо',

X = 'Хати',

так как фактически интерпретатор языка Пролог использует базу данных, представленную двумя фактами

конкр('Хати', слон).

конкр('Джамбо', слон).

где второй факт определяется в результате выполнения правила

конкр(X, слон) :- родитель(X, Y), конкр(Y, слон).

Ответом на запрос

?- родитель(X, Y).

будут два имени

X = ‘Джамбо’,
Y = ‘Хати’,

т.е. родителем ‘*Хати*’ является ‘*Джамбо*’.

Сложнее будет ответить на вопрос, какое отношение связывает ‘*Джамбо*’ и ‘*Хати*’. На следующем графе это неизвестное отношение отмечено переменной *W* (рис. 9.4).

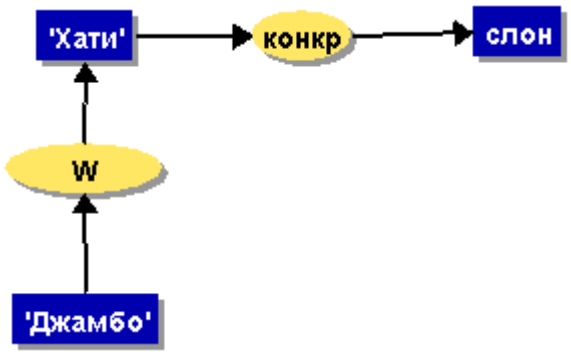


Рис. 9.4. Обозначение неизвестного отношения отмечено переменной *W*

Язык Пролог не допускает исполнения запросов такого вида:

~~?- W(‘Джамбо’, ‘Хати’)~~.

Однако проблему можно решить, заранее включив предикатную константу *родитель* в число аргументов наряду с другими именами, введя новое отношение *тройка*:

тройка(родитель, ‘Джамбо’, ‘Хати’)

и включив его в базу знаний:



конкр(‘Хати’, слон).

родитель(‘Джамбо’, ‘Хати’).

конкр(X, слон) :- родитель(X, Y), конкр(Y, слон).

тройка(родитель, ‘Джамбо’, ‘Хати’).

Теперь можно легко ответить на вопрос, связанный с нахождением отношения между концептами, обозначающими объекты ‘*Джамбо*’ и ‘*Хати*’. Запрос имеет следующий вид (рис. 9.5).

?- тройка(W, ‘Джамбо’, ‘Хати’).

Ответом будет *W* = *родитель*, то есть *Джамбо* является родителем *Хати*.

Развивая описание особенностей такого животного, как слон, приведем пример доопределенного для рассматриваемой задачи концептуального графа (рис. 9.5).

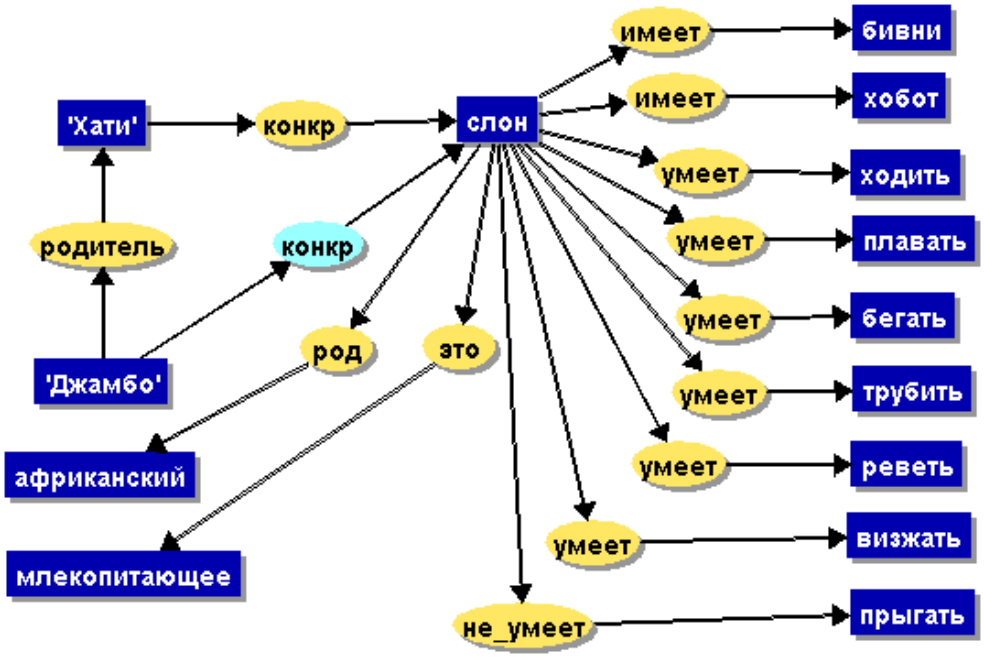


Рис. 9.5. Пример доопределенного для рассматриваемой задачи концептуального графа

Переходя к языку Пролог, получим следующие факты и добавим правило к результирующей базе знаний:



```
умеет(слон, бегать).
умеет(слон, трубить).
не_умеет(слон, прыгать).
это(слон, млекопитающее).
имеет(слон, бивни).
имеет(слон, хобот).
умеет(слон, реветь).
умеет(слон, визжать).
умеет(слон, плавать).
умеет(слон, ходить).
конкр('Хати', слон).
род(слон, африканский).
родитель('Джамбо', 'Хати').
конкр(X, слон) :- родитель(X, Y), конкр(Y, слон).
```

Факт *конкр('Джамбо', слон*) в состав приведенных фактов не входит и определяется при помощи правила *конкр(X, слон) :- родитель(X, Y), конкр(Y, слон).*

К фактам добавлено рассмотренное ранее правило, поэтому приведенные строки следует называть базой знаний о предметной области. Правило позволяет заключить, что не только '*Хати*', но и '*Джамбо*' обладают признаками и умениями слона. Например, ответом на запрос

```
?- умеет('Джамбо', X).
```

будет перечисление всех умений класса слонов, которыми обладает и конкретный слон '*Джамбо*'.

В рассмотренном примере нет описаний событий в том плане, в котором они используются при определении семантических сетей и систем имитационного моделирования. Далее будут рассмотрены модели предметных областей с событиями, которые сменяют друг друга и определяют поведение объекта.

Пролог не является в полном смысле декларативным языком в том плане, что обработка знаний с его использованием зависит от порядка следования правил или фактов в базе знаний и от порядка следования предикатов в теле правил. Это означает, что язык Пролог обладает и некоторыми процедурными, или императивными, свойствами. В дальнейшем при моделировании и программировании интеллектуальных систем с событиями нам потребуется логическое, сетевое и событийное представление знаний о семантике предметной области, что может быть достигнуто путем интеграции декларативного и процедурного (императивного) подходов. В научном плане эти проблемы решались в работах [10–16].

Логическое, сетевое и событийное представление знаний о семантике предметной области

Современные приложения интеллектуальных систем исключительно разнообразны; интеллектуальные системы являются составной частью информатики, которая, по одному из распространенных определений, является наукой о рациональной, преимущественно автоматической обработке информации для поддержки человеческих знаний и коммуникаций в технической, экономической и социальной областях. Выбор средств для манипуляции с предметной областью зависит от ее семантики. Например, смысл языковых конструкций и всей программы в целом отражают семантическую сторону программирования.



Семантическая паутина (от англ. *semantic web*) – общедоступная глобальная семантическая сеть, формируемая на базе Всемирной паутины путем стандартизации представления информации в виде, пригодном для машинной обработки.

Известен ряд языков для описания семантики предметной области. Основой для описания различных предметных областей при

создании интеллектуальных систем могут быть, например, принципы организации «Семантической паутины» – Semantic Web, которая призвана сделать размещенную в ней информацию более понятной для компьютеров [17, 18] и являющейся надстройкой над существующей «Всемирной паутиной» – World Wide Web. Особенностью предложенных принципов построения Semantic Web является повсеместное использование унифицированных идентификаторов – URI (англ. *Uniform Resource Identifier*), которые представляют собой последовательность символов, идентифицирующих абстрактный или физический ресурс. Свои глобально уникальные URI в Semantic Web есть не только у веб-страниц, но и у объектов реального мира – людей, городов, художественных произведений, а также у свойств, например, таких как «имя», «должность», «цвет». Программа-клиент может непосредственно извлекать из «паутины» факты и делать из них логические заключения.

Консорциумом «Всемирной паутины» (W3C) созданы стандарты: модель RDF (Resource Description Framework), языки OWL (Web Ontology Language), SPARQL, SWRL (Semantic Web Rule Language). Для первоначального знакомства с данными технологиями предлагаем ознакомиться с учебно-методическим пособием [19]. В этом пособии показано, как использование стандартов позволяет переносить разработанную базу знаний между разными средствами работы с ней. Например, можно создавать онтологию с использованием инструментальных средств с удобным графическим интерфейсом, например, с системой Protégé (https://protege.stanford.edu).



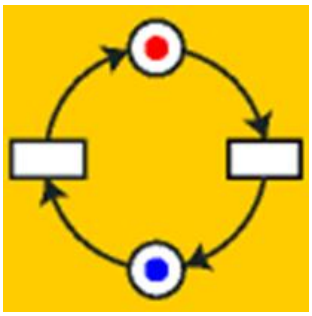
SWI-Prolog for the (semantic) web
https://www.swi-prolog.org/web/

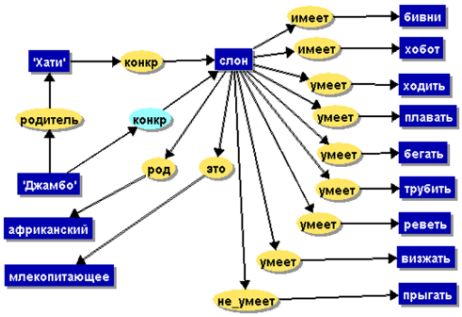
Семантическая паутина – одна из самых многообещающих областей применения SWI-Prolog. SWI-Prolog естественным образом обрабатывает семантическую модель Web Resource Description Framework (RDF), где RDF обеспечивает стабильную модель представления знаний с общей семантикой. SWI-Prolog также вполне способен предоставлять веб-службы (HTTP), особенно когда речь идет о динамическом создании HTML-страниц и предоставлении данных для веб-приложений Javascript с использованием сериализации JSON (источник: https://www.swi-prolog.org/web/).

Целью настоящей главы является автоматизация представления фактов и правил вывода при последующей программной реализации данного подхода в интеллектуальных событийных системах на примере конкретных предметных областей человеческой деятельности. Подобно Semantic Web, основой для представления интеллектуальной информации о предметной области будет семантическая сеть или формально определенный концептуальный граф, пригодные для последующей машинной обработки.

Методологическая основа исследования предметной области ориентирована на использование имитационного моделирования интеллектуальных событийных систем, в которых взаимодействия компонент задаются локально. В настоящей главе реализована и проиллюстрирована примерами методика синтеза концептуальных логических сетей на основе выявления общей семантики концептуальных графов и логических сетей Петри, в результате чего построены модели, обладающие декларативными и процедурными свойствами.

Обоснование интеграции концептуальных графов и сетей Петри





& = ?

Концептуальный граф представляет собой графическое представление логики, основанное на семантических сетях, используемых в моделях искусственного интеллекта, и на экзистенциальных графах Ч. С. Пирса. Несколько версий концептуальных графов были разработаны и внедрены в течение последних лет. При исследовании различных видов концептуальных графов разработаны новые методы проведения рассуждений, представления знаний и изучения семантики естественного языка [2, 3, 20, 21].

Концептуальные графы представляют собой графическую интерпретацию формул в исчислении предикатов первого порядка. Семантические сети, называемые иногда концептуальными сетями, часто формируются как композиции концептуальных графов частного вида типа «звезда». Каждый концептуальный граф при этом содержит центральную вершину, или первичный концепт, связанный бинарными отношениями с несколькими вторичными концептами. Однако часто семантические сети и концептуальные графы с контекстным наполнением рассматриваются как одно и то же понятие, если речь не идет о точном формульном описании в рамках первопорядковой логики предикатов [1].

Концептуальные графы и сети Петри успешно применяются при моделировании дискретно-событийных систем. В концептуальных графах могут быть определены концепты-события и причинно-следственные связи между ними. Для участников событий могут быть определены исполняемые ими роли. Сети Петри фактически представляют собой модели, похожие на семантические сети и концептуальные графы, в том смысле, что для них определены отношения между объектами-позициями и объектами-переходами [22–27].

Использование реализованного программно [28, 29] аппарата сетей Петри помогает бороться с тупиками в различных системах. Например, тупиковая ситуация создается, когда две прикладных программы взаимно блокируют необходимые им данные, и в результате ни одна из этих программ не может продолжать работу. Для устранения нарушения условий ожидания дополнительных ресурсов Хавендер (Havender J. W.) предложил следующую стратегию. Каждый процесс должен запрашивать все требуемые ему ресурсы сразу, причем не может начать выполняться до тех пор, пока все они не будут ему предоставлены. Если же процесс удерживает определенные ресурсы и получает отказ в выделении ему дополнительных ресурсов, то он должен освободить свои первоначальные ресурсы и при необходимости запросить их снова вместе с дополнительными. Периодичность повторных запросов ресурсов можно определить, например, на основе анализа имитационной модели на основе временной сети Петри. Можно действовать и в соответствии с правилом, согласно которому каждый процесс

может иметь только один ресурс в каждый момент времени. Если нужен второй ресурс, процесс должен освободить первый. Эту ситуацию можно описать сетью Петри, а затем разрешить ее.



Кроссплатформенная система PIPE2, версия 4.3.0 (Platform Independent Petri Net Editor), представляющая собой свободно распространяемый программный инструментарий, написанный на языке Java, для графического представления, редактирования, моделирования и анализа сетей Петри https://sourceforge.net/projects/pipe2/files/PIPEv4/PIPEv4.3.0/

Сеть Петри, рассматриваемая в качестве модели дискретных процессов, имеет две составляющие – *статическую* и *динамическую*. Статическая составляющая представлена двудольным графом с начальной маркировкой позиций. Эта компонента порождает динамическую компоненту – модель, основанную на смене маркировок позиций по инициативе переходов.

Определение концептуальных графов и сетей Петри показывает их сходство: графы обеих моделей являются двудольными и отношения инцидентности одинаковы по форме. Главное же различие заключается в том, что с позиций искусственного интеллекта концептуальными графами описывается декларативная составляющая модели представления знаний о предметной области, в то время как сетью Петри описывается как декларативная, так и императивная (процедурная) составляющая. Интеграция средств описания и системного анализа сетевых моделей обоих классов может повысить качество проектирования моделируемых систем. Важная разновидность сетей Петри – логические, или бинарные, сети Петри, рассматриваемые в целях последующей интеграции их с концептуальными графами, будут описаны далее в настоящей главе.



Раскрашенные сети Петри (Colored Petri Nets) – это дискретно-событийный язык моделирования, объединяющий сети Петри с языком функционального программирования Standard ML. Сети Петри обеспечивают основу графического представления и основные примитивы для моделирования параллелизма, коммуникаций и синхронизации http://cpntools.org/2018/01/09/colored-nets/

В концептуальных графах аргументы, или концепты, обозначающие понятия предметной области, на рисунках часто обозначаются прямоугольниками, а отношения между этими понятиями обозначаются овалами [1]. Овалами также обозначаются унарные отношения, характеризующие свойства объектов. Концептуальный граф, таким образом, может быть описан простой логической формулой, представляющей собой конъюнкцию бинарных и унарных предикатов. Принят международный стандарт ISO/IEC 24707, специфицирующий синтаксис и семантику моделей концептуальных графов общего вида – CGIF (англ. *Conceptual Graphs Interchange Format*). Исследования в области концептуальных логических моделей в настоящее время сопровождаются разработкой и использованием множества формальных и неформальных расширений [3, 30].

Определение модифицируемых концептуальных графов

Основная используемая в настоящей главе терминология, касающаяся определения и графического представления концептуальных графов, иллюстрируется на рис. 9.6. Унарные предикаты характеризуют свойства объектов, представленных концептами-позициями, а бинарные предикаты задают логические связи между этими объектами.

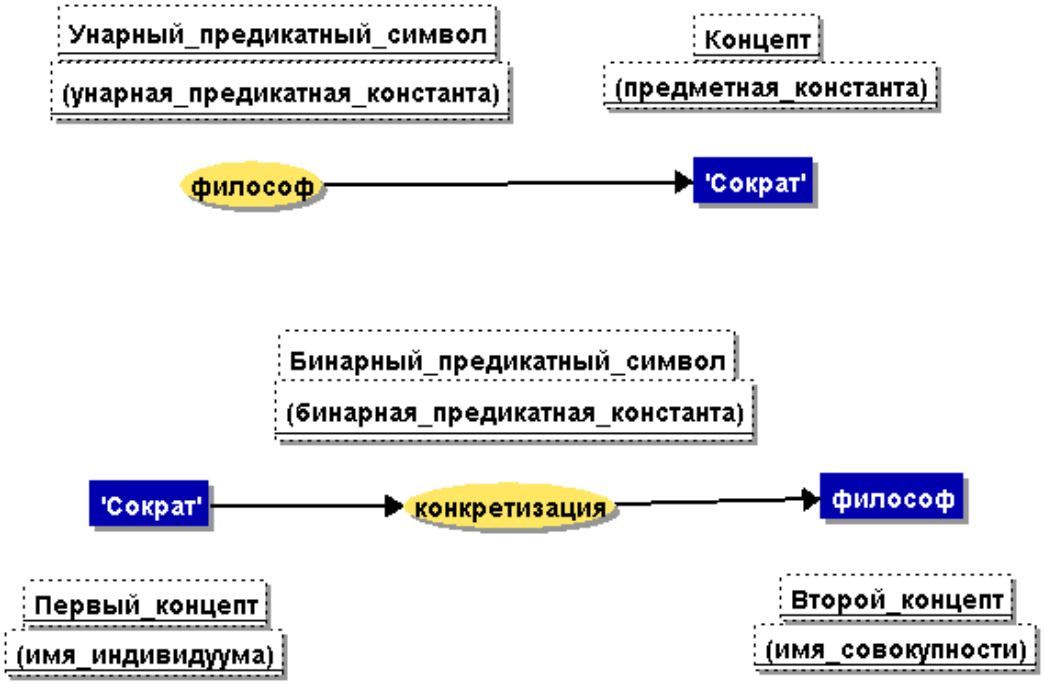


Рис. 9.6. Представление фразы «Сократ – философ» концептуальными графами:
***а* – унарным предикатом; *б* – бинарным предикатом**

В связи с тем, что вводимый формализм основан на логике высказываний и логике предикатов первого порядка, необходимо уточнить некоторые используемые понятия и определения. Рассмотрим различные способы логического и концептуального описания некоторой фразы, например, утверждения «Сократ – философ», отдавая дань уважения великому древнегреческому философу. На верхней части рис. 9.1 дано графическое представление следующего высказывания:

$$философ(\text{‘}Сократ\text{’}), \qquad (9.1)$$

где *философ* – унарное предикатное имя, или унарная предикатная константа; ‘*Сократ*’ – предметная константа; на нижней части рис. 9.1 дано представление той же самой фразы следующим высказыванием:

$$конкретизация(\text{‘}Сократ\text{’}, философ). \qquad (9.2)$$

Здесь для бинарного предиката *конкретизация* определены: в качестве значения первого аргумента – имя индивидуума, а

в качестве значения второго аргумента – имя совокупности или класса. В концептуальном графе первый и второй аргументы предиката представлены *концептами* (обозначены прямоугольниками), или *сущностями* предметной области. Овалом обозначен сам *предикат* или одноименное *отношение*, рассматриваемое как область истинности указанного предиката.

Выражения (9.1), (9.2) и соответствующие им концептуальные графы на рис. 9.6, естественно, различны, однако они представляют фразу, имеющую один и тот же смысл.

При интеграции концептуальных графов и сетей Петри в объединенной модели представления знаний о предметной области появляется возможность учета многих видов отношений: причинно-следственных (каузальных), предшествования – следования, родовидовых, класс – подкласс, целое – часть, синонимии и антонимии, логических, функциональных, атрибутивных, количественных, пространственных, временных, лингвистических, характерных для интеллектуальных систем.

Обычный концептуальный граф представляет собой размеченный двудольный ориентированный граф:

$$CG_1 = (Pred, Arg, U), \tag{9.3}$$

где $Arg \cap Pred = \varnothing$, вершины из множества *Pred* размечены именами бинарных предикатов, а вершины из множества *Arg* – именами аргументов; $U \subseteq (Pred \times Arg) \cup (Arg \times Pred)$ – отношение инцидентности, определяющее множество дуг. Размеченные вершины из множества *Pred* называются концептуальными предикатами, а размеченные вершины из множества *Arg* – вершинами-концептами, или просто концептами. Бинарным предикатам соответствуют одноименные бинарные отношения (по определению, отношение является областью истинности предиката). По определению дуг, они могут связывать предикатные вершины с вершинами-концептами и наоборот; не допускается непосредственное связывание вершин из одних и тех же множеств *Pred* и *Arg*. Вместо приведенного определения концептуальных графов вида CG_1 чаще используются их формульные описания в рамках логики высказываний и первопорядковой логики предикатов.

При учете того факта, что часть отношений может иметь функциональный характер, следует расширить определение (9.3):

$$CG_2 = (Pred, Func, Arg, U_p, U_f), \tag{9.4}$$

где $Arg \cap Pred = \varnothing$, $U_p \subseteq (Pred \times Arg) \cup (Arg \times Pred)$ – отношение инцидентности, определяющее множество дуг, связывающих предикатные вершины с аргументами-концептами; $U_f \subseteq (Func \times Arg) \cup (Arg \times Func)$ – из области определения отношение инцидентности, определяющее дуги, связывающие функциональные вершины с аргументами-концептами функций с областью их значений, $Pred \cap Func = \varnothing$, $Arg \cap Func = \varnothing$.

Модифицируемый концептуальный граф определяется следующим кортежем:

$$CG_3 = (Pred, Func, Arg, U_p, U_f, V_p, V_f), \qquad (9.5)$$

где дополнительно по сравнению с кортежем (9.4) определены два множества V_p и V_f правил модификации предикатов и функций соответственно, поэтому структура концептуального графа может изменяться при моделировании. Данные модификации выполняет внешний источник либо они осуществляются.

Определение процедурной составляющей в модели представления знаний на основе концептуального графа

Кортежами (9.3), (9.4) и (9.5) определены в основном декларативные свойства моделей представления знаний концептуальными графами. Например, модифицируемый концептуальный граф CG_3 определен кортежем (9.5), в котором процедурная составляющая знаний представлена неявно, при помощи двух множеств V_p и V_f, правил модификации предикатов и функций. Данные правила целесообразно сгруппировать в группы совместимых составных правил, образуя абстрактные модули (АМ), реализуемые впоследствии программно.

Процедурную составляющую модели представления знаний можно задать одним из следующих способов:

Во-первых, непосредственно в концептуальном графе в множество размеченных вершин-концептов *Arg* включить подмножество концептов-событий $Event \subseteq Arg$, связывая их отношениями (предикатными вершинами) предшествования-следования *Pred* или *Next* и вводя дополнительные условия, определяя алгоритм работы АМ. На этой основе возможно задавать все структурированные, так называемые дейкстровские конструкции для составляемых на основе

концептуальных графов будущих программ на конкретных алгоритмических языках [10, 31, 32].

Во-вторых, возможно определить АМ с помощью суперпозиции операций системы алгоритмических алгебр (САА) Глушкова, связывающих между собой концепты-события из множества *Event* [33].

В третьих, учитывая тот факт, что ингибиторная сеть Петри (сеть со сдерживающими дугами) является универсальной алгоритмической системой [22–24], АМ возможно определить с помощью согласованных правил срабатывания переходов подобной сети, вводя дополнительные предикаты, моделирующие условные переходы в программах. Формальное определение параллельной программы на основе сети Петри, обладающей свойствами свободного выбора, живости и безопасности, приведено в работе [24]. В определение включено множество предикатов и функция распределения предикатов по множеству позиций, обладающих свойством свободного выбора, т.е. таких позиций, которые являются единственными входными позициями для нескольких переходов.



___Пример 9.2. Интерпретация концептуального графа «Почта»

На рис. 9.7 приведен пример концептуального графа с контекстным представлением для фразы, описывающей отправку письма по электронной почте.

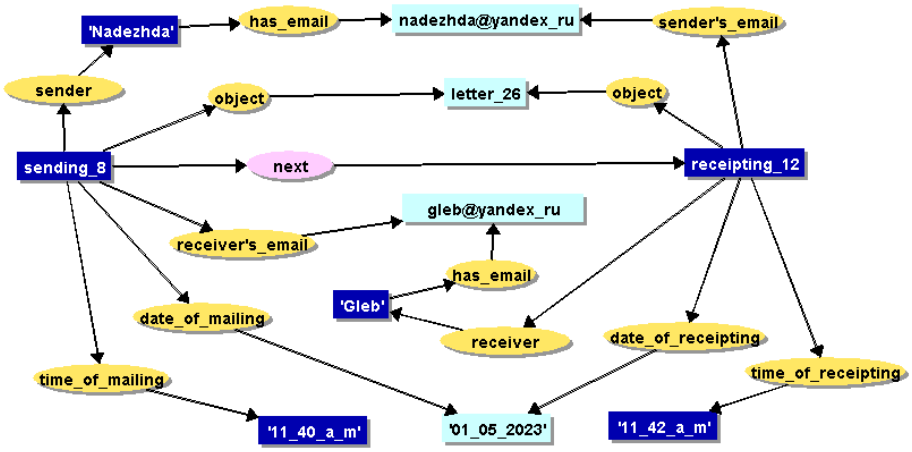


Рис. 9.7 Пример концептуального графа G_1 с контекстным представлением

Данный граф получен из двух концептуальных подграфов G_{11} и G_{12}, соединенных по правилу конъюнкции концептов *nadezhda@yandex.ru*, *letter_26*, *gleb@yandex.ru* и '*01_05_2023*', принадлежащих как одному, так и другому подграфу. Подграф G_{11} описывает событие-концепт *sending_8* отправки письма по электронной почте, а подграф G_{12} – событие-концепт *receipting_12* приема этого письма. В концептуальном графе G_1 событие-концепт *sending_8* связано с событием-концептом *receipting_12* отношением *next*. В поведенческой модели эта связь активизируется (конкретизируется) путем модификации бинарного предиката

next(*sending_8*, *receipting_12*) ← **true**,

в результате выполнения которого становится истинным высказывание *next*(*sending_8*, *receipting_12*). Это простейший случай применения правила модификации предиката.

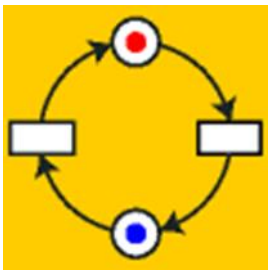
Редактор концептуальных графов *CharGer* [23, 24] позволяет по XML-коду графа G_1 получить информацию, необходимую для создания программного обеспечения интеллектуальной системы. Высказывания, приведенные ниже, получены с помощью специальной программы-переходника и полностью готовы для использования в качестве фактов для базы знаний, создаваемой, например, на языке SWI-Prolog:



```
% Facts in a Prolog program:
date_of_mailing(sending_8, '01_05_2023').
date_of_receipting(receipting_12, '01_05_2023').
has_email('Gleb', gleb_yandex_ru).
has_email('Nadezhda', nadezhda_yandex_ru).
next(sending_8, receipting_12).
object(receipting_12, letter_26).
object(sending_8,letter_26).
receiver's_email(sending_8, gleb_yandex_ru).
receiver(receipting_12, 'Gleb').
sender's_email(receipting_12, nadezhda_yandex_ru).
sender(sending_8, 'Nadezhda').
time_of_mailing(sending_8, '11_40_a_m').
time_of_receipting(receipting_12, '11_42_a_m').
```

Приведенная методика позволяет автоматизировать наиболее трудоемкий этап в создании интеллектуальных систем, а именно этап формирования базы знаний на основе концептуального графа. Ее недостатком является то, что верификация концептуальных графов распределенных алгоритмов не предусмотрена в существующих реализациях интеллектуальных редакторов концептуальных графов.

Формализация и последующая реализация третьего способа является предпочтительной, поскольку как для концептуальных графов, так и для сетей Петри разработаны подходящие редакторы и средства анализа. При интеграции этих двух моделей появится возможность создания интеллектуальных систем, где б*о*льшую роль будут играть концепты-события, роль которых в качестве элементов процедурной составляющей модели представления знаний будет усилена за счет сетей Петри. В свою очередь, сети Петри обогащаются за счет возможности указания свойств объектов, представленных переходами и позициями.



Определение сети Петри

Сеть Петри по определению, данному в основополагающих работах [22–27], представляет собой двудольный ориентированный граф, содержащий позиции, переходы, дуги и метки. Дуги связывают позиции с переходами и переходы с позициями. Переходы, их входные и выходные позиции могут представлять события обработки данных с входной и выходной логикой. При анализе динамического поведения системы, моделируемой сетью Петри, исследуется процесс смены ее состояний, моделируемый путем запуска, или срабатывания, переходов. При срабатывании какого-либо перехода изменяются состояния его входных и выходных позиций, что сопровождается перемещением меток, или маркеров, из одних позиций в другие. Метка – это примитивное понятие для сетей Петри и вводится в дополнение к позициям и переходам.

Официально история сетей Петри начинается с 1962 г., когда К. Петри защитил диссертацию и опубликовал ее под названием *"Kommunikation mit Automaten"*. Современная литература, связанная с сетями Петри, обширна и разнообразна. Было продемонстрировано, что сети Петри являются мощным инструментом моделирования различных типов дискретных динамических систем, управляемых событиями. Исследование проводилось во многих отраслях, каждая из которых изучала важный и перспективный аспект применения этого формализма.

Сетью Петри называется размеченный маркированный двудольный ориентированный граф $PN = (P, T, F, W, M_0)$, где $P = \{p_0, p_1, \ldots, p_m\}$ – конечное множество позиций; $T = \{t_1, t_2, \ldots, t_n\}$ – конечное множество переходов; $P \cap T = \varnothing$; $P \cup T \neq \varnothing$; $F \subseteq (P \times T) \cup (T \times P)$ – отношение инцидентности, или отношение потока, определяющее множество дуг; W: $F \to \{1, 2, 3, \ldots\}$ – функция кратности дуг (отметим, что существует также определение сети Петри с кратными дугами на основе использования понятия мультимножества для входных и выходных позиций переходов); M_0: $P \to \{0, 1, 2, \ldots\}$ – функция начальной маркировки позиций. В процессе срабатывания переходов происходит перемещение меток и, как следствие, смена маркировок. Подробное описание правил срабатывания переходов и связанных с ними смен маркировок дано в работах [22–25]. Для интеграции сетей Петри с концептуальными логическими моделями наиболее интересен случай, когда W: $F \to \{1\}$, т.е. кратность всех дуг равна единице и, как следствие, по каждой дуге может передаваться только одна метка; M_0: $P \to \{0, 1\}$ – при любой достижимой в сети маркировке в каждой позиции смогут находиться не более, чем по одной метке (такая сеть Петри называется безопасной, или 1-ограниченной, и может быть легко трансформирована в логическую, или бинарную, сеть).

Для практического применения аппарата сетей Петри, как и многих других поведенческих формализмов, необходима программная поддержка имитационного моделирования и построения графа достижимых состояний. Для сетей Петри это графы достижимых разметок позиций. В ряде случаев может быть полезен анализ сетей Петри на основе матричных уравнений.

Другая сторона приложений сетей Петри в отличие от имитационного моделирования связана с реализацией аппаратного и

программного обеспечения для параллельных и распределенных вычислительных систем. Одной из первых работ в области исследования связей сетей Петри с программированием является монография [23]. В частности, в этой работе большое внимание уделено семантике структур управления в программах, реализованных на основе формализма сетей Петри. Рассмотрена семантика и логика реализации событийного, потокового и динамического управления, причем условия готовности формулируются как логическая функция от некоторых событий.

Известен ряд разновидностей сетей Петри [22–24]: простые, автоматные, маркированные, свободного выбора, ингибиторные, правильно сформированные WF (англ. *Well Formed nets*), временные, цветные, иерархические, нечеткие, стохастические, функциональные. К комбинированным сетям Петри относятся следующие: иерархические цветные, временные цветные, иерархические цветные временные, цветные нечеткие, иерархические цветные временные нечеткие и др. Сети Петри моделируют широкий класс логических задач. Логические методы используются для анализа нечетких и логических (бинарных) сетей Петри, что позволяет создавать модели систем на основе нечеткой логики, исчислений высказываний и предикатов.

Правила срабатывания концептов-переходов в концептуальных графах для логических сетей Петри (КЛ СП) далее в настоящей работе будут представлены в форме продукционных правил, содержащих символ "→", разделяющий посылки, или антецеденты (условия, при истинности которых происходит срабатывание переходов), и следствия, или консеквенты (логические выражения, становящиеся истинными после срабатывания переходов).

Графические концептуальные и логические представления сетей Петри

В настоящее время предложено большое число модификаций сетей Петри. Особый интерес у многих исследователей вызвали приложения логических (бинарных, булевых) сетей Петри к решению разнообразных задач [34 – 42]. Некоторые формулировки и понятия

для логических сетей Петри вводились и уточнялись также в работах [10–12, 15, 41, 42]. В дальнейшем эти уточнения понадобятся при интеграции концептуальных графов с логическими сетями Петри.

Логическая, или бинарная, сеть Петри определяется как следующий кортеж из пяти элементов:

$$LN = (P, T, F, H, R_0), \tag{9.6}$$

где $P = \{P_1, P_2, \ldots, P_m\}$ – конечное множество позиций;

$T = \{T_1, T_2, \ldots, T_n\}$ – конечное множество переходов, $P \cup T \neq \varnothing$, и $P \cap T = \varnothing$;

$F(p, t)$: $P \times T \to \{\mathbf{true}, \mathbf{false}\}$, $p \in P$, $t \in T$ – входной по отношению к переходу бинарный предикат, при интерпретации которого определяются дуги, ведущие от позиций к переходам;

$H(t, p)$: $T \times P \to \{\mathbf{true}, \mathbf{false}\}$, $p \in P$, $t \in T$ – выходной по отношению к переходу бинарный предикат, при интерпретации которого определяющий дуги, ведущие от переходов к позициям;

$R_0(p)$: $P \to \{\mathbf{true}, \mathbf{false}\}$, $p \in P$ – начальная интерпретация унарного предиката разметки $R(p)$.

Логическая сеть Петри с дополнительными ингибиторными (сдерживающими) и информационными дугами определяется как кортеж с семью элементами :

$$LNII = (P, T, F, F_{Inh}, F_{Inf}, H, R_0), \tag{9.7}$$

где бинарные предикаты F_{Inh} и F_{Inf} задают дуги с дополнительными свойствами:

$F_{Inh}(p, t)$: $P \times T \to \{\mathbf{true}, \mathbf{false}\}$, $p \in P$, $t \in T$ – входной по отношению к переходу бинарный предикат, при интерпретации которого определяются ингибиторные, или сдерживающие, дуги, ведущие от позиций к переходам; например, ингибиторная дуга (P_i, T_j), определенная истинным высказыванием $F_{Inh}(P_i, T_j)$ является разрешающей по отношению к переходу T_j в том случае, если $R(P_i) = \mathbf{false}$, т.е. в случае, если в позиции P_i отсутствует метка;

$F_{Inf}(p, t)$: $P \times T \to \{\mathbf{true}, \mathbf{false}\}$, $p \in P$, $t \in T$ – входной по отношению к переходу бинарный предикат, при интерпретации которого определяются информационные дуги, ведущие от позиций к переходам; например, информационная дуга (P_i, T_j), определенная истинным высказыванием $F_{Inf}(P_i, T_j)$, является разрешающей по отношению к переходу T_j в том случае, если $R(P_i) = \mathbf{true}$, т.е. в случае, если

в позиции P_i присутствует метка; после срабатывания перехода T_j метка остается в прежней позиции.

Функционирование определенных выше логических сетей Петри, определенных кортежами (9.6) и (9.7), как и обычных сетей Петри, описывается формально с помощью продукционных правил срабатывания переходов, определяющих множество последовательностей срабатываний и множество достижимых разметок позиций. Дополнительные ингибиторные дуги в логических сетях Петри приходится вводить по той причине, что при моделировании часто неизвестно, истинно ли, например, высказывание $R(P_j)$ для позиции P_j, являющейся выходной для некоторого перехода T_i и связанной с ним дугой (T_i, P_j). При обычном, без «участия» ингибиторной дуги (P_j, T_i), срабатывании перехода T_i и при истинности высказывания $R(P_j)$ поступление новой метки вызовет лишь подтверждение истинности этого высказывания. Эта ситуация является ошибочной, так как эквивалентна потере метки. Введение ингибиторной дуги (P_j, T_i), что эквивалентно выполнению присваивания $F_{Inh}(P_j, T_i) :=$ **true**, позволит избежать этой ошибки путем запрета на срабатывание перехода T_i в данной ситуации.

Родственные логическим сетям Петри безопасные (1-ограниченные) сети Петри с ингибиторными (тормозящими, или сдерживающими) дугами эквивалентны ординарным безопасным сетям Петри. В работе [24] подчеркивается, что «введение дополнительных тормозящих (ингибиторных) дуг, т.е. модификация исходных спецификаций на языке ординарных сетей Петри, часто дает возможность проектировщику «исправлять» допущенные ошибки… методически модификация исходной сети Петри путем перехода от ординарной безопасной сети Петри к сети Петри с дополнительно вводимыми тормозящими связями, а затем – вновь к ординарной сети Петри весьма удобна».

Интеграция декларативного подхода на основе графической формы логической модели представления знаний с исполнимой моделью на основе сети Петри, также представимой в виде графического образа, позволит расширить состав моделей для различных предметных областей. На начальном этапе интеграции определяется логическая сеть Петри, которая может быть получена путем простой трансформации из безопасной, или 1-ограниченной сети Петри, проблема безопасности для которой разрешима [22–24].

На основе данных определений (9.6) и (9.7) возможно представление логических сетей Петри *LN* и *LN*II в форме концептуальных графов. На рис. 9.8–9.10 представлены примеры концептуальных сетей Петри (КЛ СП).

Представление логических сетей Петри концептуальными графами имеет свои особенности. Как концепты-позиции, так и концепты-переходы обозначены прямоугольниками и отличаются друг от друга только именами (и, если это возможно, цветом). Это связано с особенностями реализации используемого редактора концептуальных графов CharGer. Предикатам F, F_{Inh}, F_{Inf}, H соответствуют овалы. В целях иллюстрации для указания начальной разметки можно воспользоваться графическим представлением унарного предиката разметки R, как показано на рис. 9.8 для позиций P_1 и P_2. При реализации симулятора логической сети Петри разметку можно было бы отмечать цветом фона или именем графического объекта.

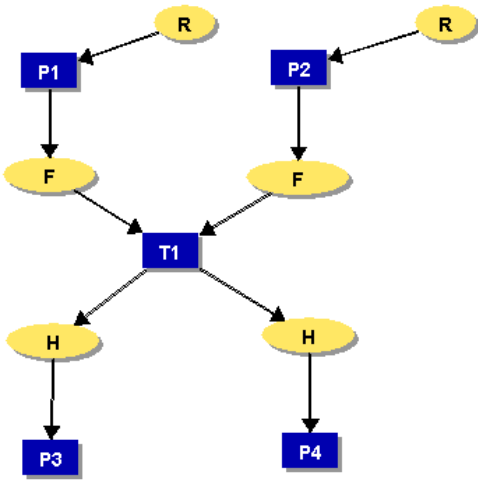


Рис. 9.8. Концептуальный граф для логической сети Петри с явно определенной разметкой позиций

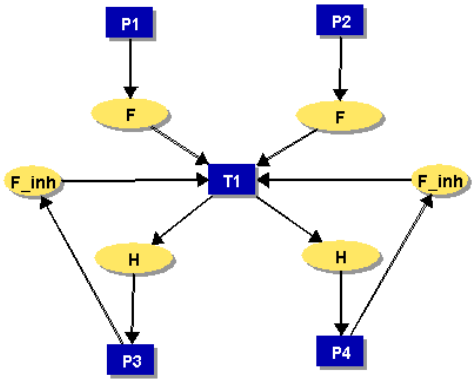


Рис. 9.9. Концептуальный граф для логической сети Петри с обычными и ингибиторными дугами

Редактор концептуальных графов позволяет получить факты, или высказывания, соответствующие структуре КЛ СП на рис. 9.8, которые автоматически включены в конъюнкцию высказываний, содержащих имена бинарных и унарных предикатов и предметных констант, обозначающих концепты-переходы и концепты-позиции:

There is a Proposition where:

H of T1 is P3 and

H of T1 is P4 and

F of P1 is T1 and

F of P2 is T1 and

R is P1 and

R is P2.

The Final Form of the Compound Proposition is:

F(P1, T1) & F(P2, T1) & H(T1, P3) & H(T1, P4) & R(P1) & R(P2). (9.8)

Здесь первая фраза означает, что далее следует составное высказывание, а вторая фраза – что сформировано составное высказывание в окончательной форме. Истинность выражения (9.8) свидетельствует о том, что граф КЛ СП сформирован.

Примечание. Здесь и далее прямым шрифтом выделены формулы в том виде, в котором они сформированы программой-редактором концептуальных графов по их внутреннему представлению в формате языка разметки XML. Другие формулы, представленные в обычном курсивном математическом стиле, сформированы вручную.

Правило срабатывания концепта-перехода T_1 при начальной разметке $R(P_1)$ = **true** и $R(P_2)$ = **true** имеет следующий вид:

$$T_1\colon R(P_1)\ \&\ R(P_2) \rightarrow \neg\, R(P_1)\ \&\ \neg\, R(P_2)\ \&\ R(P_3)\ \&\ R(P_4). \qquad (9.9)$$

Предполагается, что концепты-позиции P_3 и P_4 до срабатывания концепта-перехода T_1 не были размечены, т.е. были ложны высказывания $R(P_3)$ = **false** и $R(P_4)$ = **false**.

Если отсутствие меток в выходных позициях нельзя гарантировать, то в концептуальный граф следует добавить ингибиторные дуги $F_{inh}\,(P_3, T_1)$ и $F_{inh}\,(P_4, T_1)$, связывающие выходные концепты-позиции с концептом-переходом. Составные высказывания, описывающие граф КЛ СП на рис. 9.9, формируются редактором концептуальных графов автоматически:

There is a Proposition where:

F_inh of P3 is T1 and

F_inh of P4 is T1 and

F of P1 is T1 and

F of P2 is T1 and

H of T1 is P3 and

H of T1 is P4.

The Final Form of the Compound Proposition is:

F_inh(P3, T1) & F_inh(P4, T1) & F(P1, T1) & F(P2, T1) & H(T1, P3) & H(T1, P4).

Правило срабатывания перехода T_1 в этом случае записывается следующим образом:

$$T_1\colon R(P_1)\ \&\ R(P_2)\ \&\neg\ R(P_3)\ \&\ \neg\ R(P_4) \rightarrow \neg\ R(P_1)\ \&\ \neg\ R(P_2)\ \&\ R(P_3)\ \&\ R(P_4). \qquad (9.10)$$

В выражении для правила (9.10) учтено, что в графе КСП выходные позиции P_3 и P_4 связаны с переходом T_1 ингибиторными дугами, как и показано на рис. 9.9. От правила (9.9) правило (9.10) отличается тем, что в его левую часть добавлена конъюнкция $\neg\ R(P_3)\ \&\ \neg\ R(P_4)$.

При наличии нескольких типов дуг – обычных, ингибиторных и информационных, составные высказывания, описывающие граф КСП на рис. 9.10, формируются редактором концептуальных графов автоматически и имеют следующий вид:

There is a Proposition where:

F of P1 is T1 and

F of P2 is T1 and

F_inh of P6 is T1 and

F_inh of P5 is T1 and

F_inf of P4 is T1 and

F_inh of P3 is T1 and

H of T1 is P5 and

H of T1 is P6.

The Final Form of the Compound Proposition is:

F(P1, T1) & F(P2, T1) & F_inh(P6, T1) & F_inh(P5, T1) & F_inf(P4, T1) & F_inh(P3, T1) & H(T1, P5) & H(T1, P6).

Правило срабатывания перехода T_1 примет следующий вид:

$$T_1: R(P_1) \,\&\, R(P_2) \,\&\, \neg R(P_3) \,\&\, R(P_4) \,\&\, \neg R(P_5) \,\&\, \neg R(P_6) \rightarrow \neg \rightarrow R(P_1) \,\&\, R(P_2) \,\&\, R(P_5) \,\&\, R(P_6). \quad (9.11)$$

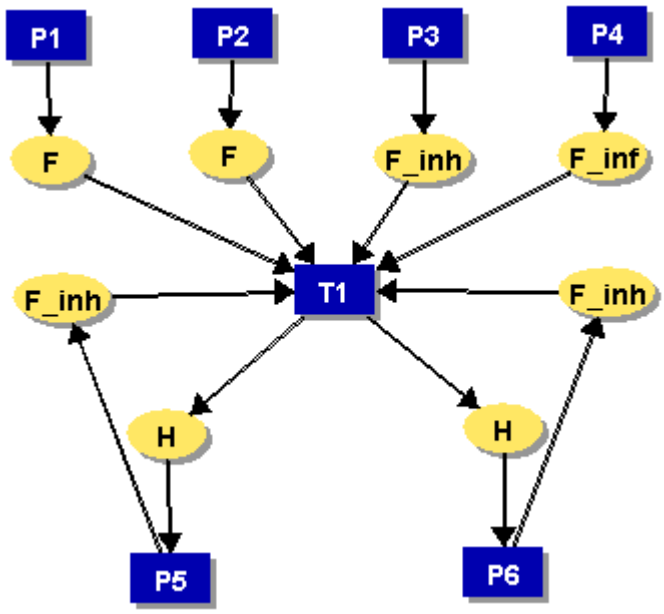


Рис. 9.10. Концептуальный граф для логической сети Петри с обычными, ингибиторными и информационной дугами

Из выражения (9.11) для этого правила следует, что после срабатывания перехода T_1 разметка позиций P_3 и P_4, связанных с данным переходом ингибиторной и информационной дугами соответственно, не изменится.

Рассматривая приведенные правила срабатывания переходов, можно заметить, что в выражениях отсутствуют высказывания с обозначениями имен предикатов H, F, F_{inh} и F_{inf}. В выражении (9.12) для правила срабатывания перехода на рис. 9.11 включение этих новых высказываний позволит учесть наличие или отсутствие связей позиций с переходами:

$$T_1: R(P_1) \,\&\, F(P_1, T_1) \,\&\, R(P_2) \,\&\, F(P_2, T_1) \,\&\, \neg R(P_3) \,\&\, \neg R(P_5) \,\&\, \neg \,\&R(P_6 \,\&\, F_{inh}(P_3, T_1) \,\&\, R(P_4) \,\&\, F_{inf}(P_4, T_1) \,\&\, F_{inh}(P_6, T_1) \,\&\, \,\&F_{inh}(P_5, T_1) \rightarrow \neg R(P_1) \,\&\, \neg R(P_2) \,\&\, R(P_5) \,\&\, H(T_1, P_5) \,\&\, \,\&R(P_6) \,\&\, H(T_1, P_6). \quad (9.12)$$

Отсутствие любой из связей $F(P_1, T_1)$, $F(P_2, T_1)$, $F_{inh}(P_3, T_1)$, $F_{inf}(P_4, T_1)$, $H(T_1, P_5)$ или $H(T_1, P_6)$ в КСП сделает невозможной передачу метки далее по сети, что эквивалентно невозможности срабатывания перехода T_1.

Определение концептуальной логической сети Петри с переменной структурой

Логическая, или бинарная, сеть Петри *LPN* (англ. *Logical Petry Net*) может быть задана следующим кортежем:

$$LPN = (A, T, R, UR), \qquad (9.13)$$

где A – множество позиций, интерпретируемых как предметные константы;

T – множество переходов, состоящее из двух непересекающихся подмножеств $T^{\&}$ и T^{*}, где $T = T^{\&} \cup T^{*}$, $T^{\&} \cap T^{*} = \varnothing$, $T^{\&}$ – подмножество переходов с конъюнктивной логикой на входах и выходах, T^{*} – подмножество переходов с конъюнктивной логикой на входах и с недетерминированным или условным выбором одного из логических правил вывода (правил модификации предиката разметки позиций $R(a)$, $a \in A$); если $T^{*} = \varnothing$, то допускается использование обозначения T для переходов типа $T^{\&}$;

R: $A \rightarrow \{$**true**, **false**$\}$ – унарный предикат разметки позиций, модифицируемый после каждого срабатывания какого-либо перехода; начальная интерпретация предиката $R(a)$, $a \in A$, соответствует начальной разметке позиций сети *LPN*;

UR – множество правил модификации (англ. *Update Rules*) унарного предиката $R(a)$, $a \in A$, в процессе срабатывания переходов; правила *UR* в неявной форме задают также и структуру связей в сети Петри типа *LPN*.

Концептуальная логическая сеть Петри КЛ СП, или *CLPN* (англ. *Conceptual Logical Petry Net*) с переменной структурой задается на основе кортежа (9.13) определения сети *LPN* следующим образом:

$$CLPN = (A, UP, MUP, BP, NBP, T, R, UR), \qquad (9.14)$$

где A – множество концептов-позиций, интерпретируемых как предметные константы;

UP – множество унарных предикатов для обозначения свойств концептов-позиций;

MUP – множество правил модификации свойств концептов-позиций;

BP – множество бинарных предикатов для обозначения отношений между концептами-позициями;

NBP – множество правил модификации отношений между концептами-позициями;

T – множество концептов-переходов, состоящее из двух непересекающихся подмножеств $T^{\&}$ и T^*, где $T = T^{\&} \cup T^*$, $T^{\&} \cap T^* = \varnothing$, $T^{\&}$ – подмножество концептов-переходов с конъюнктивной логикой на входах и выходах, T^* – подмножество концептов-переходов с конъюнктивной логикой на входах и с недетерминированным или условным выбором одного из логических правил вывода (правил модификации предиката разметки позиций $R(a)$, $a \in A$); если $T^* = \varnothing$, то допускается использование обозначения T для переходов типа $T^{\&}$;

R: $A \rightarrow \{\textbf{true}, \textbf{false}\}$ – унарный предикат разметки концептов-позиций, модифицируемый после каждого срабатывания какого-либо концепта-перехода; начальная интерпретация предиката $R(a)$, $a \in A$, соответствует начальной разметке концептов-позиций сети *CLPN*;

UR – множество правил модификации (англ. *Update Rules*) унарного предиката $R(a)$, $a \in A$, в процессе срабатывания переходов; правила *UR* в неявной форме задают также и структуру связей в сети Петри типа *CLPN*.

Элементы кортежа *UP*, *MUP*, *BP* и *NBP* при их интерпретации задают правила модификации концептуального графа, положенного в основу сети *CLPN*.

Акторы-переходы в концептуальных логических сетях Петри

Сети Петри могут рассматриваться как семантические сети с событиями частного вида или как сценарии деятельности, предназначенные для исследования статических и динамических свойств различных систем: систем отношений между людьми, последовательностей действий при выполнении какой-либо работы, последовательностей передачи сообщений в сетях ЭВМ и др.

В логической, или бинарной, сети Петри позициям соответствуют условия, которые представляются простыми или составными высказываниями, а переходы объединяют высказывания чаще всего в выражениях-конъюнкциях, истинность которых свидетельствует о готовности срабатывания переходов. При исследовании систем на основе логических сетей Петри могут быть построены как логические модели на основе декларативных языков программирования, так и продукционные модели, которые имеют не только декларативный, но и процедурный, или императивный, характер, что облегчает создание приложений на универсальных процедурных и объектно-ориентированных языках.

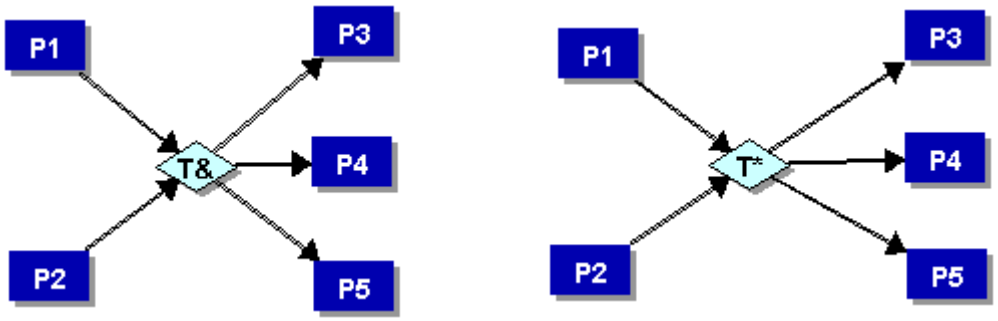
В предлагаемых в настоящей главе концептуальных логических сетях Петри (КЛ СП) для представления переходов используются так называемые *акторы*, применяемые в редакторе концептуальных графов CharGer36 [8, 9] в качестве вершин для реализации логических и арифметических функций. Новая семантика акторов-переходов предопределена таким образом, что они способны в соответствии с новой интерпретацией выполнять функции переходов сети Петри, работая в соответствии с продукционными правилами срабатывания переходов.

Определяемая далее логика событий, происходящих на входах и выходах акторов-переходов в целом соответствует известному методу графической оценки и анализа при сетевом планировании GERT [43] (англ. *Graphical Evaluation and Review Technique*), отдельные модификации которого нашли свое применение при анализе конкурирующих и параллельных программ для ЭВМ. Основным методом исследования сложных систем с применением сетей GERT и Q-GERT (модель сети GERT с очередями) является метод статистического моделирования – метод Монте-Карло. Аналитические методы исследования подобных сетей сложны и практически не реализуемы.

Методика использования акторов-переходов в концептуальных логических сетях Петри

Использование акторов-переходов иллюстрирует рис. 9.11. Количество входных (2) и выходных позиций (3) для данного примера определено из соображений компактности графов и соответствующих им правилам. Считается, что позициям соответствуют одноименные им переменные высказывания.

По умолчанию на входе и на выходе актора-перехода $T^{\&}$ (рис. 9.11,*а*) реализуются операции конъюнкции. На входе актора-перехода T^* реализуется операция конъюнкции; опционально (вручную в редакторе) на выходе актора T^* (рис. 9.11,*б*) может быть задана реализация операции выбора одной из нескольких альтернатив по условиям, недетерминированно или по вероятностям.



а) *б*)

Рис. 9.11. Акторы-переходы $T^{\&}$ и T^* в концептуальных логических сетях Петри (КЛ СП)

Выход по условиям соответствует выполнению управляющих операторов *switch* или *case*, применяемых в языках программирования. Под недетерминированным выбором подразумевается выбор одного из выходных высказываний «внешним, или независимым, наблюдателем». При вероятностном выборе, который также относится к недетерминированным, каждое исходящее действие, состоящее в появлении единственного истинного высказывания на выходе актора-перехода, имеет некоторую вероятность выполнения. Сумма вероятностей выходов должна быть равна единице.

Редактор концептуальных графов CharGer, обрабатывая внутренние XML-представления концептуальных графов на рис. 9.12,*а*,*б*, выдает следующие правила срабатывания переходов:

а) Impact Results $T^{\&}$ of P_1 and P_2 are P_3 and P_4 and P_5;

б) Impact Result T^* of P_1 and P_2 is P_3;

Impact Result T^* of P_1 and P_2 is P_4;

Impact Result T^* of P_1 and P_2 is P_5,

которые имеют следующий смысл:

– для случая *а* (рис. 9.12,*а*): результом применения актора-перехода $T^{\&}$ к истинным высказываниям P_1 и P_2 станет истинность высказываний P_3, P_4 и P_5; предполагается, что до срабатывания актора-перехода высказывания P_3, P_4 и P_5 были ложными, что для безопасной сети Петри означает отсутствие меток в одноименных позициях; далее этот факт для логических сетей Петри будет подразумеваться по умолчанию;

– для случая *б* (рис. 9.12,*б*) в результате проверки введенных условий, или недетерминированно, или по заданным вероятностям выбирается одно из трех правил; например, первое правило срабатывания перехода имеет следующий смысл: «Результатом применения актора-перехода T^* к истинным высказываниям P_1 и P_2 станет истинность высказывания P_3». Правила срабатывания переходов для случаев, когда позициям соответствуют элементарные высказывания, имеют следующий вид:

$$а)\ T^{\&}: P_1 \,\&\, P_2 \to P_3 \,\&\, P_4 \,\&\, P_5; \tag{9.15}$$

$$б)\ T^*: P_1 \,\&\, P_2 \to P_3; \quad \text{или} \quad T^*: P_1 \,\&\, P_2 \to P_4; \quad \text{или} \quad T^*: P_1 \,\&\, P_2 \to P_5. \tag{9.16}$$

Скорректированные правила содержат конъюнкции $\neg P_1 \,\&\, \neg P_2$ в правых частях, учитывающие перемещение меток через переходы:

$$а)\ T^{\&}: P_1 \,\&\, P_2 \to \neg P_1 \,\&\, \neg P_2 \,\&\, P_3 \,\&\, P_4 \,\&\, P_5; \tag{9.17}$$

$$б)\ T^*: P_1 \,\&\, P_2 \to \neg P_1 \,\&\, \neg P_2 \,\&\, P_3; \quad \text{или} \quad T^*: P_1 \,\&\, P_2 \to \neg P_1 \,\&\, \neg P_2 \,\&\, P_4; \quad \text{или} \quad T^*: P_1 \,\&\, P_2 \to \neg P_1 \,\&\, \neg P_2 \,\&\, P_5. \tag{9.18}$$

Следующие правила отличаются от предыдущих добавлением конъюнкций в левых частях, что соответствует проверкам незанятости выходных позиций:

$$а)\ T^{\&}: P_1 \,\&\, P_2 \,\&\, \neg P_3 \,\&\, \neg P_4 \,\&\, \neg P_5 \to \neg P_1 \,\&\, \neg P_2 \,\&\, P_3 \,\&\, P_4 \,\&\, P_5; \tag{9.19}$$

$$б)\ T^*: P_1 \,\&\, P_2 \,\&\, \neg P_3 \to \neg P_1 \,\&\, \neg P_2 \,\&\, P_3;$$

или

$$T^*: P_1 \,\&\, P_2 \,\&\, \neg P_4 \rightarrow \neg P_1 \,\&\, \neg P_2 \,\&\, P_4; \qquad (9.20)$$

или

$$T^*: P_1 \,\&\, P_2 \,\&\, \neg P_5 \rightarrow \neg P_1 \,\&\, \neg P_2 \,\&\, P_5.$$

Использование элементарных высказываний для одноименных позиций в формулах упрощает составление программ, интерпретирующих КЛ СП.

Следующие выражения содержат высказывания, записанные в соответствии с определениями (9.13) и (9.14) при помощи предиката R разметки позиций:

$$a)\ T^{\&}: R(P_1) \,\&\, R(P_2) \rightarrow \neg R(P_1) \,\&\, \neg R(P_2) \,\&\, R(P_3) \,\&\, R(P_4) \,\&\, R(P_5); \qquad (9.21)$$

$$б)\ T^*: R(P_1) \,\&\, R(P_2) \rightarrow \neg R(P_1) \,\&\, \neg R(P_2) \,\&\, R(P_3); \qquad (9.22)$$

или

$$T^*: R(P_1) \,\&\, R(P_2) \rightarrow \neg R(P_1) \,\&\, \neg R(\mathrm{P}_2) \,\&\, R(P_4);$$

или

$$T^*: R(P_1) \,\&\, R(P_2) \rightarrow \neg R(P_1) \,\&\, \neg R(P_2) \,\&\, R(P_5).$$

Следующие выражения для правил срабатывания переходов записаны на основе формул (9.21) и (9.22) в виде α-дизъюнкций из операций системы алгоритмических алгебр Глушкова; условия α, при истинности которых происходит срабатывание переходов, заключены в квадратные скобки, а в круглые скобки слева от символа неравнозначности "⊕" заключены одна или несколько согласованных операций модификации предиката R разметки позиций; если α-условие ложно, то переход не срабатывает, что соответствует выполнению пустого оператора E; обратной стрелкой "←" обозначена операция модификации предиката R:

$$a)\ T^{\&}: [R(P_1) \,\&\, R(P_2)](R(P_1) \leftarrow \neg R(P_1), R(P_2) \leftarrow \neg R(P_2), R(P_3) \leftarrow \mathbf{true}, R(P_4) \leftarrow \mathbf{true}, R(P_5) \leftarrow \mathbf{true} \oplus E); \qquad (9.23)$$

$$б)\ T^*: [R(P_1) \,\&\, R(P_2)](R(P_1) \leftarrow \neg R(P_1), R(P_2) \leftarrow \neg R(P_2), R(P_3) \leftarrow \mathbf{true} \oplus E); \qquad (9.24)$$

или

T^*: $[R(P_1)\ \&\ R(P_2)](R(P_1) \leftarrow \neg\, R(P_1), R(P_2) \leftarrow \neg\, R(P_2),$
$R(P_4) \leftarrow \mathbf{true} \oplus E)$;

или

T^*: $[R(P_1)\ \&\ R(P_2)](R(P_1) \leftarrow \neg\, R(P_1), R(P_2) \leftarrow \neg\, R(P_2),$
$R(P_5) \leftarrow \mathbf{true} \oplus E)$.

Позиции в КЛ СП рассматриваются как обычные концепты и могут иметь окружение в виде отношений и других концептов, как показано на рис. 9.12. Здесь овалами графически представлены бинарные отношения между концептами. Например, при функциональном характере отношений они могут указывать роли для объектов.

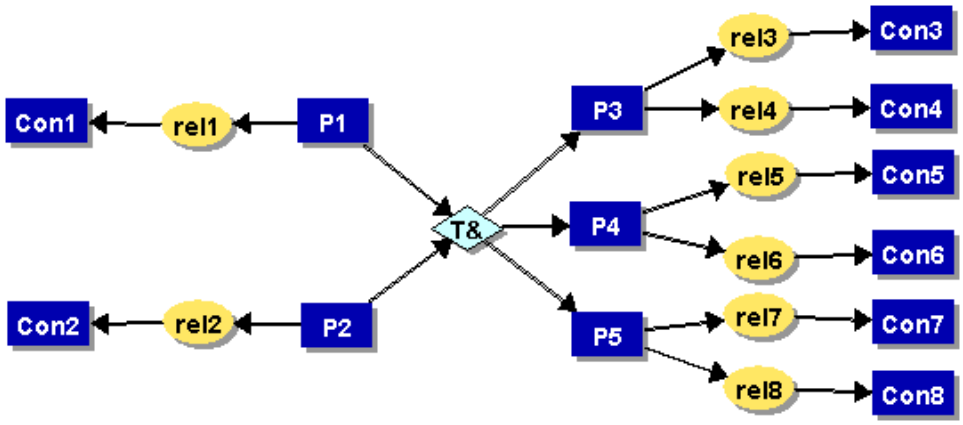


Рис. 9.12. Актор-переход в составе фрагмента концептуальной логической сети Петри (КЛ СП)

Расширенному набору операций на входах и выходах акторов-переходов КЛ СП ставятся в соответствие следующие обозначения:
$T^{\&\&}$, $(T^{\&}, T)$; $T^{\&*}$, (T^*); $T^{*\&}$; T^{**},

где первые символы “&” и “*” в верхних индексах обозначают входную логику, определенную ранее. Вторые индексы соответствуют подобной же логике, только реализуемой на выходе акторов-переходов. Рядом в скобках указаны упрощенные эквивалентные обозначения для акторов-переходов.



Построение предметно интерпретируемой концептуальной логической сети Петри для правового процесса

Уже на начальных этапах активного исследования сетей Петри и их разновидностей они использовались при моделировании большого числа систем, в том числе и отличных от вычислительных систем. Например, в основополагающей монографии [22] предлагался перечень задач на применение сетей Петри в следующих предметных областях: исследование операций, моделирование мозга, химические реакции, военный конфликт, политические системы, экономика и макроэкономические события, транспортные потоки на дорогах, биологические популяции, семантические сети для представления естественного языка, языкознание и литературоведение.

В качестве примера рассматривается гипотетический процесс подачи и продвижения условного искового заявления в юридической практике, обсуждаемый в работе [22]. Это простой процесс передачи и регистрации документов, характерный и для других областей человеческой деятельности. Процесс описывается концептуальной логической сетью Петри (КЛ СП), приведенной на рис. 9.13.

В правовых системах несколько действующих лиц (судьи, адвокаты, обвиняемые, клерки и др.) могут одновременно или последовательно выполнять действия, относящиеся к конкретному делу.

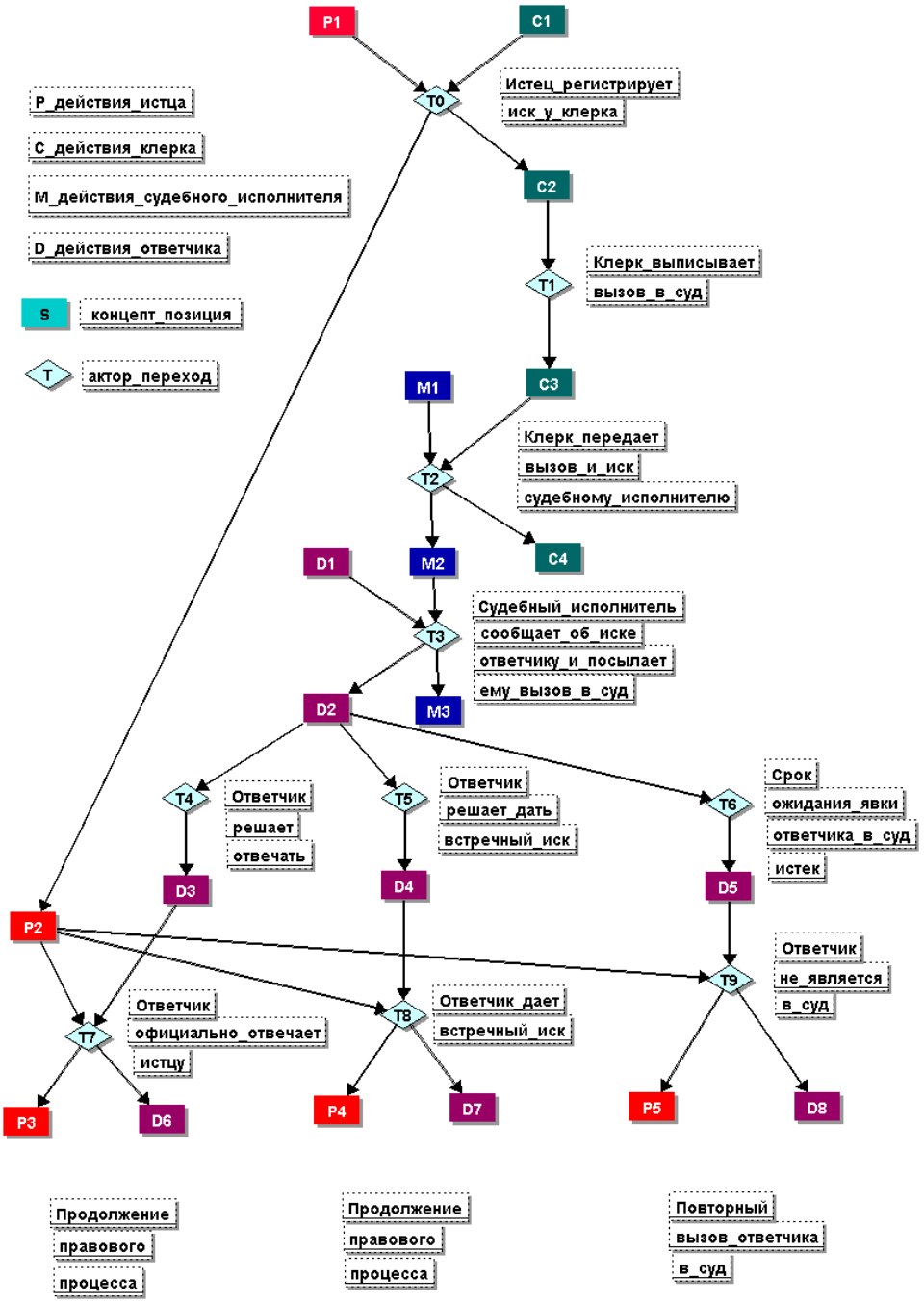


Рис. 9.13. Концептуальный граф *CLPN*$_1$ логической сети Петри (КЛ СП), описывающей процесс подачи и рассмотрения искового заявления

На рис. 9.14 в отличие от традиционных обозначений элементов обычных сетей Петри прямоугольники представляют концепты-позиции, а ромбы – акторы-переходы. Не следует путать значки

акторов-переходов с графическим изображением символа двухветвленного *if* – роль этого оператора в концептуальных логических сетях Петри выполняют концепты-позиции. Наряду с концептами-позициями в этих сетях могут использоваться обычные концепты и отношения.

Подобное описание позволит в дальнейшем упростить исследование моделируемых процессов на основе логического подхода в искусственном интеллекте, когда на начальном этапе требуется исследовать данную сеть как обычную сеть Петри и показать, что эта сеть живая и безопасная (1-ограниченная), т.е. в каждой позиции может находиться не более одной метки. Если сеть Петри обладает свойством безопасности, то далее возможно построить на ее основе логическую и простую продукционную модели представления знаний о предметной области.

Как следует из комментариев на рис. 9.13, переходы в данной сети имеют следующую содержательную интерпретацию [22]:

T_0 – ИСТЕЦ регистрирует иск у КЛЕРКА;

T_1 – КЛЕРК выписывает вызов в суд;

T_2 – КЛЕРК передает вызов и иск СУДЕБНОМУ ИСПОЛНИТЕЛЮ;

T_3 – СУДЕБНЫЙ ИСПОЛНИТЕЛЬ сообщает ОТВЕТЧИКУ об иске и вручает ему вызов в суд;

T_4 – ОТВЕТЧИК решает отвечать;

T_5 – ОТВЕТЧИК решает дать встречный иск;

T_6 – переход определяет, что время ожидания со дня передачи вызова в суд ОТВЕТЧИКУ истекло;

T_7 – ОТВЕТЧИК официально отвечает ИСТЦУ;

T_8 – ОТВЕТЧИК выставляет встречный иск;

T_9 – ОТВЕТЧИК не является в суд.

Клерк в данном процессе выполняет функции секретаря суда или делопроизводителя, работающего в канцелярии суда. Позиции в рассматриваемой сети Петри соответствуют состояниям участников процесса. Последовательности прохождения меток по концептам-позициям и акторам-переходам соответствуют процессам ИСТЦА, КЛЕРКА, СУДЕБНОГО ИСПОЛНИТЕЛЯ и ОТВЕТЧИКА:
P – процесс ИСТЦА:
$P_1 \to T_0 \to P_2 \to (T_7 \to P_3 \mid T_8 \to P_4 \mid T_9 \to P_5)$;

C – процесс КЛЕРКА:
$C_1 \to T_0 \to C_2 \to T_2 \to C_4$;
M – процесс СУДЕБНОГО ИСПОЛНИТЕЛЯ:
$M_1 \to T_2 \to M_2 \to T_3 \to M_3$;
D – процесс ОТВЕТЧИКА:
$D_1 \to T_3 \to D_2 \to (T_4 \to D_3 \to T_7 \to D_6 \vee T_5 \to D_4 \to T_8 \to D_7 \vee T_6 \to D_5 \to$
$T_9 \to D_8)$.

Символ "|" означает, что одна из последовательностей, заключенная в скобки, выбирается под воздействием внешних причин. Для процесса P ИСТЦА выбор одной из последовательностей $T_7 \to P_3 \mid T_8 \to P_4 \mid T_9 \to P_5$ зависит от поведения ОТВЕТЧИКА; для процесса D выбор одной из последовательностей $T_4 \to D_3 \to T_7 \to D_6 \vee T_5 \to D_4 \to T_8 \to D_7 \vee T_6 \to D_5 \to T_9 \to D_8$ зависит от поведения самого ОТВЕТЧИКА.

Определение концептуальной логической сети Петри для конкретной предметной области

Концептуальная логическая, или бинарная, сеть Петри для гипотетического правового процесса, представленная на рис. 9.14, формально задается следующим кортежем:

$$CLPN_1 = (A, T, R, UR), \qquad (9.25)$$

где элементы A, T, R и UR определены непосредственно для процесса подачи и прохождения искового заявления (по сравнению с сетью $CLPN$ общего вида элементы кортежа UP, MUP, BP и NBP, присутствовавшие в формуле (9.13), для данной задачи не определены):

$A = P \cup C \cup M \cup D$ – множество позиций, отмечающих состояния участников процесса;

$P = \{P_1, P_2, P_3, P_4, P_5\}$ – множество позиций, отмечающих состояния процесса ИСТЦА;

$C = \{C_1, C_2, C_3, C_4\}$ – множество позиций, отмечающих состояния процесса КЛЕРКА;

$M = \{M_1, M_2, M_3\}$ – множество позиций, отмечающих состояния процесса СУДЕБНОГО ИСПОЛНИТЕЛЯ;

$D = \{D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8\}$ – множество позиций, отмечающих состояния процесса ОТВЕТЧИКА;

$T = \{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$ – множество переходов-инициаторов действий участников процесса;

R: $A \rightarrow \{\textbf{true}, \textbf{false}\}$ – унарный предикат разметки позиций, модифицируемый после каждого срабатывания какого-либо перехода; начальная интерпретация предиката $R(a)$, $a \in A$, соответствует начальной разметке сети $CLPN_1$;

$UR = \{URT_0, URT_1, URT_2, URT_3, URT_4, URT_5, URT_6, URT_7, URT_8, URT_9\}$

– множество правил модификации предиката $R(a)$, $a \in A$, в процессе срабатывания переходов.

Выражения для правил модификации предиката разметки позиций в логической модели

В результате автоматическогог логического анализа по внутреннему XML представлению концептуального графа логической сети Петри, изображенной на рис. 9.13, автоматически получена система правил, показанная на рис. 9.14.

There are the Propositions where:
Impact Results of T0 of C1 and P1 are P2 and C2
Impact Result of T1 of C2 is C3
Impact Results of T2 of C3 and M1 are C4 and M2
Impact Results of T3 of M2 and D1 are M3 and D2
Impact Result of T4 of D2 is D3
Impact Result of T5 of D2 is D4
Impact Result of T6 of D2 is D5
Impact Results of T7 of P2 and D3 are D6 and P3
Impact Results of T8 of P2 and D4 are D7 and P4
Impact Results of T9 of P2 and D5 are D8 and P5
Impact Results of T3 of M2 and D1 are M3 and D2

Рис. 9.14. Результат логического анализа концептуального графа логической сети Петри по внутреннему XML представлению

Начальная интерпретация предиката $R(a)$, $a \in A$ соответствует состояниям готовности участников судебного процесса к действиям:

$R(P_1)$ = **true**, $R(P_2)$ = **false**, $R(P_3)$ = **false**, $R(P_4)$ = **false**, $R(P_5)$ = **false**,
$R(C_1)$ = **true**, $R(C_2)$ = **false**, $R(C_3)$ = **false**, $R(C_4)$ = **false**, $R(M_1)$ = **true**,
$R(M_2)$ = **false**, $R(M_3)$ = **false** , $R(D_1)$ = **true**, $R(D_2)$ = **false**, $R(D_3)$ = **false**,
$R(D_4)$ = **false**, $R(D_5)$ = **false**, $R(D_6)$ = **false**, $R(D_7)$ = **false**, $R(D_8)$ = **false.**

Систему правил срабатывания переходов представим в следующем виде непосредственно по правилам на рис. 9.9:

$$URT_0 : R(P_1)\,\&R(C_1) \rightarrow R(P_2)\,\&R(C_2);$$

$$URT_1 : R(C_2) \rightarrow R(C_3);$$

$$URT_2 : R(C_3)\,\&R(M_1) \rightarrow R(C_4)\,\&R(M_2);$$

$$URT_3 : R(M_2)\,\&R(D_1) \rightarrow R(M_3)\,\&R(D_2).$$

Последующие правила для переходов T_4, T_5 и T_6 несовместимы; выбор одного из этих правил в логической модели формально может осушествить, например, «внешний» наблюдатель. В реальной системе этот выбор зависит от поведения ответчика:

$$URT_4 : R(D_2) \rightarrow R(D_3);$$

$$URT_5 : R(D_2) \rightarrow R(D_4);$$

$$URT_6 : R(D_2) \rightarrow R(D_5).$$

В результате предыдущего выбора одного из правил срабатывания переходов T_4, T_5 или T_6 далее будет выполняться одно из соответствующих им правил URT_7, URT_8 или URT_9:

$$URT_7 : R(P_2)\,\&R(D_3) \rightarrow R(P_3)\,\&R(D_6);$$

$$URT_8 : R(P_2)\,\&R(D_4) \rightarrow R(P_4)\,\&R(D_7);$$

$$URT_9 : R(P_2)\,\&R(D_5) \rightarrow R(P_5)\,\&R(D_8).$$

В итоге получаем один из альтернативных ответов, о чем свидетельствует наличие меток в выходных позициях одного из сработавших переходов T_7, T_8 или T_9.

Рассмотритм, например, смысл выражения URT_0, которое соответствует срабатыванию перехода T_0:

$$URT_0 : R(P_1)\,\&R(C_1) \rightarrow R(P_2)\,\&R(C_2).$$

Это выражение означает, что если до стабатывания перехода T_0 было истинно выражение $R(P_1)\,\&R(C_1)$, то после его срабатывания будет истинно выражение $R(P_2)\&R(C_2)$.

Приведенные выше выражения полностью соответствуют концептуальному графу логической сети Петри, представленному на рис. 9.19. Однако эти данные дают лишь косвенное представление о перемещении меток, что не позволяет осуществить программную реализацию данной логической модели на процедурно ориентированных языках. Поэтому на основе данных выражений следует получить новые выражения для правил срабатывания переходов, в которых учитывается перемещение меток.

Следующие модификации предыдущих логических правил учитывают перемешение меток, выраженное, естественно, в логической форме:

$$URT_0 : R(P_1)\,\&R(C_1) \rightarrow \neg\ R(P_1)\,\&\neg R(C_1)\,\&R(P_2)\&R(C_2);$$

$$URT_1 : R(C_2) \rightarrow \neg R(C_2)\,\&R(C_3);$$

$$URT_2 : R(C_3)\,\&R(M_1) \rightarrow \neg\ R(C_3)\,\&\neg R(M_1)\,\&R(C_4)\,\&R(M_2);$$

$$URT_3 : R(M_2)\,\&R(D_1) \rightarrow \neg R(M_2)\,\&\neg R(D_1)\,\&R(M_3)\&R(D_2).$$

Например, рассмотрим подробнее следующее правило:

$$URT_0 : R(P_1)\,\&R(C_1) \rightarrow \neg\ R(P_1)\,\&\neg R(C_1)\,\&R(P_2)\&R(C_2),\ .$$

где истинности конъюнкции $R(P_1)\,\&R(C_1)$ соответствует наличие по одной метке в каждой из входных позиций перехода T_0. После срабатывания данного перехода эти метки покинут его входные позиции – это будет соответствовать тому, что оба элементарных высказывания $R(P_1) u\, R(C_1)$ станут ложными, что и отражено в формуле $\neg R(P_1)\,\&\neg R(C_1)\,\&R(P_2)\&R(C_2)$. Истинность высказываний $R(P_2) u\, R(C_2)$ соответствует появлению по одной метке в выходных полициях перехода T0.

Как ранее было декларировано, последущие авила для переходов T_4, T_5 и T_6 несовместимы, т.е. истинным может быть только одно из них: условно выбор одного из этих правил в логической модели может осушествить, например, «внешний» наблюдатель (в реальности этот выбор, как и ранее, зависит от поведения ответчика):

$$URT_4 : R(D_2) \rightarrow \neg\ R(D_2)\ \&R(D_3);$$

$$URT_5 : R(D_2) \rightarrow \neg\ R(D_2)\ \&R(D_4);$$

$$URT_6 : R(D_2) \rightarrow \neg\ R(D_2)\ \&R(D_5).$$

В результате предыдущего выбора одного из правил срабатывания переходов T_4, T_5 или T_6 далее должно выполниться одно из соответствующих правил URT_7, URT_8 или URT_9:

$$URT_7 : R(P_2)\ \&R(D_3) \rightarrow \neg R(P_2)\ \& \neg R(D_3)\ \&R(P_3)\&R(D_6);$$

$$URT_8 : R(P_2)\ \&R(D_4) \rightarrow \neg R(P_2)\ \& \neg R(D_4)\ \&R(P_4)\&R(D_7);$$

$$URT_9 : R(P_2)\ \&R(D_5) \rightarrow \neg R(P_2)\ \& \neg R(D_5)\ \&R(P_5)\&R(D_8).$$

В итоге должен быть получен один из альтернативных несовместимых ответов, о чем свидетельствует наличие меток в выходных позициях одного из сработавших переходов T_7, T_8 или T_9, которым соответствуют следующие интерпретации:

T_7 – ОТВЕТЧИК официально отвечает ИСТЦУ, $R(P_3)$ = **true** – ИСТЕЦ получил официальный ответ от ОТВЕТЧИКА, $R(D_6)$ = **true** – ОТВЕТЧИК официально ответил ИСТЦУ;

T_8 – ОТВЕТЧИК выставляет встречный иск, $R(P_4)$ = **true** – ОТВЕТЧИК выставил встречный иск ИСТЦУ, $R(D_7)$ = **true** – ИСТЕЦ получил встречный иск от ОТВЕТЧИКА;

T_9 – ОТВЕТЧИК не является в суд, $R(P_5)$ = **true** – ИСТЕЦ не получил ответ от ОТВЕТЧИКА в заданный срок, $R(D_8)$ = **true** – ОТВЕТЧИК не явился в суд в назначенное время.

Продукционная модель представления знаний о прохождении искового заявления

Систему логических правил, носящих декларативный характер, можно далее реализовать на языке Пролог. Однако представим, что для программной реализации модели на каком-либо процедурно-ориентированном языке необходимо использовать исполнимую модель представления знаний о предметной области. Под исполнимостью модели подразумевается тот факт, что она непосредственно пригодна для кодирования моделирующей программы. К подобным моделям относятся

конечные автоматы, сети Петри. Продукционные модели представления знаний о предметной области содержат как декларативную составляющую, так и процедурную. Процедурную и «исполнимую» составляющие концептуальные логические сети Петри «заимствуют», естественно, от обычных логических сетей Петри, лежащих в их основе.

Определим продукционную модель логической (бинарной) сети Петри следующим образом: элементы кортежа A, T и R определяются так же, как и для сети $CLPN_1$, а четвертым элементом кортежа является множество $PR = \{PRT_0, PRT_1, PRT_2, PRT_3, PRT_4, PRT_5, PRT_6, PRT_7, PRT_8, PRT_9\}$ продукционных правил (англ. *Production Rules*), модифицирующих предикат разметки позиций $R(a)$, $a \in A$, и определяющих порядок срабатывания переходов сети $CLPN_1$.

При построении системы продукционных правил воспользуемся рассмотренными ранее логическими правилами, которые описывали логику срабатывания переходов сети Петри. Использована следующая нотация для оператора, подобного оператору *if* в процедурно-ориентированных языках:

$$[\alpha](A \oplus B).$$

Данное выражение означает, что при истинности условия α выполняется действие A, а при его ложности – действие B. Здесь $[\alpha]$ – условная часть продукционного правила, где квадратные скобки обязательны. В круглые скобки, также обязательные, заключена операционная часть продукционного правила. Далее данное выражение будет использовано, в основном, в следующем виде:

$$[\alpha](A \oplus E),$$

где E – «пустое» действие.

В правых частях продукционных правил реализуются действия по модификация предиката $R(a)$ с помощью операции присваивания ":=" (другое допустимое обозначение – обратная стрелка "←"). Выполнению продукционных правил должна предшествовать установка начальной разметки позиций, т.е. установка начальных значений предиката $R(a)$ сети $CLPN_1$:

$$R(P_1) := \textbf{true}, R(P_2) := \textbf{false}, R(P_3) := \textbf{false}, R(P_4) := \textbf{false}, R(P_5) := \textbf{false},$$

$R(C_1)$:= **true**, $R(C_2)$:= **false**, $R(C_3)$:= **false**, $R(C_4)$:= **false**, $R(M_1)$:= **true**,

$R(M_2)$:= **false**, $R(M_3)$:= **false** , $R(D_1)$:= **true**, $R(D_2)$:= **false**, $R(D_3)$:= **false**,

$R(D_4)$:= **false**, $R(D_5)$:= **false**, $R(D_6)$:= **false**, $R(D_7)$:= **false**, $R(D_8)$:= **false.**

Далее задается последовательность продукционных правил:

$$PRT_0 : \left[R(P_1)\ \&R(C_1)\right]\left(R(P_1) := \neg R(P_1), R(C_1) := \neg R(C_1), R(P_2) := \textbf{true}, R(C_2) := \textbf{true} \oplus E\right);$$

$$PRT_1 : \left[R(C_2)\right]\left(R(C_2) := \neg R(C_2), R(C_3) := \textbf{true} \oplus E\right);$$

$$PRT_2 : \left[R(C_3)\ \&R(M_1)\right]\left(R(C_3) := \neg R(C_3), R(M_1) := \neg R(M_1), R(C_4) := \textbf{true}, R(M_2) := \textbf{true} \oplus E\right);$$

$$PRT_3 : \left[R(M_2)\ \&R(D_1)\right]\left(R(M_2) := \neg R(M_2), R(D_1) := \neg R(D_1), R(M_3) := \textbf{true}, R(D_2) := \textbf{true} \oplus E\right).$$

Последующие продукционные правила для переходов T_4, T_5 и T_6 несовместимы, т.е. выполняться может быть только одно из них; выбор одного из этих правил в модели может осуществить пользователь, вводя значения переменной K:

$$PRT_4 : \left[(K=1)\ \&R(D_2)\right]\left(R(D_2) \neg\ R(D_2), R(D_3) := \textbf{true} \oplus E\right);$$

$$PRT_5 : \left[(K=2)\ \&R(D_2)\right]\left(R(D_2) := \neg R(D_2), R(D_4) := \textbf{true} \oplus E\right);$$

$$PRT_6 : \left[(K=3)\ \&R(D_2)\right]\left(R(D_2) := \neg R(D_2), R(D_5) := \textbf{true} \oplus E\right).$$

В результате предыдущего выбора одного из правил срабатывания переходов T_4, T_5 или T_6 далее будет выполняться одно из соответствующих им правил T_7, T_8 или T_9:

$$PRT_7 : \left[R(P_2)\ \&R(D_3)\right]\left(R(P_2) := \neg R(P_2), M(D_3) := \neg M(D_3), R(P_3) := \textbf{true}, R(D_6) = \textbf{true} \oplus E\right);$$

$$PRT_8 : \left[R(P_2)\ \&R(D_4)\right]\left(R(P_2) := \neg\ R(P_2)\ \& \neg R(D_4), R(P_4) := \textbf{true}, R(D_7) := \textbf{true} \oplus E\right);$$

$$PRT_9 :\left[R(P_2)\,\&R(D_5)\right](R(P_2) := \neg\ R(P_2), R(D_5) := \neg R(D_5),$$
$$R(P_5) := \mathbf{true}, R(D_8) := \mathbf{true} \oplus E).$$

Примечание. В операционной части выражений для продукционных правил операции модификации предиката *R* разделены запятыми, означающими последовательное выполнение операций.

В итоге исполнения модели формируется один из альтернативных ответов, о чем свидетельствует наличие меток в выходных позициях одного из сработавших переходов T_7, T_8 или T_9.

Возможна ситуация, когда по каким-то причинам не удается проверить исходную сеть Петри на безопасность (1-ограниченность). Тогда для того, чтобы моделируемая сеть Петри всегда оставалась логической, достаточно замкнуть ингибиторные связи каждого перехода со своими выходными позициями, тем самым проверяя эти позиции на отсутствие меток перед срабатыванием перехода. Тогда система продукционных правил *PR* получит следующий вид:

$$PRT_0 :\left[R(P_1)\,\&R(C_1)\,\&\neg R(P_2)\,\&\neg R(C_2)\right](R(P_1) := \neg R(P_1),$$

$$R(C_1) := \neg\ R(C_1),\ R(P_2) := \mathbf{true}, R(C_2) := \mathbf{true} \oplus E)\ ;$$

$$PRT_1 :\left[R(C_2)\,\&\neg R(C_3)\right]\left(R(C_2)\ := \neg R(C_2), R(C_3) := \mathbf{true} \oplus E\right);$$

$$PRT_2 :\left[R(C_3)\,\&R(M_1)\,\&\neg R(C_4)\,\&\neg R(M_2)\right](R(C_3) := \neg R(C_3),$$

$$R(M_1) := \neg R(M_1),\ R(C_4) := \mathbf{true}, R(M_2) := \mathbf{true} \oplus E);$$

$$PRT_3 :\left[R(M_2)\,\&R(D_1)\,\&\neg R(M_3)\,\&\neg R(D_2)\right](R(M_2) := \neg R(M_2),$$

$$R(D_1) := \neg R(D_1),\ R(M_3) := \mathbf{true}, R(D_2) := \mathbf{true} \oplus E);$$

$$PRT_4 :\left[(K=1)\,\&R(D_2)\,\&\neg R(D_3)\right]\left(R(D_2) := \neg\ R(D_2), R(D_3) := \mathbf{true} \oplus E\right);$$

$$PRT_5 :\left[(K=2)\,\&R(D_2)\,\&\neg R(D_4)\right]\left(R(D_2) := R(D_2), R(D_4) := \mathbf{true} \oplus E\right);$$

$$PRT_6 :\left[(K=3)\,\&R(D_2)\,\&\neg R(D_5)\right]\left(R(D_2) := \neg R(D_2), R(D_5) := \mathbf{true} \oplus E\right);$$

$$PRT_7 : \left[R(P_2)\,\&R(D_3)\,\&\neg R(P_3)\,\&\neg R(D_6)\right](R(P_2) := \neg R(P_2),$$

$$M(D_3) := \neg M(D_3),\ R(P_3) := \mathbf{true}, R(D_6) := \mathbf{true} \oplus E);$$

$$PRT_8 : \left[R(P_2)\,\&R(D_4)\,\&\neg R(P_4)\,\&\neg R(D_7)\right](R(P_2) := \neg\ R(P_2)\ \&\ R(D_4),$$

$$R(P_4) := \mathbf{true},\ R(D_7) := \mathbf{true} \oplus E);$$

$$PRT_9 : \left[R(P_2)\,\&R(D_5)\,\&\neg R(P_5)\,\&\neg R(D_8)\right](R(P_2) := \neg R(P_2),$$

$$R(D_5) := \neg R(D_5),\ R(P_5) := \mathbf{true}, R(D_8) := \mathbf{true} \oplus E).$$

В соответствии с последней системой правил составлена программа «Иск» на языке Python, текст которой приведен ниже.



Программа «Иск» на языке Python, реализующая модель концептуальной логической сети Петри

Продукционные правила пригодны для их использования в качестве исполнимых спецификаций для компьютерной программы; другими словами, они легко программируются. Приведенная на рис. 9.15 программа на языке Python составлена при учете того факта, что выражения с предикатом R в системе продукционных правил представляют собой атомарные высказывания. Поэтому в программе вместо них используются булевы переменные, одноименные позициям концептуальной логической сети Петри.

Данный пример имеет демонстрационный характер и результаты его исследования и реализации могут после некоторых модификаций использоваться и для других областей человеческой деятельности, где участники ведут диалог, участвуют в работе над общим проектом, высказывают свои мнения о дальнейшем продвижении проекта.

Программа «Иск» (начало)

```
p1=d1=m1=c1=True
p2=p3=p4=p5=c2=c3=c4=d2=False
d3=d4=d5=d6=d7=d8=m2=m3=False
choice=0
if(p1 and c1):
    p1=not(p1)
    c1=not(c1)
    p2=True
    c2=True
    print("t0")
if(c2):
    c2=not(c2)
    c3=True
    print("t1")
  if(c3 and m1):
    c3=not(c3)
    m1=not(m1)
    c4=True
    m2=True
    print("t2")
if(m2 and d1):
    m2=not(m2)
    d1=not(d1)
    m3=True
    d2=True
    print("t3")
print("Введите номер ветки, по
которой будет выполняться
дальнейший переход: 1\t 2\t 3\t")
choice=input()
        print("t7")
    print("Ответчик официально
отвечает истцу")
```

Программа «Иск» (окончание)

```
if(choice=="1"):
    d2=not(d2)
    d3=True
    print("t4")
    if(p2 and d3):
        p2=not(p2)
        d3=not(d3)
        p3=True
        d6=True
elif(choice=="2"):
    d2=not(d2)
    d4=True
    print("t5")
    if(p2 and d4):
        p2=not(p2)
        d4=not(d4)
        p4=True
        d7=True
        print("t8")
    print("Ответчик выставляет
встречный иск")
elif(choice=="3"):
    d2=not(d2)
    d5=True
    print("t6")
    if(p2 and d5):
        p2=not(p2)
        d5=not(d5)
        p5=True
        d8=True
        print("t9")
    print("Ответчик не является
в суд")
else:
    print("Вы ввели неверные
данные!")
```

Рис. 9.15. Программа моделирования концептуальной логической сети Петри

Развитие концептуальной модели правового процесса

Ранее предполагалось, что моделирование юридических систем сетями Петри могло бы привести к лучшему пониманию их, а анализ модели мог бы привести к улучшению юридической системы. Следующий пример показывает, что представление модели правового процесса концептуальной логической сетью Петри позволяет более полно учитывать особенности предметной области за счет усиления декларативной составляющей модели представления знаний и, как следствие, повысить результативность использования модели.

Концептуальный граф логической сети Петри может включать элементы, наличествующие в традиционных концептуальных графах. Например, в графе КЛ СП на рис. 9.16 доопределены функции участников правового процесса. Результат логического анализа фрагмента концептуального графа логической сети Петри на рис. 9.16 по внутреннему XML представлению приведен на рис. 9.17.

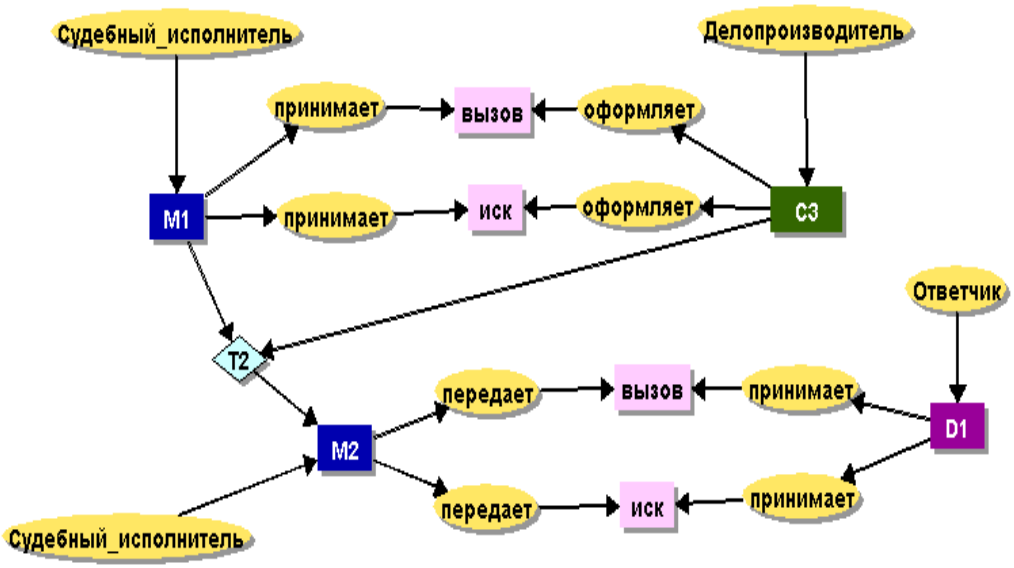


Рис. 9.16. Дополненный фрагмент концептуального графа логической сети Петри (КЛ СП)

```
There are the Propositions where:
Делопроизводитель is C3
оформляет of C3 is иск
оформляет of C3 is вызов
Судебный_исполнитель is M1
принимает of M1 is вызов
принимает of M1 is иск
Судебный_исполнитель is M2
передает of M2 is иск
передает of M2 is вызов
Ответчик is D1
принимает of D1 is иск
принимает of D1 is вызов
Impact Result T2 of M1 and C3 is M2
```

Рис. 9.17. Результат логического анализа фрагмента концептуального графа логической сети Петри по внутреннему XML представлению

Дополненному фрагменту концептуального графа логической сети Петри соответствуют получаемые автоматически следующие высказывания (факты) и правило срабатывания перехода T_2 (рис. 9.18).

```
There are the Propositions where:
        Делопроизводитель(C3),
        оформляет(C3, иск),
        оформляет(C3, вызов).
        Судебный_исполнитель(M1),
        принимает(M1, вызов),
        принимает(M1, иск).
        Судебный_исполнитель(M2),
        передает(M2, иск),
        передает(M2, вызов).
        Ответчик(D1),
        Принимает(D1, иск),
        Принимает(D1, вызов).
        Правило срабатывания перехода:
        T2:   M1 & C3 → M2.
```

Рис. 9.18. Факты и правило, формируемые автоматически по концептуальному графу КЛ СП на рис. 9.16

Полученные структурированные знания могут быть использованы при дальнейшей реализации интеллектуальной экспертной системы, сопровождающей правовой процесс.



___Пример 9.3. Составить базу знаний и правил для КЛ СП на рис. 9.16 для языка Пролог, руководствуясь данными на рис. 9.18

Ответ:

```
готов('M1').
готов('C3').
делопроизводитель('C3').
оформляет('C3', иск).
оформляет('C3', вызов).
судебный_исполнитель('M1').
принимает('M1', вызов).
принимает('M1', иск).
судебный_исполнитель('M2').
передает('M2', иск).
передает('M2', вызов).
ответчик('D1').
принимает('D1', иск).
принимает('D1', вызов).
готов('M2'):-готов('M1'),готов('C3').
```

Здесь истинности высказываний (наличию фактов в базе знаний) *готов('M1')* и *готов('C3')* соответствуют начальной разметке позиций М1 и С3 на графе КЛ СП на рис. 9.16. Правило

готов('M2') :- готов('M1'), готов('C3')

свидетельствует о том, что судебный исполнитель принимает оформленные клерком иск и вызов для ответчика и готовится к следующим действиям – отправке иска и вызова ответчику.

Концептуальный граф для процесса продвижения искового заявления

Другой вариант построения обычного концептуального графа без использования аппарата сетей Петри для процесса продвижения искового заявления в правовой системе представлен на рис. 9.19, здесь доопределены и скорректированы имена концептов и отношений.

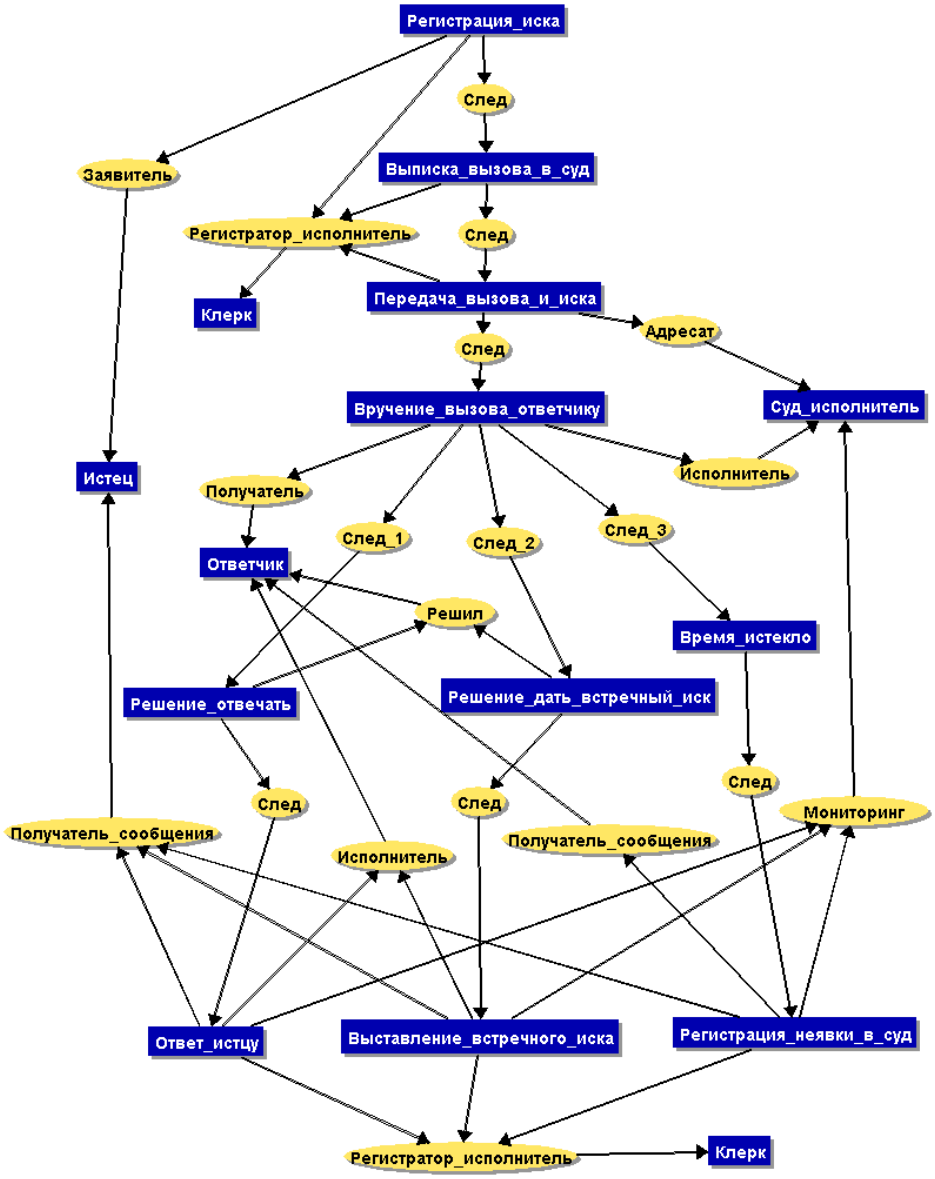


Рис. 9.19. Концептуальный граф для процесса продвижения искового заявления в правовой системе («традиционный» подход)



____Пример 9.4. Составить на основе концептуального графа на рис. 9.19 базу знаний и правил для языка Пролог

Указание. XML-код внутреннего представления данной сети далее используется для получения фактов базы знаний. Для этого надо воспользоваться редактором CharGer36 (используя команды Operation, Summarize everything и далее программу main4.exe, редактируя полученный текстовый файл «под Пролог» так, чтобы имена отношений и предметных констант начинались со строчной буквы). Возможно также использовать команды File, Show English и далее вручную отредактировать полученный текстовый файл для получения фактов базы знаний, пригодных для дальнейшего использования в программе на Прологе.

Ответ:

Факты базы знаний:

адресат(передача_вызова_и_иска, выписка_вызова_в_суд).

заявитель(регистрация_иска, истец).

исполнитель(вручение_вызова_ответчику, регистрация_неявки_в_суд).

исполнитель(ответ_истцу, выставление_встречного_иска).

мониторинг(ответ_истцу, суд_исполнитель).

получатель(вручение_вызова_ответчику,

выставление_встречного_иска).

получатель_сообщения(ответ_истцу, регистрация_неявки_в_суд).

получатель_сообщения(регистрация_неявки_в_суд, клерк).

регистратор_исполнитель(ответ_истцу, ответчик).

регистратор_исполнитель(регистрация_иска,

регистрация_неявки_в_суд).

решил(решение_отвечать, передача_вызова_и_иска).

след(время_истекло, выставление_встречного_иска).

след(выписка_вызова_в_суд, передача_вызова_и_иска).

след(передача_вызова_и_иска, вручение_вызова_ответчику).

след(регистрация_иска, выписка_вызова_в_суд).

след(решение_дать_встречный_иск, клерк).

след(решение_отвечать, ответ_истцу).

след_1(вручение_вызова_ответчику, решение_отвечать).

след_2(вручение_вызова_ответчику, решение_дать_встречный_иск).

след_3(вручение_вызова_ответчику, время_истекло).

Недостатком подобного представления знаний о предметной области является то, что процедурная составляющая модели представления знаний выражена неявно и поэтому правила, устанавливающие последовательность действий участников процесса, при необходимости придется формировать вручную. В этом и состоит преимущество в использовании модели представления знаний в виде КЛ СП, представленной на рис. 9.13.



О сложности семантического анализа фраз естественного языка *(По мотивам записок Артура Конан Дойла о Шерлоке Холмсе)*

Начало и развитие сюжета.

Место действия: Лондон, Бейкер-стрит, 221-б. Квартира миссис Хадсон.

Жильцы: доктор Ватсон, служивший в английской армии, побывавший на афганской войне, вышедший в отставку и возвратившийся на родину в Англию; мистер Шерлок Холмс – частный детектив.

– Мистер Холмс, так вы – единственный на весь мир частный детектив? – спросил доктор Ватсон, поднимая брови.

– Единственный частный детектив-консультант, – ответил Шерлок Холмс. – Последняя и высшая инстанция. Я знакомлюсь с подробностями дела и высказываю свое мнение, мнение специалиста. Я не ищу славы. Когда мне удается распутать дело, мое имя не фигурирует в газетах. Я вижу высшую награду в самой работе, в возможности применить на практике мой метод. Вы, Ватсон,

хорошо его знаете. Вспомните хотя бы дело Джефферсона Хоупа о садовых участках.

– Да, помню, ответил Ватсон, смягчаясь. – Интереснейший случай. Ответчик отозвал встречный иск. Я вас понял, Холмс. Еще есть примеры?

– Есть. Вот дело о слонах в Лондонском зоопарке…

– Да, такое не забудешь! Слон нашелся в Гайд-парке… Но давайте перейдем к более серьезным задачам.

– Миссис Хадсон, нет ли утренних газет?

– Да, мистер Холмс, вот свежая почта. Утренний выпуск «The Illustrated London News».

– Что там криминального?

– О, сколько угодно. Вот, например, из раздела «Криминальная хроника»: дело о краже в отеле. Некий ***Карл украл кораллы у Клары, а она в отместку украла у него кларнет!***

– Да, занятное дело, стоит им заняться…



Построение концептуальных графов на тему «Кража в отеле»: объектный и событийный подходы

Продолжая решать задачи определения семантической связи типа «смысл – текст», заметим, что анализ даже простых фраз приводит к сложным комбинаторным структурам на уровне семантического представления.



___Пример 9.5. Составить на основе концептуальных графов на рис. 9.20 две базы знаний на языке Пролог

Рассмотрим, например, распространенную шуточную скороговорку «*Карл у Клары украл кораллы, а Клара у Карла украла*

кларнет». На рис. 9.20 представлены концептуальные графы, составленные для данной фразы. Верхний граф построен на основе объектного подхода, что не позволяет средствами логики первого порядка установить последовательность событий. Потребовалось бы определить, например, отношение «*Позже*» между отношениями «*Украла*» и «*Владел*» и другие временные и причинно-следственные отношения второго порядка, что невозможно сделать путем графического представления отношений средствами первопорядковой логики. Логика же второго порядка, не реализованная в текущей версии языка Пролог, позволила бы задать в явной форме отношение между предикатами.

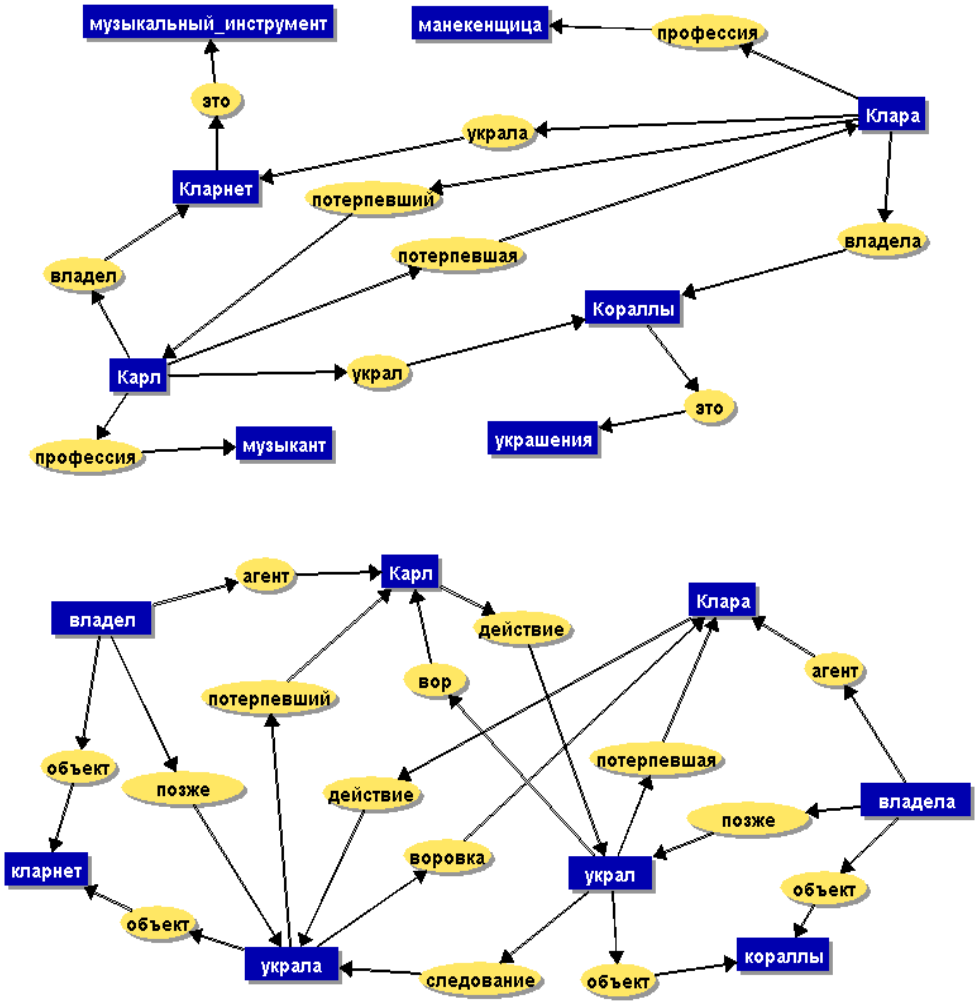


Рис. 9.20. Различные концептуальные графы для фразы «Карл у Клары украл кораллы, а Клара у Карла украла кларнет»

Нижний граф на рис. 9.20 построен на основе событийного подхода. Данная семантическая сеть с событиями называется «сценарием». Здесь уже явно определены временны*е* отношения «*Позже*» и «*Следование*». А вот отсутствующее в графе отношение «*Прекращение*» между событиями «*Украла*» и «*Владел*» могло бы указать на причинно-следственную связь между кражей и прекращением владения кларнетом.

Из рис. 9.20 видно, что одна и та же фраза может быть представлена по-разному; различна также степень полноты информации о предметной области, хотя главный смысл фразы отражен в обоих случаях.

Продолжение сюжета:

Доктор Ватсон задумался и через несколько минут снова обратился к Шерлоку Холмсу, углубившемуся в чтение утренних газет.

– Мистер Холмс! Я восхищен вашим разрешением имущественного спора между леди Кларой и джентльменом Карлом. К счастью для обоих, вам их удалось примирить! Умники из Скотланд-ярда даже не успели завести уголовное дело! Но в Скотланд-ярде часто критикуют ваши методы…

– Видите ли, Ватсон, как высказался когда-то Аристотель, великий древнегреческий философ, «есть только один способ избежать критики: ничего не делайте, ничего не говорите и будьте никем». И еще добавлю: как выразился наш великий соотечественник Сэмюэл Джонсон, «логика — это искусство приходить к непредсказуемому выводу». Вот и я, Ватсон, того же мнения и использую его на практике. Мой метод дедукции… Холмс не закончил фразу и посмотрел на утомившегося Ватсона.



… – Миссис Хадсон, что у нас сегодня на обед?

– Обед будет готов через несколько минут, господа!

– Утомил я вас логическим подходом к искусственному интеллекту, Ватсон?

– Да нет, просто я хотел бы после обеда успеть повидать мисс Морстен! Кстати, ожерелье из кораллов, украшенное жемчужинами – неплохой подарок к свадьбе… не так ли, мистер Холмс?

– Зайдите тогда в ювелирную лавку на улице Мэрилбоун Хай, у меня там знакомый ювелир.

– А, вы, наверное, вспомнили ту историю про кражу берилловой диадемы?

– Вы угадали, Ватсон! С этим делом не справились даже Грегсон, Лестрейд и Этелни Джонс из Скотланд-ярда; они были в тупике, а это их нормальное состояние, они тогда позвали меня. Кстати, мистер Ватсон, я вам мог бы рекомендовать чудесный учебник по математической логике, его написал Огастес де Мóрган (англ. Augustus de Morgan) – шотландский математик и логик, профессор математики в Университетском колледже Лондона. К своим идеям в алгебре логики Огастес де Морган пришел независимо от Дж. Буля. В 1847 он изложил элементы логики высказываний и логики классов, дал первую развитую систему алгебры отношений. С его именем связаны известные теоретико-множественные соотношения (законы де Моргана). Вы знали такого ученого, доктор Ватсон?

– Конечно, он еще известен тем, что был домашним учителем Ады Лавлейс, урожденной Байрон, которая известна прежде всего созданием описания вычислительной (аналитической) машины, проект которой был разработан Чарльзом Бэббиджем. Она же составила первую в мире программу для этой машины. Ввела в употребление термины «цикл» и «рабочая ячейка», считается первым программистом в истории. С ее заметками по искусственному интеллекту был знаком сам Алан Тьюринг!

– Господа, стол накрыт!



Построение предметно интерпретируемой концептуальной логической сети Петри, моделирующей строительство промышленного объекта

В монографии [22] приведен пример сети Петри для строительства дома; в качестве контрпримера была выбрана PERT-диаграмма (англ. *Program* (*Project*) *Evaluation and Review Technique*), используемая для оценки и анализа проектов [44]. Граф КЛ СП для данного примера приведена на рис. 9.21.

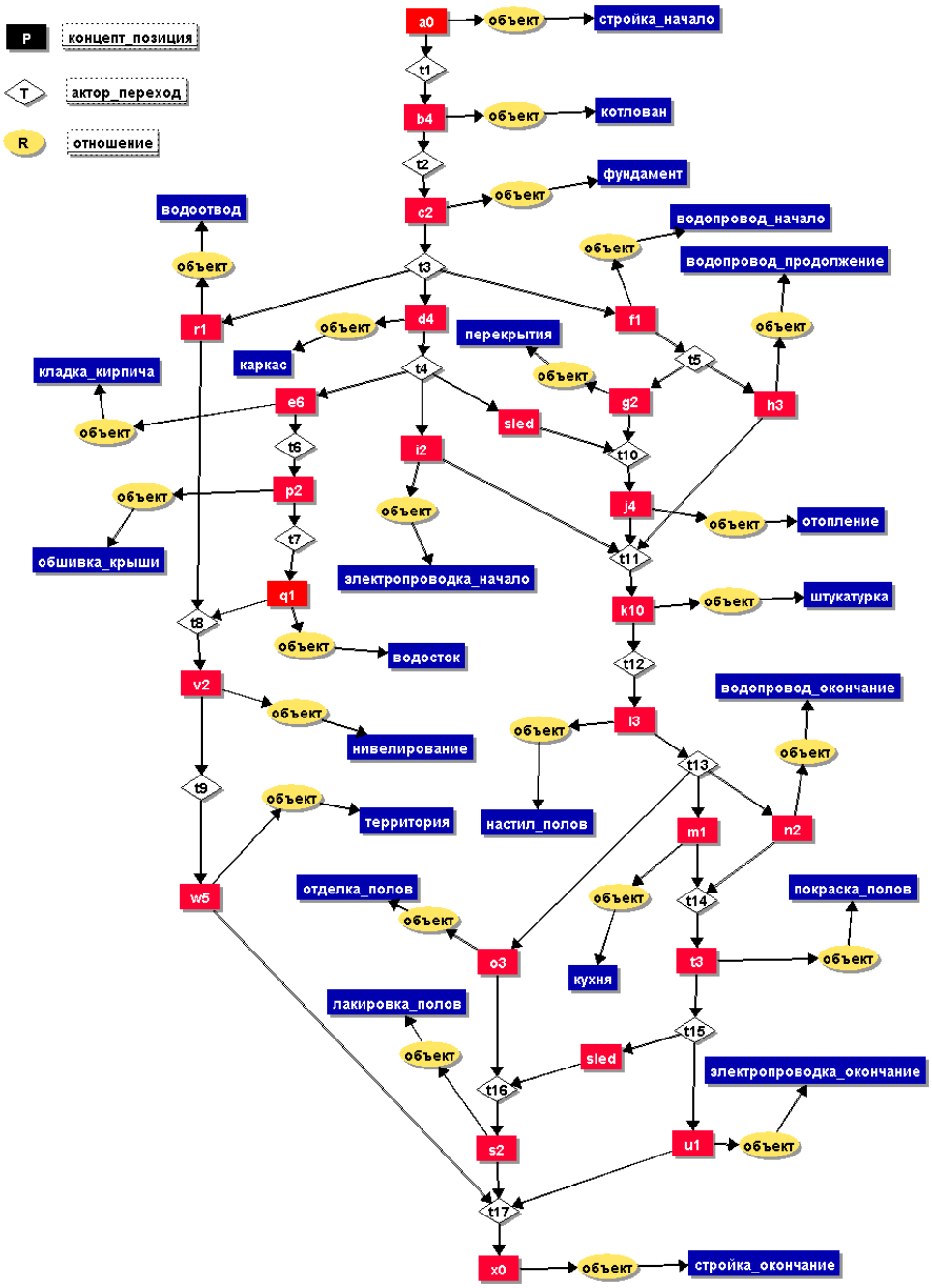


Рис. 9.21. Концептуальный граф логической сети Петри (КЛ СП), описывающий процесс строительства дома

Каждый этап PERT-диаграммы представлен позицией, а причинно-следственные связи – переходами. В отличие от сети Петри, приведенной в монографии [22], описание процесса строительства

дома сетью типа КЛ СП расширено за счет указания названий этапов. На рис. 9.21 представлен концептуальный граф логической сети Петри, описывающей процесс строительства дома. Индексы при именах объектов показывают время реализации этапа. В случае использования временных сетей Петри могут быть указаны моменты начала и окончания выполнения каждого этапа, что позволит получить те же временные характеристики, которые определяются по PERT-диаграмме. Могут быть указаны также материалы и число рабочих, занятых выполнением каждого этапа.

Факты и правила получены автоматически путем трансформации КЛ СП на рис. 9.21 в текст, содержащий факты и правила, т.е. цель исследования (рис. 9.22).

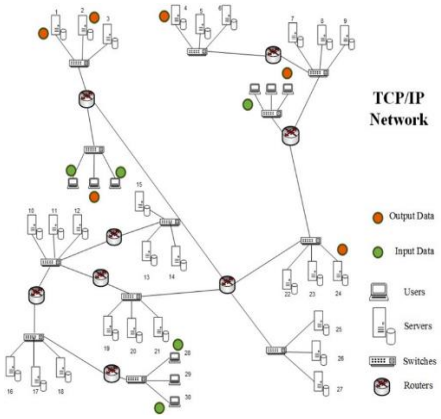
Facts	Rules
объект of a0 is стройка_начало объект of b4 is котлован объект of c2 is фундамент объект of d4 is каркас объект of r1 is водоотвод объект of f1 is водопровод_начало объект of e6 is кладка_кирпича объект of i2 is электропроводка_начало объект of g2 is перекрытия объект of h3 is водопровод_ продолжение объект of p2 is обшивка_крыши объект of q1 is водосток объект of j4 is отопление объект of k10 is штукатурка объект of v2 is нивелирование объект of l3 is настил_полов объект of w5 is территория объект of m1 is кухня объект of n2 is водопровод_окончание объект of o3 is отделка_полов объект of t3 is покраска_полов объект of s2 is лакировка_полов объект of u1 is электропроводка_ окончание объект of x0 is стройка_окончание	t1 of a0 is b4 t2 of b4 is c2 t3 of c2 are f1 and r1 and d4 t4 of d4 are sled and e6 and i2 t5 of f1 are h3 and g2 t6 of e6 is p2 t7 of p2 is q1 t8 of q1 and r1 is v2 t9 of v2 is w5 t10 of g2 and sled is j4 t11 of i2 and j4 and h3 is k10 t12 of k10 is l3 t13 of l3 are o3 and n2 and m1 t14 of m1 and n2 is t3 t15 of t3 are sled and u1 t16 of sled and o3 is s2 t17 of s2 and w5 and u1 is x0

Рис. 9.22. Трансформация КЛ СП в текст



___Пример 9.6. Предлагаем самостоятельно создать базу знаний и правил на основе концептуального графа КЛ СП на рис. 9.21 для программы на языке Пролог, а также составить моделирующую программу на языке Python

Построение концептуальной логической сети Петри, моделирующей приложение для глобальной компьютерной сети



___Пример 9.7. Предлагаем самостоятельно составить базу знаний и правил на основе концептуального графа КЛ СП сетевого приложения на рис. 9.23 для программы на языке Пролог, а также составить моделирующую программу на языке Python

Изучение данного примера позволит освоить элементы технологии распределенного программирования, позволяющей перейти от декларативного стиля программирования к процедурному и далее к объектно-ориентированному. На рис. 9.23 представлен концептуальный граф КЛ СП сетевого приложения для гипотетической вычислительной грид-системы, реализованной на платформе компьютерной сети, описанной в восьмой главе (см. рис. 8.1). Здесь предикаты, концепты-позиции, простые концепты, переходы-акторы и правила срабатывания переходов-акторов определяют функциональную архитектуру модели распределенного грид-приложения.

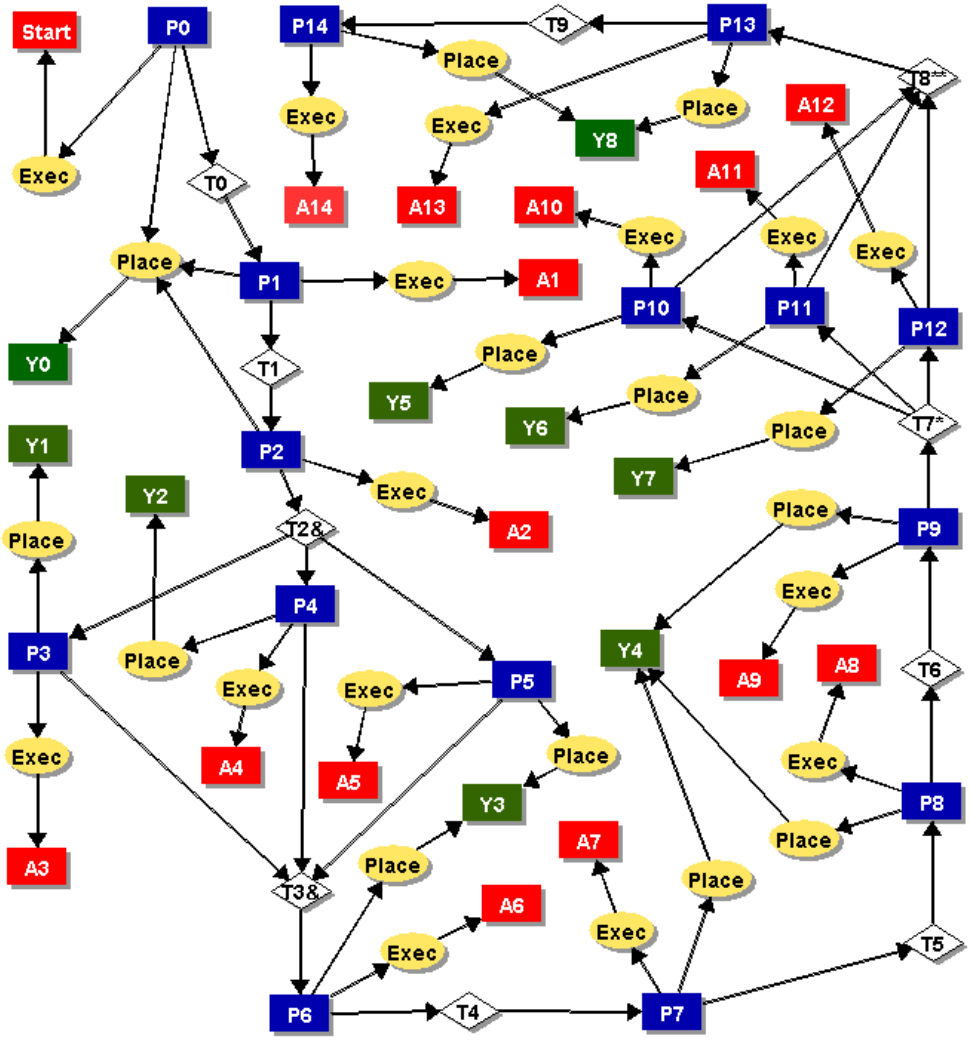


Рис. 9.23. Концептуальная логическая сеть Петри (КЛ СП), моделирующая выполнение распределенного алгоритма в сетевой грид-системе

Простыми концептами здесь представлены операторы из множества $A = \{A_1, A_2, \ldots, A_{14}\}$ и логические узлы из множества $Y = \{Y_0, Y_1, \ldots, Y_8\}$, на которых размещаются концепты-позиции $P = \{P_0, P_1, \ldots, P_{14}\}$. Передача управления и данных в системе задается путем передачи сообщений с данными. В модели все передачи инициируются переходами-акторами $T = \{T_0, T_1, \ldots, T_9\}$. Предикат *Exec* связывает позиции из множества P с операторами из множества A:

$$Exec: P \times A \rightarrow \{\textbf{true}, \textbf{false}\}.$$

Предикат *Place* связывает позиции из множества P с логическими узлами из множества Y:

$$Place: P \times Y \rightarrow \{\textbf{true}, \textbf{false}\}.$$

В модели обработка данных имитируется позициями. По окончании обработки результаты включаются в сообщения (инкапсулируются) и передаются в следующие позиции. Там они выбираются из сообщений (деинкапсулируются) и подвергаются обработке с привлечением локальных данных. Технические подробности этих процессов касаются сетевых технологий, и мы их опускаем.

В рассматриваемой модели правила срабатывания переходов-акторов определяются их типом. Перейдем к рассмотрению этих правил непосредственно для КЛ СП на рис. 9.23.

Обрабатывая XML-представление данного графа интеллектуальным редактором CharGer, получим следующую таблицу (рис. 9.24).

There are the Execution Facts	There are the Placement Facts	There are the Transition Rules
Exec of P0 is Start	Place of P0 is Y0	T0 of P0 is P1
Exec of P1 is A1	Place of P1 is Y0	T1 of P1 is P2
Exec of P2 is A2	Place of P2 is Y0	T2& of P2 are P3 and P4 and P5
Exec of P3 is A3	Place of P3 is Y1	T3& of P3 and P4 and P5 is P6
Exec of P4 is A4	Place of P4 is Y2	T4 of P6 is P7
Exec of P5 is A5	Place of P5 is Y3	T5 of P7 is P8
Exec of P6 is A6	Place of P6 is Y3	T6 of P8 is P9
Exec of P7 is A7	Place of P7 is Y4	T7* of P9 are P10 or P11 or P12
Exec of P8 is A8	Place of P8 is Y4	T8** of P10 or P11 or P12 is P13
Exec of P9 is A9	Place of P9 is Y4	T9 of P13 is P14
Exec of P10 is A10	Place of P10 is Y5	
Exec of P11 is A11	Place of P11 is Y6	
Exec of P12 is A12	Place of P12 is Y7	
Exec of P13 is A13	Place of P13 is Y8	
Exec of P14 is A14	Place of P14 is Y8	

Рис. 9.24. Обработка XML-представления графа интеллектуальным редактором CharGer

Содержимое этой таблицы сформировано редактором и его можно легко привести к виду программы, т.е. базы знаний с фактами и правилами, на языке SWI-Prolog. В первой колонке таблицы содержатся факты – высказывания на основе предиката *Exec*, во второй колонке содержатся факты – высказывания на основе предиката *Place*. Последняя колонка содержит правила срабатывания переходов-акторов, которые рассмотрим детальнее.

Строка следующего вида: T0 of P0 is P1 и ей подобные соответствуют перемещению метки из позиции P0 в позицию P1 под воздействием перехода-актора T0. Содержимое данной строки можно записать в обычном для логических сетей Петри формульном продукционном виде:

$$T_0\colon P_0 \to \neg\, P_0 \,\&\, P_1.$$

В этом выражении (а также в выражениях для остальных продукционных правил) имена позиций интерпретируются как булевы переменные в логике высказываний; истинность какой-либо переменной P_i интерпретируется как наличие метки в одноименной позиции. В предикатной форме выражение для правила срабатывания перехода-актора представимо в следующем продукционном виде:

$$T_0\colon R(P_0) \to \neg\, R(P_0)\ \&\ R(P_1),$$

где R – унарный предикат разметки позиций. Аналогично составляются продукционные правила срабатывания переходов-акторов T_0, T_1, T_4, T_5, T_6 и T_{13}.

Выражению T2& of P2 are P3 and P4 and P5 в таблице соответствует следующее продукционное правило с булевыми переменными:

$$T_2^{\&}\colon P_2 \to \neg\, P_2 \,\&\, P_3 \,\&\, P_4 \,\&\, P_5,$$

что в КЛ СП соответствует срабатыванию перехода-актора $T_2^{\&}$, выполняющего сетевую операцию *fork*; на выходе данного перехода будут сформированы три метки, которые перейдут в позиции-концепты P_5 & P_3 & P_4.

Выражению T3& of P3 and P4 and P5 is P6 в таблице соответствует следующее продукционное правило с булевыми переменными:

$$T_3^{\&}\colon P_3 \,\&\, P_4 \,\&\, P_5 \to \neg\, P_3 \,\&\, \neg\, P_4 \,\&\, \neg\, P_5 \,\&\, P_6.$$

Здесь переход-актор $T_3^{\&}$ реализует сетевую операцию *join*, при выполнении которой на его входе реализуется конъюнктивная сборка меток, а на выходе появляется одна метка, поступающая в позицию-концепт P_6.

Строке T7* of P9 are P10 or P11 or P12 соответствует следующее выражение для правила срабатывания перехода T_7^*:

$$T_7^*: P_9 \rightarrow \neg P_9 \,\&\, (\alpha_{10}\&P_{10} \vee \alpha_{11}\&P_{11} \vee \alpha_{12}\&P_{12}),$$

где только одно из условий α_{10}, α_{11}, α_{12} может быть истинным, т.е. под управлением перехода T_7^* метка передается из позиции P_9 в одну из его выходных позиций.

Строке T8** of P10 or P11 or P12 is P13 соответствует следующее выражение для правила срабатывания перехода T_8^{**}:

$$T_8^{**}: (\alpha_{10}\&P_{10} \rightarrow P_{13}\&\neg P_{10}) \vee (\alpha_{11}\&P_{11} \rightarrow P_{13}\&\neg P_{11}) \vee$$

$$(\alpha_{12}\&P_{12} \rightarrow P_{13}\&\neg P_{12}),$$

где принято, что значения всех условий α_{10}, α_{11}, α_{12} «передаются» вместе с меткой из одной из выбранных переходом T_7^* выходных позиций P_{10} , P_{11} или P_{12} и используются переходом T_8^{**} при удалении метки из одной из выбранных входных позиций P_{10} , P_{11} или P_{12} в выходную позицию P_{13}.

Возможности формализма КЛ СП позволяют провести дальнейшую детализацию модели распределенного приложения, представленной на рис. 9.23. Например, выполнение оператора A1 в сетевой среде задается графом КЛ СП, изображенным на рис. 9.25.

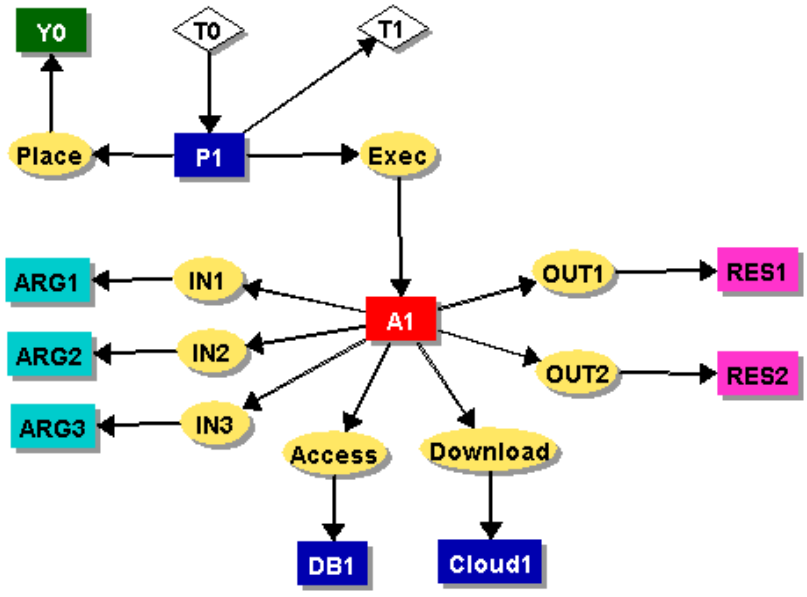


Рис. 9.25. Фрагмент концептуальной логической сети Петри (КЛ СП), моделирующей выполнение одного из операторов распределенного алгоритма в сетевой грид-системе

Здесь, помимо общих объектов, заданных в графе на рис. 9.23, представлены следующие концепты и предикаты:

- ARG1, ARG2 и ARG3 – простые концепты, представляющие в модели данные, которые передаются из позиции-концепта P0 в позицию-концепт P1 и далее обрабатываются оператором A1;
- IN1, IN2 и IN3 – «входные» унарные предикаты, задающие порты ввода данных;
- DB1 и Cloud1 – простые концепты, представляющие в модели базы данных, размещенные на текущем узле сети Y0 (DB1) и в «облаке» (Cloud1) и используемые при обработке данных, принимаемых из сети;
- Access, Download – предикаты, задающие способы доступа к базе данных DB1 и к облачному сервису Cloud1;
- RES1 и RES2 – простые концепты, представляющие в модели результаты обработки данных оператором A0;
- IN2 и IN3 – «выходные» унарные предикаты, задающие порты вывода данных.

Применяя команду File / Show English в интеллектуальным редакторе, получим следующий результат – отношения, представленные в машинной форме (рис. 9.26).

There are the Facts for A1-P1-Y0

Exec of P1 is A1
IN1 of A1 is ARG1
IN2 of A1 is ARG2
IN3 of A1 is ARG3
… of T0 is P1
T1 of P1 is …
OUT1 of A1 is RES1
OUT2 of A1 is RES2
Download of A1 is Cloud1
Access of A1 is DB1
Place of P1 is Y0

Рис. 9.26. Отношения, представленные в машинной форме

Далее можно использовать полученную информацию для программирования модели обработки данных.

Как видно из рис. 9.23 и 9.25, позиции-концепты и переходы-акторы образуют основу или остов (англ. *backbone*) модели, близкой к обычной логической сети Петри. Отличие состоит в описанной и использованной в настоящей главе интерпретации переходов-концептов, реализуемых акторами. На остов «навешиваются» другие объекты, связанные с понятием «концептуальный» граф: обычные, или простые, концепты и обычные же, или простые предикаты, задающие отношения, характерные для семантических сетей. Принято, что переходы-акторы не связываются с простыми концептами. Их можно связывать только с позициями-концептами.

Далее программу для исполнимой модели обработки данных в сетевой грид-среде можно записать на обычном алгоритмическом языке типа Python, руководствуясь примером разработанной в настоящей главе программой «Иск» на языке Python, реализующей модель концептуальной логической сети Петри для правового процесса.



Вопросы и задания для самопроверки

1. Дайте характеристику основным элементам семантической сети.
2. Назовите отличие однородных семантических сетей от неоднородных.
3. Перечислите достоинства и недостатки семантических сетей.
4. Используя соответствующие отношения, постройте семантическую сеть для перечисленных предметных областей:

4.1. География какого-либо региона. Отношения: государство, страна, континент, широта.

4.2. Диагностика заболеваний внутренних органов. Отношения: категории болезней, патофизиологическое состояние, жалобы пациента, наблюдения, симптомы.

4.3. Распознавание химических структур. Отношения: формула вещества, свойства вещества, область применения, меры предосторожности.

4.4. Процедура поиска полезных ископаемых. Отношения: наименование ископаемого, расположение месторождения, глубина залегания, методы добычи.

4.5. Судебная процедура. Отношения: юридическое лицо, событие, меры воздействия, способы расследования.

4.6. Распределение продуктов по магазинам. Отношения: источник снабжения, наименование продукта, способ транспортировки, конечный пункт транспортировки.

4.7. Определение принадлежности животного к определенному виду, типу, семейству. Отношения: место обитания, строение, особенности поведения, вид питания.

4.8. Классификация пищевых продуктов. Отношения: наименование продукта, составляющие части, способ приготовления, срок хранения.

4.9. Распознавание типа компьютера. Отношения: страна изготовитель, стандартная конфигурация, область применения, используемое программное обеспечение.

Глава 10

Концептуальные интеллектуальные модели роботизированного производства



Станок-автомат для производства тортов

Автоматизация и роботизация производства

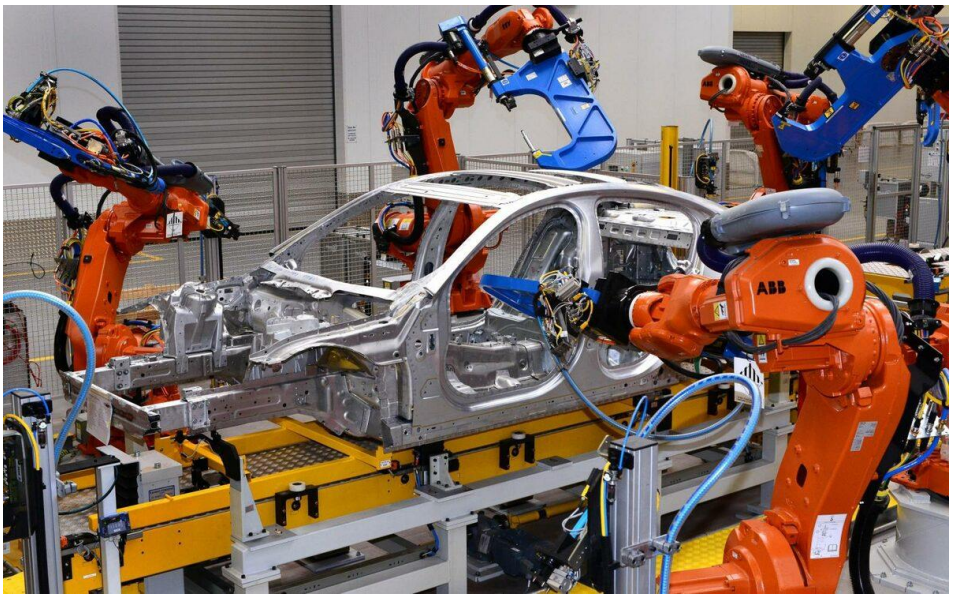
Автоматизация производства связана с заменой процессов, выполняемых человеком, на операции, производимые с помощью специализированной техники. Роботизация – это одна из частей автоматизации. Мехатроника и робототехника – это области науки и техники, основанные на объединении узлов точной механики с электронными, электротехническими и компьютерными компонентами, обеспечивающими проектирование и производство качественно новых модулей, машин и роботизированных систем с интеллектуальным управлением.

Промышленные роботы уже используются на самых разных предприятиях и учреждениях. Они могут быть связаны с медициной, пищевой промышленностью, металлообработкой, автомобильной промышленностью… От внедрения промышленных роботов на предприятиях обычно ожидают минимизацию некорректного влияния человеческого фактора, быструю окупаемость вложенных средств,

сокращение издержек в перспективе, снижение процента брака, повышение производительности, повышение надежности выполнения операций, простоту перепрограммирования и перенастройки оборудования.

В качестве примера подходящей предметной области в настоящей рассматривается участок гибкого автоматизированного производства, сокращенно ГАП (FMS – Flexible Manufacturing Systems). Применяемые на различных участках ГАП промышленные роботы (роботы-манипуляторы, робокары, погрузчики-робокары, роботы-погрузчики, складские роботы-штабелеры и др.) различаются по специализации, грузоподъемности, возможностям передвижения, способам установки на рабочем месте, адаптивности к окружающей среде, видам управления, способам программирования. Управляющие программы для роботов с программным управлением реализуют различные алгоритмы для исполнительных устройств, связанные с выполнением функций робота и позволяющие ему преодолевать различные препятствия в работе.

Приведенные ниже фотографии взяты из открытых источников фирм-производителей и иллюстрируют некоторые приложения промышленных роботов (рис. 10.1–10.3).



а)

Рис. 10.1. Промышленные роботы на конвейере сборки автомобилей (начало)



б)

Рис. 10.1. Окончание



Рис. 10.2. Роботизированная «умная» линия для упаковки печенья



Рис. 10.3. Роботизированная «умная» линия для производства шоколадных конфет

Представление и структурирование данных и знаний о роботизированном производстве

Структурирование информации обычно заключается в ее делении на группы и подгруппы по определенным критериям (существенным или системно значимым признакам). Для лучшего запоминания и надежного хранения информации между группами, подгруппами и их элементами строятся логические связи. По мере того, как между объектами предметной или проблемной областей устанавливаются отношения либо для объектов устанавливаются их свойства, *данные*, характеризующие эти отношения и объекты, становится уместным называть *знаниями*.

В литературных источниках, касающихся логического подхода к искусственному интеллекту, отличия знаний от данных формулируются следующим образом:

1) интерпретация. Хранимые данные могут быть интерпретированы только человеком или программой, составленной человеком. Знания содержат как данные, так и их описание (правила интерпретации);

2) наличие связей классификации. Данные не имеют эффективного описания связей между различными типами данных. Знания структурированы, поэтому можно установить соответствие между единицами знаний;

3) наличие статических и динамических ситуационных связей. Связи описывают множество текущих ситуаций объекта. Данные трудно поддаются анализу. На основе структуры и состава знаний по ситуации возможно построение процедур анализа знаний.

Структурированные данные и знания отличаются хорошей организацией и точным форматированием; они обычно хранятся в таблицах либо в других структурах в упорядоченном виде со связанными строками и столбцами для легкого нахождения и обработки конкретной информации, а также для выделения отношений между ее частями. Для аналитических целей наряду с реляционными базами данных часто используются и хранилища неструктурированных данных.

Рассмотрим представление знаний на содержательном уровне в виде следующего предложения, несколько модифицируя и комбинируя соответствующие описания известной из работ [45, 47] задачи, «традиционно» служащей для иллюстрации различных технологий создания роботизированных производств. Концептуальные графы для роботизированного производства приведены в работах [13, 14].



Автомат для производства тортов

Ключевая фраза для представления данных и знаний о работе роботизированного производства

«Автомат подает партию заготовок на станок (точнее, в подающий накопитель – часть станка), затем партия деталей обрабатывается по одной детали и оказывается в выходном накопителе станка, откуда их забирает первый робот-манипулятор, который упаковывает партию деталей в кассету. Затем второй робот-погрузчик грузит кассеты с деталями на робокар (число кассет, размещаемых на робокаре, ограничено), который перевозит их на склад. На складе в зависимости от некоторого условия α робот-штабелер помещает кассеты в ячейки (при α = true) или же при наличии транспорта (при α = false) третий робот упаковывает имеющиеся кассеты в контейнер (объем контейнера ограничен), который по заполнении грузится роботом-автопогрузчиком на грузовой автомобиль по его прибытии».

КОНЦЕПТУАЛЬНЫЙ ГРАФ ДЛЯ МОДЕЛИ РОБОТИЗИРОВАННОГО ПРОИЗВОДСТВА И ЕГО КОНТЕКСТНОЕ ПРЕДСТАВЛЕНИЕ

Сценарий, или сценарную сеть, работы участка ГАП возможно представить в виде семантической сети с событиями, ролевыми и причинно-следственными (каузальными) связями между объектами. Для этой сети должны быть определены следующие множества концептов и отношений:

– множество концептов-событий (фактов):

$$F = \{F_0, F_1, F_3, F_4, F_5, F_6, F_7, F_8, F_9\},$$

имеющих, соответственно, следующие названия (имена):

X = {*Подача*, *Обработка*, *Упаковка_1*, *Погрузка_1*, *Перевозка*, *Размещение*, *Упаковка_2*, *Погрузка_2*, *Местонахождение_1*, *Местонахождение_2*};

– множество *R* всех отношений:

Rel = {*Имя*, *Агент*, *Объект*, *Место*, *След*, *След_склад*, *След_контейнер*};

– множество R_1 ролевых отношений, т.е. отношений без отношений следования:

Rel_1 = {*Имя*, *Агент*, *Объект*, *Место*};

– множество *Con* остальных концептов (концептов-объектов), представленно на рис. 10.4 и 10.5.

Сценарий роботизированного производства, или семантическая сеть с событиями, представлен на рис. 10.4.

Концептуальное представление сценария работы участка ГАП приведено на рис. 10.5.

Здесь прямоугольниками представлены концепты – концепты-объекты и концепты-события, а овалами – отношения между концептами. Причинно-следственные (каузальные) связи между событиями представлены бинарным отношением *След*. Так, истинность высказывания *След*(F_i, F_j), где *След* – одноименный отношению бинарный предикатный символ, означает, что вслед за событием F_i выполняется событие F_j. В овале, связывающем в концептуальном графе

концепты-события F_4 и F_5, записана формула $\alpha\&След(F_4, F_5)$, а в овале, связывающем концепты-события F_4 и F_6, записана формула $\neg\alpha\&След(F_4, F_6)$, т.е. первая связь реализуется при α = **true**, а вторая – при α = **false**. Однако в целях упрощения последующего анализа концептуального графа и представления его в реляционной базе данных будем использовать отношения *След_склад* и *След_контейнер*.

При построении концептуального графа использовался редактор *CharGer*, позволяющий представить граф в виде XML-файла. Такое представление позволяет осуществить в дальнейшем автоматический переход от концептуального графа к отношениям реляционной базы данных. Полученные отношения, представленные табл. 10.1–10.7, могут использоваться далее при исследовании сценария, представленного концептуальным графом на рис. 10.6.

Таблица 10.1

Имя

Факт	Имя
F_0	Подача
F_1	Обработка
F_2	Упаковка_1
F_3	Погрузка_1
F_4	Перевозка
F_5	Размещение
F_6	Упаковка_2
F_7	Погрузка_2
F_8	Местонахождение_1
F_9	Местонахождение_2

Таблица 10.2

Агент

Факт	Агент
F_0	Автомат
F_1	Станок
F_2	Робот_1
F_3	Робот_2
F_4	Робокар
F_5	Штабелер
F_6	Робот_3
F_7	Автопогрузчик
F_8	–
F_9	–

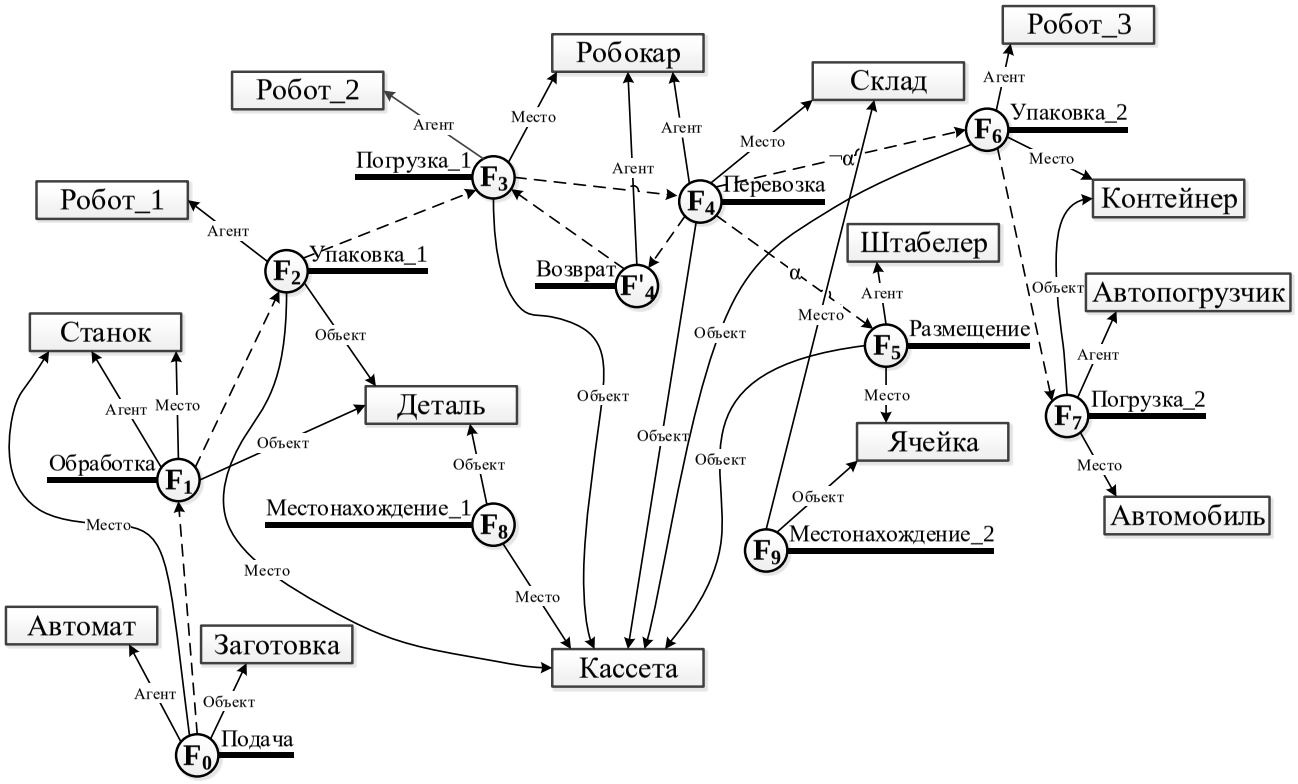


Рис. 10.4. Семантическая сеть с событиями (сценарий) для участка роботизированного производства

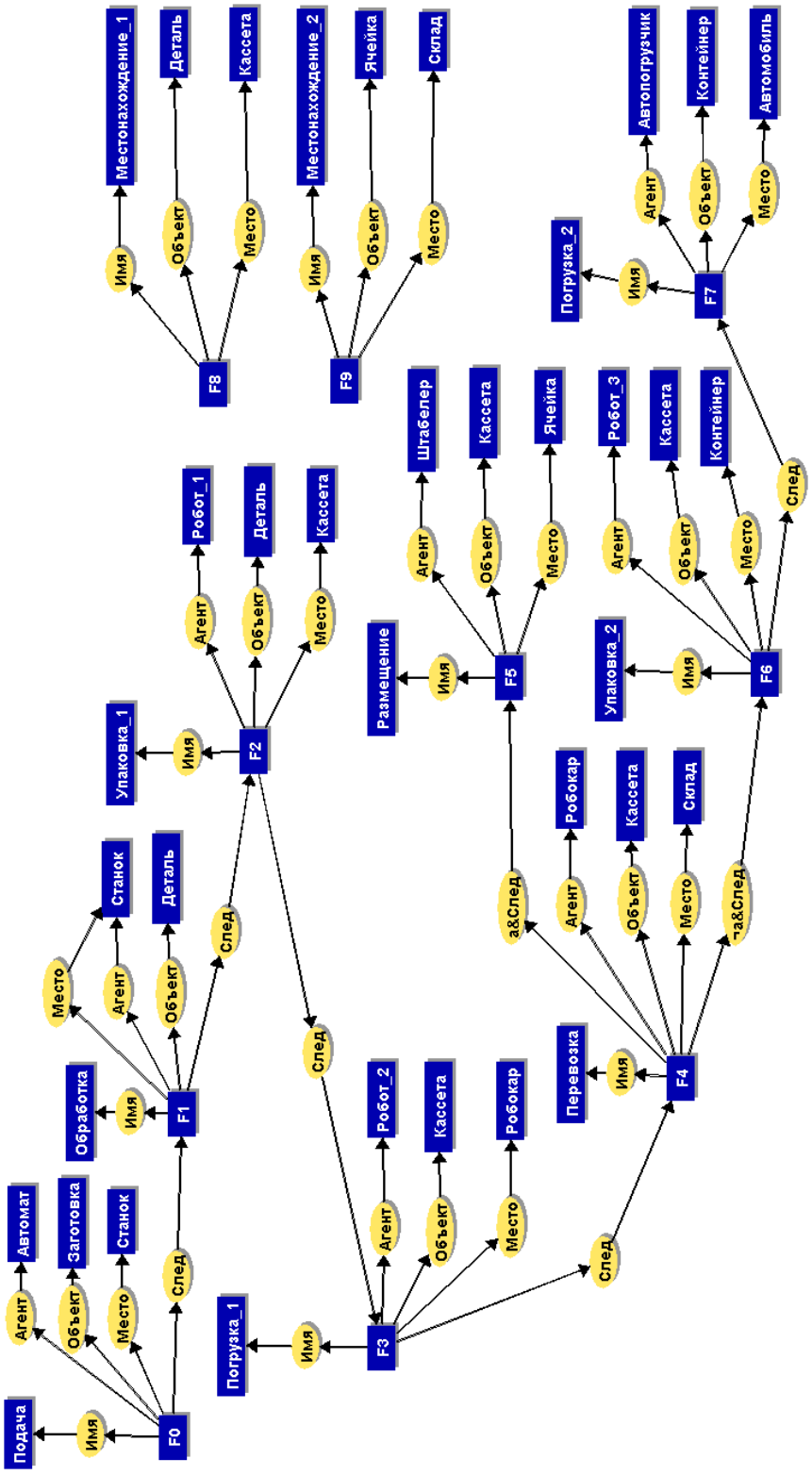


Рис. 10.5. Концептуальный граф сценария роботизированного производства

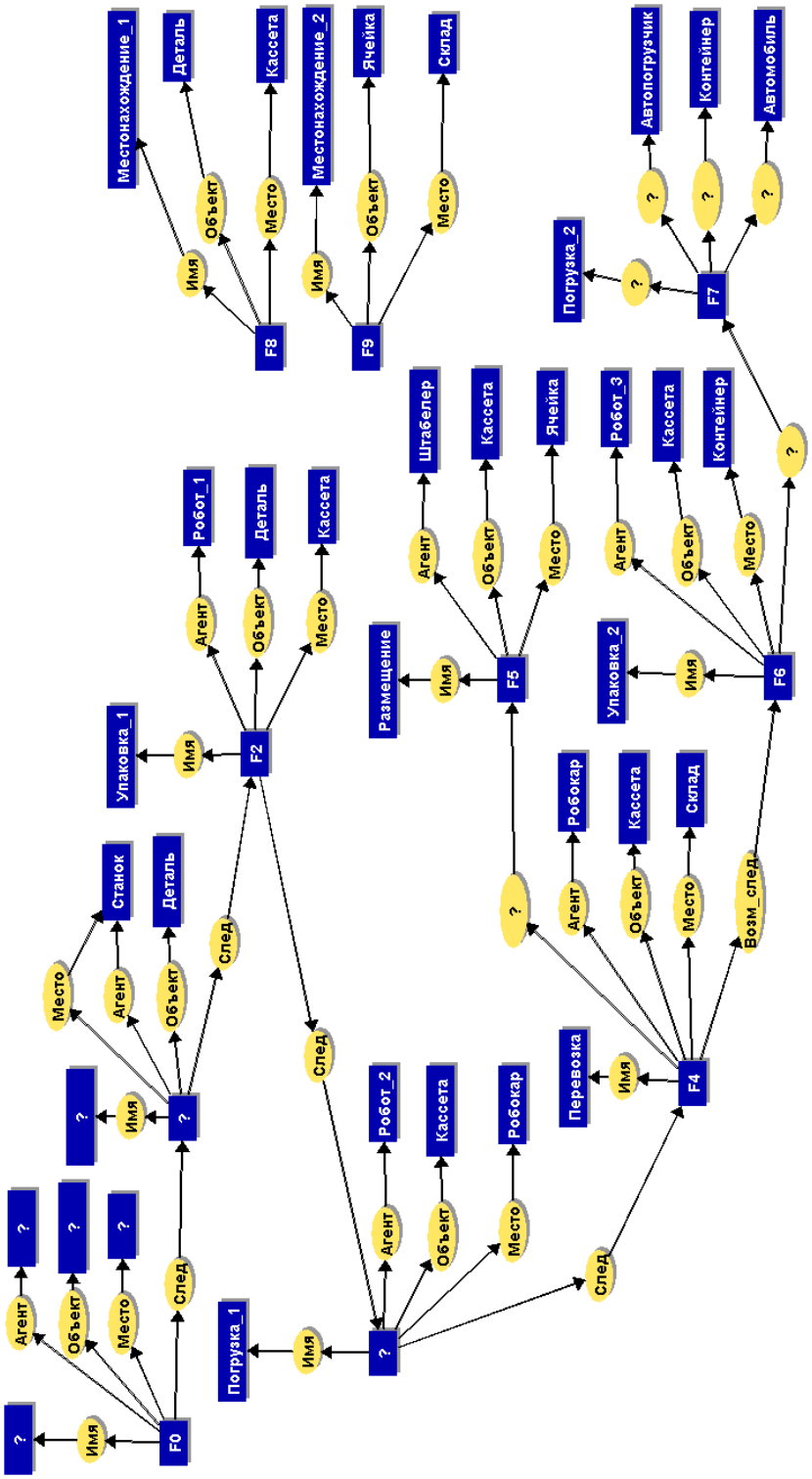


Рис. 10.6. Вид концептуального графа с некоторыми неизвестными концептами и отношениями, которые необходимо найти при помощи запросов к базе знаний

Таблица 10.3

Объект

Факт	Объект
F_0	Заготовка
F_1	Деталь
F_2	Деталь
F_3	Кассета
F_4	Кассета
F_5	Кассета
F_6	Кассета
F_7	Контейнер
F_8	Деталь
F_9	Ячейка

Таблица 10.4

Место

Факт	Место
F_0	Станок
F_1	Станок
F_2	Кассета
F_3	Робокар
F_4	Склад
F_5	Ячейка
F_6	Контейнер
F_7	Автомобиль
F_8	Кассета
F_9	Склад

Таблица 10.5

След

Факт_1	Факт_2
F_0	F_1
F_1	F_2
F_2	F_3
F_3	F_4
F_6	F_7

Таблица 10.6

След_склад

Факт_1	Факт_2
F_4	F_5

Таблица 10.7

След_контейнер

Факт_1	Факт_2
F_4	F_6



___ Задания по теме «Роботизированное производство» (см. рис. 10.4, 10.5, 10.6)

Указание. На основе табл. 10.1–10.7, составьте базу знаний с бинарными и тернарными отношениями, используя язык SWI-Prolog, и определите запросы к ней

1. Какие ролевые отношения связывают событие F_7 с концептами *Погрузка_2*, *Автопогрузчик*, *Контейнер* и *Автомобиль*?

2. Какие несобытийные концепты связаны с реализацией события F_0?

3. Определить концепты-события, необходимо следующие за событием F_2.

4. Определить концепты-события, возможно следующие за событием F_4.

5. Какое отношение связывает концепты-события F_4 и F_5? Другой вариант вопроса: «Возможно ли следование события F_5 непосредственно после события F_4?».

6. В каких событиях и в каких ролях участвовал *робокар*?

7. Связаны ли события F_6 и F_7 отношением непосредственного следования?

При поиске ответа на последний вопрос проверяется истинность или ложность высказывания *След*(F_6, F_7). Для рассматриваемого примера это высказывание истинно.

Переход от концептуальной сценарной модели роботизированного производства к имитационной модели

Логические формализмы позволяют описывать структурные и логические связи между сущностями предметной области в виде составных высказываний, истинность которых соответствует ситуациям, возникающим в заданной предметной области.

При построении поведенческих – исполнимых, или операционных, моделей необходима операционная поддержка, обеспечиваемая механизмами вывода используемого логического формализма.

Модели, описанные графами на рис. 10.4–10.6, имеют декларативный характер, и мы их относим к классу концептуальных. Эти модели могут быть положены в основу построения других, поведенческих моделей.

Граф, представленный на рис. 10.5, описывает явно прохождение лишь одной кассеты с деталями. Вместе с тем очевидно, что некоторые события могут выполняться параллельно. Поэтому из ряда исполнимых, или операционных, моделей дискретно-событийных систем выберем сети Петри.

Рассмотрим в качестве примера сеть Петри, описывающую работу участка роботизированного производства ГАП. На рис. 10.7 ресурсными позициями R_0, R_1, …, R_8 в модели представлены ресурсы-агенты участка ГАП – *автомат* (R_0), подающий партии заготовок *деталей* к *станку*; *станок* (R_1), обрабатывающий *детали*; *робот_1* (R_2), упаковывающий готовые изделия в *кассеты*; *робот_2* (R_3), осуществляющий погрузку *кассет* с *деталями* на *робокар* – мобильный объект (R_4); *штабелер* – робот-погрузчик (R_5), размещающий *кассеты* с *деталями* в *ячейки* на *складе*; *робот_3* (R_6), упаковывающий *кассеты* в *контейнер*; *автопогрузчик* (R_7), грузящий *контейнеры* с *кассетами* на *автомобиль* (R_8). Наличие меток в ресурсных позициях соответствует незанятости данных ресурсов, или готовности выполнения своих функций. При имитации занятия ресурса метка удаляется из соответствующей позиции.

Позиции W_0, W_1 имитируют текущее размещение заготовок для будущих деталей до упаковки в кассеты, а позиции W_2, W_3, W_4, W_5, W_6, W_7 используются для имитации текущего размещения кассет уже с деталями. Предполагается, что число заготовок в каждой партии, поступающей на обработку, равно числу деталей, упакованных в кассету. Позиция W'_4 – вспомогательная.

Переходы, соответствующие событиям F_1, F_2, F_3, F_4, F'_4, F_5, F_6, F_7, в модели на рис. 10.7, реализуют события, представленные ранее в семантической сети (сценарии) на рис. 10.4 и в концептуальном графе на рис. 10.5. Введен дополнительный переход, реализующий в сети событие F_k, имитирующий учет изделий в кассетах и завершающий работу сети Петри. Вместимости робокара (не более C кассет) и контейнера (не более M кассет) ограничены, что учтено при определении правил срабатывания переходов F_3, F_6 и F_7.

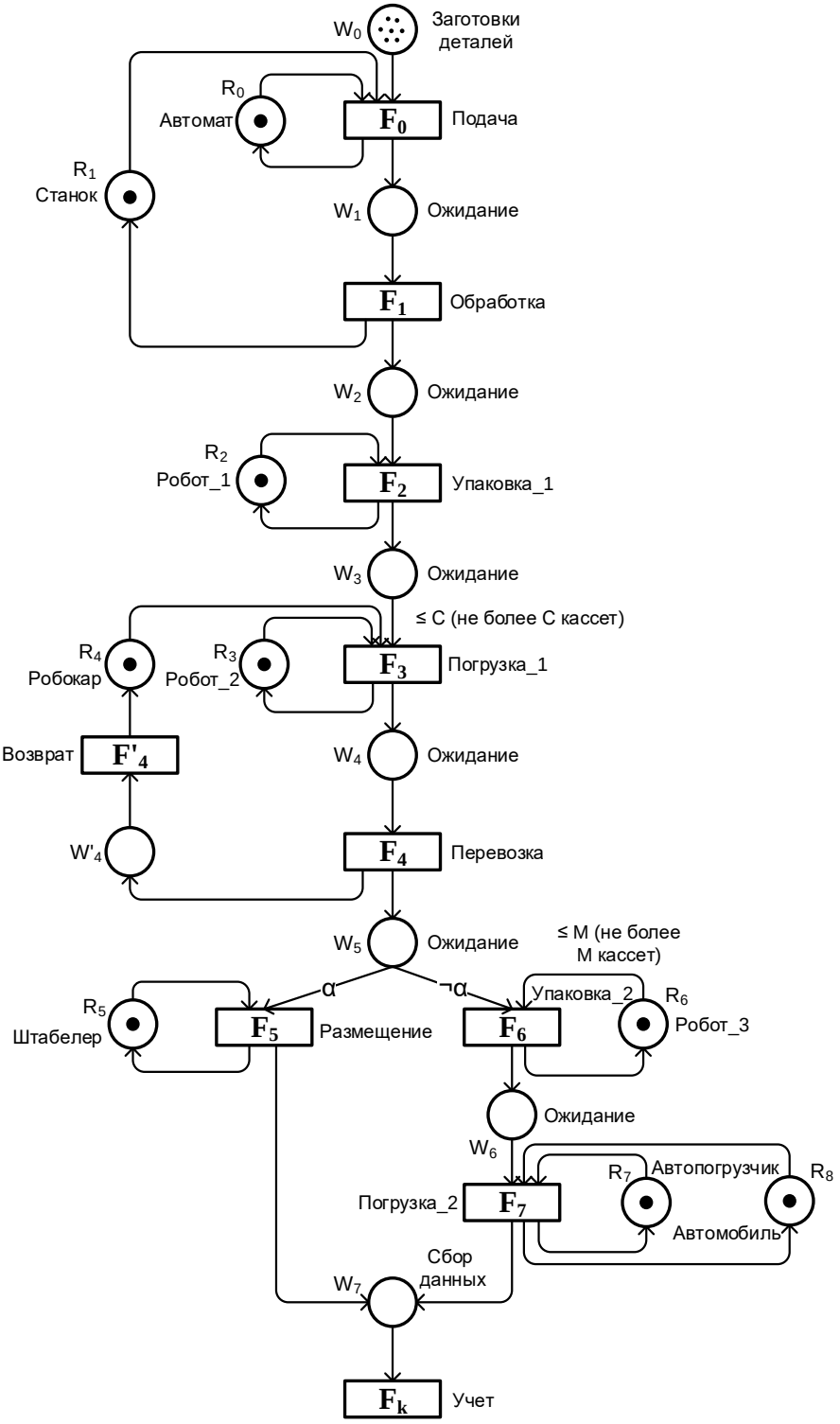


Рис. 10.7. Сеть Петри для сценария работы участка роботизированного производства

Как обычно в сетях Петри, асинхронные срабатывания переходов приводят к изменению текущей разметки сети. В отличие от семантической сети с событиями (см. рис. 10.4) и концептуального графа (см. рис. 10.5), сетью Петри (см. рис. 10.7) эффективно представляется динамика модели, ее параллельное выполнение.

Например, обработка заготовок может выполняться одновременно с перемещением робокара, т.е. в отмеченном и в очевидных других случаях в модели реализуется «конвейерный» параллелизм. Поэтому модель, построенная в терминах сетей Петри и доопределенная правилами и процедурами срабатывания переходов, обладает всеми признаками поведенческой.

Интеграция концептуальных и поведенческих моделей дискретно-событийных систем: концептуальные сети Петри

Достоинством концептуальных моделей дискретно-событийных систем, представленных семантическими сетями с событиями и концептуальными графами, является их декларативный характер, позволяющий строить экспертные системы и системы ситуационного управления. В дискретно-событийных моделях систем на основе разновидностей сетей Петри проще учитывается динамика процессов, занятие и освобождение ресурсов, управление очередями. Предложим вариант интеграции моделей обоих классов в виде концептуальных сетей Петри (КСП). Пример КСП-сети для сценария работы участка ГАП роботизированного производства представлен на рис. 10.8.

КСП отличается от определенного ранее от графа типа КЛ СП и представляет собой концептуальный граф, в котором события, или первые концепты, представлены переходами обычной (не логической) сети Петри, связаны ролевыми отношениями с вторыми концептами – именами *событий*, *агентами*, *объектами*, *местами*. Представленные графически ролевые отношения соответствуют семантической сети (см. рис. 10.4) и концептуальному графу (см. рис. 10.5). Подобное представление позволяет строить удобные в использовании экспертно-имитационные модели дискретно-событийных систем, сочетающие признаки декларативных инфологических моделей с операционными (исполнимыми) моделями типа сетей Петри.

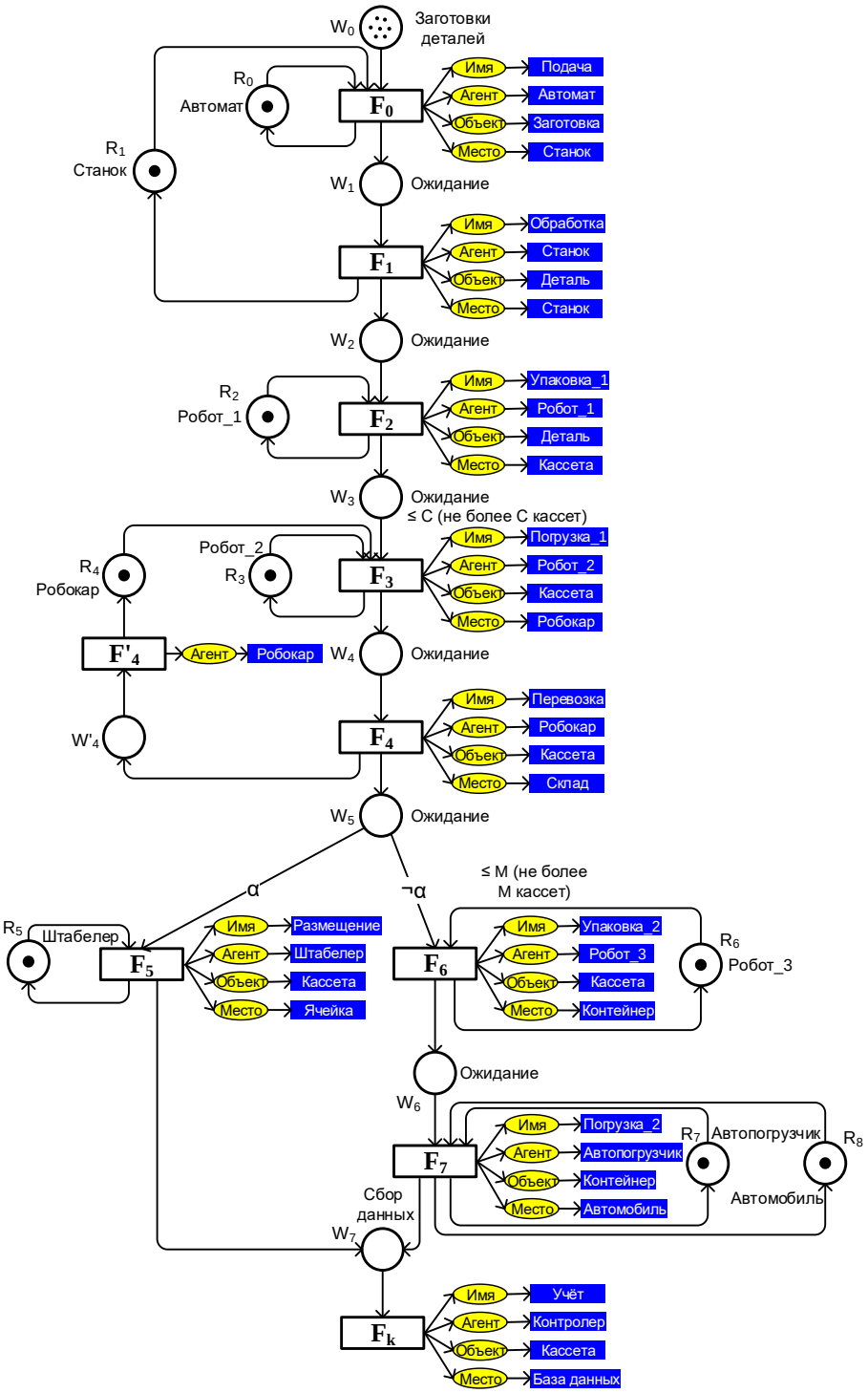


Рис. 10.8. Концептуальная сеть Петри для сценария работы участка роботизированного производства

Модель беспроводной сети MANET, управляющей работой участка роботизированного производства

Промышленная мобильная связь, сокращенно ПМС (IMC – *Industrial Mobile Communication*), обеспечивающая возможность обмена данными через беспроводные каналы связи сетей, позволяет использовать для управления промышленными роботами различные радиосети, например, WLAN, GSM, GPRS, UMTS; она охватывает программные и аппаратные компоненты.

Управление производственными процессами ГАП возможно реализовать с помощью беспроводных мобильных *ad hoc* сетей MANET с переменной топологией (*Mobile Ad hoc NETwork*). В *ad hoc* сетях каждый узел может выполнять функции маршрутизатора и способен ретранслировать пакеты. Узлы сети типа MANET предназначены для работы на подвижных объектах и могут перемещаться независимо друг от друга. В сетях MANET возможно использовать как реактивные, так и проактивные протоколы маршрутизации. Для рассматриваемых в настоящей работе производственных процессов, управляемых беспроводной сетью MANET, возможно ограничиться реактивным протоколом AODV (*Ad hoc On-Demand Distance Vector protocol*). Возможно использовать другие виды протоколов маршрутизации; при выборе протокола следует учитывать частое изменение топологии сети.

Уточним размещение подвижных и стационарных объектов при организации роботизированного производства (рис. 10.9). *Оператор* (S_0), *автомат* (R_0), *станок* (R_1), *робот_1* (R_2) и *робот_2* (R_3) размещены на участке ГАП и не выходят из зоны радиовидимости сети. *Штабелер* (R_5), *склад* (S_1), *робот_3* (R_6) и *автопогрузчик* (R_7) находятся на участке логистики и также не покидают зону радиовидимости сети. На всех перечисленных объектах размещены узлы сети MANET. Для обеспечения непрерывной связности этих узлов сети MANET в ее составе используются транзитные узлы T_1, T_2 и T_3, число которых определяется расстоянием между участками ГАП и логистики.

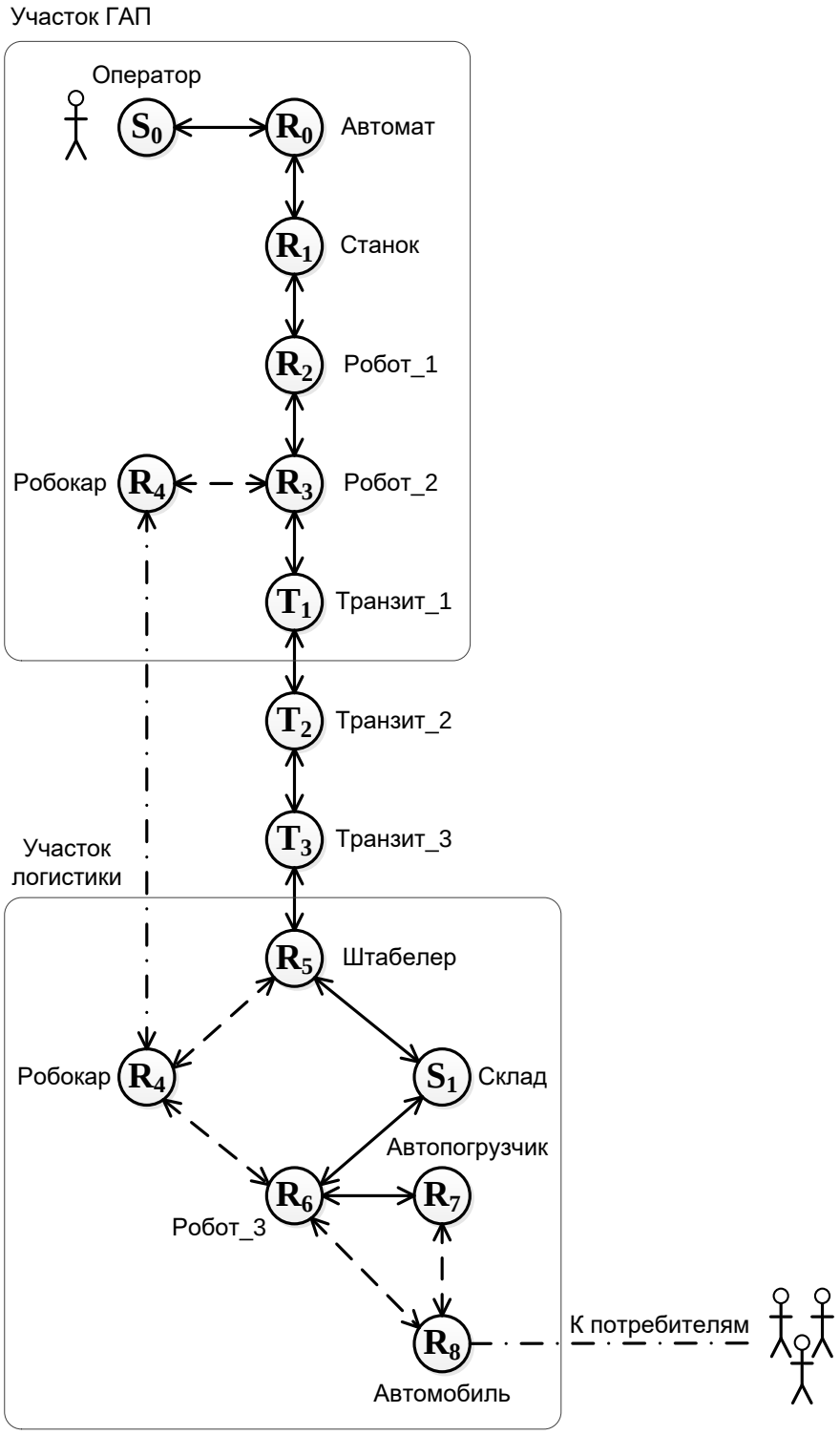


Рис. 10.9. Схема развертывания беспроводной управляющей сети MANET в производственной среде

Мобильные узлы сети MANET, размещенные на *робокаре* (R_4) *роботе_3* (R_6) и на *автомобиле* (R_8), время от времени выходят из зоны радиовидимости сети: *робокар* выполняет перевозку готовых изделий в кассетах между участками, а *автомобиль* доставляет контейнеры с произведенной продукцией потребителю. На рис. 10.9 временные связи мобильных узлов с сетью MANET представлены штриховыми стрелками. Штрих-пунктирной стрелкой условно показано направление перемещения мобильного объекта (*робокара*) с установленным на нем узлом сети MANET. Работа распределенного программного обеспечения сети определяется в целом сетями Петри (см. рис. 10.7 и 10.8).



__ Задание по теме «Роботизированное производство»: используя схему развертывания беспроводной управляющей сети MANET на рис. 10.9 и редактор CharGer, разработайте соответствующий концептуальный граф

О реализации вычислительной и коммуникационной компоненты роботизированной системы

В киберфизических системах вычислительная компонента, как правило, распределена по всей физической системе и связана с ее составляющими элементами. Вычислительная и коммуникационная платформы являются основой для построения большинства современных эргатических и киберфизических систем. Эти платформы реализуют необходимые для прикладной функциональности алгоритмы. Взятые вместе, эти платформы называются сетевыми. Для классификации компьютерных сетей используются различные признаки, но чаще всего сети делят на типы по территориальному признаку, то есть по величине территории, которую покрывает сеть.

В настоящее время существует большое разнообразие вычислительных сетей, которые нашли свое применение в промышленности и на транспорте (источники: https://aveon.ru; https://www.tadviser.ru/index.php/Статья:TADетали:Как_построить_надежную_беспроводную_сеть_на_промышленном_предприятии). Например, на основе коммерческих беспроводных сетей стандартов IEEE 802.11 a/b/g/n появились промышленные сети Industrial Wi-Fi, для которых разрабатываются и тестируются устройства промышленного iWifi стандарта IEEE 802.11ac. Устройства для промышленных Wi-Fi сетей, предназначенные для работы в тяжелых промышленных условиях, могут характеризоваться следующими признаками: иметь прочный металлический корпус с защитой от вибраций, пыли и влаги; выполняться с учетом промышленного рабочего диапазона температур от –40° до +75 °C; иметь гальваническую развязку интерфейсов и вводов/выводов для защиты от высокого уровня электромагнитных помех; обладать резервированием по электропитанию; иметь коннекторы, которые обеспечивают физическую защиту Ethernet-портов при работе в условиях агрессивной внешней среде и на движущихся объектах, которые препятствуют самопроизвольному разъединению и попаданию в устройство пыли и влаги.

Приведенные ниже рисунки иллюстрируют применение промышленных сетей Wi-Fi в роботизированном производстве (рис. 10.11–10.14).

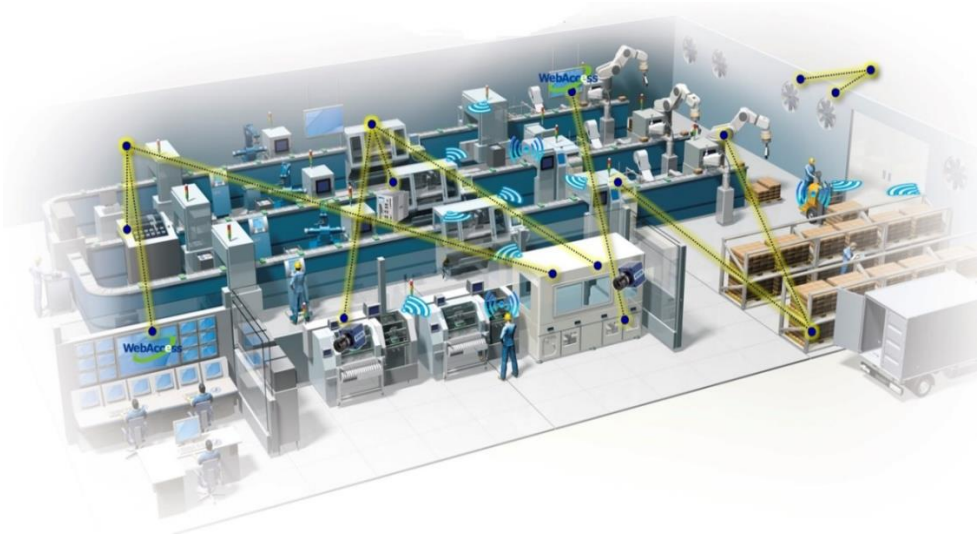


Рис. 10.10. Роботизированное производство под управлением беспроводной сети

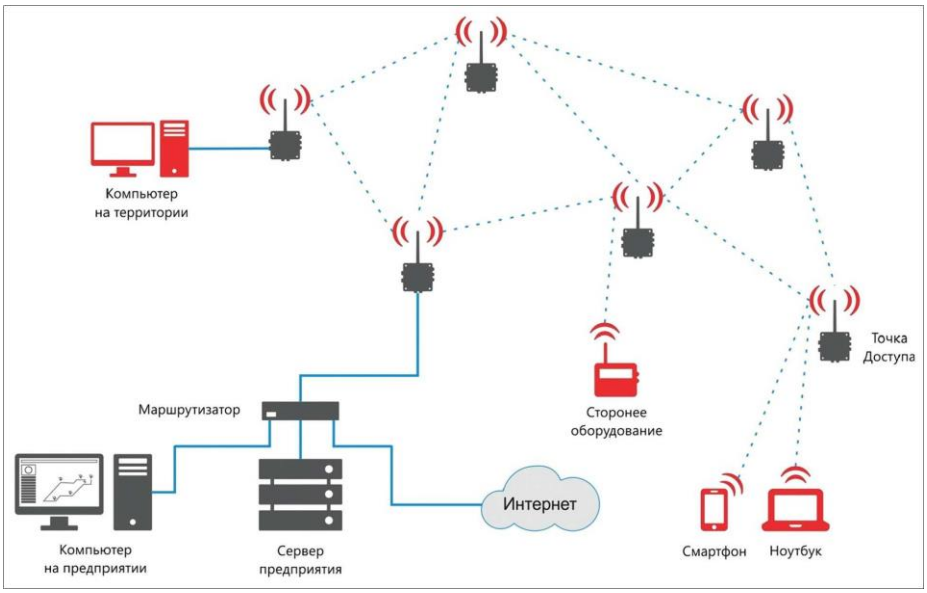


Рис. 10.11. Структурная схема промышленной сети Wi-Fi

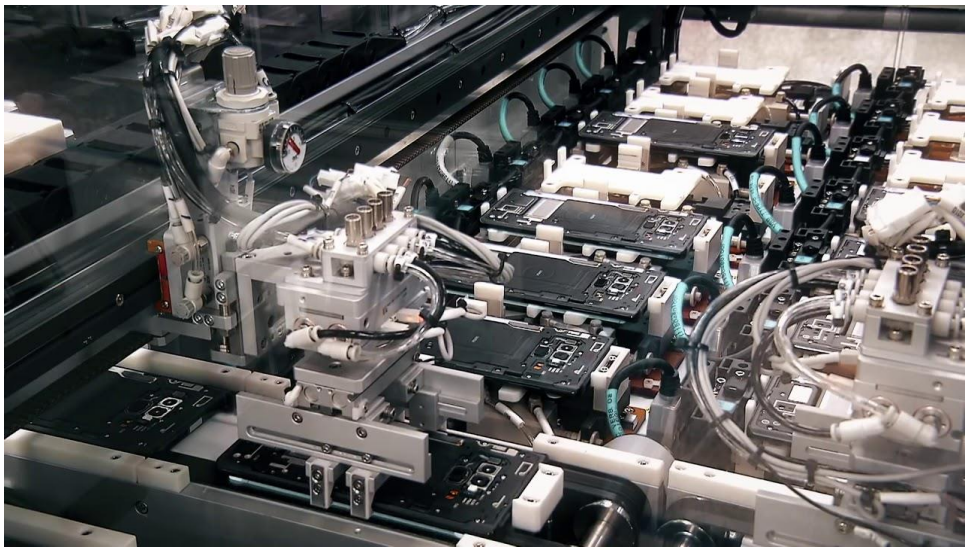


Рис. 10.12. Роботизированная линия автоматизированной сборки смартфонов

Робот-андроид по имени ASIMO, созданный японской корпорацией HONDA (рис. 10.13).

Исследователи считают, что в один прекрасный день ASIMO сможет помогать пожилым людям, которым необходим постоянный уход, а также людям с ограниченными возможностями. А пока он учится играть в футбол и разносить кофе!





Рис. 10.13. Робот-андроид ASIMO

Робот Федор, или Skybot-F850 – первый российский космодроид. Разработан НПО «Андроидная техника» и Фондом перспективных исследований по заказу МЧС России. Робот оснащен речевой системой, способной распознавать слова и выдавать ответы. Робот может работать автономно, под управлением оператора на большом расстоянии через спутниковую связь. Был отправлен на МКС и возвращен обратно. Это был первый полет первого российского антропоморфного робота в космос. При полете Федора получены данные о том, как нагрузки при старте и невесомость могут повлиять на робототехнические системы, какие проблемы могут возникнуть.

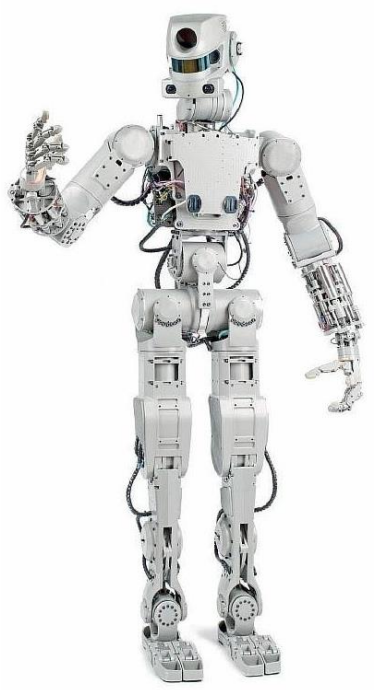


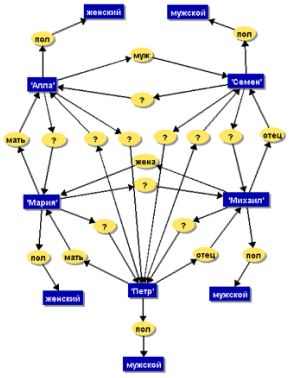
Рис. 10.14. Робот Федор

Индустриальные беспроводные Mesh-сети – это масштабируемые, защищенные, управляемые ячеистые сети, обеспечивающие Wi-Fi покрытия больших территорий. Все узлы Mesh-сетей автоматически конфигурируются, при выходе любого узла сети MESH абоненты переключаются на соседние узлы, а сами узлы переконфигурируются без потери связи. Трафик от абонентов такой сети автоматически перенаправляется по лучшему маршруту. Узлы Mesh-

сетей могут получать начальную конфигурацию и от контроллера, как и обычные точки доступа.

Решения Industrial Wi-Fi нашли широкое применение на многих производственных площадках, особенно на предприятиях в нефтегазовой, энергетической и горнодобывающей отраслях, на транспорте, железной дороге и метрополитене.

Глава 11





Концептуальные модели искусственного интеллекта, основанные на знаниях и правилах

Примеры решения задач

Задача 1. Составить программу на языке Пролог для нахождения неизвестных отношений в концептуальном графе, представленном на рис. 11.1. Здесь считается, что первоначально даны идентификаторы конкретных детей, выделенных из группы детей по каким-то признакам, – *мальчика_2* и *девочки_3*, которых родители и знакомые зовут *Андрей* и *Катя*. Далее для данного примера могли бы быть, например, определены имена классов «*мальчик*» и «*девочка*».

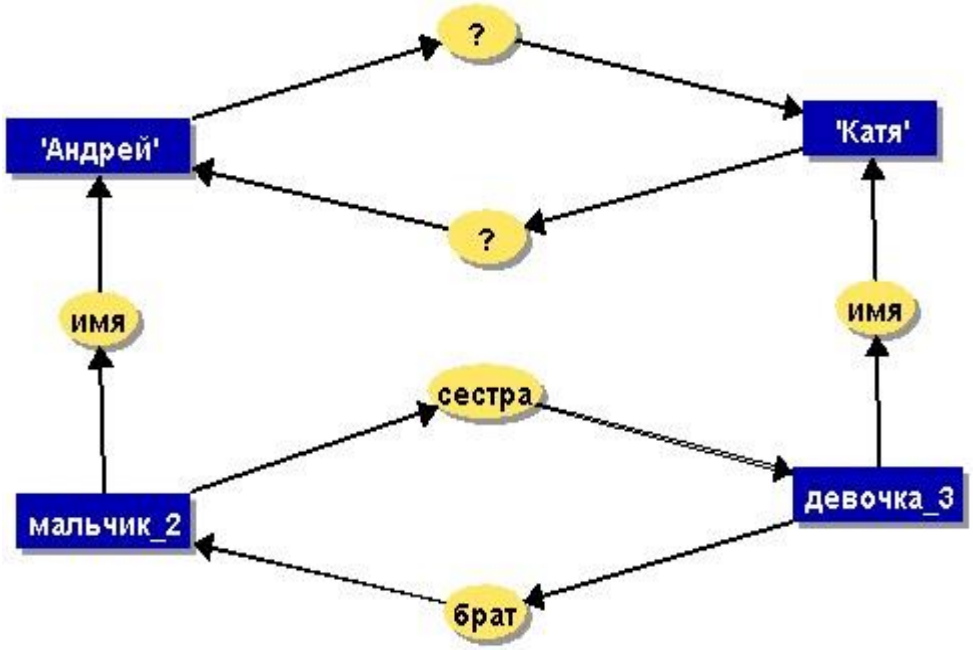


Рис. 11.1. Концептуальный граф с неизвестными отношениями

Решение. В представленной ниже программе использован тернарный, или трехместный, предикат «*родство*», в котором заданные ранее факты с именами бинарных, или двуместных, предикатов «*брат*» и «*сестра*» используются в качестве значений первого аргумента:



```
имя(мальчик_2,'Андрей').
имя(девочка_3,'Катя').
сестра(мальчик_2, девочка_3).
брат(девочка_3, мальчик_2).
родство(сестра, мальчик_2, девочка_3).
родство(брат, девочка_3, мальчик_2).
родство(C, A, B) :- родство(C, X, Y), имя(X, A), имя(Y, B).
родство(D, B, A) :- родство(D, Y, X), имя(X, A), имя(Y, B).
```

В данной программе используются рекурсивные правила. В результате выполнения запроса «*какими отношениями связаны определенные в графе концепты*?» получены следующие результаты:

```
? - родство(X, Y, Z).

X = сестра,
Y = мальчик_2,
Z = девочка_3;
X = брат,
Y = девочка_3,
Z = мальчик_2;
X = сестра,
Y = 'Андрей',
Z = 'Катя';
X = брат,
Y = 'Катя',
Z = 'Андрей';
```

Значения переменной *X* «*брат*» и «*сестра*» соответствуют именам искомых отношений.

На следующий запрос – "*кем является Катя для Андрея?*":

```
?- родство(X, 'Андрей', 'Катя').
```

получен ответ:

```
X = сестра,
```

т.е. для Андрея Катя является сестрой.

Задача 2. Следующий пример является модификацией предыдущего. Здесь требуется дополнить экстенсиональные отношения «*брат*» и «*сестра*» в базе знаний интенсиональными отношениями между детьми *Андреем* и *Катей*, получаемыми в результате логического вывода. Концептуальный граф представлен на рис. 11.2.

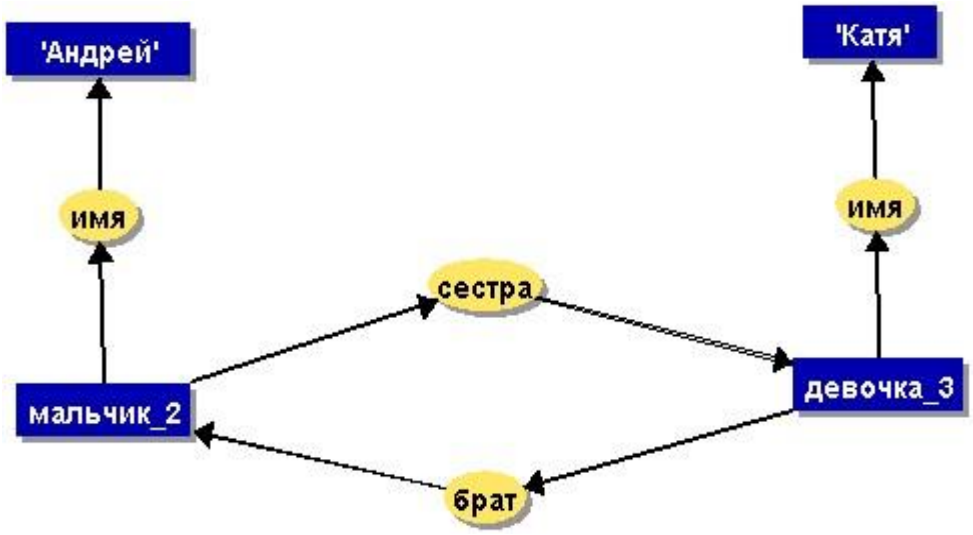


Рис. 11.2. Концептуальный граф с неполными отношениями «*брат*» и «*сестра*»

Решение. Программа имеет следующий вид:



```
имя(мальчик_2,'Андрей').
имя(девочка_3,'Катя').
сестра(мальчик_2, девочка_3).
брат(девочка_3, мальчик_2).
сестра(A, B) :- сестра(X, Y), имя(X, A), имя(Y, B).
брат(A, B) :- брат(X, Y), имя(X, A), имя(Y, B).
```

Как и в предыдущем примере, в данной программе используются рекурсивные правила. Например, запрос и ответ на этот запрос “*выдать все пары брат-сестра*” имеет следующий вид:

```
?- сестра(X,Y).

X = мальчик_2,
Y = девочка_3;
```

X = 'Андрей',

Y = 'Катя';

Запрос и ответ на вопрос "*правда ли, что Андрей – брат Кати?*" имеют следующий вид:

?- брат('Катя', 'Андрей').

true.

Задача 3. На рис. 11.3 задан концептуальный граф. Используя рекурсивное правило, составить программу на Прологе для определения отношения, отмеченного вопросительным знаком (выполнить самостоятельно!).

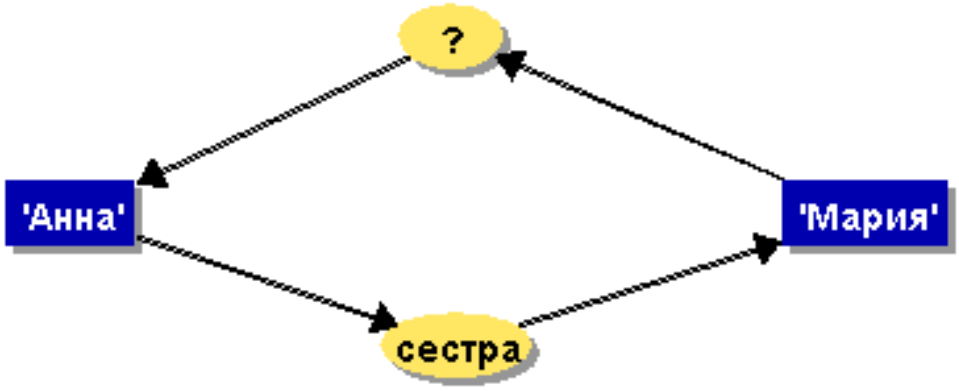


Рис. 11.3. Концептуальный граф с двумя вершинами-концептами

Задача 4. Доопределить по смыслу предметной области неизвестные отношения в концептуальном графе «Родство», представленном на рис. 11.4, разработать программу на языке Пролог с правилами для интенсиональной базы знаний с отношениями, первоначально обмеченными символом вопроса «?» на рисунке (выполнить самостоятельно!).

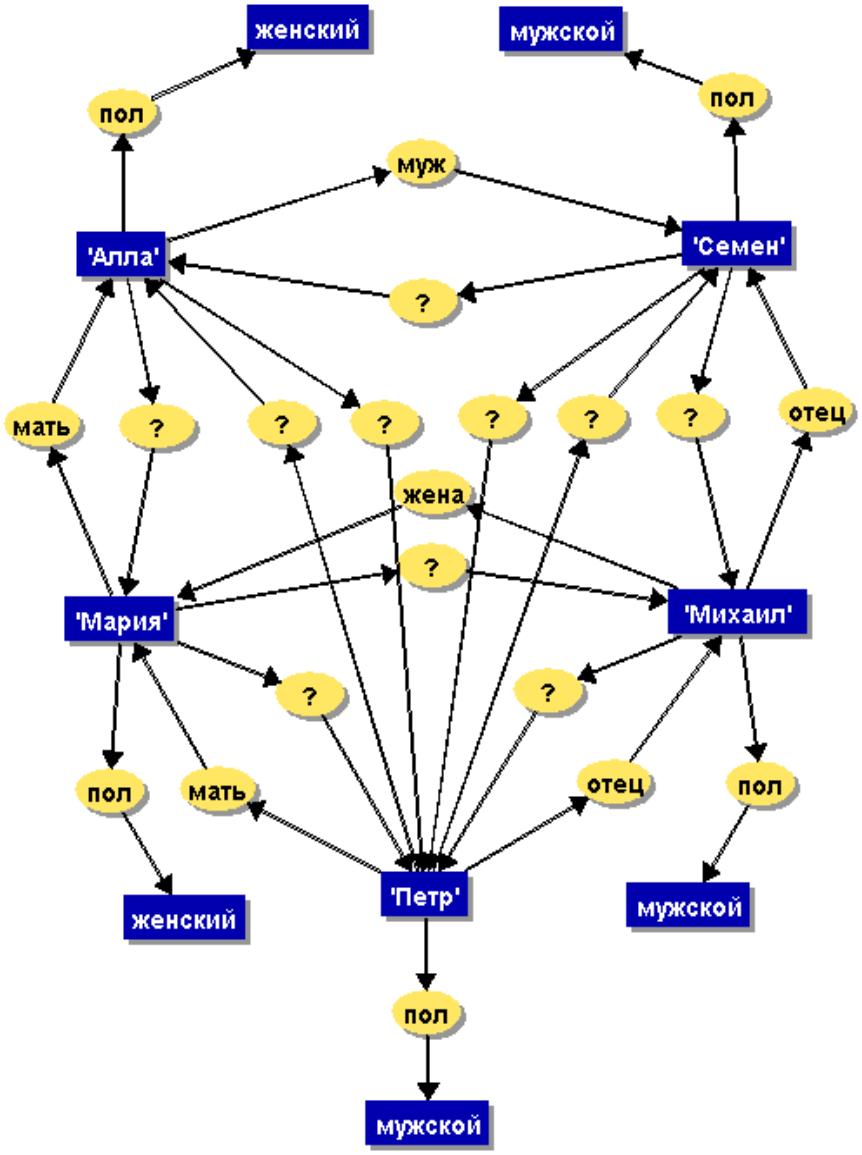


Рис. 11.4. Концептуальный граф «Родство»

Задача 5. Используя унарные предикаты и правила, разработать программу на языке Пролог для простой экспертной системы. Например, задавая различные признаки, требуется угадать животное – лев, тигр или зебра. Разработать программу на каком-либо недекларативном языке типа C++, Java, Python и др. (по указанию преподавателя).

Решение. Концептуальный граф представлен на рис. 11.5. Соответствующая этому графу программа представлена ниже:



признак(имеет_клыки).
признак(млекопитающее).
признак(плотоядный).
признак(семейство_кошачьих).
признак(хищник).
признак(охота_круглосуточная).
признак(обитает_в_Африке_и_Индии).
признак(живет_в_прайдах).
признак(самец_имеет_гриву).
признак(умеет_плавать).
признак(поддается_дрессировке).
признак(имеет_хвост).
признак(обитает_в_Африке).
признак(имеет_гриву).
признак(не_приручается).
признак(имеет_копыта).
признак(непарнокопытное).
признак(травоядное).
признак(защищается_лягание).
признак(имеет_полосы).
признак(обитает_в_Азии).
признак(охота_в_одиночку).
признак(охота_ночью_из_засады).
это(лев) :- признак(имеет_клыки),
признак(млекопитающее),
признак(плотоядный), признак(семейство_кошачьих),
признак(хищник), признак(охота_круглосуточная),
признак(обитает_в_Африке_и_Индии),
признак(самец_имеет_гриву), признак(умеет_плавать),
признак(поддается_дрессировке), признак(имеет_хвост).
это(тигр) :- признак(млекопитающее),
признак(семейство_кошачьих),
признак(хищник), признак(плотоядный),
признак(обитает_в_Азии),
признак(поддается_дрессировке),
признак(умеет_плавать),
признак(охота_в_одиночку), признак(имеет_хвост),
признак(охота_ночью_из_засады), признак(имеет_клыки),
признак(имеет_полосы).
это(зебра) :- признак(обитает_в_Африке),

признак(имеет_хвост),
признак(имеет_гриву), признак(имеет_клыки),
признак(не_приручается), признак(имеет_копыта),
признак(непарнокопытное), признак(травоядное),
признак(защищается_лягание), признак(имеет_полосы).

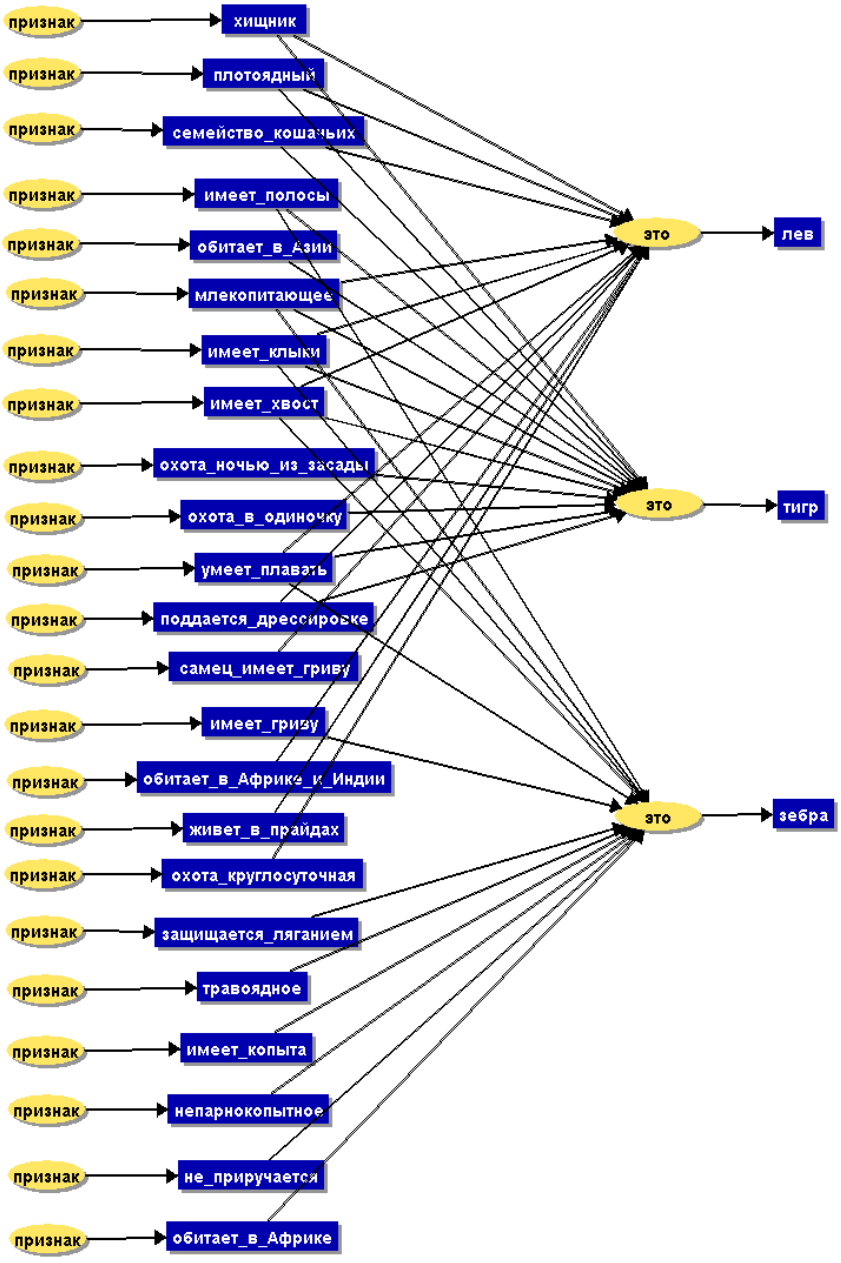


Рис. 11.5. Концептуальный граф для прототипа экспертной системы

Построение концептуального графа, изображенного на рис. 11.5, основано на графическом представлении унарных предикатов типа «*признак(X)*» и «*это(Y)*», где переменная *X* принимает значения в множестве признаков, характерных для различных животных, а *Y* – в множестве имен животных.

Напомним, что запятая «,» между литералами в теле правила означает конъюнкцию. Тело правила может быть представлено и в дизъюнктивной форме, при этом между литералами ставится точка с запятой «;».

Отрицание предиката задается с помощью унарного предиката *not*. Например, тот факт, что животное не имеет полос, записывается в Прологе как *not(признак(имеет_полосы))*. Строго говоря, в теле правила возможно использовать отрицание, при этом литерал начинается с функции *not*. Например, такой литерал, как *not(признак(самец_имеет_гриву))*, является допустимым в правой части правила.

При использовании недекларативных языков для решения данной задачи можно использовать вместо унарных предикатов булевы переменные. Более сложный вариант решения предполагает использование элементов нечеткой логики, позволяющей сделать выводы при совпадении не всех признаков с эталоном.

Задача 6. Работа с *n*-арными ($n > 2$) отношениями. При построении концептуальных графов используются в основном унарные и бинарные предикаты и отношения (отношение – это область истинности одноименного в общем случае предиката). Предикаты с арностью (местностью) более двух редактором концептуальных графов *CharGer* не поддерживаются, что не позволяет обозначать все связи в графическом представлении *n*-арного предиката ($n > 2$). В качестве примера рассмотрен граф 4-арного (четырехместного) предиката на рис. 11.6.

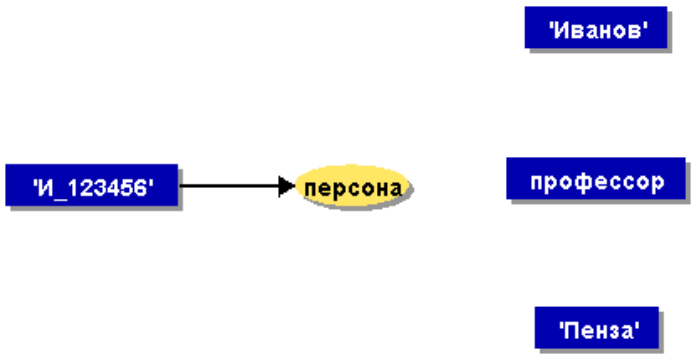


Рис. 11.6. Концептуальный граф с недостроенными связями для 4-арного предиката «*персона*»

Концептуальный граф 4-арного предиката «*персона*» с недостроенными, но подразумеваемыми связями, формализует фразу «*персона с табельным номером И_123456 является профессором Ивановым, проживающим в Пензе*». Соответствующее данному графу высказывание имеет следующий вид:



персона(‘И_123456’, ‘Иванов’, профессор, ‘Пенза’),

где «*персона*» – имя предиката, а в скобках указаны предметные константы – значения аргументов предиката. Подобное высказывание можно использовать в качестве факта экстенсиональной базы знаний. Этого высказывания недостаточно для раскрытия смысла (семантики) фразы «*персона с табельным номером И_123456 является профессором Ивановым, проживающим в Пензе*». Как известно, понятиям логического программирования соответствуют понятия реляционных баз данных: предикат – отношение, аргумент предиката (предметная переменная) – атрибут, факт – кортеж, цель – запрос. Отношению реляционной алгебры для рассматриваемой задачи соответствует следующая таблица:

Персона			
Таб_номер	Фамилия	Должность	Место_проживания
…	…	…	…
‘И_123456’	‘Иванов’	профессор	‘Пенза’
…	…	…	…

Описание предметной области, или область экспертизы, здесь расширено за счет введения атрибутов «*Таб_номер*», «*Фамилия*», «*Должность*», «*Место_проживания*». Эти новые, но подразумеваемые ранее знания можно представить в концептуальном графе (рис. 11.7).

Этот граф более информативен, но для достройки необходимых связей потребовался бы другой редактор концептуальных графов, отличный от редактора *CharGer*.

На рис. 11.8 приведен концептуальный граф другого логического представления той же фразы бинарными предикатами. Введенным ранее именам атрибутов здесь соответствуют одноименные предикаты.

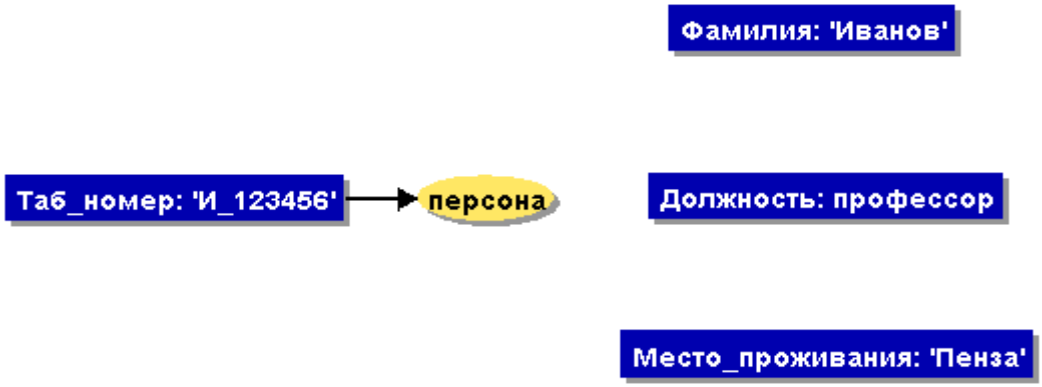


Рис. 11.7. Концептуальный граф с недостроенными связями 4-арного предиката «*персона*» с указанием имен атрибутов

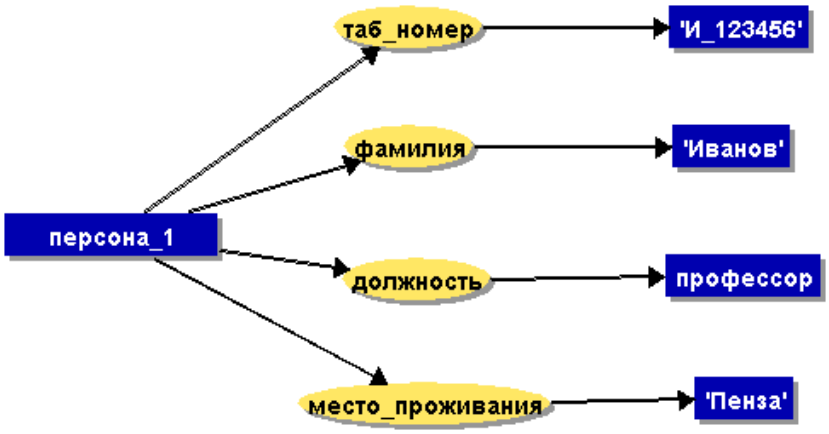


Рис. 11.8. Концептуальный граф логического представления фразы бинарными предикатами

Подобное описание более информативно и расширяет семантику предметной области по сравнению с графом на рис. 11.6. Теперь факты экстенсиональной базы знаний в Прологе следует представить таким образом:



```
таб_номер(персона_1, ‘И_123456’).
фамилия(персона_1, ‘Иванов’).
должность(персона_1, профессор).
место_проживания(персона_1, ‘Пенза’).
```

Предметная константа «*персона_1*» указывает на конкретную личность. Имя «*персона*» может быть использовано в качестве имени класса персон, как представлено на рис. 11.9. Здесь отношение «*например*» означает, что объект «*персона_1*» является экземпляром класса «*персона*».

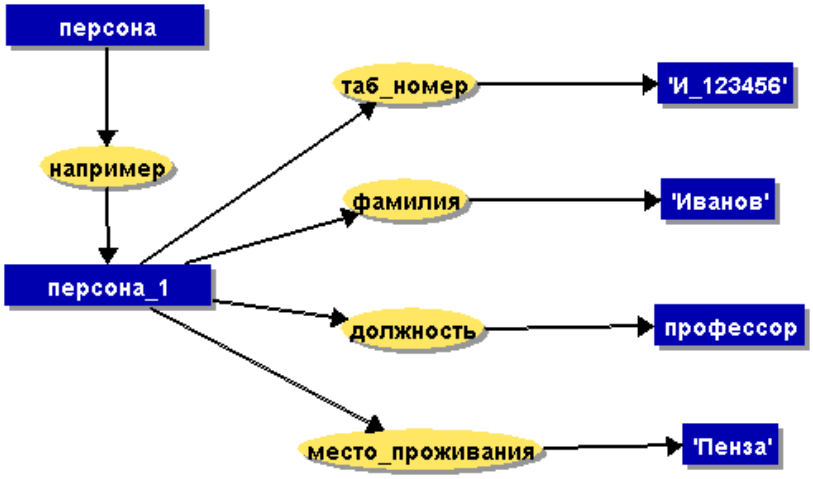


Рис. 11.9. Концептуальный граф с отношением «*например*» типа «*класс – элемент класса*»

При решении задач на построение и анализ концептуальных графов и семантических сетей от читателя предполагается знание некоторых элементов языка Пролог. Считается, что в дальнейшем связь концептуальных графов и Пролога поможет легче разобраться в реализации Семантической паутины (от англ. *Semantic Web*) – общедоступной глобальной семантической сети, формируемой на базе Всемирной паутины (англ. *World Wide Web*) путем стандартизации представления информации в виде, пригодном для машинной обработки. Понятие «семантическая паутина» было впервые введено Тимом Бернерсом-Ли (изобретателем Всемирной паутины) и называется им «следующим шагом в развитии Всемирной паутины» (см. https://ru.wikipedia.org/wiki/Семантическая_паутина).

Ранее в настоящем пособии были определены понятия экстенсионала и интенсионала базы знаний. Аналогично определяются и соответствующие свойства семантических сетей. Экстенсиональная семантическая сеть описывает факты, или экстенсиональные знания,

о моделируемых объектах и является как бы моментальным снимком текущего состояния предметной области. Интенсиональная семантическая сеть описывает общую структуру моделируемой предметной области на основе абстрактных объектов и отношений. Оба вида знаний ранее были определены для канонических концептуальных графов.

Под концептуализацией принято понимать процесс перехода от представления предметной области к точной спецификации этого описания на некотором формальном языке, ориентированном на компьютерное представление. Семантические сети в общем случае определяются как соединения концептуальных графов. Концептами могут быть понятия, события и свойства. Понятия – это сведения об абстрактных или физических объектах предметной области; события – это действия, которые могут внести изменения в предметную область, в результате чего изменяется ее состояние; свойства используются для уточнения понятий, событий или самих свойств. Применительно к понятиям, свойства описывают их особенности или характеристики; применительно к событиям – продолжительность, место и участников реализации и др. Пример семантической сети с событиями приведен на рис. 11.10.

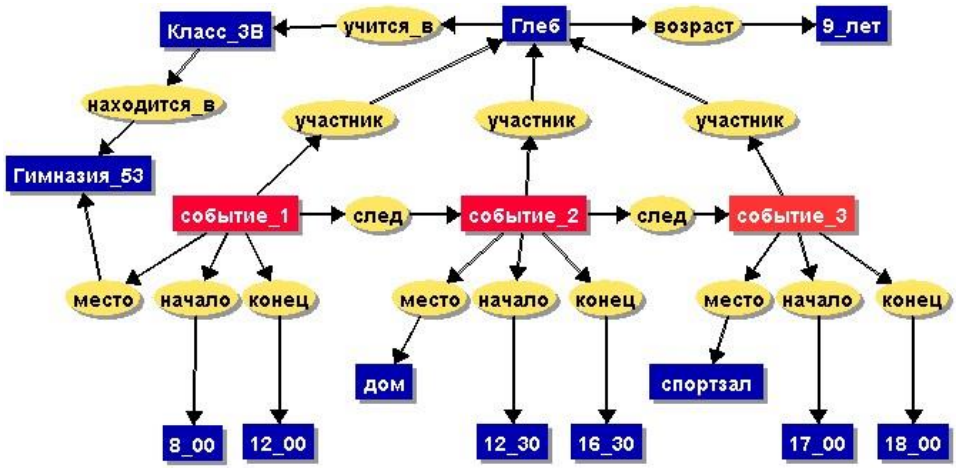


Рис. 11.10. Пример семантической сети с событиями, или сценария

Здесь для каждого события определены его длительность, место в цепочке событий и участники в их реализации. Семантические сети такого вида также называются **сценариями** и имеют важное значение в информатике и вычислительной технике.

Виды отношений в семантических сетях

При выполнении лабораторных и самостоятельных работ, связанных с построением и использованием семантических сетей, следует определиться с типами и именами используемых отношений. В литературе, как в русскоязычной, так и англоязычной, и особенно в переводной, используются различные имена для одних и тех же отношений. При разработке представлений в виде семантических сетей важную роль играют следующие бинарные отношения:

– отношение «***является***» (синоним: «***конкретизация***»), отражающее принадлежность объекта некоторому классу объектов; в английском варианте это отношение имеет обычно имя "***is_a***";

– отношение «***класс – элемент_класса***», или «например» ("***for_instance***");

– отношение «***это***», которое относится к классу родовидовых отношений или отношений «подкласс-класс»; в английском варианте это отношение имеет обычно имя "***a_kind_of***", или "***ako***". Иногда для обозначения отношения «***это***» используют англоязычное имя "***is_a***", что не всегда верно;

– отношение «***имеет_частью***» ("***has_a_part***"), указывающее на то, что одно понятие представляет часть другого; противоположное отношение – «***является_частью***» ("***a_part_of***");

– отношение «***подмножество***» ("***subset***");

– отношение «***надмножество***» ("***superset***");

– отношение «***быть_элементом_множества***» ("***element***", "***instance_of***");

– отношение «***есть***», указывающее на то, что одно понятие служит атрибутом другого.

Следует отметить, что не все перечисленные термины и переводы точно соответствуют исследуемой предметной области. Кроме того, они не стандартизованы.

Другие отношения отражают функциональные связи (определяемые обычно глаголами «производит», «влияет» и др.); качественные связи («больше», «меньше», «равно» и др.); пространственные связи («далеко от», «близко от», «за», «под», «над» и др.); временные связи («раньше», «позже», «в течение» и др.); атрибутивные связи («иметь свойство», «иметь значение» и др.).

В литературе распространена следующая классификация семантических отношений:

– ***лингвистические*** отношения (глагольные, атрибутивные и падежные), отображающие смысловую взаимосвязь между событиями, между событиями и понятиями или свойствами;

– ***логические*** и ***квантифицированные*** отношения, соответствующие операциям, используемым в алгебре логики и логике предикатов (дизъюнкция, конъюнкция, отрицание, импликация и операции, связанные использованием кванторов общности и существования);

– ***теоретико-множественные*** отношения – это отношения между множеством и подмножеством, частью и целым, множеством и элементом множества.

Проблема поиска решения в базе знаний типа семантической сети сводится к задаче поиска фрагмента сети, соответствующего некоторой подсети. Известно, что проблема изоморфного подграфа – это вычислительная задача, в которой для двух графов *A* и *B* нужно определить, содержит ли граф *A* подграф, изоморфный графу *B*. Задача поиска изоморфного подграфа является обобщением задачи поиска максимальной клики и задачи определения наличия в графе гамильтонова цикла, и, следовательно, относится к *NP*-полным задачам.

Во многих случаях использование языков логического программирования типа Пролог или языков запросов к базам данных типа *SQL* помогает найти приемлемое по времени и качеству решение задач, связанных с семантическими сетями.

Задача «Глеб и его друзья» на определение свойств отношений

Иногда при построении концептуальных графов возникает ситуация, связанная с тем, что некоторые отношения не имеют функционального характера. Поэтому в редакторе *CharGer* не допускается построение всех (кроме единственной!) связей в концептуальном графе, представленном на рис. 11.11.

Построение подобного графа следует заменить построением графа, представленного на рис. 11.12. Как в первом, так и во втором случае из анализа ситуации предметной области видно, что отношение «*дружит*» не имеет функционального характера, поскольку

требуется установить это отношение между всеми парами концептов, где каждая пара начинается с концепта «*Глеб*». Этому условию отвечает концептуальный граф на рис. 11.12.

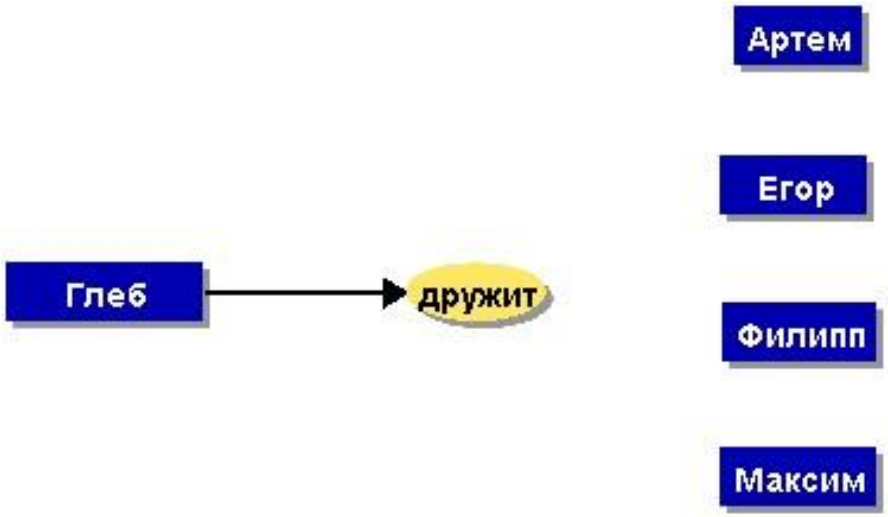


Рис. 11.11 Концептуальный граф с недостроенными в редакторе *CharGer* связями

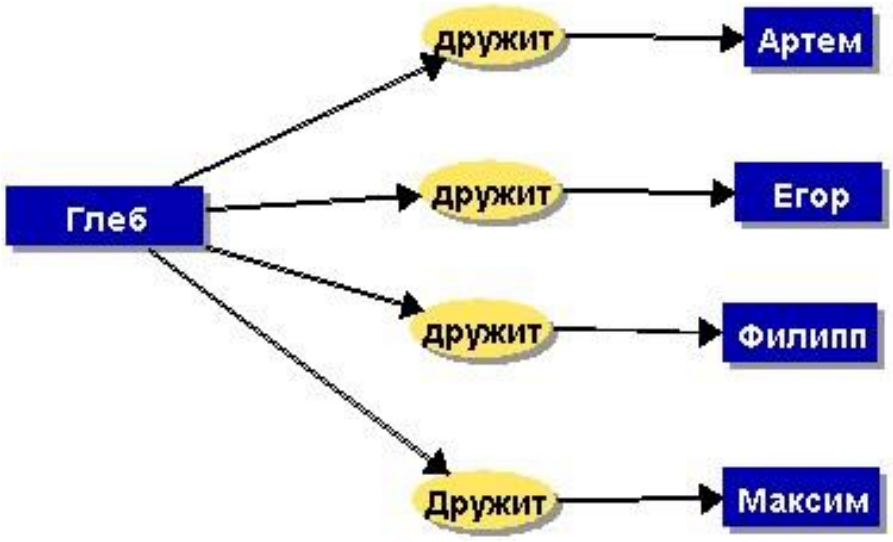


Рис. 11.12. Корректный концептуальный граф для бинарного отношения «*дружит*»

Другой вариант построения корректного концептуального графа для бинарного отношения «*например*» типа «*класс – экземпляр класса*» представлен на рис. 11.13.

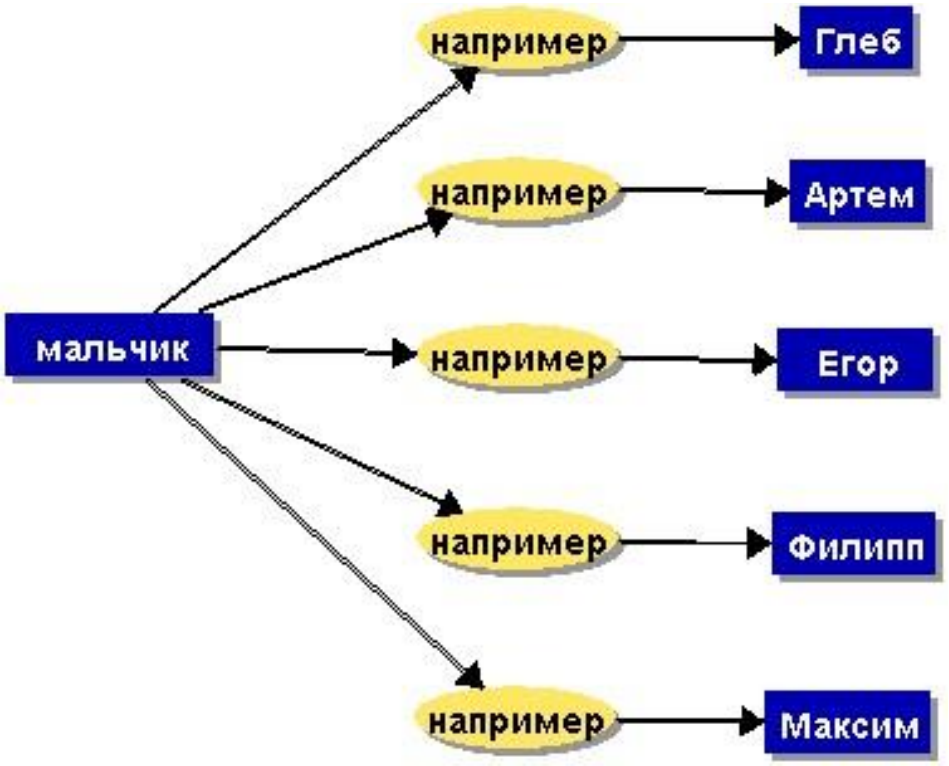


Рис. 11.13. Корректный концептуальный граф для бинарного отношения «например»

Выполнение команды **Operation | Summarize Everything** над графом, представленном на рис. 11.13, в редакторе *CharGer* и дальнейшая обработка результата с помощью программы **main4.exe** даст следующий результат для экстенсионального отношения, или фактов программы на языке Пролог:



```
например(мальчик, 'Глеб').
например(мальчик, 'Артем').
например(мальчик, 'Егор').
например(мальчик, 'Филипп').
например(мальчик, 'Максим').
```

Эти же факты можно получить, выполнив команды **File | Show English** и немного подкорректировав полученный результат.

Задача «Школьники»

При решении следующей задачи рассматривается концептуальный граф, приведенный на рис. 11.14. Здесь графически заданы факты, задающие экстенсионал базы знаний. Требуется с помощью правила вывода получить факты, соответствующие интенсионалу базы знаний: в данном примере необходимо по ранее известным фактам определить дополнительные свойства концепта с именем «*Глеб*», т.е. умения мальчика Глеба. Отношение «*конкретизация*» типа «*экземпляр класса – класс*» эквивалентно по смыслу отношению «*является*». Концепты «*мальчик*», «*личность*», «*школьник*» обозначают классы объектов, между которыми установлены отношения «*это*», «*статус*» типа «*подкласс – класс*». Отношение «*умеет*» относится к типу «*класс – экземпляр класса*».

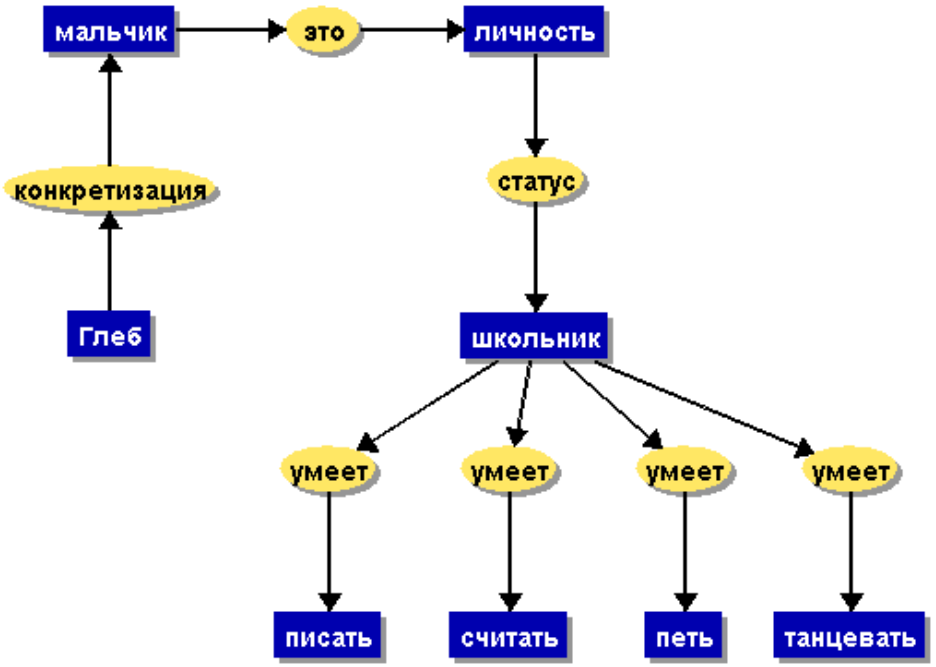


Рис. 11.14. Экстенсиональные отношения, заданные при помощи концептуального графа

Решение данной задачи может быть получено с помощью запроса

?- умеет(X, U)

к следующей программе на языке Пролог:

```
% Факты
конкретизация('Глеб', мальчик).
это(мальчик, ребенок).
это(ребенок, школьник).
умеет(школьник, писать).
умеет(школьник, считать).
умеет(школьник, петь).
умеет(школьник, танцевать).
% Правило вывода для получения
% интенсионала базы знаний
умеет(X, U):-конкретизация(X, Z), это(Z, W), статус(W, V), умеет(V,U).
```

Дополнительных фактов, или высказываний, здесь немного, что позволяет, не слишком усложняя исходный концептуальный граф на рис. 11.14, достроить новые связи «*умеет*», показанные на рис. 11.15.

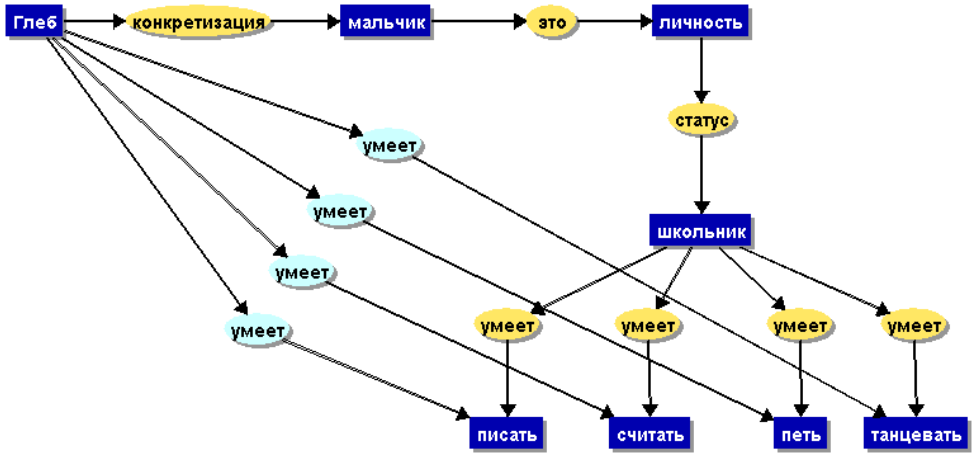


Рис. 11.15. Концептуальный граф с дополнительными связями, заданными с помощью интенсионала «*умеет*»

С помощью программы на языке Пролог можно таким же образом, как и ранее, получить дополнительные интенсиональные знания о предметной области, например, об умениях группы школьников (рис. 11.16).

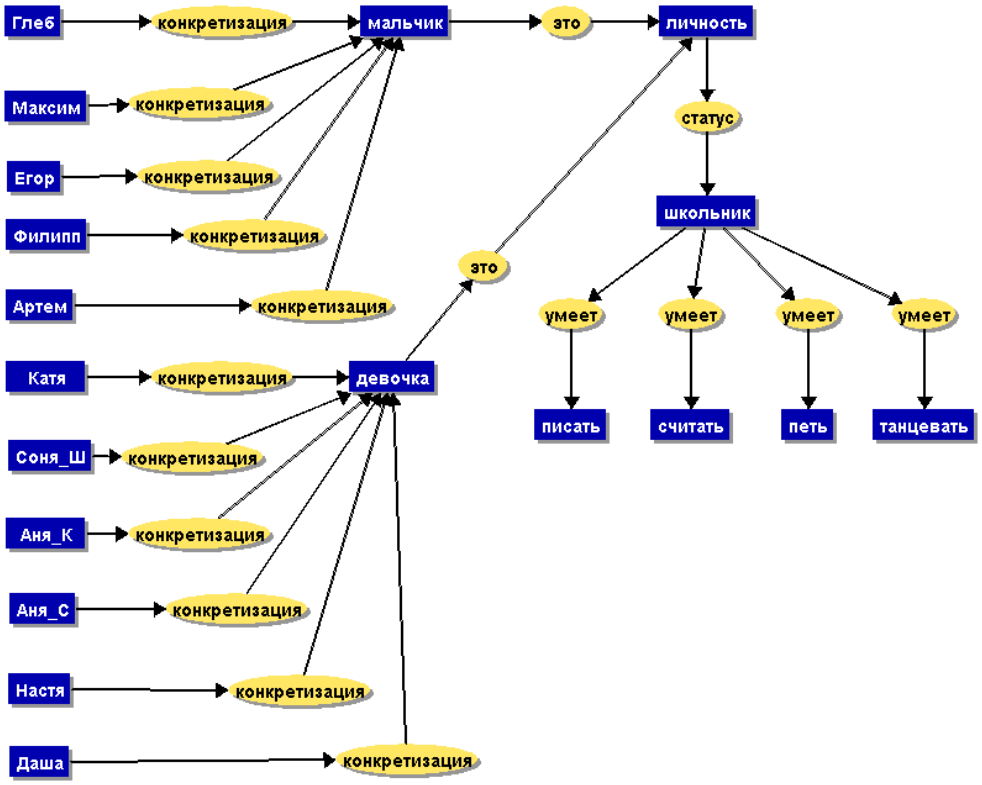


Рис. 11.16. Экстенсиональные отношения, заданные на концептуальном графе для множества школьников

Для автоматического составления базы знаний на языке Пролог выполним в редакторе *CharGer* команду **Show English**, с помощью которой получим следующее текстовое представление концептуального графа, представленного на рис. 11.16:

конкретизация of Настя is девочка
конкретизация of Филипп is мальчик
конкретизация of Егор is мальчик
конкретизация of Даша is девочка
конкретизация of Глеб is мальчик
конкретизация of Аня_С is девочка
конкретизация of Максим is мальчик
конкретизация of Аня_К is девочка
конкретизация of Соня_Ш is девочка
умеет of школьник is танцевать

умеет of школьник is петь
конкретизация of Артем is мальчик
это of девочка is личность
умеет of школьник is считать
умеет of школьник is писать
статус of личность is школьник
это of мальчик is личность and
конкретизация of Катя is девочка

Затем, после несложного редактирования, сделанного вручную, получим следующие факты и одно правило:



```
% Факты (экстенсионал базы знаний)
конкретизация('Настя', девочка).
конкретизация('Артем', мальчик).
конкретизация('Катя', девочка).
конкретизация('Филипп', мальчик).
конкретизация('Егор', мальчик).
конкретизация('Даша', девочка).
конкретизация('Глеб', мальчик).
конкретизация('Аня_С', девочка).
конкретизация('Максим', мальчик).
конкретизация('Аня_К', девочка).
конкретизация('Соня_Ш', девочка).
это(девочка, личность).
это(мальчик, личность).
статус(личность, школьник).
умеет(школьник, танцевать).
умеет(школьник, считать).
умеет(школьник, писать).
умеет(школьник, петь).
% Правило вывода для получения
% интенсионала базы знаний
умеет(X, U):-конкретизация(X, Z), это(Z, W), статус(W, V), умеет(V,U).
```

Графическое представление дополнительных связей, характеризующих умения группы школьников, приведет к сильному загромождению рисунка, поэтому рекомендуется ограничиться использованием интенсионала базы знаний, получаемого при выполнении запроса

?- умеет(X, U)

к приведенной программе на языке Пролог.



БУДЬТЕ ЗДОРОВЫ !!!

Задача «Диагностика гриппа»

Особенности формирования правил логического вывода необходимо учитывать при решении многих задач. Например, на рис. 11.17 представлен концептуальный граф, используемый при выставлении диагноза «*грипп*». Здесь предполагается, что диагноз «*это(диагноз, грипп)*» выставляется при наличии ряда исходных признаков, т.е. высказывание «*это(диагноз, грипп)*» становится истинным после проверки на истинность конъюнкции высказываний-жалоб.

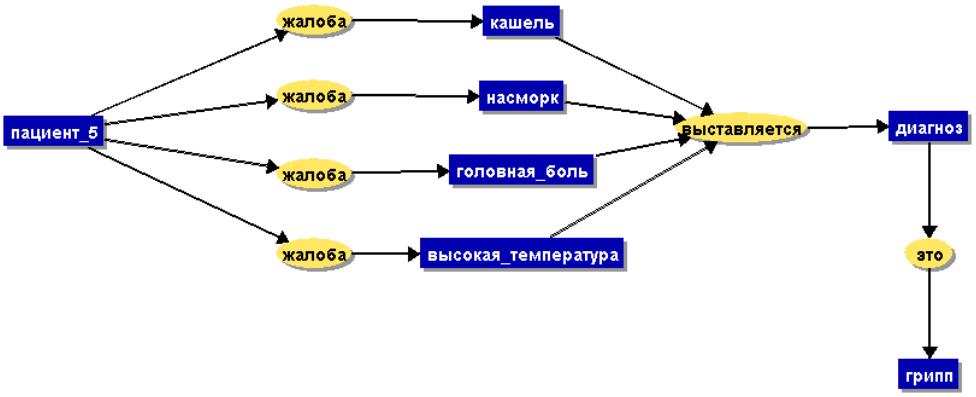


Рис. 11.17. Пример концептуального графа для выставления диагноза «*грипп*»

Рассмотрим случай (рис. 11.18), когда концептом является высказывание, что допускается редактором *CharGer*. Здесь вместо отношения «*это*» использовано более уместное по практическим причинам отношение «*вероятно_это*», а вместо не вполне определенного имени отношения «*выставляется*» использовано логическое отношение «*конъюнкция*». Например, врач выставляет предварительный диагноз заболевания «*вероятно_это(диагноз, грипп*)» для конкретного пациента «*пациент_5*» по совокупности признаков.

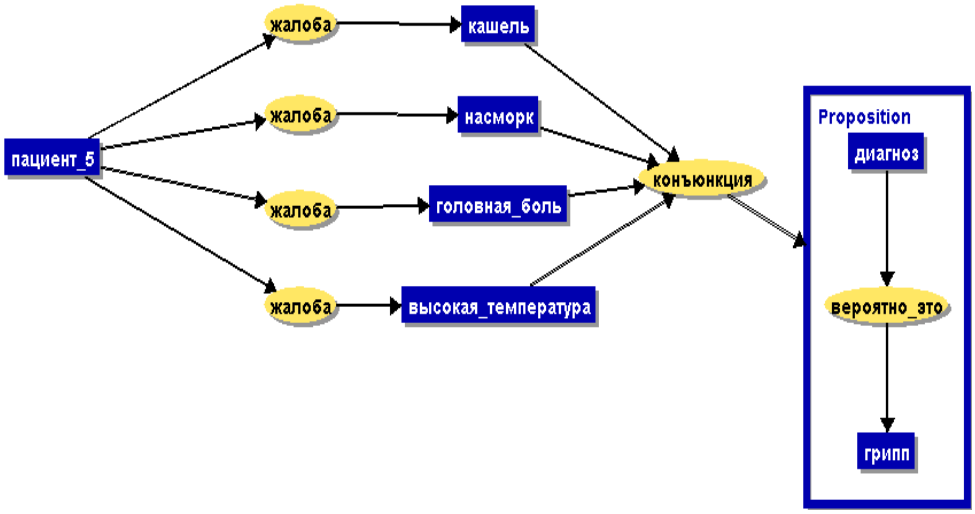


Рис. 11.18. Использование высказывания «*вероятно_это(диагноз, грипп)*» в качестве концепта

Выполнение команды **Operation | Summarize Everything** над графом, представленном на рис. 11.8, в редакторе *CharGer* дает следующий результат:

a Proposition
consists of a диагноз and a грипп
a грипп
is the вероятно_это of a диагноз

a высокая_температура
is the жалоба of a пациент_5
a головная_боль
is the жалоба of a пациент_5

a Proposition
is the конъюнкция of a кашель and
a насморк and
a головная_боль and
a высокая_температура

a насморк
is the жалоба of a пациент_5
a кашель
is the жалоба of a пациент_5

Примечание. Если не учитывать высказывание-концепт «Proposition», то для получения остальных фактов Пролога по концептуальному графу можно воспользоваться программой **main4.exe**, а затем вручную добавить другие отношения и правила вывода.

Отношение «*конъюнкция*» представляется в Пролог-программе в правой части правила вывода (само имя отношения «*конъюнкция*» будет отсутствовать, так как его роль выполняют запятые, разделяющие литералы). Для указания на симптомы «*кашель*», «*насморк*», «*головная_боль*», «*высокая_температура*» в правой части правила будем использовать следующие высказывания (факты), которые указывают на жалобы и самого источника жалоб на недомогания, т.е. на конкретного пациента:



```
жалоба(пациент_5, кашель).
жалоба(пациент_5, насморк).
жалоба(пациент_5, головная_боль).
жалоба(пациент_5, высокая_температура).
```

Факты и правила Пролога в результате имеют следующий вид:



```
% Формирование отношений и правила вывода,
% которые в целом не поддерживаются
% программой main4.exe
% Факты:
жалоба(пациент_5, кашель).
жалоба(пациент_5, насморк).
```

жалоба(пациент_5, головная_боль).
жалоба(пациент_5, высокая_температура).
%Правило:
вероятно_это(диагноз, грипп):-
жалоба(пациент_5, кашель),
жалоба(пациент_5, насморк),
жалоба(пациент_5, головная_боль),
жалоба(пациент_5, высокая_температура).

Эту же программу на Прологе можно было получить и на основе анализа концептуального графа, представленного на рис. 11.19.

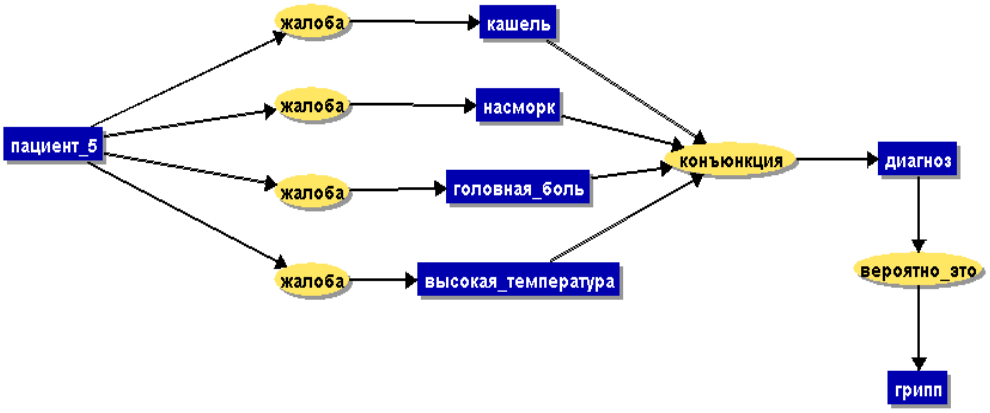


Рис. 11.19. Концептуальный граф, построенный на основе высказываний одного уровня

При уточнении диагноза лечащий врач использует, естественно, и другие признаки заболевания, а также результаты проводимых анализов. Нередко заболевание заподозрить можно и по одному из симптомов, тем более, что сразу затруднительно проанализировать все признаки начавшегося заболевания. Соответствующая модификация правила постановки диагноза имеет следующий вид:



% Правило:
вероятно_это(диагноз, грипп):-
жалоба(пациент_5, кашель);
жалоба(пациент_5, насморк);
жалоба(пациент_5, головная_боль);
жалоба(пациент_5, высокая_температура).

Точки с запятыми («;») в правой части правила вывода в Прологе обозначают логическую операцию «*дизъюнкция*». Концептуальные графы, описывающие данную ситуацию, представлены на рис. 11.20 и 11.21.

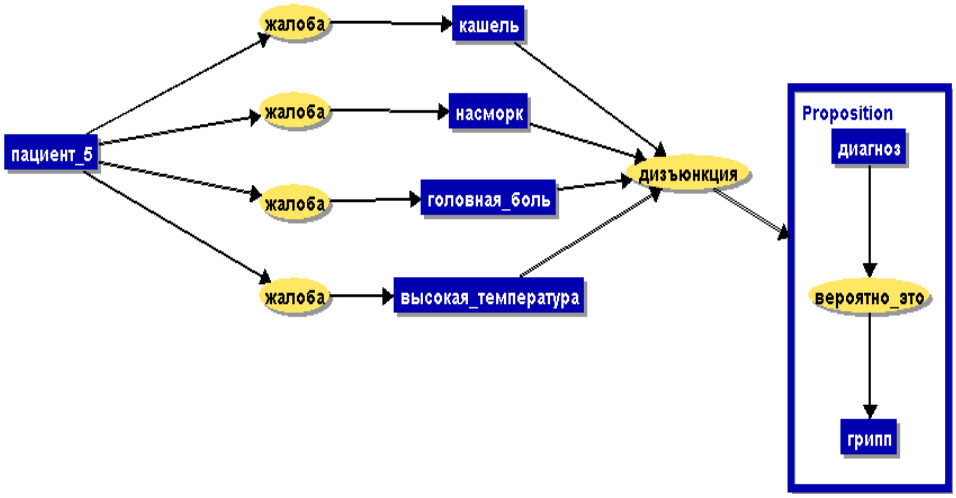


Рис. 11.20. Концептуальный граф с концептом-высказыванием, описывающий постановку вероятного диагноза по одному из симптомов

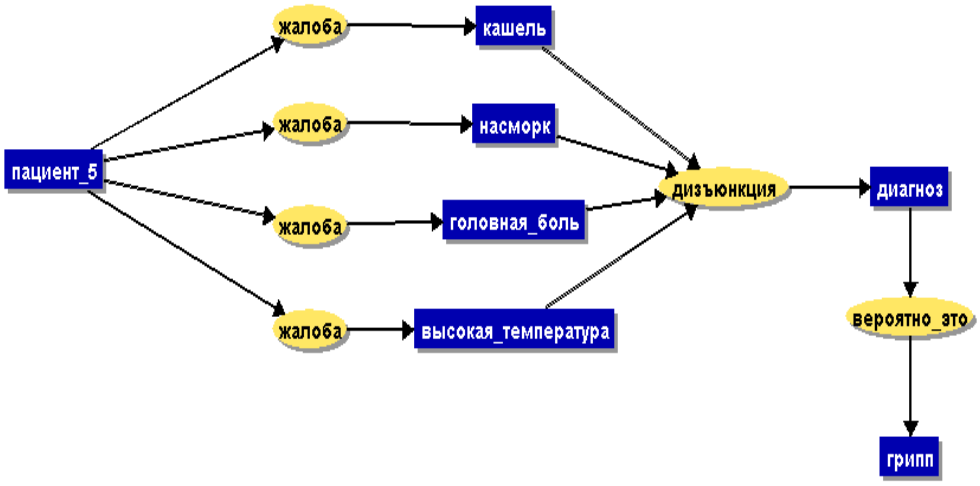


Рис. 11.21. Концептуальный граф с высказываниями одного уровня, описывающий постановку вероятного диагноза по одному из симптомов

Задача «Диагностика и лечение пневмонии»

Продолжая рассматривать **условный** пример диагностики заболевания (т.е. не предназначенный для лечения конкретных больных!), сформулируем обратную задачу: построить концептуальный граф, описывающий ход лечения больного с диагнозом «*пневмония*», используя готовую программу на языке Пролог. При составлении программы авторы настоящего учебного пособия получали консультацию у практикующих врачей. Программа имеет следующий вид (для перевода незнакомых медицинских терминов и других ключевых слов воспользуйтесь медицинским англо-русским словарем):



```
% Факты базы знаний
% Knowledge base facts
this_is('Mucolytics', 'Ambroxol').
this_is('NSAID', 'Paracetamol').
this_is('Detoxification', 'Reamberin').
is_a('Therapy', 'Positive_dynamics').
this_is('Diagnosis', 'Pneumonia').
this_is('Antibiotics', 'Levofloxacin').
value('Name', 'Ivanov_Ivan_Ivanovich').
has_a('Patient', 'Age').
has_a('Patient', 'Name').
has_a('Patient', 'Work').
has_a('Patient', 'Gender').
is_a('Continuation_of_therapy', 'Reconvalescence').
conduction(‘Pneumonia’, ‘Therapy’).
has_a('Patient', 'Complaints').
is_a('Correction_of therapy', 'Reconvalescence').
conduction('Discharge', 'Secondary_prevention').
value('Weight', '80_kg' ).
value('Height', '178_cm').
value('Age', '72_years').
this_is('Secondary_prevention', 'Vaccination').
treatment('Pneumonia', 'Antibiotics').
treatment('Pneumonia', 'NSAIDs').
```

treatment('Pneumonia', 'Detoxification').
this_is('Vaccination', 'Pneumococcal_vaccine').
this_is('Vaccination', 'Influenza_vaccine').
value('Gender', 'Male').
is_a('Sign', 'Dullness_on_percussion').
is_a('Sign', 'Wheezing_in_lungs').
treatment('Pneumonia', 'Mucolitics').
conduction('Reconvalescence', 'Discharge').
has_a('Patient', 'Height').
multiplicity ('500_mg', '1p_day').
has_a('Patient', 'Weight').
multiplicity ('500_mg', '1p_day').
multiplicity ('30_mg', '2_3p_day').
multiplicity ('500_mg', '1_2p_day').
multiplicity('500_ml', '1_2p_day').
has_a('Patient', 'Sign').
is_a('Sign', 'Leukocytosis').
is_a('Sign', 'Stab_shift_left').
is_a('Sign', 'Increase_CRP').
is_a('Sign', 'Focal_infiltration_on_Rg').
this_is('Sanatorium_resort_treatment', 'Climatic_resorts').
this_is('Sanatorium_resort_treatment', 'Mud_resorts').
this_is('Physiotherapy', 'Electro-sleep').
necessary('Positive_dynamics','Continuation_of_therapy')
necessary('Negative_dynamics', 'Correction_therapy').
is_a('Complaint', 'Shortness of breath').
is_a(Complaint'', 'Cough_with_sputum').
is_a('Complaint', 'Weakness').
is_a('Complaint', 'Fever').
is_a('Complaint', 'Fatigue').
is_a('Sign', 'Epid_anamnesis').
this_is('Physiotherapy', 'Bar_Therapy').
this_is('Exercise therapy', 'Gentle_training_mode').
this_is('Exercise', 'Training_mode').
dose('Azithromycin','500_mg').
dose('Levofloxacin', '500_mg').
this_is('Physiotherapy', 'Balneotherapy').
dose('Ambroxol', '30_mg').

```
dose('Paracetamol', '500_mg').
dose('Reamberin', '500_ml').
conduction('Discharge', 'Rehabilitation').
has_a('Job', 'Name').
has_a('Job', 'Address').
is_a('Therapy', 'Negative_dynamics').
this_is('Rehabilitation', 'Sanatorium_resort_treatment').
this_is('Rehabilitation', 'Physiotherapy').
this_is('Rehabilitation', 'Exercise_therapy').
required('Work', 'Sick_list').
this_is('Antibiotics', 'Azithromycin').
is_a('Sign', 'Streptococcus_pneumoniae_in_sputum').

% Правило вывода
% Выставляется диагноз
% Inference rule
set('Diagnosis', 'Pneumonia') :-
is_a('Complaint', 'Fever'),
is_a('Complaint', 'Shortness of breath'),
is_a('Complaint', 'Cough_with_sputum'),
is_a('Complaint', 'Weakness'),
is_a('Complaint', 'Fatigue'),
is_a('Sign', 'Wheezing_in_lungs'),
is_a('Sign', 'Dullness_on_percussion'),
is_a('Sign', 'Leukocytosis'),
is_a('Sign', 'Stab_shift_left'),
is_a('Sign', 'Increase_CRP'),
is_a('Sign', 'Focal_infiltration_on_Rg'),
is_a('Sign', 'Streptococcus_pneumoniae_in_sputum'),
is_a('Sign', 'Epid_analysis').
```

Данная программа постановки диагноза могла бы использоваться в экспертных системах типа «Помощник врача» или «Обучение молодого специалиста».



Задача «Отель»

Целью анализа фраз естественного языка является структуризация семантической информации для ее представления в экспертной системе, соответствующая системному или функциональному проектированию объекта. Такое представление будет выглядеть более удачным: недостающие семантические связи могут быть достроены впоследствии, в том числе семантические интенсиональные связи быть достроены при помощи логического вывода.

Приведем примеры подобных информативных концептуальных графов с их контекстным наполнением. На рис. 11.22 представлен концептуальный граф для предметной области «Отель», а на рис. 11.23 в качестве предметной области выбраны «Компьютеры». Данные примеры разработаны магистрантами Д. Г. Курышевой и А. С. Гусаровым при выполнении лабораторных заданий по дисциплине «Интеллектуальные системы».

В графе «Отель» концептами являются классы объектов, но сами объекты здесь не определены. Имена классов могут использоваться как имена атрибутов (т.е. в качестве имен колонок реляционных таблиц). Для пополнения базы знаний в концептуальном графе необходимо определить отношение «Например» и добавить необходимые факты в список фактов с новым концептом *Пенза*:



```
имеет(гостиница, название).
например(название, 'Пенза').
```

и добавить новое правило языка Пролог:

```
название(гостиница, X):- имеет(гостиница, название),
например(название, X).
```

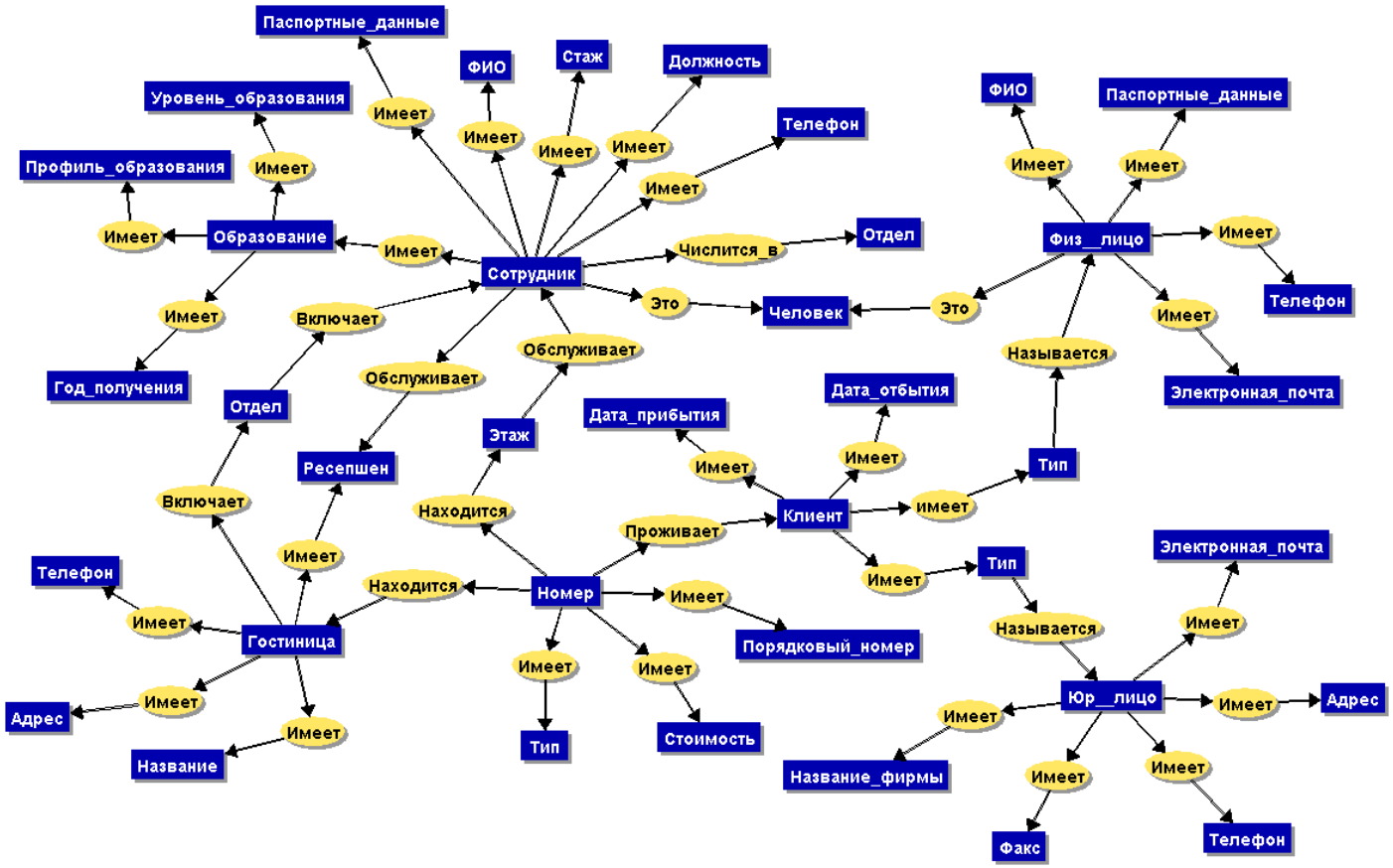


Рис. 11.22. Концептуальный граф для предметной области «Отель»

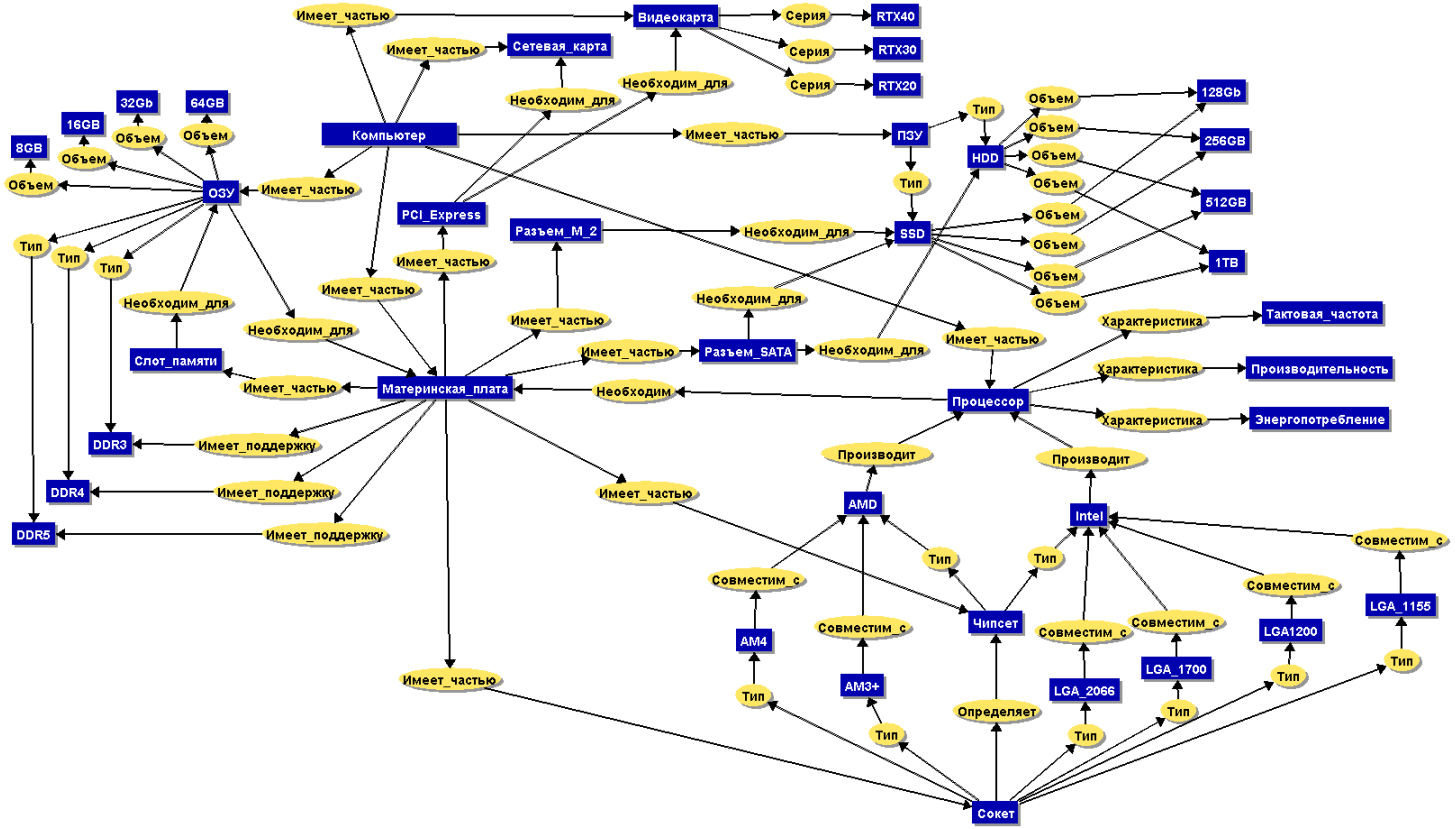


Рис. 11.23. Концептуальный граф на тему «Компьютеры»

Тогда ответом на запрос

? - название(гостиница, X).

будет '*Пенза*'.

Примечание. Точку в конце фактов и правил убирать нельзя, так как это элемент синтаксиса Пролога.

Задание. Дополните рис. 11.22 концептами – объектами соответствующих классов и добавьте в программу на языке Пролог необходимые факты и правила.



Задача «Компьютеры»

Следующий пример «Компьютеры» рассмотрим подробнее, вплоть до программы на Прологе и работы с ней.

В ходе выполнения лабораторной работы был построен концептуальный граф с 40 объектами (концептами) предметной области, которые представлены в виде вершин графа (см. рис. 11.23). Примеры вершин: «*Материнская_плата*», «*Разъем_SATA*», «*DDR5*», «*AM4*», «*Видеокарта*».

Овалами заданы отношения между объектами. Примеры отношений: «*Имеет_частью*», «*Совместим_с*», «*Тип*», «*Производит*», «*Необходим_для*», «*Объем*».

При помощи интеллектуального редактора *CharGer36* преобразуем концептуальный граф в текстовый вид в формате «*a/an второй_концепт – is the отношение – of a/an первый_концепт*» (рис. 11.24).

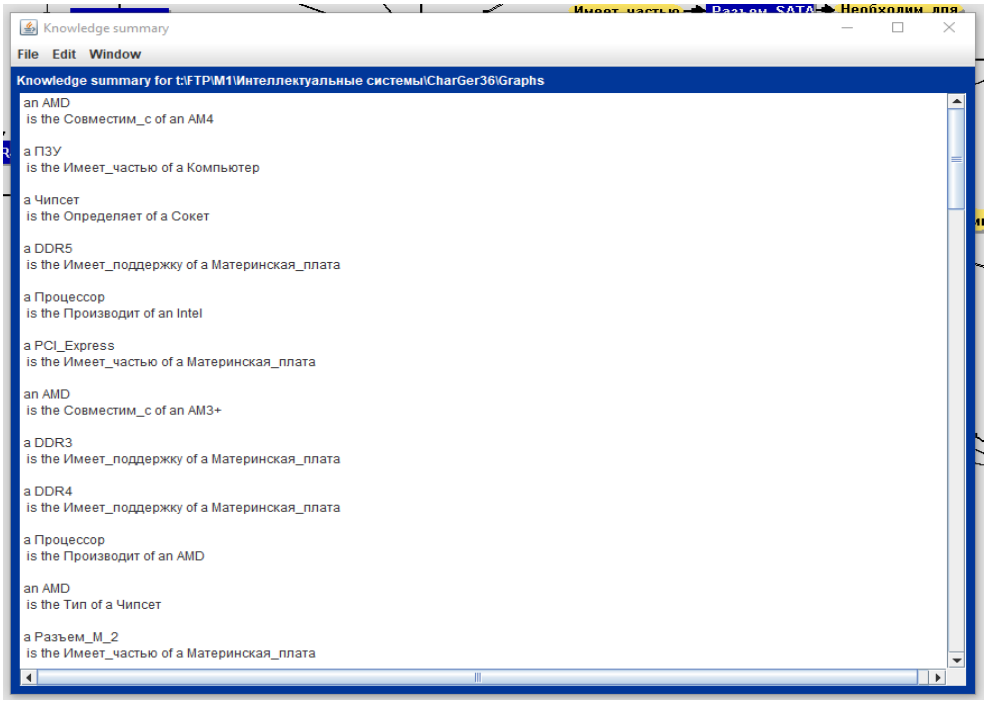


Рис. 11.24. Преобразование графа в текстовый вид

При помощи программы-конвертора ***main4.exe*** преобразуем текстовый вариант концептуального графа в код на Прологе (рис. 11.25).

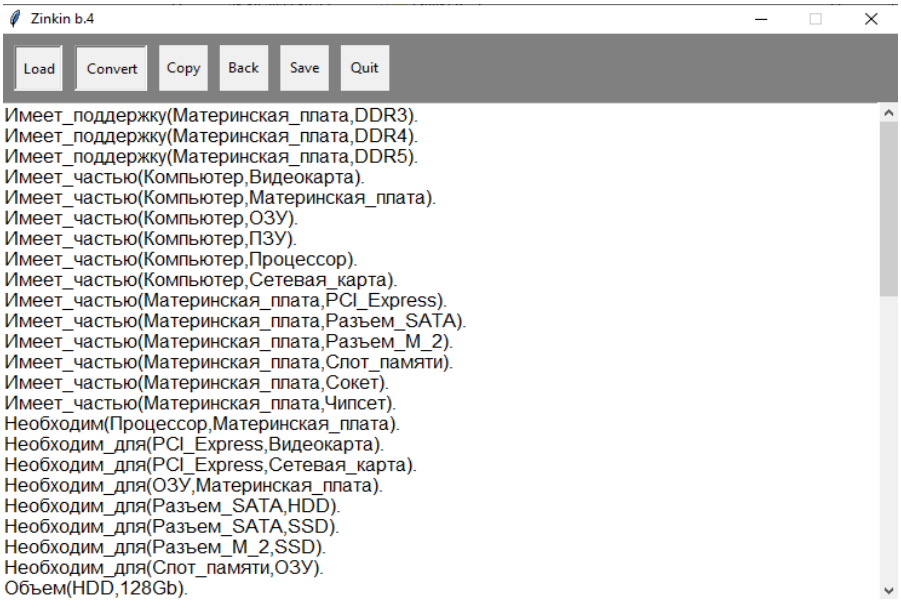


Рис. 11.25. Преобразование текстового варианта концептуального графа в код на Прологе

Далее необходимо отформатировать полученный результат в нижний регистр во избежание проблем запуска программы на языке Пролог.



Представление концептуального графа фактами языка SWI-Prolog:

```
имеет_поддержку(материнская_плата,ddr3).
имеет_поддержку(материнская_плата,ddr4).
имеет_поддержку(материнская_плата,ddr5).
имеет_частью(компьютер,видеокарта).
имеет_частью(компьютер,материнская_плата).
имеет_частью(компьютер,озу).
имеет_частью(компьютер,пзу).
имеет_частью(компьютер,процессор).
имеет_частью(компьютер,сетевая_карта).
имеет_частью(материнская_плата,pci_express).
имеет_частью(материнская_плата,разъем_sata).
имеет_частью(материнская_плата,разъем_м_2).
имеет_частью(материнская_плата,слот_памяти).
имеет_частью(материнская_плата,сокет).
имеет_частью(материнская_плата,чипсет).
необходим(процессор,материнская_плата).
необходим_для(pci_express,видеокарта).
необходим_для(pci_express,сетевая_карта).
необходим_для(озу,материнская_плата).
необходим_для(разъем_sata,hdd).
необходим_для(разъем_sata,ssd).
необходим_для(разъем_м_2,ssd).
необходим_для(слот_памяти,озу).
объем(hdd,gb_128).
объем(hdd,tb_1).
объем(hdd,gb_256).
объем(hdd,gb_512).
объем(ssd,gb_128).
объем(ssd,tb_1).
```

объем(ssd,gb_256).
объем(ssd,gb_512).
объем(озу,gb_16).
объем(озу,gb_32).
объем(озу,gb_64).
объем(озу,gb_8).
определяет(сокет,чипсет).
производит(amd,процессор).
производит(intel,процессор).
серия(видеокарта,rtx20).
серия(видеокарта,rtx30).
серия(видеокарта,rtx40).
совместим_с(am3,amd).
совместим_с(am4,amd).
совместим_с(lga1200,intel).
совместим_с(lga_1155,intel).
совместим_с(lga_1700,intel).
совместим_с(lga_2066,intel).
тип(озу,ddr3).
тип(озу,ddr4).
тип(озу,ddr5).
тип(пзу,hdd).
тип(пзу,ssd).
тип(сокет,am3).
тип(сокет,am4).
тип(сокет,lga1200).
тип(сокет,lga_1155).
тип(сокет,lga_1700).
тип(сокет,lga_2066).
тип(чипсет,amd).
тип(чипсет,intel).
характеристика(процессор,производительность).
характеристика(процессор,тактовая_частота).
характеристика(процессор,энергопотребление).

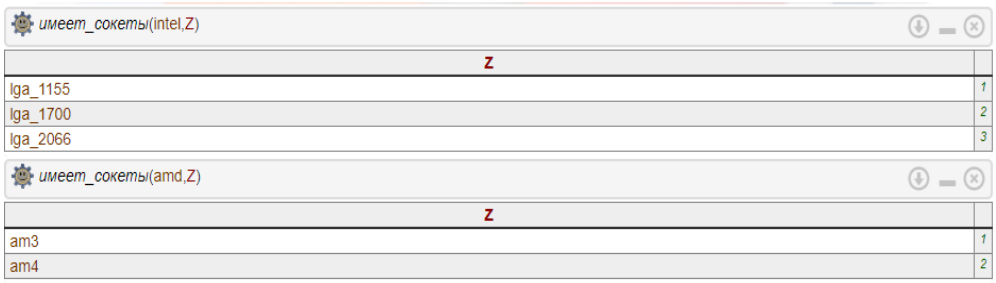
При работе было добавлено несколько правил.



Правила на языке SWI-Prolog:

состоит_из(X,Z) :- имеет_частью(X,Y), состоит_из(Y,Z);
имеет_частью(X,Z).
часть(X,Y) :- имеет_частью(Y,X).
нужен(X,Y) :- необходим_для(Y,X).
является_типом_для(X,Y) :- тип(Y,X).
имеет_сокеты(X,Z) :- тип(чипсет,X), совместим_с(Z,X), тип(сокет,Z).

Было составлено несколько запросов (рис. 11.26–11.29).



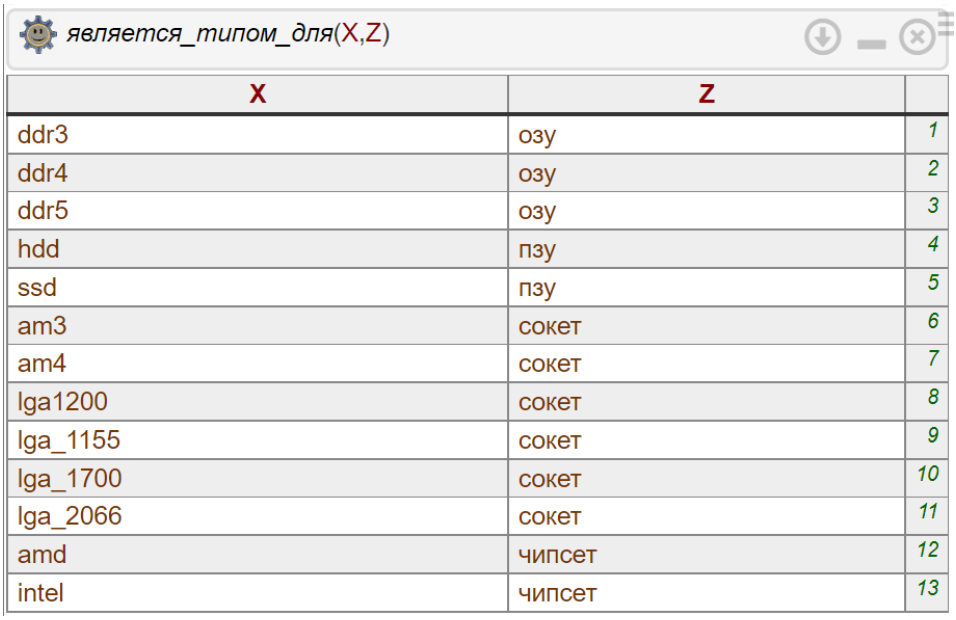
имеет_сокеты(intel,Z)

Z	
lga_1155	1
lga_1700	2
lga_2066	3

имеет_сокеты(amd,Z)

Z	
am3	1
am4	2

Рис. 11.26. Пример исполнения запроса 1



является_типом_для(X,Z)

X	Z	
ddr3	озу	1
ddr4	озу	2
ddr5	озу	3
hdd	пзу	4
ssd	пзу	5
am3	сокет	6
am4	сокет	7
lga1200	сокет	8
lga_1155	сокет	9
lga_1700	сокет	10
lga_2066	сокет	11
amd	чипсет	12
intel	чипсет	13

Рис. 11.27. Пример исполнения запроса 2

содержит_в_себе(компьютер,Z)

Z	
микросхемы	1
pci_express	2
разъем_sata	3
разъем_м_2	4
слот_памяти	5
сокет	6
чипсет	7
видеокарта	8
материнская_плата	9
озу	10
пзу	11
процессор	12
сетевая_карта	13

Рис. 11.28. Пример исполнения запроса 3

часть(X,Z)

X	Z	
видеокарта	компьютер	1
материнская_плата	компьютер	2
озу	компьютер	3
пзу	компьютер	4
процессор	компьютер	5
сетевая_карта	компьютер	6
pci_express	материнская_плата	7
разъем_sata	материнская_плата	8
разъем_м_2	материнская_плата	9
слот_памяти	материнская_плата	10
сокет	материнская_плата	11
микросхемы	сокет	12
чипсет	материнская_плата	13

Рис. 11.29. Пример исполнения запроса 4

Данная программа послужила основой для создания экспертной системы.

Заключение

В результате изучения материала настоящего пособия могут быть получены навыки алгоритмизации задач и процедурного программирования на языке высокого уровня Python. Использовано расположение материала по принципу «от простого к сложному». Даны учебные примеры алгоритмизации и программирования, завершающиеся разработкой реально работающих программ. Детально разобраны примеры решения задач, касающихся экологических проблем города, сетевых грид-приложений, задач из области информатизации различных предметных областей: правового процесса, строительства, распределенного программирования, позволяющие освоить методику перехода от декларативного стиля программирования к процедурному и далее к объектно-ориентированному. При иллюстрации логического подхода к искусственному интеллекту некоторые примеры реализованы на языке SWI Prolog.

Алгоритмический язык высокого уровня Python в числе прочих достоинств имеет широкий набор инструментов для работы с компьютерными сетями. Большая часть Интернета – Всемирная паутина, серверы баз данных и многочисленные сервисы основаны на протоколе TCP, работающем поверх протокола IP, для краткости – TCP/IP. Интернет основан на правилах, по которым создаются соединения, идет обмен данными, закрываются соединения, обрабатывается истечение срока действия и т.д. Эти правила называются протоколами, и они упорядочены в стеке. Протокол TCP поддерживает клиент-серверное соединение с помощью нескольких вызовов сокетов и запоминает IP-адрес клиента.

При работе сетевые приложения на языке Python могут использовать следующие наиболее популярные интернет-службы: доменную систему имен, модули для работы с электронной почтой, веб-службы, средства удаленной обработки. Последние средства создавались первоначально как реализации способов удаленного выполнения программ, но позже они стали полноценной платформой для управления системами, содержащими тысячи серверов.

К достоинствам языка программирования Python можно отнести наличие встроенных библиотек, простоту интеграции, легкость в создании прототипов, открытый код, объектно-ориентированную

парадигму, переносимость, высокую производительность, платформенную независимость и др. Эти характеристики языка имеют большое значение при разработке проектов, связанных с искусственным интеллектом и машинным обучением.

В последнее время ситуация с искусственными нейронными сетями изменилась: появилось большое количества готовых к использованию библиотек для обучения нейронных сетей.

TensorFlow Python – библиотека сквозного машинного обучения Python для выполнения высококачественных численных вычислений. С помощью TensorFlow можно построить глубокие нейронные сети для распознавания образов и рукописного текста и рекуррентные нейронные сети для NLP (англ. *Natural Language Processing* – обработка естественных языков).

Keras Python – одна из основных библиотек Python с открытым исходным кодом, написанная для построения нейронных сетей и проектов машинного обучения. Keras особенно удобна для начинающих разработчиков, которые хотят проектировать и разрабатывать собственные нейронные сети, в том числе сверточные.

Theano Python – это научная математическая библиотека, которая позволяет определять, оптимизировать и вычислять математические выражения, в том числе и в виде многомерных массивов. Она может работать с очень большими нейронными сетями.

PyTorch Python – это полностью готовая к работе, одна из наиболее популярных библиотек машинного обучения Python с примерами, приложениями и вариантами использования. Библиотека PyTorch хорошо адаптирована к графическому процессору (GPU), что позволяет эффективно использовать ее, например, в приложениях NLP (обработка естественных языков).

В дополнительном списке литературы к пособию приведен список книг по тематике, связанной с машинным обучением, с искусственными нейронными сетями и их обучением, использованием языка Python для программирования искусственного интеллекта и киберфизических систем. Эти книги рекомендуются читателям, заинтересованным в дальнейшем освоении технологий прикладного искусственного интеллекта.

Девятая, десятая и одиннадцатые главы посвящены рассмотрению примеров построения концептуальных моделей, основанных на знаниях и правилах. Изложение проиллюстрировано примерами

концептуальных графов для различных предметных областей и программами на языках Python и SWI Prolog. Определены формально концептуальные графы и концептуальные логические сети Петри. Показано, что они могут использоваться при построении декларативных, логических и исполнимых продукционных моделей представления знаний. Рассмотрен пример реализации исполнимых продукционных моделей представления знаний на языке Python. Логические и продукционные модели легко программируются, а реализованные программы могут быть прототипами для построения на их основе различного рода справочных и экспертных систем, применяемых в различных отраслях науки и техники. Реализована и проиллюстрирована примерами методика синтеза концептуальных логических сетей Петри на основе выявления общей семантики концептуальных графов и сетей Петри, в результате чего построены модели, обладающие в совокупности декларативными, императивными и динамическими свойствами. Автоматизировано выделение фактов и правил вывода при последующей программной реализации данного подхода в интеллектуальных событийных системах на примере конкретных предметных областей человеческой деятельности. Подобно Semantic Web, основой для представления интеллектуальной информации о предметной области являются семантическая сеть или концептуальный граф с событиями, пригодные для последующей машинной обработки.

В приложениях 1 и 2 приведены справочные сведения по языкам Python и SWI Prolog.

Авторы настоящего учебного пособия полагают, что самостоятельное изучение его разделов позволит совершенствовать алгоритмические и логические умения и навыки разработки прикладных программ для различных предметных областей.

Для продолжения изучения принципов алгоритмизации и программирования на различных языках авторы рекомендуют литературу, указанную в основных и дополнительном списках. В частности, рекомендуется книга авторов: Хайнеман Дж., Поллис Г., Селков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python. 2-е изд. СПб. : Альфа-книга, 2017. 432 с. Логический подход к искусственному интеллекту детально описан в книге авторов: Рассел С., Норвиг П. Искусственный интеллект: современный подход. Решение проблем: знания и рассуждения. Т. 1. 4-е изд. СПб. : Диалектика, 2021. 704 с.



Список литературы

Список литературы к главам 1–7

1. Ахмад И. 40 алгоритмов, которые должен знать каждый программист на Python. СПб. : Питер, 2023. 368 с.
2. Адитья Бхаргава. Грокаем алгоритмы. СПб. : Питер, 2022. 288 с.
3. Борзунов С. В., Кургалин С. Д. Задачи по дискретной математике с алгоритмами на Python. СПб. : БХВ-Петербург, 2022. 592 с.
4. Бэрри П. Изучаем программирование на Python. М. : Изд-во «Э», 2017. 624 с.
5. Вирт Н. Алгоритмы и структуры данных. М. : ДМК Пресс, 2010. 272 с.
6. Гагарина Л. Г., Колдаев В. Д. Алгоритмы и структуры данных. М. : Финансы и статистика : ИНФРА-М, 2009. 304 с.
7. Грис Д. Наука программирования. М. : Мир, 1984. 416 с.
8. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. М. : Мир, 1981. 386 с.
9. Гэддис Т. Начинаем программировать на Python. 5-е изд. СПб. : БХВ-Петербург, 2022. 880 с.
10. Дейкстра Э. Дисциплина программирования. М. : Мир, 1978. 274 с.
11. Доусон М. Программируем на Python. СПб. : Питер, 2023. 416 с.
12. Дронов В. А., Прохоренок Н. А. Python 3. Самое необходимое. СПб. : БХВ-Петербург, 2018. 608 с.
13. Зингаро Д. Алгоритмы на практике. СПб. : Питер, 2023. 432 с.
14. Златопольский Д. М. ЕГЭ по информатике. Решение задач по программированию. СПб. : БХВ-Петербург, 2013. 304 с.

15. Иванов М. Алгоритмический тренинг. Решения практических задач на Python и C++. СПб. : БХВ-Петербург, 2023. 416 с.

16. Кнут Д. Искусство программирования. М. : Диалектика, 2019. Т. 3. 832 с.

17. Князева М. Д. Алгоритмика: от алгоритма к программе : учеб. пособие. М. : КУДИЦ-ОБРАЗ, 2006. 191 с.

18. Кольцов Д. М., Дубовик Е. В. Справочник Python. Кратко, быстро, под рукой. СПб. : Наука и Техника, 2021. 288 с.

19. Кормен Т. Х. Алгоритмы. Вводный курс. М. : Вильямс, 2019. 208 с.

20. Любанович Б. Простой Python. Современный стиль программирования. СПб. : Питер, 2021. 592 с.

21. Москвитин А. А. Решение задач на компьютерах. Спецификация задачи : учеб. пособие. СПб. : Лань, 2018. 228 с.

22. Мэтиз Э. Изучаем Python. Программирование игр, визуализация данных, веб-приложения. СПб. : Питер, 2017. 496 с.

23. Нилаб Нисчал. Python это просто. Пошаговое руководство по программированию и анализу данных. СПб. : БХВ-Петербург, 2022. 848 с.

24. Пойа Дж. Математическое открытие. Решение задач: основные понятия, изучение и преподавание. М. : Наука, 1976. 448 с.

25. Сазонов Э. В. Экология городской среды : учеб. пособие для вузов. М. : Юрайт, 2023. 275 с.

26. Скиена С. С. Алгоритмы. Руководство по разработке. 3-е изд. СПб. : БХВ-Петербург, 2022. 848 с.

27. Такфилд Б. Алгоритмы неформально. Инструкция для начинающих питонистов. СПб. : Питер, 2022. 272 с.

28. Хайнеман Д. Алгоритмы. С примерами на Python. СПб. : Питер, , 2023. 304 с.

29. Херманс Ф. Ум программиста. Как понять и осмыслить любой код. СПб. : БХВ-Петербург, 2023. 272 с.

30. Кондратьева О. Е. [и др.] Экология : учебник и практикум для вузов / под ред. О. Е. Кондратьевой. М. : Юрайт, 2023. 283 с.

31. Хайнеман Дж., Поллис Г., Селков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python. 2-е изд. СПб. : Альфакнига, 2017. 432 с.

Список литературы к главе 8

1. Васильчиков В. В. Основы разработки сетевых Windows-приложений. Ярославль : Изд-во ЯрГУ, 2007. 212 с.

2. Васильчиков В. В. Разработка сетевых приложений для ОС Windows (практические примеры). Ярославль : Изд-во ЯрГУ, 2009. 216 с.

3. Глейзер Д., Мадхав С. Многопользовательские игры. Разработка сетевых приложений. СПб. : Питер, 2017. 368 с.

4. Зинкин С. А., Джафар М. С. Развитие информационно-коммуникационных инфраструктур распределенных вычислительных систем на основе концепции «Сеть это компьютер» // Известия Юго-Западного государственного университета. 2018. Т. 22, № 4 (79). С. 75–93.

5. Иванов В. Б. Прикладное программирование на C/C++. С нуля до мультимедийных и сетевых приложений. М. : Литрес, 2016. 241 с.

6. Кумар В., Кровчик Э., Лагари Н. [и др.]. .NET. Сетевое программирование. М. : Лори, 2020. 400 с.

7. Митрохин М. А., Карамышева Н. С., Егоров В. Ю., Зинкин С. А. Интеллектуальная обработка информации : учеб. пособие. Пенза : Изд-во ПГУ, 2022. 320 с.

8. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы. СПб. : Питер, 2021. 1008 с.

9. Попов К. В., Карамышева Н. С., Зинкин С. А. Методы и технологии разработки сетевых приложений для распределенных вычислительных систем : учеб.-метод. пособие. Пенза : Изд-во ПГУ, 2022. 182 с.

10. Таненбаум Э., Уэзеролл Д. Компьютерные сети. 5-е изд. М. : ЛитРес, 2019. 961 с.

11. Федоров Д. Ю. Программирование на языке высокого уровня Python : учеб. пособие для прикладного бакалавриата. М. : Юрайт, 2019. 161 с.

12. Чан У. Python: создание приложений. Библиотека профессионала. М. : Вильямс, 2015. 816 с.

Список литературы к главам 9, 10 и 11

1. Тейз А., Грибомон П., Луи Ж. [и др.]. Логический подход к искусственному интеллекту: от классической логики к логическому программированию. М. : Мир, 1990. 429 с.

2. Палагин А. В., Кривой С. Л., Петренко Н. Г. Концептуальные графы и семантические сети в системах обработки естественно-языковой информации // Математические машины и системы. 2009. № 3. С. 67–79.

3. Власенко С. В., Варчев Д. А. Использование аппарата концептуальных графов в задачах представления и обработки знаний : учеб. пособие. СПб. : Изд-во СПбГЭТУ "ЛЭТИ", 2009. 64 с.

4. Sowa J. F. Conceptual Graphs // Handbook of Knowledge Representation / ed. by F. van Harmelen, V. Lifschitz, and B. Porter. Elsevier, 2008. P. 213−237.

5. Sowa J. F. Cognitive Architectures for Conceptual Structures // Proc. 19th International Conference on Conceptual Structures / ed. by S. Andrews, S. Polovina, R. Hill, & B. Akhgar, LNAI 6828. Heidelberg : Springer, 2011. P. 35–49.

6. Polovina S. An Introduction to Conceptual Graphs. Conceptual Structures: Knowledge Architectures for Smart Applications / ed. by U. Priss, S. Polovina, R. Hill // Lecture Notes in Artificial Intelligence (LNAI 4604). Springer. 2007. P.1–15.

7. SWI-Prolog. URL: https://www.swi-prolog.org/ (дата обращения: 31.01.2023).

8. CharGer Manual v3.5b1 2005-11-30. P. 1–58. URL: http://charger.sourceforge.net/ (дата обращения: 31.01.2023).

9. Delugah H. CharGer A Conceptual Graph Editor written by Harry Delugah. URL: http://www.cs.uah.edu/~delugach/CharGer/ (дата обращения: 31.01.2023).

10. Волчихин В. И., Карамышева Н. С., Зинкин С. А., Гурин Е. И. Алгоритмика, логика и моделирование агентно-базированных метакомпьютерных систем с повышенным уровнем параллельности // Известия высших учебных заведений. Поволжский регион. Технические науки. 2022. № 2. С. 5–25. doi: 10.21685/2072-3059-2022-2-1

11. Zinkin S. A., Volchihin V. I., Karamysheva N. S., Jaafar M. S. Synthesis of the Functional Architecture of Agent-based Computing Systems when Using Conceptual Models of Artificial Intelligence // 2021

International Conference on Engineering Management of Communication and Technology, EMCTECH 2021. Proceedings. Austria, Vienna, 2021. P. 1–6. doi: 10.1109/EMCTECH53459.2021.9619174

12. Zinkin S. A., Volchihin V. I., Karamysheva N. S., Jaafar M. S. Dynamic Topology Transformation of Cloud-Network Computer Systems: Conceptual Level // 2020 International Conference on Engineering Management of Communication and Technology, EMCTECH 2020. Proceedings. Austria, Vienna, 2020. P. 1–10. doi: 10.1109/EMCTECH49634.2020.9261554

13. Зинкин С. А, Пащенко Д. В., Пучкова У. Н., Джафар М. С. Интеграция методов концептуального и поведенческого моделирования дискретно-событийных систем: II. Логико-алгебраические операционные модели и инфокоммуникационные технологии // Кибернетика и программирование. 2017. № 1. С. 75–93.

14. Зинкин С. А., Джафар М. С., Карамышева Н. С. Концептуальные представления и модификации сетей Петри для приложений в области синтеза функциональной архитектуры распределенных вычислительных систем с переменной структурой // Известия Юго-Западного государственного университета. 2018. Т. 22, № 6 (81). С. 143–167.

15. Волчихин В. И., Карамышева Н. С., Митрохин М. А., Зинкин С. А. Представление и структурирование знаний в семантико-ориентированной вычислительной среде. I. Интеграция концептуальных графов и логических сетей на основе формализации структурированных ситуаций // Известия высших учебных заведений. Поволжский регион. Технические науки. 2023. № 2. С. 24–51. doi: 10.21685/2072-3059-2023-2-3

16. Timothy John Berners-Lee. Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential. The MIT Press, 2005.

17. Uniform Resource Identifier. URL: https://ru.wikipedia.org/ (дата обращения: 31.01.2023).

18. Денисов М. П. Лабораторные работы по описанию предметных областей с использованием системы Protégé. Казань : Казан. ун-т, 2020. 53 с.

19. Кузнецов И. П., Мацкевич А. Г. Семантико-ориентированные системы на основе баз знаний. М. : МТУСИ, 2007. 173 с.

20. Корешкова Т. Семантический анализ для автоматической обработки естественного языка. URL: https://rdc.grfc.ru/2021/09/semantic_analysis/ (дата обращения: 31.01.2023).

21. Питерсон Дж. Теория сетей Петри и моделирование систем. М. : Мир, 1984. 254 с.

22. Котов В. Е. Сети Петри. М. : Наука, 1984. 160 с.

23. Мараховский В. Б., Розенблюм Л. Я., Яковлев А. В. Моделирование параллельных процессов. Сети Петри. СПб. : Профессиональная литература, 2014. 400 с.

24. Кулагин В. П. Моделирование структур параллельных вычислительных систем на основе сетевых моделей : учеб. пособие. М. : МГИРЭА(ТУ), 2008. 102 с.

25. Petri Nets. Fundamental Models, Verification and Applications / ed. by M. Diaz. John Wiley and Sons, 2009. 613 p.

26. Petri Nets. Theory and Applications / ed. by V. Kordic. I-TECH Education and Publishing. Vienna, Austria, 2008. 544 p.

27. PIPE. Platform Independent Petri Net Editor. URL: https:// sourceforge.net/projects/pipe2/ (дата обращения: 31.01.2023).

28. CPN Tools. A tool for editing, simulating, and analyzing Colored Petri nets. URL: https://cpntools.org/ (дата обращения: 31.01.2023).

29. A World of Conceptual Graphs. URL: http://conceptual-graphs.org/ (дата обращения: 31.01.2023).

30. Мустафа С. Д., Зинкин С. А., Карамышева Н. С. Организация функционирования распределенных вычислительных систем с переменной архитектурой в виде облачного сервиса, формируемого по запросу клиента (концептуальные графы распределенных алгоритмов) // XXI век: итоги прошлого и проблемы настоящего плюс. 2018. Т. 7, № 4 (44). С. 136–146.

31. Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л. Методы символьной мультиобработки. Киев : Наукова думка, 1980. 252 с.

32. Sileno G. Operationalizing Declarative and Procedural Knowledge: a Benchmark on Logic Programming Petri Nets (LPPNs) // International Conference on Logic Programming (ICLP2020) Workshop on Causal Reasoning and Explanation in Logic Programming (CAUSAL2020). P. 1–13.

33. Koichi Kobayashi, Kunihiko Hiraishi. Optimal Control of Asynchronous Boolean Networks Modeled by Petri Nets // Proceedings of the 2nd International Workshop on Biological Processes & Petri Nets (BioPPN2011) . Vol. 724. P. 7–20. URL: http://ceur-ws.org/

34. Akutsu T., Hayashida M., Ching W.-K., Ng M. K. Control of Boolean networks: Hardness results and algorithms for tree structured networks // Journal of Theoretical Biology. 2007. Vol. 244. P. 670–679.

35. Yaqub Omar, Li Lingxi. Modeling and Analysis of Connected Traffic Intersections Based on Modified Binary Petri Nets // Hindawi Publishing Corporation, International Journal of Vehicular Technology. 2013. Vol. 2013. 10 p. Article ID 192516.

36. Iordache M. V., Antsaklis P. J. Supervisory Control of Concurrent Systems. A Petri Net Structural Approach. Boston : Birkhauser, 2006. 281 p.

37. van der Aalst W. M. P. A class of Petri nets for modeling and analyzing business processes // Computing Science Reports 95/26, Eindhoven University of Technology. Eindhoven, 1995. P. 1–25.

38. Du Y., Jiang C. Formal Representation and Analysis of Batch Stock Trading Systems by Logical Petri Net Workflows // Lecture Notes in Computer Science. 2002. Vol. 2495. P. 221–225.

39. Митрохин М. А., Карамышева Н. С., Егоров В. Ю., Зинкин С. А. Интеллектуальная обработка информации : учеб. пособие / Пенза : Изд-во ПГУ, 2023. 320 с.

40. Карамышева Н. С., Зинкин С. А., Митрохин М. А. Методы и модели интеллектуализации вычислительных и киберфизических систем : учеб. пособие. Пенза : Изд-во ПГУ, 2022. 176 с.

41. Graphical Evaluation and Review Technique Simulation. URL: https://cyberpedia.su/9x12977.html (дата обращения: 31.01.2023).

42. Program (Project) Evaluation and Review Technique (PERT). URL: https://ru.wikipedia.org/wiki/PERT (дата обращения: 31.01.2023).

43. Ющенко Е. Л., Цейтлин Г. Е., Грицай В. П., Терзян Т. К. Многоуровневое структурное проектирование программ. Теоретические основы, инструментарий. М. : Финансы и статистика, 1989. 208 с.

44. Назаретов В. М., Ким Д. П. Робототехника и гибкие автоматизированные производства : в 9 кн. Техническая имитация интеллекта / под ред. И. М. Макарова. М. : Высш. шк., 1986. Кн. 6. 144 с.

45. Рассел С., Норвиг П. Искусственный интеллект: современный подход. 4-е изд. Решение проблем: знания и рассуждения. СПб. : Диалектика, 2021. Т. 1. 704 с.

Дополнительный список литературы

1. Вакуленко С. А., Жихарева А. А. Практический курс по нейронным сетям. СПб. : Университет ИТМО, 2018. 71 с.

2. Гафаров Ф. М., Галимянов А. Ф. Искусственные нейронные сети и приложения : учеб. пособие. Казань : Изд-во Казан. ун-та, 2018. 121 с.

3. Гифт Н. Прагматичный искусственный интеллект. Машинное обучение и облачные технологии. СПб. : Питер, 2019. 304 с.

4. Джонс М. Т. Программирование искусственного интеллекта в приложениях. М. : ДМК Пресс, 2011. 312 с.

5. Джоши Пратик. Искусственный интеллект с примерами на Python. СПб. : Диалектика, 2019. 448 с.

6. Митрохин М. А., Карамышева Н. С., Егоров В. Ю., Зинкин С. А. Интеллектуальная обработка информации : учеб. пособие. Пенза : Изд-во ПГУ, 2023. 320 с.

7. Карамышева Н. С., Зинкин С. А., Митрохин М. М. Методы и модели интеллектуализации вычислительных и киберфизических систем : учеб.-метод. пособие. Пенза : Изд-во ПГУ, 2022. 176 с.

8. Макмахан Рао. Знакомство с PyTorch: глубокое обучение при обработке естественного языка. СПб. : Питер, 2020. 256 с.

9. Мишулина О. А., Трофимов А. Г., Щербинина М. В. Лабораторный практикум по курсу «Введение в теорию нейронных сетей» : учеб.-метод. пособие. М. : МИФИ, 2007. 112 с.

10. Пойтнер, Ян: Программируем с PyTorch. Создание приложений глубокого обучения. СПб. : Питер, 2020. 256 с.

11. Рашид Тарик. Создаем нейронную сеть. СПб. : Альфа-книга, 2017. 272 с.

12. Таулли Т. Основы искусственного интеллекта: нетехническое введение. СПб. : БХВ-Петербург, 2021. 288 с.

13. Шакла Нивант. Машинное обучение и TensorFlow. СПб. : Питер, 2019. 336 с.

14. Шолле Ф. Глубокое обучение на Python. СПб. : Питер, 2018. 400 с.

Приложение 1



Справочная информация по языку Python

Создание программного кода

Чтобы запустить программу на языке Python 3.10, необходимо установить используемые библиотеки. Основные библиотеки встроены, однако для некоторых требуется использование команды **pip** командной строки. Следует открыть командную строку и ввести **pip install** <название_библиотеки>, затем в программе перед основным кодом написать **import** <название_библиотеки> (**from** <название_библиотеки> **import** <название_модуля> – для импортирования отдельного модуля).

Python не требует особого оформления программы и наличия синтаксиса, информирующего о начале или завершении программы.

Идентификаторы

Идентификатор – это имя или название, идентификаторы должны быть у всех видов данных, которые используются в программе: переменные, константы, функции, процедуры и другие компоненты программы.

Исключением являются числовые константы. При присвоении идентификатора следует придерживаться следующих правил:

- имя должно начинаться с буквы;
- имя не может содержать пробел и специальные символы: ‘.’, ‘!’, ‘@’, ‘#’, ‘$’, ‘%’;

- имена не должны содержать более 79 символов (Python doc разрешает использование идентификаторов неограниченной длины, однако имя размерностью более 79 символов вызовет ошибку в PEP-8);
- не следует использовать имена, совпадающие с названиями функций, инструкций и методов языка Python, так как при этом прекращается выделение в тексте одноименных ключевых слов языка;
- не рекомендуется ставить '_' в начале или конце идентификатора, это нотация встроенных типов;
- не допускается использование повторяющихся имен на одном уровне области определения;
- регистр значения не имеет.

Комментарии

Для пояснений отдельных конструкций в программном коде используются комментарии. Однострочные комментарии начинаются с символа '#' (решетка). Многострочные комментарии заключаются в тройные кавычки ' " " " ' в начале и конце комментируемого блока. Комментарии можно вставлять в любом месте процедуры. Когда процедура выполняется, интерпретатор игнорирует комментарии. В IDE наиболее распространен CTRL+/ шорткат для быстрого комментирования.

Данные

Данные – информация, представленная в формальном виде, который обеспечивает возможность ее хранения, обработки и передачи. В программном коде используются данные, постоянные (константы) и переменные величины. Для неизменных значений в программе используются ***константы.*** Константы могут быть в виде числового значения или идентификатора константы.

Переменная – параметр, значение которого в процессе выполнения программы может принимать различные значения.

Python относится к языкам с неявной динамической типизацией, по умолчанию переменной присваивается значение **int**, т.е. a = 5 будет эквивалентно a = int(5).

Чтобы узнать тип переменной, следует использовать функцию **type**(<Имя_переменной>).

Типы данных

Название	Тип	Пример
NullType	Отсутствие значения	None
Bool	Логические данные	True/ False
Complex	Комплексное число	1+3,5j
Float	Числа с плавающей точкой	3.876
Int	Целое число неограниченного размера	234567890987654
Str	Строковый тип Заключается в одинарные или двойные кавычки	'Hello world'/"Hello world"
Long	Длинные числа (не рекомендуется в версиях старше Python 2.x)	234567890987654

Объявление данных

Python не требует объявления переменных в начале программы, их можно создать в процессе. Переменной можно задать пустое значение, например int <имя_переменой>, a = ''(для строковых) или a = [] (для списков). Лучше всего сразу тип переменной и/или задавать значение.

Переменные в Python могут быть **локальные** (local), **нелокальные** (nolocal) или **глобальные** (global).

Локальные переменные (local) – переменная, объявленная внутри тела функции или локальной области видимости.

Нелокальная переменная (nolocal) – переменная, объявленная во вложенной функции, область видимости которой не определена.

Глобальная переменная (global) – переменная, доступ которой можно получить как внутри, так и из вне области видимости функции. Переменная должна быть объявлена за пределами функции.

Если внутри функции будет создана переменная с тем же именем, что и глобальная, то локальная будет в приоритете, все операции

будут выполняться над ней. Не рекомендуется создавать локальные и глобальные переменные с одинаковым именем, если того не требует поставленная задача.

Выражения

Выражение – это единица программного кода, которая определяет способ вычисления некоторого значения. Выражения в языке программирования формируются в соответствии с определенными правилами из констант и переменных, компонент массивов, полей записей, функций и знаков операций. Запись в тексте программного кода отличается от обычной математической записи. Операции подразделяются на арифметические, логические, строковые.

Различают унарные и бинарные операции. Унарный знак плюс «+» перед величиной не изменяет ее значения. Унарный знак минус «–» перед величиной изменяет ее знак. К бинарным операциям (участвуют не одна величина, а две) относятся сложение, вычитание, умножение, деление:

Операция	Приоритет	Тип действий
( )	1	Вычисление в круглых скобках
******	2	Возведение в степень
+x, -x, ~x	3	Унарные плюс, минус, битовое отрицание
***, /, //, %**	4	Умножение, деление, остаток
+, –	5	Сложение, вычитание
<<, >>	6	Битовые сдвиги
&	7	Битовое «И»
^	8	Битовое исключающее «ИЛИ»
|	9	Битовое «ИЛИ»
==, !=, >, >=, <, <=, is, is not, in, not in	10	Сравнение, проверка идентичности, проверка вхождения
not	11	Логическое «НЕ»
and	12	Логическое «И»
or	13	Логическое «ИЛИ»

Для выполнения арифметических операций над числовыми переменными разных типов, используется неявная типизация. В случае

сложения, например int и str интерпретатор выдаст ошибку – необходимо использовать явное приведение типов:

<требуемый_тип_данных>(выражение)

Пример: num_sum = num_int + int(num_str)

В выражениях используются круглые скобки для заключения в них тех частей выражения, которые необходимо выполнить в первую очередь. Количество открывающих скобок должно быть равно количеству закрывающих. Вычисление выражений выполняется в определенном порядке, установленном круглыми скобками, и в соответствии с приоритетом операций.

Операции с равным приоритетом обычно выполняют слева направо. Наиболее часто используемые стандартные арифметические функции для работы с числовыми данными приведены ниже в таблице.

Результат операции можно присвоить первому операнду, в таком случае выражение будет выглядеть следующим образом:

<первый_операнд><знак_арифметической_операции>=<второй_операнд>

Для округления результата над переменными типа float используется функция **round(<операнд>,<количество_знаков_после_запятой>)**

Арифметические операции

Операция	Описание
+	Сложение
–	Вычитание
*	Умножение
/	Деление
//	Целочисленное деление двух чисел
%	Остаток от деления двух чисел
**	Возведение в степень

Логические операции

Тип **Boolean.** Память 2 байта. Интервальные значения True/ False

Логические операции

Операция	Описание
a and b	a и b (логическое умножение)
a or b	a или b (Логическое сложение)
not a	Не a (Логическое отрицание)

Операции сравнения

Операция	Описание
==	True, если операнды равны
!=	True, если операнды не равны
>	True, если первый операнд больше второго
<	True, если первый операнд меньше второго
>=	True, если первый операнд больше или равен второму
<=	True, если первый операнд меньше или равен второму

Оператор in

Возвращает True, если выбранный элемент есть в некотором наборе значений, например, в массиве.

Пример: print(num in array_with_num)

Строковые операции

Строковые переменные используются для работы с текстом или с символьными переменными. Строки различают переменной и фиксированной длины. Переменная длина возможна по умолчанию до 2 млрд символов. Если строковой переменной присвоить некоторое значение, то размер переменной изменяется в соответствии с длиной присвоенного значения.

Для строки фиксированной длины – размер строки постоянная величина. Поэтому при присвоении слишком длинного значения, часть строки отсекается в соответствии с объявленным размером. Если присваиваемое значение короче, то объявленная строка дополняется пробелами. При объявлении строки фиксированной длины необходимо указать точное количество символов в строке:

*<имя_строковой_переменной>***.zfill(***<количество_символов_в_строке>*)

Длина строки – переменная или константа целого типа.

Две возможные операции для работы со строками – операция сравнения и операция конкатенации. Операция конкатенации – это объединение (сложение) строк знаком «+».

В одном операторе можно объединить любое количество строковых выражений. Если складываются числовые значения, то получается сумма. При сложении переменных строкового и числового типа результатом будет строка.

Стандартные функции для работы со строками

Функция (аргумент)	**Возвращает / действие**
int(строка)	Преобразование строкового типа в числовые данные целого типа
float(строка)	Преобразование строкового типа в числовые данные числа с плавающей точкой
str(число любого типа)	Преобразование числового выражения в строковый тип
Строка**[начальный индекс: конечный индекс: шаг]**	Извлечение среза
len(строка)	Длина текущей строки
Строка**.split**(символ)	Разбиение строки по разделителю
Строка**.join**(список)	Создание списка из строки с разделителем

Операторы

Оператор присваивания

Оператор присваивания – основной оператор в любом языке программирования. Оператор присваивания заменяет текущее значение переменной новым значением, которое определяется выражением, или определяет выражение, значение которого должно возвращаться функцией.

Операция присваивания позволяет изменять значения свойств формы и размещенных на ней управляющих элементов. Синтаксис оператора присвоения следующий:

<Идентификатор> = <выражение>

Допустимо присваивание значений переменным и функциям любого типа. Исключением являются переменные файлового типа.

Тип выражения и тип переменной (или функции) должны быть совместимы для присваивания.

Стандартные функции

В таблице приведены наиболее часто используемые стандартные математические функции. Для их использования необходимо импортировать библиотеку math.

Import math

Или обращаться напрямую к библиотеке **math**.имя_функции()

Функция (аргумент)	**Возвращает / действие**
fabs(x)	Модуль числа (абсолютное значение)
factorial(x)	Факториал целого числа
ceil(x)	Округление вверх
floor(x)	Округление вниз
exp(x)	e^x
log(x[<основание_логарифма>, base])	Натуральный логарифм или логарифм по основанию первого передаваемого значения
pow(x, y)	X в степени y (приводит аргументы к типу float)
sqrt(x)	Квадратный корень из x
cos(x)	Косинус x в радианах
sin(x)	Синус x в радианах
tan(x)	Тангенс x в радианах
hyp(x, y)	Вычисление гипотенузы
degrees(x)	Конвертер из радиан в градусы
radians(x)	Конвертер из градусов в радианы

Оператор условного перехода

Использование условных операторов и операторов цикла, называемых также управляющими структурами, позволяет написать программу, которая будет «принимать решения» из альтернативно возможных вариантов и повторять операции.

Условные операторы проверяют, имеет ли условие значение **True** или **False**, и в зависимости от результата указывает один или

несколько вариантов возможных решений. Обычно условие является выражением, которое использует оператор сравнения для сравнения одного значения переменной с другим.

Оператор условного перехода выполняет определенные действия в зависимости от значения условия. Синтаксис оператора условного перехода следующий:

Короткая форма ⇨
```
If <условие>:
    <оператор>
```

Дословно:

Если условие истинно, то выполнить <оператор>

Полная форма ⇨
```
If <условие>:
    <оператор1>
Else <условие>:
    <оператор2>
```

Дословно:

Если условие истинно, то выполнить <оператор 1>, иначе выполнять <оператор 2>

Если при проверке условия должны выполняться сразу несколько операторов, то условный оператор записывается в блочной форме:

```
If      <условие>:
        <оператор1>
        <оператор2>
Или
If      <условие>:
        <группа операторов 1>
Else:
        <группа операторов 2>
```

Допускаются многократно вложенные конструкции **If…Else…**, имеющие столько уровней вложения, сколько необходимо. Однако вместо многократно вложенных конструкций **If…Elif…Else…** в некоторых случаях удобнее использовать оператор выбора.

Оператор выбора

Если имеется несколько операторов (групп операторов), которые необходимо исполнить в случае истинности некоторого условия, то конструкция **If…Else…** может быть очень громоздкой. Оператор

выбора **match case** предназначается альтернативой условному оператору и позволяет выполнить несколько вариантов выбора, в этом случае оператор **match case** оценивает выражение только один раз – в начале управляющей структуры, и выполняет выбор ветви из набора условий. Синтаксис оператора выбора следующий:

```
match case <переменная
      или выражение> case
      <A1>:
            <операторы 1>
      case <A2>:
            <операторы 2>
            .....
            case <An>:
                  <операторы n>
            case __:
                  <операторы>
```

Управление переходит к той строке, для которой значение Ai совпадает со значением «переменной или выражения». Если «переменная или выражение» не равно ни одному из значений A1, A2, …, An, то выполняется «иначе». Если при нескольких значениях Ai необходимо выполнить один (одни) и тот же оператор, то список следует создать через логическое или, знак «|».

Каждая конструкция **case** может содержать несколько значений, диапазон значений или комбинацию значений и операторов сравнений. Необязательная конструкция **case_** выполняется, если оператор выбора не находит подходящего значения ни в одном из **case**.

Операторы цикла

Использование циклов позволяет повторно выполнять операторы в теле цикла. Некоторые циклы выполняют повторение, пока условие имеет значение **False**, другие – пока значение условия равно **True**.

Имеются также циклы, которые повторяют набор инструкций определенное число раз. Такие циклы называются циклами с параметром **For**. Циклы **For** используют в качестве счетчика параметр – переменную, значение которой увеличивается или уменьшается при каждом выполнении цикла. Синтаксис записи оператора цикла с параметром следующий:

For *<итератор>* **in range**(*<начальное_значение>, <конечное_значение>, <шаг>*):
<Тело цикла>

Параметр цикла – идентификатор, параметр изменяется в цикле от начального значения до конечного на каждом шаге на величину шага. Если шаг не указан, то по умолчанию он равен 1. Тело цикла это операторы – инструкции, которые необходимо повторить. Если в цикле нет ни одного оператора, то это пустой цикл.

Цикл завершается не в тот момент, когда значение счетчика равняется конечному значению, а когда оно превышает его. Если начальное значение совпадает с конечным значением, то цикл выполняется один раз. Для уменьшения значения счетчика можно использовать отрицательное значение шага. В этом случае конечное значение должно быть меньше начального значения.

Неопределенные циклы

Неопределенные циклы выполняются в зависимости от условия. Если условие существования цикла в конце цикла – условие выхода из цикла, то такой цикл называется **с постусловием**. Цикл может выполняться или нет в зависимости от результата, вычисляемого внутри цикла. Синтаксис записи оператора цикла с постусловием следующий:

While True:
<Тело цикла>
If not <условие**>:**
break

Условие проверяется после того, как операторы, входящие в цикл, будут выполнены хотя бы один раз. Цикл работает до тех пор, *пока условие является ложным*.

Можно передвинуть проверку условия на начало цикла. Такой цикл называется **циклом с предусловием**. Синтаксис записи оператора цикла с предусловием следующий:

While *<Условие>*:
<Тело цикла>

Цикл выполняется до тех пор, пока выражение условия ложно. Для выхода из цикла используется оператор break. При использовании оператора continue – цикл не прекращается, а переходит на

следующую итерацию, пропуская все последующие операции. Цикл while может быть реализован как бесконечный цикл, т.е. будет выполняться до окончания работы программы:

While True:

<Тело цикла>

Python разрешает вложенность циклов.

Ввод данных

Ввод значений с клавиатуры можно осуществить с помощью функции **input()**, при выполнении ввод будет осуществляться в консоли. Синтаксис записи функции вызова диалогового окна следующий:

<имя_переменной> = **input**(*<сообщение_об_ожидании_ввода>*)

Имя переменной и сообщение об ожидании ввода указывать не обязательно.

По умолчанию input возвращает строковое значение, если требуется ввод численного значения, input поэтому необходимо предварительно преобразовать получаемое значение в числовой вид:

A= int(**input**(“Введите значение”))

Оператор Print

Для вывода данных в консоль можно использовать функцию вывода **print**(). Синтаксис записи функции для создания стандартного вывода следующий:

print(<Сообщение_или_переменная>)

Можно осуществлять вывод нескольких сообщений или переменных в одной функции вывода, в таком случае переменные будут перечисляться через запятую. Атрибут sep указывает, какой разделитель будет между значениями, по умолчанию – запятая:

print(«несколько», «сообщений», «для», «вывода», sep=”\n”)

Функция содержит и другие необязательные атрибуты, которые указываются в скобках и перечисляются через запятую после сообщения или переменной, которые требуется вывести:

print(“Hello world”, **end**=”!”)

End – символ, добавляемый в конце вывода, по умолчанию – перенос на новую строку:

print(“{0} – первая переменная, {1} – вторая переменная”.**format**(a,b))

Метод **format** позволяет использовать функцию print как шаблон, в строку-шаблон в фигурных скобках добавляются индексы переменных, которые необходимо вывести. Сами переменные являются аргументами format.

print(“**%d** – числовая переменная, **%s** – строковая переменная”.**%**(a,b)):

Метод format можно заменить знаком процента «**%**», в таком случае можно выводить переменные разного формата. %d – вывод числовых и десятичных значений, %S – строковых.

Массивы

В программном коде при необходимости можно ссылаться как на весь массив, так и на отдельные его элементы. Последовательность может состоять из большого количества элементов, а ссылка на каждый элемент будет состоять только из имени этой последовательности и индекса – номера элемента в этой последовательности. В Python по умолчанию используются списки. Разница между списками и массивами заключается в том, что списки могут хранить значения разных типов данных одновременных, а массивы – только одного. Списки и массивы на Python по умолчанию динамические. Список можно создать с помощью литерала:

<*имя_списка*> = **[]**

Или с помощью встроенной функции **list**:

l**ist**(<элементы_списка>)

Список можно создать пустым, сразу заполнить значениями, наполнять через циклы for или while или генераторы списков:

<имя_списка> =**[c for i in <последовательность_которой_будет_список>]**

Основные функции для работы со списками:

List.append(x) – добавить элемент в конец списка;

List.insert(i,x) – добавить элемент на i-позицию списка;

List.remove(x) – удалить элемент списка по значению;

List.sort(key=<функция>) – сортировка списка на основе функции;

List.reverse() – развернуть список;

List.copy() – создание неглубокой копии списка (объекты нового списка будут ссылаться на оригинал);

List.clear() – очистить список.

Для создания массива необходимо импортировать библиотеку:

Import array
<имя_массива> = **array.array()**

Массивы заполняются так же, как и списки.

Процедуры пользователя

Процедуры, не связанные с объектами и событиями, называются процедурами пользователя или функциями. Пользовательские процедуры можно использовать неоднократно.

Пользователь может задать свою функцию с помощью **def**. Новая функция, принимая значения своих аргументов, возвращает некоторое значение. **Функция пользователя** – это программный блок, который имеет свое уникальное имя – идентификатор, включает группу операторов, обеспечивающих вычисление одного параметра или одного действия, определенного программным кодом. Синтаксис записи для def следующий:

def <*имя_функции*>(<*аргументы_функции*>):
    <программный код функции>
    return <возвращаемые_параметры>

Функция не обязательно должна передавать и возвращать параметры. Функция может возвращать и сложные объекты, например списки, массивы, кортежи и т.д.

Обращение в программном коде приложения к этой подпрограмме может быть следующим:

<*Имя Функции*> (<*список фактических параметров*>)

Фактические параметры должны совпадать с формальными параметрами по типу, количеству и последовательности расположения. Идентификаторы параметров могут быть другими. В качестве

фактических параметров могут быть использованы массивы, арифметические выражения, переменные и константы.

Анонимные функции могут содержать только одно выражение, за счет этого выполняются быстрей. Анонимные функции создаются с помощью ключевого слова **lambda**:

<*имя_функции*> = **lambda** <*аргументы_функции*>: <*код_функции*>

Фреймворки Python

Фреймворки Python – это наборы модулей Python, предоставляющих набор общих функций, которые могут быть использованы в качестве основы для создания приложений любого типа. Фреймворки Python помогают улучшить организацию кода, повысить производительность и сделать процесс разработки программного обеспечения более простым и эффективным. При составлении таблицы использовались многочисленные сайты и блоги, размещенные в Интернете.	
Тип функциональности и название	**Комментарии**
1	2
Микрофреймворки	*Предоставляют только основные компоненты, необходимые для создания какого-либо приложения. Они легко расширяемы, что делает их хорошим выбором для небольших проектов*
Flask	*Используется для быстрого создания простых веб-приложений*
Bottle	*Разработан для создания небольших или средних веб-приложений; совместим с большинством веб-серверов и версий Python*
CherryPy	*Объектно-ориентированная среда (ООП), которая позволяет создавать веб-приложения так же, как и любая другая ООП на Python; широко применяется в производственных средах на самых разных веб-сайтах – от простых функциональных до требующих высокой сложности*
Веб-фреймворки	*Предназначены для обеспечения веб-разработок*
Django	*Полнофункциональный фреймворк, который включает в себя все необходимое для создания полноценного веб-приложения; для освоения Django желательно знать основы объектно-ориентированного программирования на Python, а также элементы функционального программирования*

Продолжение табл.

1	2
FastAPI	*Современный веб-фреймворк, отличающийся высокой производительностью и простотой, поддерживает асинхронное программирование*
Pyramid	*Гибкий и расширяемый веб-фреймворк, который хорошо подходит как для небольших, так и для крупных приложений, включает в свой состав наиболее распространенные функции, необходимые при разработке веб-приложений на Python*
Tornado	*Асинхронный веб-фреймворк с открытым исходным кодом и сетевая библиотека для создания веб-приложений с использованием Python, широко используется в различных приложениях, включая веб-сервисы, аналитику в реальном времени и другие высокоскоростные приложения, оптимизирован для обработки большого количества одновременных соединений, что делает его подходящим для приложений, требующих длительного соединения с каждым пользователем (например, чат-приложения, боты и парсеры)*
Графические интерфейсы пользователя (GUI)	*Предоставляют широкий спектр инструментов и возможностей для создания настольных приложений, включая поддержку событийно-управляемого программирования, виджетов и графики*
PyQt	*Сочетает в себе простоту Python как языка общего назначения и мощность прикладной среды Qt, построенной на C++; обеспечивает интеграцию баз данных*
Tkinter	*Позволяет быстро создавать небольшие программы с графическим интерфейсом пользователя*
Kivy	*Используется при создании кроссплатформенных мобильных и настольных приложений на языке Python, хорошо подходит для создания интерактивных и визуально привлекательных приложений, таких как игры и мультисенсорные приложения*
PySide	*Предназначен для создания многоплатформенных приложений; основан на библиотеке Qt (аналогично PyQt) и предоставляет доступ к широкому спектру ее функциональных возможностей*
PySimpleGUI	*Позволяет создавать простые и удобные графические пользовательские интерфейсы в Python, позволяет легко добавить графический интерфейс к уже работающим скриптам; PySimpleGUI объединяет в себе возможности четырех различных библиотек графического интерфейса пользователя: PySide, Tkinter, wxPython и Remi*

Продолжение табл.

1	2
Системы машинного обучения	*Программные библиотеки Python, которые предоставляют готовые функциональные возможности и инструменты, помогающие разработчикам создавать и развертывать модели машинного обучения, поддерживая такие общие задачи, как предварительная обработка данных, обучение и оценка моделей, а также инструменты для развертывания моделей на производстве*
scikit-learn	*Библиотека машинного обучения, предоставляющая эффективные инструменты для решения общих задач машинного обучения и построенная на базе других научных библиотек, таких как Numpy и SciPy*
TensorFlow	*Платформа с открытым исходным кодом для построения и развертывания моделей машинного обучения, представляющая собой мощную библиотеку для глубокого обучения с широким набором инструментов для построения и обучения нейронных сетей, используется для решения задач обработки изображений и языков, распознавания речи и предсказательной аналитики*
PyTorch	*Наряду с TensorFlow, PyTorch является одним из наиболее используемых инструментов для построения моделей глубокого обучения при решении различных задач, таких как компьютерное зрение, обработка естественного языка и построение генеративных моделей*
Keras	*Высокоуровневый фреймворк глубокого обучения, способный работать «поверх» TensorFlow, Theano и CNTK, содержит модули для быстрого построения моделей машинного обучения*
Научные вычисления	*Фреймворки для научных вычислений часто включают функциональность для линейной алгебры, оптимизации, интерполяции, интегрирования и других распространенных задач в научных вычислениях*
NumPy	*Библиотека для научных вычислений, предназначенная для работы с многомерными массивами и матрицами числовых данных, предоставляет широкий спектр математических функций для работы с этими массивами и является основой для многих других библиотек и фреймворков Python, используемых в науке о данных, включая SciPy, Pandas, scikit-learn и Tensorflow*
SciPy	*Предоставляет реализации алгоритмов и функций, построенных поверх NumPy, помогает выполнять общие научные и инженерные задачи, такие как оптимизация, обработка сигналов, интегрирование, линейная алгебра и многое другое*

Окончание табл.

1	2
Pandas	*Мощная и гибкая библиотека с открытым исходным кодом, используемая для анализа данных на языке Python, предоставляет полезные структуры данных (например, DataFrame) и инструменты анализа данных, которые облегчают работу со структурированными данными*
Matplotlib	*Предоставляет широкий спектр возможностей для создания графиков, диаграмм и других визуализаций*
Фреймворки для тестирования	*Фреймворки, необходимые в вашем рабочем процессе в разработках, ориентированных на тестирование*
Pytest	*Прост в освоении и использовании, доступны более 800 плагинов, позволяет создавать модульные тесты, предоставляет подробную информацию об ошибках*
Unittest	*Встроенная среда тестирования Python, имеет простой и удобный интерфейс для написания и запуска тестов*
nose2	*Предоставляет более мощный и гибкий способ написания и запуска тестов*
Асинхронные фреймворки	*Асинхронные коды позволяют выполнять определенные задачи параллельно, не блокируя выполнение других задач*
asyncio	*Основа для многих других асинхронных фреймворков; позволяет разработчикам писать параллельный код, используя синтаксис async/await и рассчитан на обработку тысяч параллельных соединений; asyncio предоставляет единый API для нескольких транспортных протоколов, таких как TCP, UDP, SSL/TLS, и обеспечивает взаимодействия подпроцессов*
Aiohttp	*Асинхронный клиент-серверный фреймворк HTTP, построенный на основе пакета asyncio, предоставляет простой и интуитивно понятный интерфейс для обработки HTTP-запросов и ответов, а также поддержку промежуточного ПО, сессий и других инструментов веб-разработки*

Приложение 2



Справочная информация по языку Prolog

В настоящем пособии приведен ряд примеров, развивающих логическое мышление и позволяющих овладеть основными навыками построения концептуальных моделей представления знаний в вычислительных и киберфизических системах на основе использования механизмов логического вывода языка SWI-PROLOG.

Рассматриваются способы представления экстенсиональных и интенсиональных знаний о предметной области средствами языка Пролог (PROLOG, реализация интерпретатора SWI-PROLOG).

При выполнении ряда лабораторных работ далее потребуются знания некоторых особенностей языка Пролог. В связи с тем, что декларативные языки программирования, к числу которых относится и Пролог, изучаются, как правило, на младших курсах, потребовалось включение в данное учебно-методическое пособие дополнительных сведений, касающихся интеграции логического программирования и баз данных и использования Пролога при разработке концептуальных моделей и экспертных систем.

Интеграция средств логического программирования и баз данных

В настоящее время уделяется большое внимание элементам интеграции логического программирования и баз данных, которая позволяет образовать системы нового типа, расширяющие границы

информатики как науки и позволяющие удовлетворять требованиям новых применений. Для таких систем используются следующие названия:

а) дедуктивная база данных (отражает использование стиля логического программирования для реализации дедукции на основе содержимого базы данных);

б) система управления базами знаний (отражает управление сложными знаниями, а не простыми данными);

в) экспертная система баз данных (отражает использование опыта экспертов в определенной прикладной области для решения класса задач при сохранении доступа к большой базе данных).

Слияние логических программ и баз данных происходит в соответствии с общей тенденцией в информатике – совместное исследование различных областей с целью генерации новых идей и получения материального выигрыша за счет использования общих концепций.

Логическое программирование и базы данных развивались одновременно. Пролог – самый популярный язык ПРОграммирования в ЛОГике – появился как результат упрощения более общих методов доказательства теорем для достижения возможности эффективного программирования. Аналогично реляционная модель данных возникла в результате упрощения сложных иерархической и сетевой моделей для обеспечения возможности непроцедурного манипулирования множественными данными. Пролог и реляционные базы данных получили широкое распространение не только в академической или научной среде, но и в деловом мире.

О РЕАЛИЗАЦИИ ЯЗЫКА SWI-PROLOG

В настоящем пособии использован SWI-Prolog – открытая реализация языка программирования Prolog, часто используемая для преподавания и приложений Semantic Web. Эта реализация предоставляет богатый набор возможностей, библиотеки для программирования на основе логики, многопоточности, тестирования, графический пользовательский интерфейс GUI, интерфейс к языку программирования Java, интерфейс ODBC, содержит реализацию веб-сервера, библиотеки для SGML, RDF, RDFS (SGML – англ. *Standard Generalized Markup Language* – стандартный обобщенный язык разметки, или метаязык, на котором можно определять язык разметки для документов; RDF – англ. *Resource Description*

Framework – среда описания ресурса; RDFS – англ. *RDF Schema* – схема RDF), средства разработчика, включая интегрированную среду разработки IDE (англ. *Integrated Development Environment*) и обширную документацию.

SWI-Prolog обладает богатым интерфейсом с другими IT-компонентами за счет поддержки многих типов файлов и сетевых протоколов, а также за счет комплексного низкоуровневого интерфейса для языка C, который является основой для высокоуровневых интерфейсов для C++, Java, C#, Python и др. SWI-Prolog работает на платформах Unix, Windows и Macintosh и постоянно развивается. Полные сведения о реализации языка SWI-Prolog приведены на сайтах производителя (https://www.swi-prolog.org/features.html и http://www.swi-prolog.org).

Общие принципы логического программирования и управления базами данных

Изучение логического программирования и управления базами данных позволяет обнаружить следующие общие их черты:

а) *базы данных.* Системы логического программирования управляют небольшими однопользовательскими базами данных, располагаемыми в оперативной памяти и содержащими правила вывода и фактическую информацию. Системы баз данных, напротив, управляют большими, находящимися в коллективном пользовании совокупностями данных в массовой памяти и обеспечивают технологию поддержки эффективного поиска и надежного изменения долговременно хранимых данных;

б) *запросы.* Запрос обозначает процесс, с помощью которого релевантная информация извлекается из базы данных. В логическом программировании запрос (или цель) реализуется построением цепочек логических выводов, комбинирующих правила и фактическую информацию для доказательства истинности или отрицания справедливости первоначального утверждения. В системах баз данных запрос (выражаемый с помощью специализированного языка манипулирования данными) обрабатывается при использовании наиболее эффективных путей доступа в массовой памяти к большим совокупностям данных для извлечения релевантной информации;

в) *ограничения.* Ограничения задают условия правильности баз данных. Проверка ограничения – процесс, с помощью которого обеспечивается правильность базы данных и предотвращается помещение неправильных данных в базу данных. В логическом программировании ограничения выражаются с помощью универсальных правил, активизируемых при модификации базы данных. В системах баз данных лишь небольшое число типовых ограничений обычно выражается на языке определения данных.

Логическое программирование обладает большей выразительной способностью задания запросов и ограничений в сравнении с языками определения данных и манипулирования данными систем баз данных. Более того, представление запросов и ограничений возможно при этом в однородном формализме, и их проверка требует одних и тех же механизмов вывода, что позволяет вести более сложные рассуждения о содержимом базы данных. Вместе с тем системы логического программирования не обеспечивают технологии управления большими, надежными, долговременно хранимыми совокупностями данных с коллективным доступом.

Естественное расширение логического программирования и управления базами данных заключается в построении новых классов систем, расположенных на пересечении этих двух областей. Такие системы основаны на использовании логического программирования в качестве языка запросов и соединяют формулирование запросов и ограничений в стиле логического программирования с технологией баз данных для эффективного и надежного запоминания данных в массовой памяти.

Правила Пролога можно использовать для построения *интенсиональной* базы данных (ИБД) на основе *экстенсиональной* базы данных (ЭБД). ЭБД – это просто реляционная база данных.

Примечание. Пролог возвращает результат по одному кортежу, а не как множество результирующих кортежей (для получения всех кортежей в Прологе необходимо нажимать клавишу пробела).

Пролог представляет собой мощный формализм для выражения запросов и алгоритмов. Если задачам присуща рекурсия, то они не могут быть реализованы с помощью стандартных языков запросов. В дополнительной литературе определены архитектура и механизмы, необходимые для связывания механизма Пролога и системы управления базой данных. Для простоты выбрана реляционная модель

данных, поскольку многие существующие интерфейсы между Прологом и базами данных в настоящее время используют реляционные базы данных. Реляционная модель данных и соответствующий ей язык являются наиболее удобными средствами для выражения свойств этих интерфейсов. Данные методы связывания легко распространить на нереляционные системы. Некоторые существующие прототипы связанных систем разработаны в основном в промышленных и академических центрах.

Некоторые важные свойства Prolog как языка запросов

Ранее были рассмотрены основные причины для интеграции технологии баз данных и логического программирования, установлено соответствие между основными чертами реляционных баз данных и языков логического программирования. Важные свойства Пролога как языка запросов существенно увеличивают выразительную мощность классических языков запросов, основанных на реляционной алгебре или исчислении:

– *интенсиональное определение базы данных.* Правила в языке Пролог являются мощным средством для определения и использования отношений, полученных путем логического вывода. Множество выведенных отношений, определенных с помощью правил Пролога, образуют ИБД;

– *рекурсия.* Предикаты и правила в Прологе могут задаваться рекурсивно. Поэтому выразительная мощность Пролога как языка запросов строго выше выразительных возможностей традиционных языков запросов;

– *неполная информация.* Можно отмечать несущественные значения в правилах и фактах программы с помощью анонимной переменной;

– *упорядоченность правил и фактов в базе.* Программист, использующий Пролог, может упорядочить факты и правила наиболее подходящим образом с точки зрения вычислительной эффективности. Механизм Пролога чувствителен к порядку правил и фактов;

– *управление выводом.* Пролог позволяет программисту увеличить эффективность программ с помощью некоторых процедурных средств, таких как предикат отсечения;

– *отрицания.* Пролог позволяет реализовать отрицания. Например, такие литералы, как *not(has(leo, strips))* являются допустимыми в правой части правила.

Некоторые необходимые для выполнения практических работ сведения о языке PROLOG и о работе в среде интерпретатора SWI-PROLOG

Программа на Прологе состоит из фактов и правил, которые образуют базу знаний Пролог-программы, и запроса к этой базе, который задает цель поиска решений.

Предикат описывает отношение между объектами, которые являются аргументами предиката.

Факты констатируют наличие заданного предикатом отношения между указанными объектами.

Программирование на языке Пролог состоит из следующих этапов:

1) объявления некоторых фактов об объектах и отношениях между ними;

2) определения некоторых правил об объектах и отношениях между ними;

3) формулировки вопросов об объектах и отношениях между ними.

В SWI-PROLOG различаются строчные и прописные буквы.

1. Факты

Предположим, что надо сформулировать на Прологе факт, что «Оператор выполняется на узле_сети». Этот факт включает в себя два объекта «оператор» и «узел_сети», связанные отношением «выполняется». В языке Пролог используется стандартная форма записи фактов:



выполняется(оператор, узел сети).

Важно соблюдать при записи фактов следующие правила:

– имена всех отношений и объектов (предметных констант) должны начинаться со строчной буквы. Например, *выполняется*, *оператор*, *узел_сети*;

– сначала записывается имя отношения, затем в круглых скобках через запятую записываются имена объектов;

– каждый факт должен заканчиваться точкой;

– определяя с помощью фактов отношения между объектами, необходимо учитывать порядок перечисления имен объектов внутри круглых скобок; этот порядок может быть произвольным, но, выбрав один раз определенный порядок, необходимо следовать ему и дальше.

Имена объектов (концептов), список которых в каждом факте заключен в круглые скобки, называют *аргументами*. *Имя отношения*, которое записывается непосредственно перед скобками, называется также *именем предиката*. Строго говоря, отношение является областью истинности одноименного предиката.

2. Запросы

Имея некоторую совокупность фактов, мы можем обращаться к Прологу с вопросами о них. В Прологе вопрос записывается почти так же, как и факт, за исключением того, что перед ним ставится специальный символ « ?- ». Например:



?- выполняется(оператор, узел_сети).

Обращение к Прологу с вопросом инициирует процедуру поиска в базе данных, ранее введенной в систему. Пролог ищет факты, сопоставимые с фактом в запросе. Два факта сопоставимы (или соответствуют один другому), если их предикаты одинаковы (побуквенное совпадение) и их соответствующие аргументы попарно совпадают. Если Пролог находит факт, сопоставимый с вопросом, то он отвечает ***true***. Если в базе данных такого факта не существует, то ***false***.

Все запросы должны заканчиваться точкой. Если вы забудете ее поставить, то Пролог выведет символ '|' и будет ожидать дальнейшего ввода. В этом случае надо ввести точку и нажать клавишу ***Enter***.

3. Переменные

Для того, чтобы обеспечить поиск ответов на более сложные вопросы (например, «Что выполняется на узле_сети?»), в Прологе используются *переменные*. В Прологе можно не только присваивать имена конкретным данным, но и использовать имена, подобные *X* (неизвестное), для обозначения объектов, которые должны быть определены системой. Имена такого типа называются переменными. В Прологе используется соглашение, согласно которому каждое имя, начинающееся с прописной буквы, рассматривается как переменная. При поиске ответа на вопрос Пролог организует просмотр всех фактов в базе данных, чтобы обнаружить объект, который эта переменная могла бы обозначать. Так, при вопросе «выполняется ли на узле_сети *X*?», программа просматривает все известные ему факты для обнаружения тех объектов, которые могут обрабатываться на узле_сети. Такая переменная, как *X*, сама по себе не является именем какого-то конкретного объекта, но она может быть использована для обозначения объектов, для которых пока не определено имя.

Переменная в SWI-PROLOG обозначается как последовательность латинских букв, кириллицы и цифр, начинающаяся с заглавной буквы.

Если значение или имя предметной константы – аргумента предиката – начинается с заглавной буквы, то оно должно заключаться в апострофы:



звание('Александр', майор). % Александр имеет звание майора.

Комментарий в строке программы начинается с символа % и заканчивается концом строки. Блок комментариев выделяется специальными скобками: */* (начало)* и **/ (конец).*

Имена предикатов и объектов (концептов) должны начинаться со строчной буквы и могут содержать латинские буквы, кириллицу, цифры и символ подчеркивания (_). Кириллица используется наравне с латинскими буквами. Обычно предикатам дают такие имена, чтобы они отражали смысл отношения. Два предиката могут иметь одинаковые имена, тогда система распознает их как разные предикаты, если они имеют различное число аргументов (арность). Например, *выполняется/2*, *выполняется/3*.

Имя предиката может совпадать с именем какого-либо встроенного предиката SWI-PROLOG. Однако если совпали имена пользовательского и встроенного предиката, то при обращении к нему (либо из интерпретатора, либо из программы) будет вызван пользовательский предикат, т.е. пользовательское определение «перекроет» предопределенное в SWI-PROLOG.

4. Правила

Правила описывают связи между предикатами. Правило *B:-A* соответствует импликации $A \rightarrow B$ («ЕСЛИ *A* , ТО *B*»).

В общем виде правило – это конструкция вида

$$P0:-P1,P2,\ldots,Pn,$$

которая читается «*P*0 истинно, если *P*1 и *P*2 и ... *Pn* истинны».

Предикат *P*0 называется заголовком правила, выражение *P*1, *P*2, …, *Pn* – телом правила, а предикаты *Pi* – подцелями правила. Запятая означает логическое «И». Точка с запятой означает логическое «ИЛИ».

Факты и правила называются также утверждениями или клозами. Факт можно рассматривать как правило, имеющее заголовок и пустое тело.

5. Процедура – это совокупность утверждений, заголовки которых имеют одинаковый функтор и одну и ту же арность. Процедура задает определение предиката. Конец предложения всегда отмечается точкой, поэтому все факты, правила и запросы должны заканчиваться точкой. Заметим также, что между именем предиката и скобкой не должно быть пробелов.

6. Унификация – это процесс получения свободной переменной значения в результате сопоставления при логическом выводе в SWI-PROLOG.

Набор текста программы и ее выполнение

Для набора текста программы используется встроенный текстовый редактор. Для создания нового файла используется команда **File/New**, в диалоговом окне следует указать имя нового файла. Для

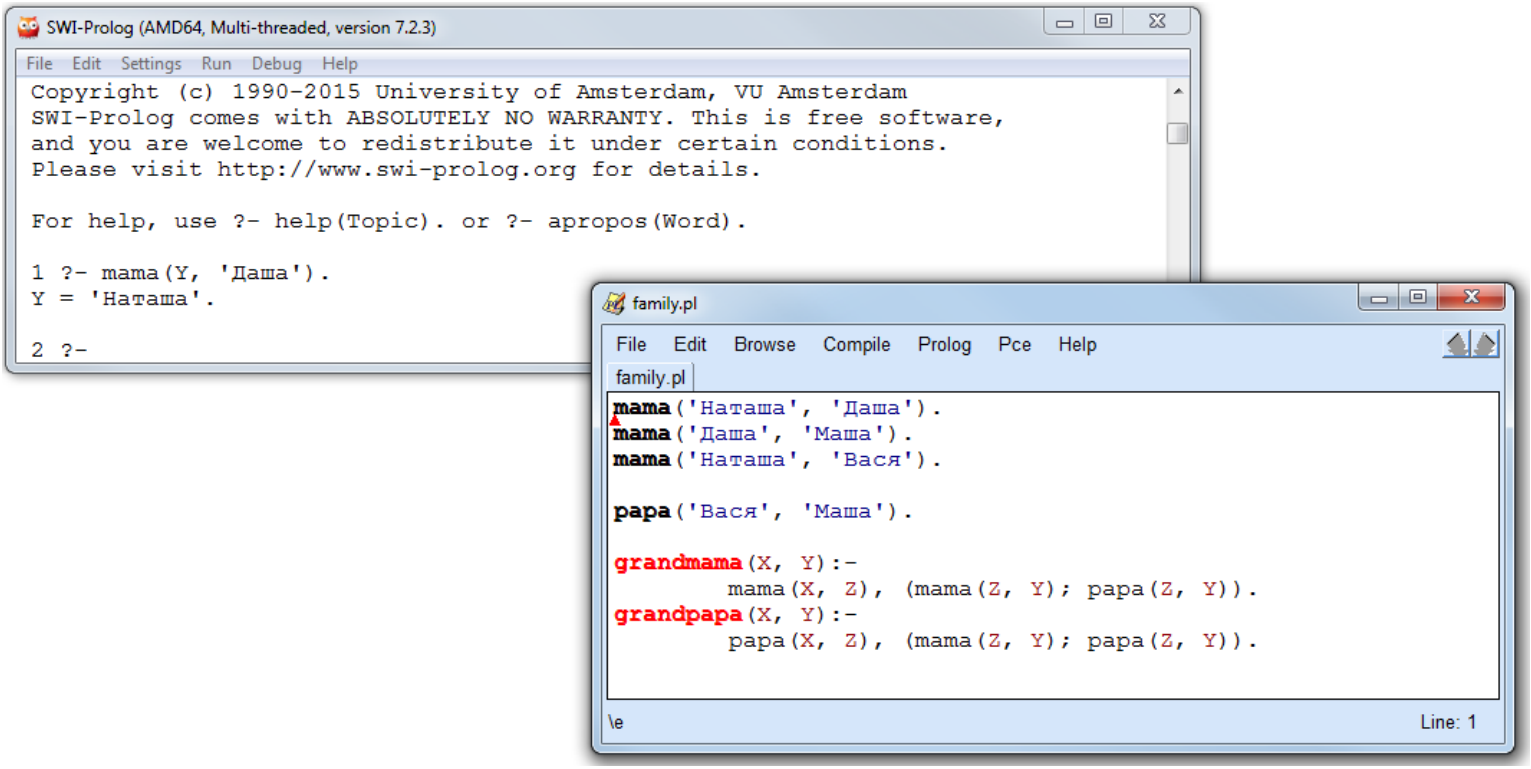
редактирования уже созданного файла с использованием встроенного редактора можно воспользоваться командой меню **File/Edit**. После набора программы следует ее сохранить, выполнив команду **File/Save buffer**.

Красным цветом подсвечиваются предикаты в заголовках предложений, которые с точки зрения синтаксиса SWI-PROLOG корректны. Указатель «*курсор*» можно использовать для выверки (например, корректности) расстановки скобок. Зеленым цветом выделяются комментарии, темно-красным цветом – переменные. Подчеркиванием выделяются предикаты в теле правила, которые совпадают с предикатом заголовка: таким образом акцентируется внимание на возможном зацикливании программы.

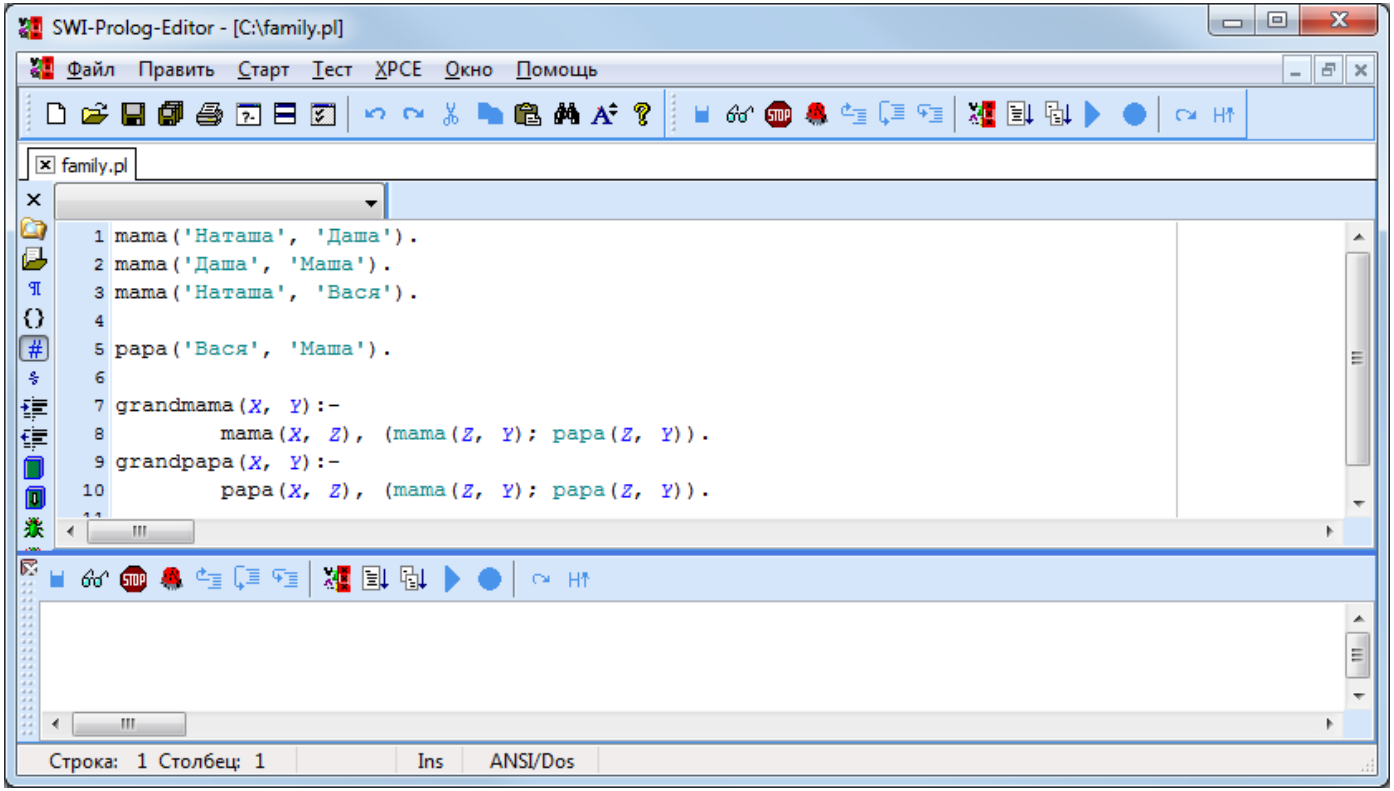
Чтобы запустить программу, сначала необходимо ее загрузить в SWI-PROLOG для выполнения. Это делается выбором опции **Compile/Compile buffer** из окна редактора. Результат отображается в окне интерпретатора SWI-PROLOG. Там же указываются ошибки, возникшие на данном этапе; чаще всего они отображаются и во всплывающем окне ошибок. Обычно перед компиляцией предлагается сохранить файл.

Если необходимо продолжить поиск в базе по этому же запросу и получить альтернативные решения, то вводится точка с запятой «;».

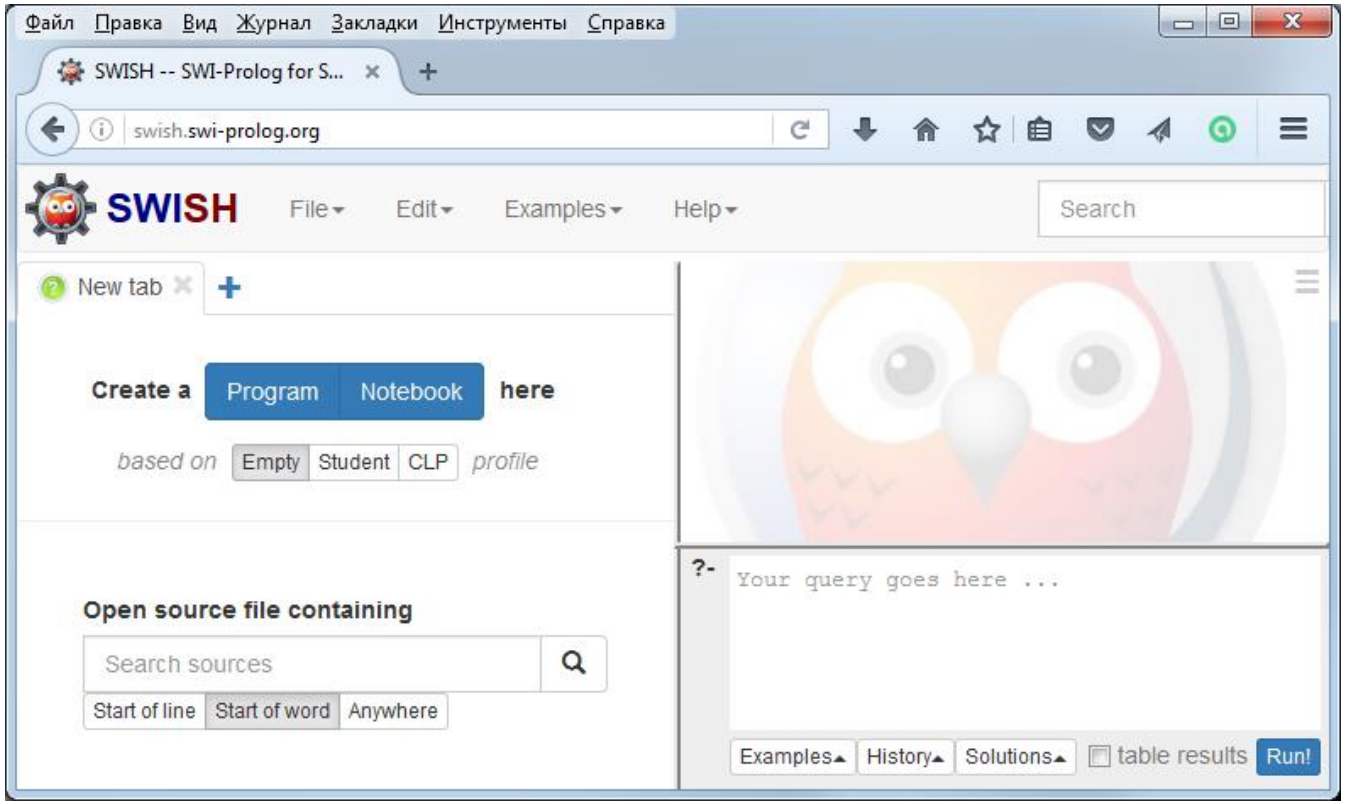
Набор и исполнение программы в среде SWI-PROLOG возможны в разных вариантах [9]:
– при помощи стандартной offline-среды программирования (http://www.swi-prolog.org):



– при помощи offline-среды программирования SWI-Prolog-Editor (http://lakk.bildung.hessen.de/ netzwerk/ faecher/informatik/swiprolog/indexe.html):



– при помощи стандартной online-среды программирования SWI-PROLOG (http://swish.swi-prolog.org):



Приложение 3



Реализация чата в одноранговой сети пиринговых узлов на языке Python

Социальная сеть – это широкое понятие, которое включает в себя практически все виды веб-сайтов и других сетевых сервисов, от деловых до личных. Люди используют эти сервисы, чтобы общаться с другими людьми и делиться общими интересами. Близкие по функциональному назначению системы дистанционного обучения позволяют установить контакт педагогов с обучаемыми и достаточно давно известны преподавателям вузов. Большинство современных социальных сетей и систем дистанционного обучения содержат приложения для обмена мгновенными сообщениями, иногда называемые «социальными мессенджерами», «приложениями для обмена сообщениями», «**приложениями для чата**» или «**чат-клиентами**» с добавлением других функций, таких как эмодзи (или графические смайлики), передача файлов, чат-боты, передача голоса по IP сети или видеочат.



Для справки. *Чат (англ. to chat – «болтать») – средство обмена различной информацией по компьютерной сети в режиме реального времени, а также программное обеспечение, позволяющее организовывать такое общение. Характерной особенностью чата является коммуникация в реальном времени или близкая к этому, что отличает чат от форумов и других «медленных» средств общения. Под словом «чат» обычно понимается групповое общение* (*https://ru.wikipedia.org/wiki/Чат*).

Пример реализации простого чата (P2P Chat) на языке Python

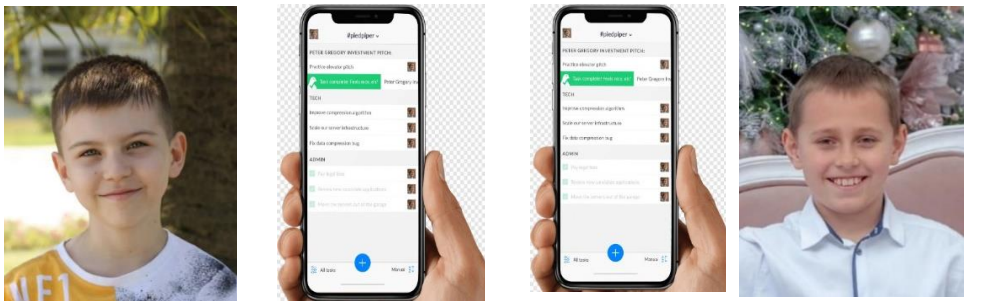
При изучении свойств отношений в концептуальных и логических моделях искусственного интеллекта (глава 11) мы познакомили вас с одноклассниками: Катей, Соней, Егором, Максимом, Глебом, Филиппом и другими школьниками. Современные школьники – народ очень занятый. Но…

Уроки в школе, домашние задания занимают много времени, а ведь еще есть кружки и секции – спортивная и художественная гимнастики, тхэквондо, программирование, английский язык, театральный кружок… Переделав все дела, можно хоть полчаса пообщаться в чате – обсудить текущие события, компьютерные игры, уроки на завтра! Программу-чат можно разработать самостоятельно после внимательного изучения принципов сетевого программирования, рассмотренных в 8-й главе. Авторы настоящего учебного пособия полагают, что справиться с данным заданием под силу студентам первого или второго курсов бакалавриата или специалитета.

Поставим задачу разработки приложения-чата на платформе одноранговой сети, в которой отсутствует центральный сервер, а все участники сети равноправны и взаимодействуют между собой напрямую. Удовлетворяющие данным требованиям одноранговые P2P сети (англ. *peer-to-peer*, что означает «равный к равному»), используются в следующих случаях:



Катя Соня



Егор Максим



Глеб Филипп

– в сетях, где требуется высокая пропускная способность или большое количество одновременных подключений; использование технологии P2P может быть более эффективным, чем использование традиционных серверов;

– P2P технология позволяют создать защищенные соединения, так как информация передается напрямую между пользователями, а не через центральный сервер, который может подвергаться атакам или мониторингу со стороны третьих сторон;

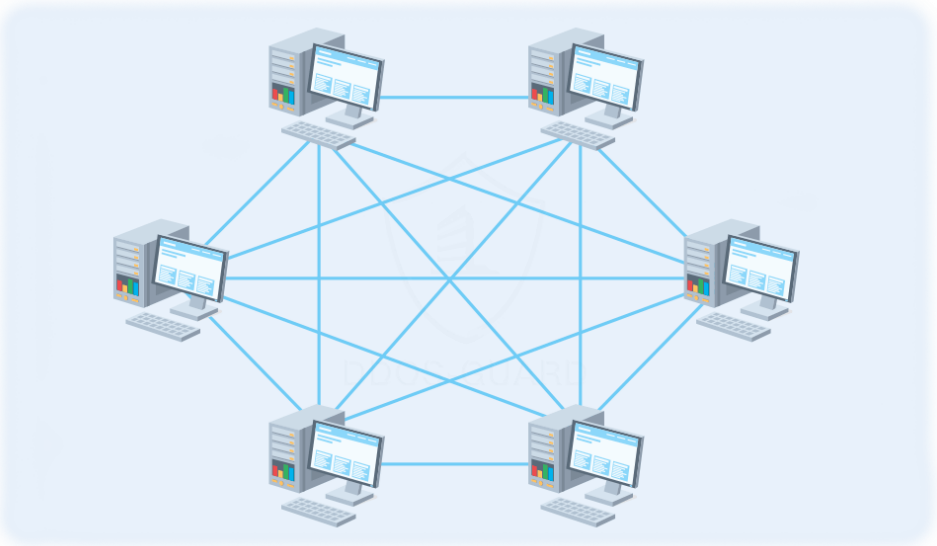
– P2P технология позволяет пользователям контролировать свои данные и определять, как они используются, что может быть важно для обеспечения конфиденциальности и защиты персональных данных.

P2P чат – это такой тип чата, в котором, как говорилось ранее, пользователи могут общаться друг с другом напрямую, без необходимости обращения к центральному серверу. Это позволяет пользователям общаться более эффективно и безопасно, так как их сообщения не проходят через третьи стороны.

В традиционных чатах сообщения проходят через сервер, который может быть подвержен взлому или некорректному мониторингу. В P2P чате сообщения отправляются напрямую от одного пользователя к другому, что делает их более безопасными. Такие чаты могут работать быстрее и использовать меньше ресурсов, что делает их использование особенно эффективным для мобильных устройств и других гаджетов с ограниченными возможностями.

Пользователи могут контролировать участников беседы и блокировать нежелательных пользователей, что обеспечивает дополнительный уровень приватности.

P2P чат работает на основе децентрализованной сети компьютеров или мобильных устройств пользователей, которые выступают в роли узлов:



Реализация P2P чата на языке Python

Для написания P2P чата на Python потребуются следующие библиотеки:

- tkinter: используется для создания и управления графическим пользовательским интерфейсом (GUI) в приложениях Python; это стандартный инструмент для разработки GUI в Python, который предоставляет разнообразные виджеты, такие как кнопки, метки и текстовые поля для интерактивного взаимодействия с пользователем;
- socket: эта библиотека позволяет программам Python взаимодействовать с сетевыми интерфейсами на низком уровне; с ее помощью можно создавать клиентские и серверные приложения, работающие по протоколам TCP и UDP, что делает ее идеальной для разработки сетевых приложений, таких как чаты и серверы данных;
- threading: предоставляет функциональность для создания многопоточных приложений; благодаря многопоточности можно выполнять несколько задач одновременно, что увеличивает эффективность и производительность программ, особенно тех, что взаимодействуют с сетью или выполняют длительные вычисления;
- psycopg2: это адаптер системы управления базой данных (СУБД) PostgreSQL для языка программирования Python; он позволяет Python-приложениям подключаться к базе данных PostgreSQL и взаимодействовать с ней, выполняя операции CRUD (создания, чтения, обновления, удаления данных), управляя транзакциями и обрабатывая сложные запросы.

Функции библиотеки tkinter:

Функция	Описание
Tk	Создает основное окно приложения
Label	Создает текстовую метку в окне
Button	Создает кнопку в окне
Entry	Создает поле для ввода текста

Функции библиотеки socket:

Функция	Описание
socket	Создает новый сокет
bind	Привязывает сокет к адресу
listen	Слушает входящие соединения
accept	Принимает входящее соединение

Функции библиотеки threading:

Функция	Описание
Thread	Создает новый поток
start	Запускает поток
join	Ожидает завершения потока

Функции библиотеки psycopg2:

Функция	Описание
connect	Устанавливает соединение с базой данных
cursor	Создает объект курсора для выполнения операций с базой данных
execute	Выполняет SQL-запрос
commit	Фиксирует текущую транзакцию

Реализация главного окна

Для начала необходимо инициализировать главное окно приложения, где будут располагаться основные элементы интерфейса; для этого используем библиотеку tkinter:

```
window = Tk()
window.title("P2P Chat")
window.geometry("360x620")
window.mainloop()
```

Затем добавим основные элементы интерфейса:

```
window = Tk()
window.title("P2P Chat")
window.geometry("360x620")
```

```
        lbl = Label(window, text="Чат")
        lbl.grid(column=0, row=1, sticky='w')
        btnHost = Button(window, text="Стать хостом чата", com-
mand=host)
        btnHost.grid(column=0, row=0, sticky='w', padx=5, pady=5)
        btnClient = Button(window, text="Подключится к чату", com-
mand=client)
        btnClient.grid(column=1, row=0, sticky='w', padx=0, pady=5)
        txtList = scrolledtext.ScrolledText(window, width=40,
height=30)
        txtList.grid(column=0, row=2, columnspan=2, padx=5, pady=5)
        txtMassage = Entry(window, width=30)
        txtMassage.grid(column=0, row=3, columnspan=2, padx=0,
pady=5)
        lbl = Label(window, text="Введите")
        lbl.grid(column=0, row=3, columnspan=2, padx=5, pady=5,
sticky='w')
        btnClear = Button(window, text="Очистить", com-
mand=lambda: Clear())
        btnClear.grid(column=0, row=4, sticky='w', padx=0, pady=5)

        window.mainloop()
```

При нажатии на элементы управления btnClient и btnHost вызываются определенные функции, указанные в качестве параметров функции Button. В данном случае это функции client и host соответственно.

Функции client и host представляют собой небольшие окна, в которых указываются IP-адрес, порт и имя пользователя. IP-адрес и порт для хоста могут быть реализованы автоматически, а в клиенте должны быть введены данные, указанные в хосте.

После ввода данных в окнах Client или host вызываются функции сохранения данных: saveHost, отвечающая за открытие и прослушивание соединения, или saveClient, в которой реализована логика подключения к соединению.

Пользователь, ставший хостом, позволяет другому пользователю подключиться к чату напрямую, не используя промежуточный сервер. В функциях сохранения данных необходимо реализовать два потока: один для отправки сообщений и второй для получения сообщений – для того, чтобы само приложение смогло принимать и отправлять сообщения.

Работа с базой данных PostgreSQL

PostgreSQL является одной из продвинутых и функционально богатых, открытых систем управления базами данных (СУБД). Вот основные аспекты, которые характеризуют PostgreSQL, являющейся объектно-реляционной СУБД. Это означает, что он поддерживает как традиционные реляционные функции, такие как таблицы и SQL запросы, так и объектно-ориентированные функции, такие как наследование, типы данных и функции. Для работы с PostgreSQL рекомендуется использовать приложение pgAdmin 4 (можно скачать по ссылке https://www.pgadmin.org/download/). Подробное описание можно найти по ссылке:
https://www.pgadmin.org/docs/pgadmin4/latest/index.html.

Работа с PostgreSQL в Python

Рассмотрим основные инструменты работы с PostgreSQL в Python.

1. Установка необходимых пакетов

Для работы с PostgreSQL в Python обычно используется библиотека psycopg2. Установите ее, используя утилиту pip:

```
pip install psycopg2
```

2. Подключение к базе данных

Чтобы подключиться к базе данных PostgreSQL, следует использовать следующий код:

```
import psycopg2
try:
connection = psycopg2.connect(
host="hostname",    # Адрес сервера
database="database_name",    # Имя базы данных
user="username",    # Имя пользователя
password="password"    # Пароль
)
cursor = connection.cursor()
```

```
print("Connected to PostgreSQL")
except Exception as e:
print(f"Error: {e}")
```

В параметрах Host, Database, User, password укажите соответствующие данные к базе данных, созданной в pgAdmin.

После выполнения connection.cursor() создается объект cursor, который позволяет взаимодействовать с базой данных посредством выполнения SQL-запросов и извлечения данных. Cursor играет центральную роль в управлении операциями с базой данных. Вот основные аспекты и функции курсора:

1. Курсор используется для выполнения SQL-запросов к базе данных. Это могут быть запросы на выборку (SELECT), вставку (INSERT), обновление (UPDATE) или удаление (DELETE) данных.

2. После выполнения запроса на выборку курсор позволяет извлекать данные. Это может быть сделано построчно или путем извлечения всех строк сразу.

3. Курсоры часто используются в транзакциях, где несколько операций с базой данных должны быть выполнены как единое целое. Курсор помогает управлять этим процессом, позволяя делать commit или rollback транзакций.

3. Выполнение запросов

Для выполнения SQL-запросов используется метод execute.

Метод execute принимает в качестве параметра SQL запрос к базе:

```
cursor.execute("SELECT * FROM your_table")
rows = cursor.fetchall()
for row in rows:
print(row)
```

После выполнения cursor.execute метод cursor.fetchall() вернет ответ от базы данных, в данном примере в переменную rows запишется результат SQL запроса:

```
SELECT * FROM your_table
```

после чего можно, например, вывести данные в консоль:

```
for row in rows:
    print(row)
```

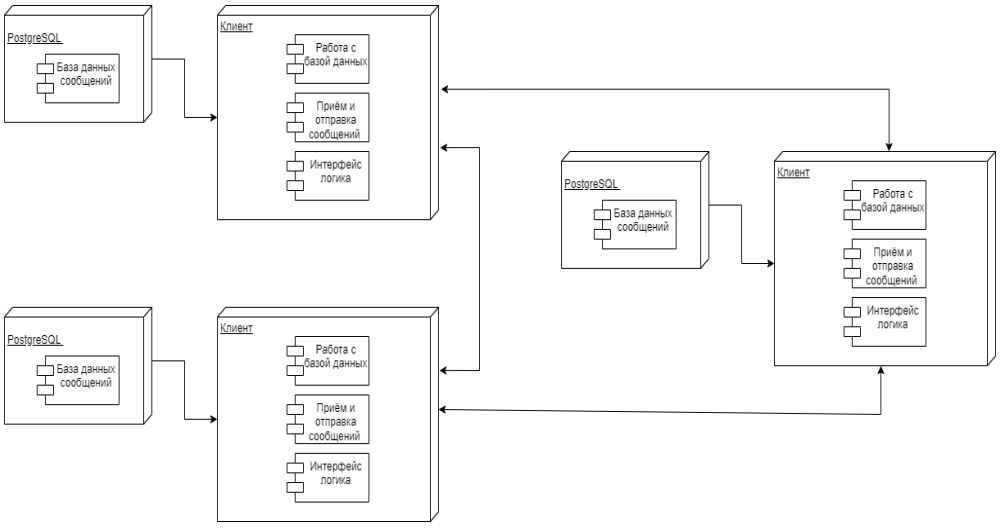
4. Вставка данных

Для вставки данных используются параметризованные запросы:

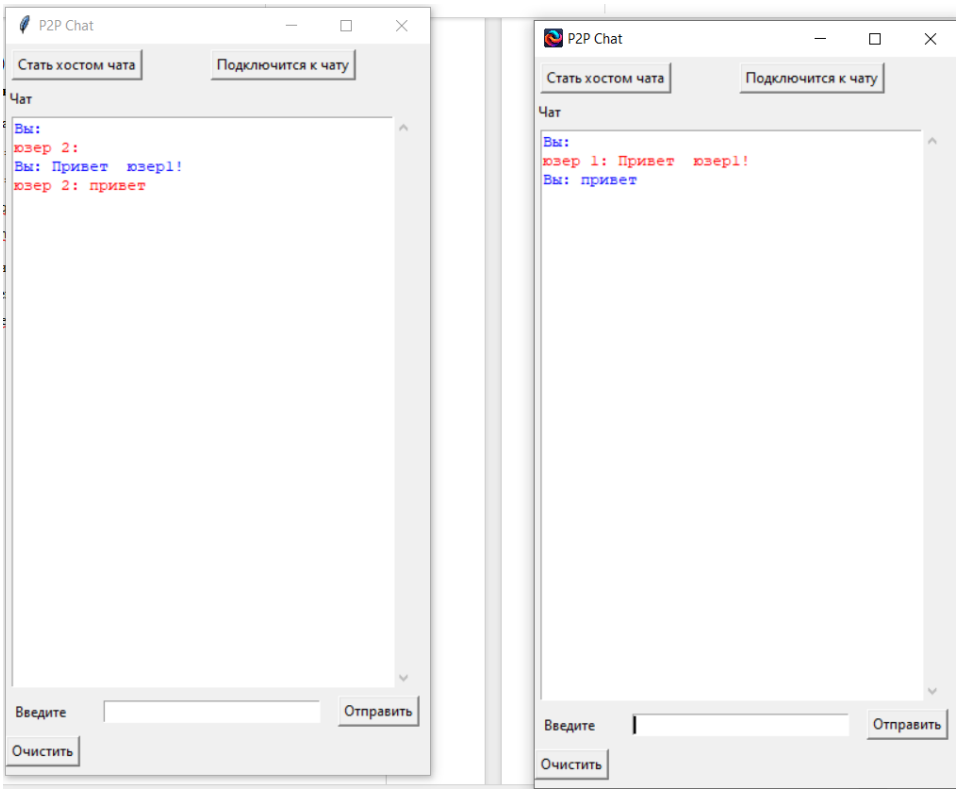
```
sql_insert_query = """ INSERT INTO your_table (id, name) VALUES (%s,%s)"""
record_to_insert = (5, 'SomeName')
cursor.execute(sql_insert_query, record_to_insert)
connection.commit()
```

В данном случает метод execute принимает два параметра, первый (sql_insert_query) SQL-запрос на вставку данных, а второй (record_to_insert) – данные, которые запишутся в базу.

Диаграмма развертывания приложения P2P Chat



Результаты работы программы P2P Chat



Полный текст программы P2P Chat

(составил студент бакалавриата В. С. Александров)

```
from tkinter import *
from tkinter import scrolledtext
import socket
import threading
import psycopg2

db_params = {
        'dbname': 'p2pChat',
        'user': 'postgres',
        'password': 'postgres',
        'host': 'localhost',
}
```

```
# CREATE TABLE messages (
#         id serial PRIMARY KEY,
#         content text NOT NULL,
#         created_at timestamp
# );

def saveHost(txtIp, txtPort, windowHost, txtUserName):
        try:
                # port = int(txtPort.get())
                clientName = txtUserName.get()
                server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
                server_socket.bind(('127.0.0.1', 3333))
                server_socket.listen(5)
                windowHost.destroy()

                def server_thread():
                        client_socket, addr = server_socket.accept()
                        print(f"Подключено к {addr[0]}:{addr[1]}")
                        global sock
                        sock = client_socket

                        send_thread = threading.Thread(target=sendMassage, args=(cli-
ent_socket,clientName))
                        receive_thread = threading.Thread(target=receiveMessage,
args=(client_socket,))

                        send_thread.start()
                        receive_thread.start()

                        send_thread.join()
                        receive_thread.join()

                server_thread = threading.Thread(target=server_thread)
                server_thread.daemon = True
                server_thread.start()
        except NameError:
```

```
            openMessageWindowChatError()
def saveClient(txtIp, txtPort, windowClient, txtUserName):
    try:
            # port = int(txtPort.get())

            clientName = txtUserName.get()

            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client_socket.connect(('127.0.0.1', 3333))
            windowClient.destroy()
            global sock
            sock = client_socket
            # Создаем потоки для отправки и приема сообщений
            send_thread = threading.Thread(target=sendMassage, args=(cli-
ent_socket, clientName))
            receive_thread = threading.Thread(target=receiveMessage, args=(cli-
ent_socket,))

            send_thread.start()
            receive_thread.start()
    except NameError:
            openMessageWindowChatError()
def host():
    windowClient = Tk()
    windowClient.title("Стать хостом")
    windowClient.geometry("290x90")

    lbl = Label(windowClient, text="Имя:")
    lbl.grid(column=0, row=0)

    txtUserName = Entry(windowClient, width=25)
    txtUserName.grid(column=1, row=0)

    lbl = Label(windowClient, text="Ip:")
    lbl.grid(column=0, row=1)

    lbl = Label(windowClient, text="Порт:")
    lbl.grid(column=0, row=2)
```

```
    txtIp1 = Entry(windowClient, width=25)
    txtIp1.grid(column=1, row=1)

    txtPort1 = Entry(windowClient, width=25)
    txtPort1.grid(column=1, row=2)

    btn = Button(windowClient, text="Ок",
                 command=lambda: saveHost(txtIp1, txtPort1, windowClient,
txtUserName), width=20)
    btn.grid(column=1, row=3)

    windowClient.mainloop()

def client():
    windowClient = Tk()
    windowClient.title("Подключение к чату")
    windowClient.geometry("290x90")

    lbl = Label(windowClient, text="Имя:")
    lbl.grid(column=0, row=0)

    txtUserName = Entry(windowClient, width=25)
    txtUserName.grid(column=1, row=0)

    lbl = Label(windowClient, text="Ip:")
    lbl.grid(column=0, row=1)

    lbl = Label(windowClient, text="Порт:")
    lbl.grid(column=0, row=2)

    txtIp1 = Entry(windowClient, width=25)
    txtIp1.grid(column=1, row=1)

    txtPort1 = Entry(windowClient, width=25)
    txtPort1.grid(column=1, row=2)

    btn = Button(windowClient, text="Подключиться",
                 command=lambda: saveClient(txtIp1, txtPort1, windowClient,
txtUserName), width=20)
    btn.grid(column=1, row=3)
```

```
    windowClient.mainloop()

def sendMassage(sock, txtUserNamew):
    btnSend = Button(window, text="Отправить", command=lambda: send-
Massage(sock, txtUserNamew))
    btnSend.grid(column=1, row=3, sticky='e')

    if txtMassage.get() == "/clear":
        clear_messages()
        send_clear_command(sock)
    else:
        massage = txtMassage.get()
        txtList.tag_configure("blue", foreground="blue")
        txtList.insert(INSERT, "Вы: " + massage + "\n", "blue")
        save_message(massage)

        massage = txtUserNamew + ": " + txtMassage.get()

        txtMassage.delete(0, 'end')
        sock.send(massage.encode())

def receiveMessage(sock):
    while True:
        try:
            txtList.tag_configure("red", foreground="red")
            message = sock.recv(1024).decode()
            if message == "/clear":
                txtList.delete(1.0, END)
                clear_messages()
            else:
                save_message(message)
                txtList.insert(INSERT, message + "\n", "red")
        except:
            break

def save_message(content):
    conn = psycopg2.connect(**db_params)
    cursor = conn.cursor()
```

```
        cursor.execute("INSERT INTO messages (content, created_at) VALUES (%s,
NOW())", (content,))
        conn.commit()
        conn.close()


def load_messages():
        conn = psycopg2.connect(**db_params)
        cursor = conn.cursor()
        cursor.execute("SELECT content, TO_CHAR(created_at, 'HH24:MI') FROM
messages ORDER BY created_at")
        messages = cursor.fetchall()
        conn.close()
        return messages

def load_and_display_messages():
        messages = load_messages()
        for message, timestamp in messages:
                txtList.insert(INSERT, f"{timestamp}: {message}\n")

def Clear():
        try:
                txtList.delete(1.0, END)
                clear_messages()
                send_clear_command(sock)
        except NameError:
                openMessageWindowClearError()

def openMessageWindowClearError():
        window = Tk()
        window.title("Ошибка!")
        window.geometry("300x100")

        label = Label(window, text="Вы не подключены к чату! \n Станьте хостом
или подключитесь к чату")
        label.pack()

        ok_button = Button(window, text="OK", command=window.destroy)
        ok_button.pack()
```

```
        window.mainloop()
def openMessageWindowChatError():
        window = Tk()
        window.title("Ошибка!")
        window.geometry("300x100")

        label = Label(window, text="Вы уже подключены к чату! \n Позже будет до-
бавлено переподключение")
        label.pack()

        ok_button = Button(window, text="OK", command=window.destroy)
        ok_button.pack()

        window.mainloop()

def clear_messages():
        conn = psycopg2.connect(**db_params)
        cursor = conn.cursor()
        cursor.execute("DELETE FROM messages")
        conn.commit()
        conn.close()
def send_clear_command(sock):
        clear_command = "/clear"
        sock.send(clear_command.encode())

window = Tk()
window.title("P2P Chat")
window.geometry("360x620")
window.iconbitmap('img.ico')

lbl = Label(window, text="Чат")
lbl.grid(column=0, row=1, sticky='w')

btnHost = Button(window, text="Стать хостом чата", command=host)
btnHost.grid(column=0, row=0, sticky='w', padx=5, pady=5)

btnClient = Button(window, text="Подключится к чату", command=client)
btnClient.grid(column=1, row=0, sticky='w', padx=0, pady=5)

txtList = scrolledtext.ScrolledText(window, width=40, height=30)
```

```
txtList.grid(column=0, row=2, columnspan=2, padx=5, pady=5)

txtMassage = Entry(window, width=30)
txtMassage.grid(column=0, row=3, columnspan=2, padx=0, pady=5)

lbl = Label(window, text="Введите")
lbl.grid(column=0, row=3, columnspan=2, padx=5, pady=5, sticky='w')

btnClear = Button(window, text="Очистить", command=lambda: Clear())
btnClear.grid(column=0, row=4, sticky='w', padx=0, pady=5)

load_and_display_messages()

window.mainloop()
```

Концептуальное представление пиринговых сетей как платформы для организации чатов

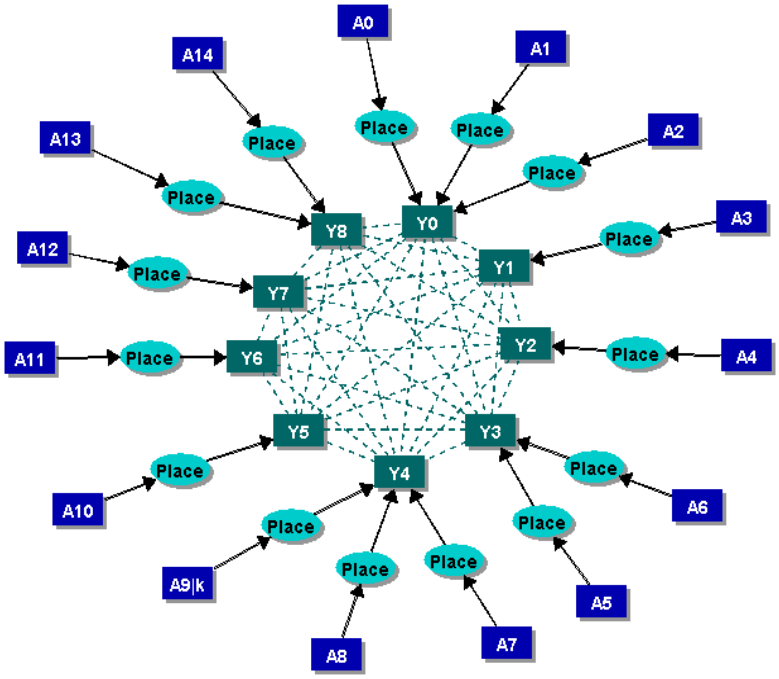
Обычно пиринговые P2P сети используются для обмена файлами. Кроме того, с их помощью можно производить распределенные вычисления, позволяющие задействовать удаленные компьютеры пользователей для выполнения сложной обработки данных. Пиринговая одноранговая сетевая архитектура распределенных вычислительных систем (РВС) отличается большой гибкостью и позволяет реализовывать сложные глобальные вычисления на основе обычных дейкстровских и других известных структур распределенного и параллельного программирования.

P2P-системам органически присуща масштабируемость, что делает их применение особенно эффективным для приложений, которые используют большие данные или с которыми работает большое число пользователей. Поскольку узлы P2P сетей слабо зависят от центрального вычислительного устройства или сервера (если такой вообще существует), появляется возможность повышения уровня масштабируемости сетей и увеличения надежности их работы. Основной задачей становится создание новых алгоритмов обнаружения ресурсов, которые используются в новых P2P платформах.

Для обеспечения *интероперабельности* между функциональной и системной архитектурами распределенной вычислительной системы, с одной стороны, и реализуемым приложением, с другой стороны, в настоящей работе использован подход, основанный на логическом подходе в искусственном интеллекте, а именно, на логике предикатов первого порядка с графическим представлением формул в виде концептуальных графов.

На следующем рисунке представлен концептуальный граф развертывания децентрализованной логической системной архитектуры P2P сети с равноправными участниками.

Концептуальный граф развертывания децентрализованной пиринговой сети с равноправными участниками:



В центре расположены узлы Y_0, Y_1, …, Y_8 P2P сети, равнодоступные друг другу. Пунктирными линиями обозначены соответствующие логические связи. Размещению модулей A_0, A_1, …, A_{14} распределенного приложения соответствует графическое представление высказываний вида $Place(A_i, Y_j)$, где $Place$ – бинарный предикатный символ, A_i – предметная константа из множества имен модулей реализуемого на сети приложения (первый концепт), Y_j – предметная

константа из множества имен узлов сети (второй концепт). Бинарный предикат *Place* для данного примера определяется следующим образом:

$$Place: A \times Y \rightarrow \{\textbf{true}, \textbf{false}\},$$

где $A = \{A_0, A_1,\ldots, A_{14}\}$, $Y = \{Y_0, Y_1,\ldots, Y_8\}$. Одноименное предикату $Place(a, y)$, где $a \in A$, $b \in B$, бинарное отношение носит функциональный характер.

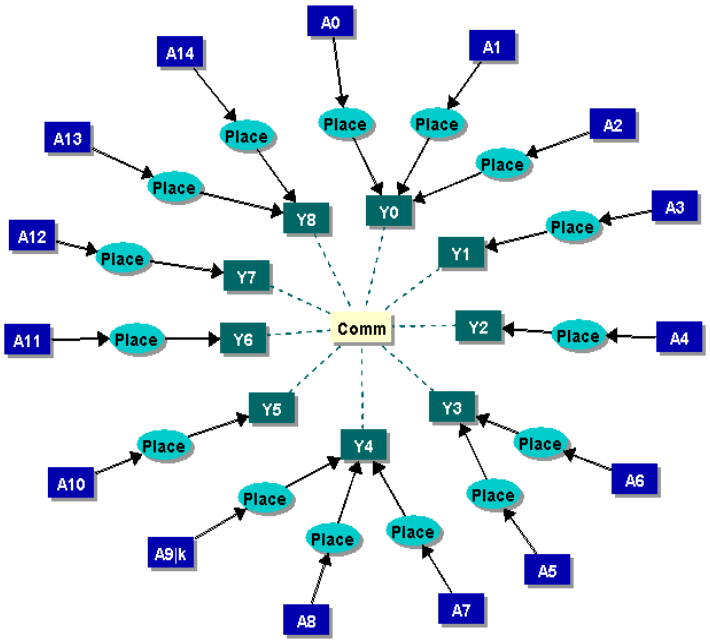
Внутренним логическим связям в РВС соответствует тривиальный бинарный предикат $Link(y, z)$, описывающий полный орграф с петлями:

$$Link: Y \times Z \rightarrow \{\textbf{true}\},$$

где Y – множество источников сообщений и данных; Z – множество приемников сообщений и данных; для децентрализованной P2P сети $Y = Z$, т.е. каждый узел в P2P сети может быть как передающим, так и принимающим сообщения и данные.

Другой вариант реализации логической системной архитектуры централизованной P2P сети представлен на следующем рисунке.

Концептуальный граф централизованной пиринговой сети с равноправными участниками и выделенным узлом для временного хранения передаваемых сообщений и данных:



Здесь все передачи осуществляются через центральный узел *Comm*, который в общем случае не является сервером, а служит звеном передачи и контроля информации в виде сообщений, передаваемых между равноправными узлами. В простейшем случае системная архитектура подобной описывается следующим тернарным предикатом:

$$LinkC\colon Y \times \{Comm\} \times Z \to \{\mathbf{true}\},$$

где $Y = Z$, т.е. все связи реализуются через центральный компьютер-коммутатор *Comm*, который в приведенном выражении обозначен одноименной предметной константой.

Содержание

Предисловие.....3

ЧАСТЬ I. АЛГОРИТМИЧЕСКОЕ МЫШЛЕНИЕ И ПРОГРАММИРОВАНИЕ

Глава 1.....10

Основные принципы алгоритмизации.....10
Алгоритм и его свойства.....12
Постановка задачи.....14
Построение модели.....16
Разработка алгоритма.....17
Величины в алгоритмах.....20
Способы записи алгоритмов.....21
Структуры алгоритмов.....26
Переменные с индексом. Массивы.....37
Исполнение (тестирование) алгоритма.....42

Глава 2.....46

Реализация алгоритмов.....46
От простых алгоритмов к сложным….....50
Рекурсивные алгоритмы.....62
Примеры циклических алгоритмов.....67

Глава 3.....85

Алгоритмы поиска данных.....85
Поиск номера элемента последовательности с заданным значением.....86
Алгоритм поиска в упорядоченной последовательности.....89
Поиск минимального и максимального элемента в заданной последовательности.....90
Примеры алгоритмов – поиск данных.....93

Глава 4.....118

Сортировка данных.....118
Простой выбор.....121
Простой обмен.....124
Простые вставки.....128
Метод Шелла.....132

Глава 5.....**139**

Численное решение уравнений.....139
Метод половинного деления (или дихотомии).....140
Метод простой итерации.....143
Метод Ньютона.....147
Метод хорд.....150

Г л а в а 6 ... 156

Алгоритмы для численного интегрирования и решения дифференциальных уравнений ... 156
Метод прямоугольников ... 157
Метод трапеций ... 161
Метод Симпсона ... 165
Решение обыкновенных дифференциальных уравнений ... 169
Метод Эйлера ... 170

Г л а в а 7 ... 177

Экологические проблемы города ... 177
Экология питания ... 178
Освещение ... 179
Экология литосферы ... 182
Оценка степени экологической безопасности ... 184
Определение ущерба от загрязнения атмосферы ... 186
Плата за загрязнение атмосферы от передвижных источников ... 188

Г л а в а 8 ... 190

Сетевой суперкомьютер –это просто! ... 190
Грид-технологии и вычисления ... 190
Реализации крупномасштабных вычислительных проектов ... 193
Глобальные вычислительные проекты, использующие архитектуру “Master-Worker” ... 194
Прототипная архитектура «Клиент-сервер» для создания грид-приложения: некоторые понятия и определения ... 195
Создание клиентского приложения ... 199
Создание серверного приложения ... 201
Ход выполнения работы над созданием грид-приложения с архитектурой “Master-Worker” (“Мастер-Исполнители”) на языке Python ... 202
Разработка программы Master (Мастер) ... 204
Разработка программы Worker (Исполнитель) ... 206

ЧАСТЬ II
СЕТЕВЫЕ И ЛОГИЧЕСКИЕ МОДЕЛИ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА, ОСНОВАННЫЕ НА ЗНАНИЯХ И ПРАВИЛАХ

Г л а в а 9 ... 214

Программирование интеллектуальных систем с событиями ... 214
Декларативный и процедурный (императивный) подходы в программировании ... 214
Логическое, сетевое и событийное представление знаний о семантике предметной ... 224
Обоснование интеграции концептуальных графов и сетей Петри ... 226
Определение модифицируемых концептуальных графов ... 229
Определение процедурной составляющей в модели представления знаний на основе концептуального графа ... 232
Определение сети Петри ... 235
Графические концептуальные и логические представления сетей Петри ... 237

Определение концептуальной логической сети Петри
с переменной структурой 244
Акторы-переходы в концептуальных логических сетях Петри 245
Методика использования акторов-переходов в концептуальных
логических сетях Петри 246
Построение предметно интерпретируемой концептуальной
логической сети Петри для правового процесса 251
Определение концептуальной логической сети Петри для конкретной
предметной области 254
Выражения для правил модификации предиката разметки позиций
в логической модели 255
Продукционная модель представления знаний о прохождении искового
заявления 258
Развитие концептуальной модели правового процесса 263
Концептуальный граф для процесса продвижения искового заявления 267
О сложности семантического анализа фраз естественного языка
(По мотивам записок Артура Конан Дойла о Шерлоке Холмсе) 269
Построение концептуальных графов на тему «Кража в отеле»:
объектный и событийный подходы 270
Построение предметно интерпретируемой концептуальной логической
сети Петри, моделирующей строительство промышленного объекта 273
Построение концептуальной логической сети Петри, моделирующей
приложение для глобальной компьютерной сети 276

Г л а в а 10 284

*Концептуальные интеллектуальные модели роботизированного
производства* 284
Автоматизация и роботизация производства 284
Представление и структурирование данных и знаний о роботизированном
производстве 287
Ключевая фраза для представления данных и знаний о работе
роботизированного производства 288
Концептуальный граф для модели роботизированного производства и его
контекстное представление 289
Переход от концептуальной сценарной модели роботизированного
производства к имитационной модели 295
Интеграция концептуальных и поведенческих моделей
дискретно-событийных систем: концептуальные сети Петри 298
Модель беспроводной сети MANET, управляющей работой участка
роботизированного производства 300
О реализации вычислительной и коммуникационной компонент
роботизированной системы 302

Г л а в а 11 307

*Концептуальные модели искусственного интеллекта, основанные на знаниях
и правилах* 307
Примеры решения задач 307
Виды отношений в семантических сетях 320
Задача «Глеб и его друзья» на определение свойств отношений 321
Задача «Школьники» 324
Задача «Диагностика гриппа» 326
Задача «Диагностика и лечение пневмонии» 333

Задача «Отель» ... 336
Задача «Компьютеры» ... 339

З а к л ю ч е н и е ... 345

С п и с о к л и т е р а т у р ы ... 348
Список литературы к главам 1–7 ... 348
Список литературы к главе 8 ... 350
Список литературы к главам 9, 10 и 11 ... 351
Дополнительный список литературы ... 354

П р и л о ж е н и е 1 ... 356

Справочная информация по языку Python ... 356
Создание программного кода ... 356
Идентификаторы ... 356
Комментарии ... 357
Данные ... 357
Операторы ... 362
Массивы ... 368

П р и л о ж е н и е 2 ... 374

Справочная информация по языку Prolog ... 374
Интеграция средств логического программирования и баз данных ... 374
О реализации языка SWI-Prolog ... 375
Общие принципы логического программирования и управления базами данных ... 376
Некоторые важные свойства Prolog как языка запросов ... 378
Некоторые необходимые для выполнения практических работ сведения о языке PROLOG и о работе в среде интерпретатора SWI-PROLOG ... 379
Набор текста программы и ее выполнение ... 382

П р и л о ж е н и е 3 ... 388

Реализация чата в одноранговой сети пиринговых узлов на языке Python ... 388
Пример реализации простого чата (P2P Chat) на языке Python ... 389
Концептуальное представление пиринговых сетей как платформы для организации чатов ... 405

Учебное издание

Князева Марина Даниловна,
Карамышева Надежда Сергеевна,
Григорьева Дарья Дмитриевна

ОТ АЛГОРИТМА К ПРОГРАММЕ И ИСКУССТВЕННОМУ ИНТЕЛЛЕКТУ

С примерами на языках Python и Prolog

Под редакцией
доктора технических наук, профессора
Зинкина Сергея Александровича

Редактор *В. В. Чувашова*
Технический редактор *Р. Б. Бердникова*
Компьютерная верстка *Р. Б. Бердниковой*
Дизайн обложки *И. В. Шваревой*

Подписано в печать 30.10.2023. Формат 70×100$^{1}/_{16}$.
Усл. печ. л. 33,48.
Заказ № 557. Тираж 65.

Издательство ПГУ.
440026, г. Пенза, ул. Красная, 40.
Тел.: (8412) 66-60-49, 66-67-77; e-mail: iic@pnzgu.ru