Федеральное агентство по образованию
Нижегородский государственный университет им. Н.И. Лобачевского

Национальный проект «Образование»
Инновационная образовательная программа ННГУ
Образовательно-научный центр «Информационно-телекоммуникационные системы:
физические основы и математическое обеспечение»

Н.Ю. Золотых

MATLAB
Алгоритмы и структуры данных

Учебно-методические материалы по программе повышения квалификации
«Информационные технологии и компьютерная математика»

Нижний Новгород
2007

Учебно-методические материалы подготовлены в рамках инновационной образовательной программы ННГУ: *Образовательно-научный центр «Информационно-телекоммуникационные системы: физические основы и математическое обеспечение»*

Золотых Н.Ю. MATLAB. Алгоритмы и структуры данных. Учебно-методический материал по программе повышения квалификации «Информационные технологии и компьютерная математика». Нижний Новгород, 2007, 128 с.

Пособие продолжает изложение, начатое в учебнике [4], посвященном в основном численным методам в MATLAB'е. Здесь рассматриваются возможности использования системы MATLAB в дискретной математике. Излагаются основные типы данных, организация программ, рекурсивные функции, динамические структуры данных (списки, стеки, очереди, сбалансированные поисковые деревья, разделенные множества, кучи, графы).

Для преподавателей, научных работников, аспирантов и студентов.

Оглавление

1. Основные типы данных. Управляющие конструкции. Программы

1.1. Типы данных

Абстрактным типом данных (АТД) в программировании называют некоторое заданное множество элементов, которые могут иметь произвольную природу, вместе с операциями над элементами этого множества. Часто встречаются математические типы данных, когда рассматриваемые элементы — математические объекты, например, целые числа, вещественные числа, векторы и матрицы с вещественными элементами и т.п. В этом случае операции над элементами — это операции, заданные на множестве этих объектов (для приведенных примеров — операции над числами, векторами, матрицами).

При программировании на компьютере возникает задача представления АТД в адресуемой памяти. Существуют типы данных, для которых эта задача находит естественное решение. Например, очевиден способ хранить в памяти 32-разрядной машины целые числа из диапазона $-2^{31} \leq \alpha < 2^{31}$. Также естественно записывать элементы вектора фиксированного размера в последовательные ячейки памяти. Для более изощренных случаев (т.е. более сложных АТД) необходимо разрабатывать более изощренные способы представления объектов в памяти компьютера. Например, если векторы предполагают вставку и исключение своих компонент (это уже новый АТД, так как появились новые операции: вставки и удаления компонент), то записывать компоненты вектора в последовательные ячейки памяти не целесообразно, так как при каждой вставки/удалении придется перезаписывать большие части вектора. Под *структурой данных* (или *структурой хранения*) понимают тот или иной способ представления АТД в адресуемой памяти компьютера.

Абстрактные типы данных в различных языках программирования могут быть представлены «*конкретными*» *типами данных*, например: *int* в C, *set* в Паскале или (более сложный тип данных) ассоциативный массив (хэш) на Perl'e, Python'e и др. При этом один и тот же АТД в языке может иметь несколько воплощений. Так, в C целые числа представлены типами данных *int*, *short int*, *long int* и др. В объектно-ориентированных языках программирования типы данных представляются *классами*. При этом у программиста есть возможность конструировать новые классы. Конкретные представители класса называются его *экземплярами* или *объектами*.

Итак, «конкретные» типы данных — это воплощение абстрактных типов данных, использующих тот или иной способ представления объектов в памяти компьютера с помощью некоторой структуры данных.

На рис. 1.1 представлены все типы данных (классы), имеющиеся в MATLAB'е. Из диаграммы видно, что все они порождаются от класса *array* (массив), и, таким образом, все типы данных в MATLAB'е являются массивами: массивы логических значений (*logical*), массивы символов (*char*), массивы вещественных чисел с плавающей запятой двойной точности (*double* — исторически первый появившийся в MATLAB'е тип данных) и т.д.

Массив — это тип данных, представляющий собой коллекцию элементов (*компонент массива*) других типов, причем доступ к элементам осуществляется по индексу (индексам)[1]. Каждый массив в любой момент времени имеет определенные размеры (в MATLAB'е они могут меняться в ходе работы программы): если массив имеет d размерностей, которые равны $n_1, n_2, \ldots, n_d$ соответственно, то он содержит $n_1 \cdot n_2 \cdot \cdots \cdot n_d$ компонент. Доступ к компонентам осуществляется по индексам, например: $A(i_1, i_2, \ldots, i_d)$, где A — имя переменной (имя конкретного массива). Нумерация элементов в MATLAB'е всегда начинается с 1, поэтому $1 \le i_j \le n_j$ $(j = 1, 2, \ldots, d)$. Массивы размерности 2 называются матрицами. Минимальное число размерностей массивов в MATLAB'е — 2, поэтому одиночные значения (скаляры) представлены как массивы размера 1×1, а векторы — как массивы $1 \times n$ (строки) или $m \times 1$ (столбцы). Сверху число размерностей никак не ограничивается, и, таким образом, возможна работа с многомерными массивами, например, с массивом размера $3 \times 2 \times 5$ и т.п. Если одна из размерностей равна нулю, например, 0×0, $0 \times 5 \times 0$, $1 \times 5 \times 0$, то массив *пустой*.

Двумерные логические массивы (*logical*) и двумерные массивы вещественных/комплексных чисел с плавающей запятой двойной точности (*double*) и только они могут быть *разреженными* (*sparse*).

В [4] достаточно подробно рассматривались числовые массивы с элементами *double*, в том числе разреженные. Ниже приводится краткое описание всех MATLAB'овских классов:

- *logical* — массив *логических значений*: 1 соответствует логической истине, 0 — лжи. Для хранения одного логического значения требуется 1 байт памяти. Ло-

[1] Массив называется *однородным* (а точнее, массивом с однородной структурой), если все его компоненты имеют один тип. Некоторые языки программирования, например, Pascal, допускают только однородные массивы. Другие, такие, как Perl, Python, — позволяют работать с неоднрородными массивами. Как мы увидим далее, в MATLAB'е поддерживаются разнородные массивы — массивы ячеек *cell*

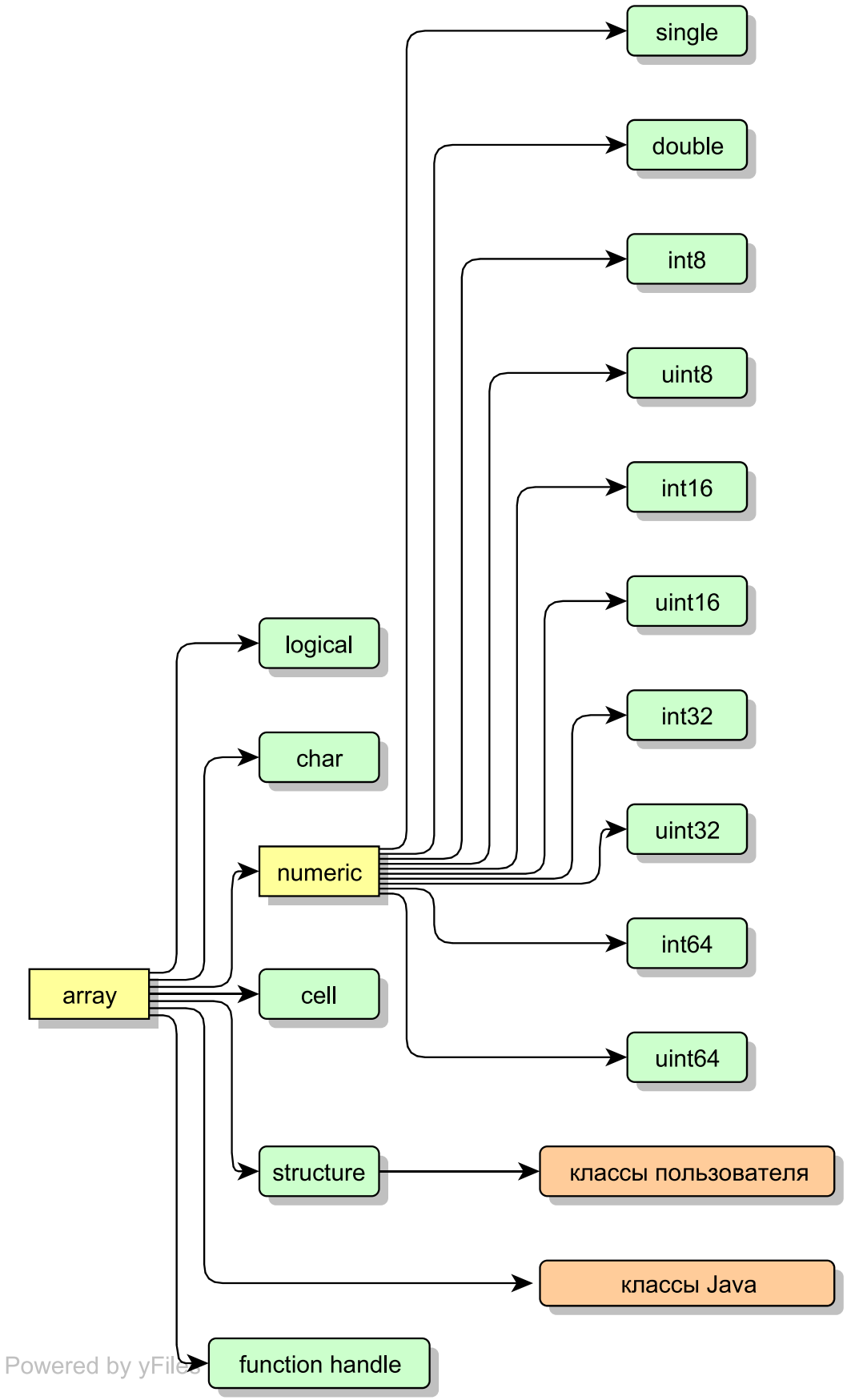


Рис. 1.1. Типы данных (классы) в MATLAB'е. Классы, отмеченные желтым цветом являются *абстрактными* — не возможно создать экземпляров этих классов.

гические массивы появляются как результаты сравнения элементов числовых и др. массивов, например, $A > B$, в результате логических операций, таких, как $A > B \,\&\, C > D$, как результат, возвращаемый некоторыми функциями, например, $any(A)$, $all(B)$, $isnumeric(C)$ и т.п. Двумерные логические массивы могут быть разреженными.

- *char* — массив *символов*. Каждый символ хранится в двух байтах в формате Inicode. Символьные массивы размера $1 \times m$ воспринимаются как обыкновенные символьные строки, например, `'Matlab'`. Символьные двумерные массивы могут, таким образом, использоваться как контейнеры таких строк, однако нужно помнить, что в этом случае все строки должны иметь одинаковое число символов. Для хранения коллекций симольных строк разной длины можно использовать массивы ячеек (*cell*).

- *single* — массив вещественных или комплексных чисел с плавающей запятой одинарной точности. Каждое вещественное значение хранится в 4 байтах в формате, предусмотренном стандартом IEEE 754. Комплексные числа представляются парой вещественных.

- *double* — массив вещественных или комплексных чисел с плавающей запятой двойной точности. Каждое вещественное значение хранится в 8 байтах в формате, предусмотренном стандартом IEEE 754. Комплексные числа представляются парой вещественных. Двумерные массивы *double* могут быть разреженными.

- *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64* — массивы знаковых (*int8*, *int16* и т.д.) и беззнаковых (*uint8*, *uint16* и т.д.) целых чисел. Цифры указывают на разрядность — число бит, используемых для представления чисел. Например, *int32* представляет целые числа из диапазона $-2^{31} \leq \alpha < 2^{31}$, а *uint32* — из диапазона $0 \leq \alpha < 2^{32}$. Операции возможны над представителями всех этих типов, кроме *int64*, *uint64*.

- *cell* — массив *ячеек*. В отличие от остальных массивов в MATLAB'е, каждый массив ячеек может состоять из компонент, относящихся к разным типам данных, т.е. быть разнородным. Компонентами массива ячеек в том числе могут служит другие массивы ячеек и т.д.

- *struct* — массив *структур*. Каждая структура представляет собой контейнер, содержащий разнородные данные. Данные размещаются в *полях* структуры. Каждое поле имеет имя. Доступ к данным происходит по имени поля. Структуры в

MATLAB'е соответсвуют структурам в C и записям (*record*) в Паскале, однако они гибче. Так, возможно *вычисление* имени поля, добавление и удаление полей в ходе работы программы, чего не возможно в C и Паскале. Это приближает структуры в MATLAB'е к ассоциативным массивам (хэшам), присутствующим в таких языках, как Perl или Python.

- На основе структур строятся *классы*. Подробности см. в главе 2, посвященной объектно-ориентированному программированию.
- Возможно конструирование классов на основе классов Java. Это позволяет в MATLAB'е использовать код, написанный на Java.
- *function handle* — массив *указателей на функцию*. Чтобы создать указатель на функцию достаточно к ее имени спереди приписать @, например, *@log*. Указатель на функцию можно передавать другим функциям в качестве параметра.

Еще раз обратим внимание, что в MATLAB'е скалярные значения (логического, символьного и т.д. типов) не выделяются в отдельные типы данных. Все типы данных в MATLAB'е — это массивы.

1.2. Массивы структур

Структурой называется тип данных, представляющий собой коллекцию значений (*полей*) разных типов, доступ к которым осуществляется по имени (*имени поля*). Как и любой другой тип данных структуры в MATLAB'е организуются в массивы (в том числе многомерные).

В MATLAB'е не нужны предварительные объявления структур. Чтобы создать ее, нужно просто указать значения соответсвующих полей. Имя поля отделяется от имени переменной–структуры точкой:

```
S.name = 'Isaac Newton';
S.age = 38;
```

Мы получили структуру (а точнее массив структур 1×1) со следующими полями и значениями полей:

name	*age*
'Isaac Newton'	38

Как мы видели, набор полей структуры может изменяться динамически. Также динамически могут меняться размеры массива структур:

$S(2).name$ = 'Blaise Pascal' ;

$S(2).age = 23$;

Теперь S — это массив структур размера 1×2:

№	*name*	*age*
1	'Isaac Newton'	38
2	'Blaise Pascal'	23

С помощью функции *struct* можно задать знаяения сразу нескольким полям структуры:

$S(3)$ = *struct*('name', 'Carl F. Gauss', 'age', 43);

Имеем:

№	*name*	*age*
1	'Isaac Newton'	38
2	'Blaise Pascal'	23
3	'Carl F. Gauss'	43

Добавим еще одно поле:

$S(3).profession$ = 'mathematician'

Получим:

№	*name*	*age*	*profession*
1	'Isaac Newton'	38	[]
2	'Blaise Pascal'	23	[]
3	'Carl F. Gauss'	43	'mathematician'

1.3. Массивы ячеек

Ячейкой (cell) называется контейнер, который может содержать в себе объект произвольного типов данных (т. е. это может быть массив чисел с плавающей запятой, массив символов, массив структур и др.)

Массив ячеек — это коллекция ячеек, доступ к которым происходит по индексу. Таким образом, массив ячеек может объединять разнотипные данные. Массив может быть одномерным, двумерным или многомерным.

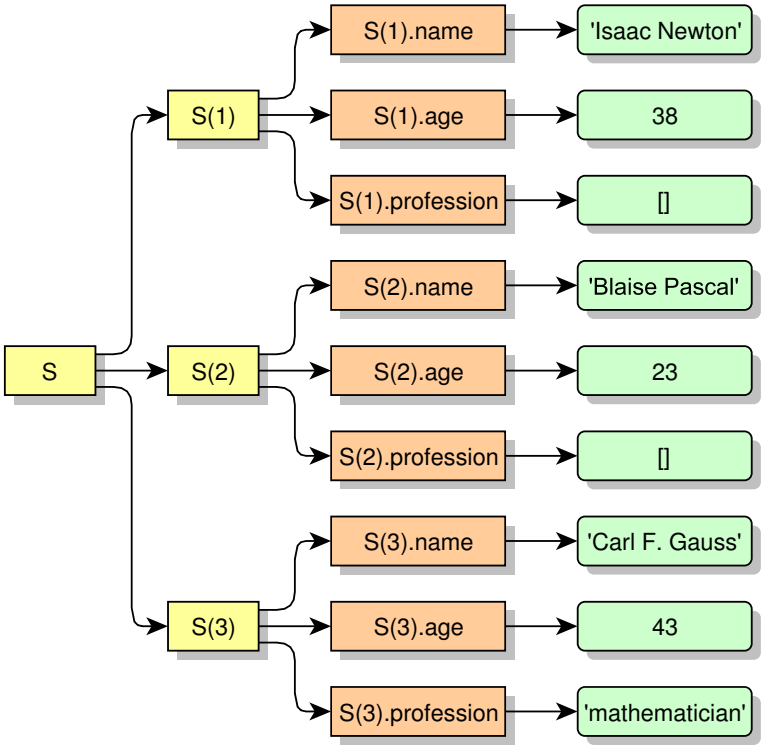


Рис. 1.2. Массив структур из трех элементов с тремя полями

Доступ к ячейкам осуществляется указанием после имени массива индекса элемента в *фигурных* скобках.

```
for n = 1:5
    M{n} = hadamard(2^n);
end
M{2}
```

Другой способ создать массив ячеек — это перечислить его элементы построчно, разделяя элементы в одной строке пробелами или запятыми, а сами строки — точкой с запятой или символом перехода на новую строку. Все элементы должны быть заключены в фигурные скобки. Например,

```
A = hadamard(4)
M = {A, sum(A), prod(A), 'matrix A'}
```

Чтобы создать массив пустых ячеек достаточно воспользоваться функцией `cell` с указанием размеров массива:

```
M = cell(5, 2, 3)
```

Одно из самых распространенных применений массива ячеек — его использования для хранения символьных строк. Например,

$M = \{$'Isaac Newton', 'Blaise Pascal', 'Carl F. Gauss', 'Nikolai I. Lobachevski'$\}$

Обращение к элементам массива ячеек с помощью индекса (или индексов, а также индексных выражений), заключенных в *круглые* скобки, приводит к созданию *срезов* массивов. Для последнего примера $M\{2\}$ — это строка 'Blaise Pascal', а $M(2)$ — массив ячеек 1×1, содержащий строку 'Blaise Pascal'. Другой пример:

$M(2$:**end**$)$

— это массив ячеек {'Blaise Pascal', 'Carl F. Gauss', 'Nikolai I. Lobachevski'}.

Заметим, что, выражения вида $M\{2$:**end**$\}$ и т. п. (индексное выражение стоит в *фигурных* скобках) также возможны. Их результатом являются уже не срезы, а *списки значений*. Они могут появляться в списках элементов массивов, в списках входных и выходных аргументов функций и др.

Например,

$M = \{225,\ 13,\ 49\}$;
$A = [M\{:\}]$;

эквивалентно

$A = [225,\ 13,\ 49]$;

а

$args = \{x,\ y,$ 'b-'$\}$;
$plot(args\{:\})$;

равносильно

$plot(x,\ y,$ 'b-'$)$;

1.4. Управляющие конструкции

В любом языке программирования, в том числе в языке, предоставляемом системой MATLAB, есть специальные конструкции, предназначенные для управления порядком выпонения команд. Такие конструкции иногда называют управляющими операторами.

В MATLAB'е к ним относятся:

- условный оператор **if**,
- оператор цикла **while**,

- оператор цикла с параметром **for**,
- оператор выбора **switch**.

1.4.1. Оператор if

Оператор **if** — это оператор ветвления. Самая простейшая его форма:

if *условие*
    команды
end

Проверяется заданное условие. Если оно выполнено, то выполняются команды, следующие за этим условием. Если не выполнено, то управление передается командам после оператора **if** (после ключевого слова **end**). *условие* можно получить в результате логических операций «меньше» $<$, «больше» $>$, «равно» $==$, «меньше или равно» $<=$, «больше или равно» $>=$ «не равно» $\tilde{}=$, «и» &, «или» |, «не» $\tilde{}$.

Например, в следующем фрагменте, на экране печатается x — *целое*, если x — целое число:

if *fix*$(x) == x$
    disp(`'x - целое'`)
end;

Возможен более развернутый вариант оператора **if**:

if *условие*
    команды1
else
    команды2
end

Проверяется условие. Если оно выполнено, то выполняется блок команд 1. Если не выполнено, то выполняется блок команд 2.

Например, в следующем фрагменте, на экране печатается x — *целое*, если x — целое число; в противном случае будет напечатано x — *дробное*:

if *fix*$(x) == x$
    disp(`'x - целое'`)
else
    disp(`'x - дробное'`)
end;

Операторы **if** могут быть вложенными. В следующем фрагменте программы слово «ворона» печатается в нужно числе и падеже в зависимости от значения переменной x, в котором хранится количество ворон: 1 ворона, 2 вороны, 3 вороны и т. д.

```
if rem(fix(x/10), 10) == 1
    disp([num2str(x) ' ворон']);
else
    if rem(x, 10) == 1
        disp([num2str(x) ' ворона']);
    else
        if rem(x, 10) >= 2 && rem(x, 10) <= 4
            disp([num2str(x) ' вороны']);
        else
            disp([num2str(x) ' ворон']);
        end;
    end;
end;
```

Самая общая схема использования оператора **if** следующая:

```
if условие1
    команды1
elseif условие2
    команды2
elseif условие3
    команды3
...
else
    команды
end
```

Вначале проверяется условие 1. Если оно выполнено, то выполняется блок команд 1. Если не выполнено, то проверяется условие 2. Если оно выполнено, то выполняется блок команд 2. В противном случае проверяется условие 3 и т. д. Если ни одно из условий не выполнено, выполняется блок команд, следующий за ключевым словом **else**.

Пример с воронами лучше переписать с использованием такого расширенного варианта оператора **if**:

```
if rem(fix(x/10), 10) == 1
```

```
    disp([num2str(x) ' ворон']);
elseif rem(x, 10) == 1
    disp([num2str(x) ' ворона']);
elseif rem(x, 10) >= 2 && rem(x, 10) <= 4
    disp([num2str(x) ' вороны']);
else
    disp([num2str(x) ' ворон']);
end;
```

Возможен вариант оператора **if** без последнего блока **else** *команды*:

```
if условие1
    команды1
elseif условие2
    команды2
elseif условие3
    команды3
...
end
```

1.4.2. Оператор while

Оператор **while** используется в составе следующей конструкции:

```
while условие
  команды
end
```

Вначале проверяется условие. Если оно выполнено, то выполняются команды внутри тела цикла. Далее снова проверяется условие, и если оно выполнено, снова выполняются команды в теле цикла и т. д. до тех пор, пока не выполнится условие. Как только условие перестанет быть выполненным, произойдет выход из цикла и управление будет передано следующим за блоком **while** (за ключевым словом **end**) командам.

В качестве примера вычислим константу *eps*:

```
e = 1;
while 1 + e ~= 1,
    e = e/2;
end;
```

```
e = 2*e
```

1.4.3. Оператор for

Оператор **for** используется в составе следующей конструкции:

```
for переменная = выражение
    команды
end
```

Если результат вычисления выражения — вектор, то указанной переменной по очереди будет присвоена каждая из компонент этого и вектора и вскияй раз будут выполнены команды, расположенные в теле цикла.

Для примера вычислим матрицу Гильберта:

```
n = 10;
H = zeros(n,n);
for i=1:n
    for j=1:n
        H(i,j)=1/(i+j-1);
    end;
end;
H % матрица Гильберта
```

Если результат вычисления выражения — матрица, то указанной переменной по очереди будет присвоен каждый из столбцов этой матрицы.

1.4.4. Оператор switch

Оператор **switch** — это оператор выбора. Схема его использования следующая:

```
switch выражение
case {список значений 1}
    команды1
case {список значений 2}
    команды2
...
otherwise
    команды
end
```

Вычисляется выражение и по очереди сравнивается с перечисленными после ключевых слов **case** значениями. Если найдено совпадение, то выполняется соответствующий блок команд. После этого управление передается на следующую после блока (после ключевого слова **end**) команду. Если совпадений не найдено, выполняются команды за ключевым словом **otherwise**. Блок **otherwise** *команды* может отсутствовать. Значения в фигурных скобках разделяются запятыми. В случае, если какой-либо список содержит только одно значение, то фигурные скобки можно опустить.

Рассмотрим пример:

```
switch lower(method)
  case {'linear', 'bilinear'}
    disp('Method is linear')
  case 'cubic'
    disp('Method is cubic')
  case 'nearest'
    disp('Method is nearest')
  otherwise
    disp('Unknown method')
end
```

В зависимости от значения символьного массива *method* в командном окне будет напечатано одно из перечисленных сообщений. Функция

lower(*str*)

заданную строку переводит в нижний регистр.

1.5. М-файлы

Программой на языке MATLAB мы будем называть последовательность команд, записанную в файле. Чтобы система принимала программу, у файла должно быть расширение *.m*, поэтому программы в MATLAB'е часто называют *m*-файлами. Об одном типе *m*-файлов — сценариях — мы уже говорили в разделе 1.5.1. К программам можно обращаться из командного окна и из других программ. При первом обращении к программе MATLAB ищет ее на диске по имени файла (без расширения). В первую очередь поиск производится в текущей папке. Как уже отмечалось, сменить текущую папку можно командой

cd имя_ папки

или с помощью меню. Программу можно подготовить во внешнем редакторе (например, блокноте Windows), а можно воспользоваться встроенным редактором-отладчиком (Editor-Debugger). Для его вызова воспользуйтесь пунктом меню FILE|NEW|M-FILE или командой *edit*.

После того, как программа найдена на диске, производится ее синтаксический анализ. Если в результате этого анализа обнаружены ошибки, информация о них выдается в рабочем окне. В случае успешного выполнения синтаксического анализа программы создается ее псевдокод, который загружается в рабочее пространство и исполняется. Если во время исполнения происходят ошибки, то сообщения о них также отражаются в командном окне.

При повторном обращении к программе, она будет найдена уже в рабочем пространстве и поэтому не потребуется времени на ее синтаксический анализ. Удалить псевдокод из рабочего пространства можно командой

clear имя_функции

В общем случае поиск очередного встретившегося в командах имени (это может быть имя переменной или программы) начинается с рабочего пространства. Если имя не найдено, то оно ищется среди встроенных (built-in) функций. Исходный код этих функций не доступен пользователю. Далее поиск продолжается в каталогах, записанных в списке доступа. Изменить эти пути можно либо через меню, либо с помощью команды *addpath*. Команда

addpath folder1 folder2 ... −begin

добавлет указанные директории в начало списка, а команда

addpath folder1 folder2 ... **−end**

добавляет директории в конец.

Команды в программе отделяются друг от друга так же, как и в командном окне: либо символом перехода на новую строку, либо знаками ; , — отличие двух последних такое же, как и в командном окне. Специальным образом обозначать конец программы никак не требуется. Внеочередное прекращение работы программы выполняется командой **return**. Символ \% означает начало комментариев: все, что записано после него и до конца строки при синтаксическом анализе игнорируется. Все, что записано в первых строках-комментариях, автоматически подключается в систему справки и может быть вызвано с помощью команды

help имя_файла

По использованию переменных и оперативной памяти программы делятся на программы-сценарии и программы-функции.

1.5.1. Программы-сценарии

С программами сценариями мы уже встречались в разделе . Программа-сценарий (script) — простейший тип программы. Программа-сценарий может использовать не только создаваемые ей самой переменные, но и использовать все переменные в рабочем пространстве командного окна. Созданные переменные так же располагаются в этом рабочем пространстве. Получить к ним доступ можно с помощью команды *keyboard*, которую нужно записать в программу. Эта команда передает управление в командное окно, в котором меняется вид приглашения:

K>>

Теперь в окне можно выполнять любые действия, в том числе просматривать значения переменных и изменять их. Чтобы продолжить выполнение программы необходимо выполнить команду **return**. Вместо команды *keyboard* для временного приостановления работы программы-сценария можно воспользоваться средствами встроенного отладчика: установить точку прерывания (breakpoint).

1.5.2. Программы-функции

После возможных пустых строк и строк, содержащих только комментарии, первая строка программы-функции должна иметь вид

function [*y1*, *y2*, ..., *ym*] = *ff*(*x1*, *x2*, ..., *xn*)

где *ff* — имя программы-функции (оно должно совпадать с именем файла), *x1*, *x2*, ..., *xn* — входные формальные параметры, *y1*, *y2*, ..., *yn* — выходные формальные параметры. Эту строку мы будем называть заголовком функции. Если функция содержит только один выходной формальный параметр, то окружающие его квадратные скобки в заголовке функции можно опустить. Функция может не содержать вовсе входных или/и выходных параметров. В этом случае скобки (круглые — для входных параметров, квадратные — для выходных) также можно опустить. Формальные параметры (входные и выходные) используются в функции как локальные переменные. Конечно же, функция может создавать и использовать другие локальные переменные, которые, как и обычно, объявлять специальным образом не нужно.

MATLAB'овские функции мы иногда будем называть процедурами или методами, чтобы не путать их с математическими функциями.

Вызов программы осуществляется командами

$[u1,\ y2,\ ...,\ ul] = ff(v1,\ v2,\ ...,\ vk)$

где $v1, v2, \ldots, vk$ и $u1, u2, \ldots, ul$ — фактические входные и выходные параметры функции. При вызове функции фактические входные параметры засылаются по порядку в соответствующие выходные формальные параметры. Выполнение функции заканчивается после выполнения ее последней команды. Досрочный выход осуществляется командой **return**. После того, как функция завершила свою работу, формальные выходные параметры засылаются в фактические выходные параметры. Количество фактических параметров должно быть не больше количества формальных параметров, но может с ним и не совпадать. Это удобно, если используются значения аргументов по умолчанию. Количество фактических входных параметров и фактических выходных параметров, с которыми была вызвана функция, всегда можно определить с помощью обращения к функциям *nargin* (количество входных параметров), *nargout* (количество выходных параметров).

Еще один способ вызова функции — это использование ее имени внутри выражения, например: $a = ff(k,\ 2).\hat{}2$. Здесь первый выходной параметр функции возводится в квадрат и результат присваивается переменной a. Заметим, что количество выходных параметров функции может быть больше, но остальные аргументы при таком обращении к функции теряются. Если выражение состоит только из одного имени функции:

$ff(v1,\ v2,\ ...\ ,\ vk)$

то первый выходной параметр присваивается перменной *ans*.

В качестве примера рассмотрим MATLAB'овскую функцию *rank*:

```
function r = rank(A, tol)
% RANK Matrix rank.
% RANK(A) provides an estimate of the number of linearly
% independent rows or columns of a matrix A.
% RANK(A,tol) is the number of singular values of A
% that are larger than tol.
% RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

% Copyright 1984-2000 The MathWorks, Inc.
% Revision : 5.9 Date : 2000/06/0102 : 04 : 15

s = svd(A);
```

```
if nargin==1
    tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);
```

Распечатать ее текст можно командами

```
type rank
```

Файл можно также открыть и во встроенном редакторе, например, командами

```
edit rank
```

Если вы это сделали, ничего не меняйте в исходном тексте!

Функция *rank* вычисляет ранг заданной матрицы A. Для этого MATLAB вычисляет ее сингулярное разложение и в качестве ранга берет количество сингулярных чисел, отличающихся от нуля больше, чем на величину *tol*. Функцию можно вызвать либо с одним входным аргументом:

```
rank(A)
```

либо с двумя:

```
rank(A, tol)
```

Количество фактических выходных аргументов внутри функции *rank* определяется с помощью обращения к функции *nargin*. Если параметр *tol* на входе не задан, то его значение вычисляется по формуле

```
tol = max(size(A)') * max(s) * eps;
```

В качестве еще одного примера рассмотрим стандартную функцию *humps*:

```
function [out1,out2] = humps(x)
%HUMPS A function used by QUADDEMO, ZERODEMO and FPLOTDEMO.
% Y = HUMPS(X) is a function with strong maxima near x = .3
% and x = .9.
%
% [X,Y] = HUMPS(X) also returns X. With no input arguments,
% HUMPS uses X = 0:.05:1.
%
% Example:
% plot(humps)
%
```

```
% See QUADDEMO, ZERODEMO and FPLOTDEMO.

% Copyright 1984-2002 The MathWorks, Inc.
% Revision : 5.8 Date : 2002/04/1503 : 34 : 07

if nargin==0, x = 0:.05:1; end

y = 1 ./ ((x-.3).^2 + .01) + 1 ./ ((x-.9).^2 + .04) - 6;

if nargout==2,
   out1 = x; out2 = y;
else
   out1 = y;
end
```

Функция вычисляет значения y некоторой тестовой функции в точках, заданных в векторе x. Возможные способы вызова функции:

```
y = humps;
y = humps(x);
[x, y] = humps;
[x, y] = humps(x);
```

В двух последних случаях ($nargout == 2$) возвращается также сам вектор x. Если входных аргументов нет ($nargin == 0$), то в качестве x берется вектор $0 : 0.05 : 1$.

После того, как функция выполнила свою работу и произошло присваивание формальных выходных параметров фактическим, все локальные переменные функции удаляются. Переменные можно открыть для использования в командном окне, скриптах и других функциях (т. е. сделать *глобальными*) с помощью команды

```
global v1 v2 ... vk
```

Это команда должна появиться во всех функциях, которые хотят использовать одни и те же переменные, упомянутые в списке. Для получения доступа к переменным функции из командного окна (или программы-сценария) необходимо выполнить эту команду в программе-сценарии или в командной строке. Использовать глобальные переменные не рекомендуется.

В MATLAB'е возможно создание функций с неопределенным числом входным или/и выходных аргументов. Чтобы создать функцию с неопределенным числом входных ар-

гументов, нужно список ее формальных входных аргументов завершить ключевым словом *varargin*. Например:

function *myfun*(*arg1*, *arg2*, *varargin*)

Здесь *arg1*, *arg2* — первый и второй аргумент функции, остальным аргументам соответствует *varargin*. Внутри функции к *varargin* можно обращаться как к массиву ячеек, содержащему знаячения «хвостовых» входных параметров. В частности, можно использовать *varargin*{:}, чтобы, например, передать список аргументов на обработку другой функции. Список аргументов может содержать только слово *varargin*.

Например, создадим функцию *plotter*, рисующую график (или графики) и делающую сверху подпись. Первым аргументов функции пусть будет строка символов, содержащая заголовок, остальные аргументы будут передаваться MATLAB'овской функции *plot*:

function *plotter*(*caption*, *varargin*)

title(*caption*)
plot(*varargin*{:})

Теперь мы можем попробовать *plotter* в работе:

$x = linspace(-2*pi, 2*pi)$;
plot('Trigonometric functions', x, $sin(x)$, 'b', x, $cos(x)$, 'r', x, $tan(x)$, 'k')

Чтобы создать функцию с неопределенным числом выходных аргументов, нужно список ее формальных выходных аргументов завершить ключевым словом *varargout*. Работа с *varargout* аналогична работе с *varargin*.

1.5.3. Подфункции

M-файл с программой-функцией может содержать описание не одной, а нескольких функций. Имя первой функции должно совпадать с именем файла. Только эта функция (*основная*) доступна извне. Остальные функции будем называть *подфункциями*. Каждая из них может быть вызвана либо из основной функции, либо из другой подфункции того же самого *m*-файла. Сразу же оговоримся, что подфункции может содержать только файл с программой-функцией. Программы-сценарии подфункций иметь не могут. Конец описания основной функции или подфункций никаким специальным образом не обозначается: описание подфункции заканчивается либо в конце файла, либо непосредственно перед началом описания следующей подфункции (т. е. перед строкой, содержащей ключевое слово **function**). Например:

```
function [...] = main(...)

    a = ...;
    b = ...;

function [...] = sub1(...)

    a = ...;
    b = ...;

function [...] = sub2(...)

    a = ...;
    b = ...;
```

Все переменные, используемые внутри подфункции, являются локальными: их область видимости ограничивается только этой подфункцией. Все переменные, используемые в основной функции, также являются локальными: их область видимости распространяется только на саму функцию, но не ее подфункции. В примере выше переменные с одинаковым именем a в основной функции и двух подфункциях различны. То же относится к переменной b.

Рассмотрим пример с использованиер рекурсии — построение кривой, называемой ковром Серпинского. Подфункция $do\_serpinsky$ вызывается из основной функции, а также вызывает себя рекурсивно.

Листинг m/sierpinski.m

```
function serpinsky(n)

% Функция serpinsky
% serpinsky строит ковер Серпинского 5-го порядка
% serpinsky(n) строит ковер Серпинского порядка n

if nargin < 1
    n = 5;
end
```

```
[x, y] = do_serpinsky(n);

shg
plot([x, x(1)], [y, y(1)])
axis equal
axis off

pause
fill(x, y, 'r', 'EdgeColor', 'None')
axis equal
axis off

function [x, y] = do_serpinsky(n)

if n == 0 % На нулевом уровне
    x = [0, -1, 0, 1];
    y = [-1, 0, 1, 0];
else % На n-м уровне функция вызывает сама себя
    [x, y] = do_serpinsky(n - 1);

    d = 2^n;
    [xlt, ylt] = move(x, y, 0, -d, d); % сдвигаем кривую влево вверх
    [xrt, yrt] = move(x, y, 1/4, d, d); % сдвигаем вправо вверх и поворачиваем на -pi/2
    [xrb, yrb] = move(x, y, 1/2, d, -d); % сдвигаем вправо вниз и поворачиваем на pi
    [xlb, ylb] = move(x, y, 3/4, -d, -d); % сдвигаем влево вниз и поворачиваем на pi/2

    x = [xlt, xrt, xrb, xlb]; % составляем новую кривую из четырех частей
    y = [ylt, yrt, yrb, ylb];

    [x, y] = move(x, y, 5/8, 0, 0); % поворачиваем кривую на -5/4pi
end

function [x, y] = move(x, y, alpha, dx, dy)
% поворачиваем на угол -2*alpha*pi и сдвигаем на вектор (dx, dy)
```

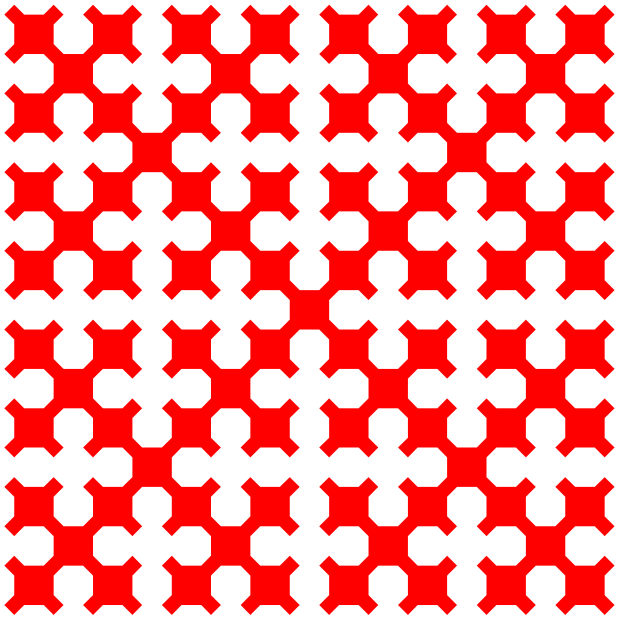


Рис. 1.3. Ковер Серпинского 4-го порядка

```
m = alpha*length(x);
x = x([(m + 1):end, 1:m]) + dx;
y = y([(m + 1):end, 1:m]) + dy;
```

Другой математический объект, связанный с именем Серпинского, — тесьма Серпинского — также может быть построена рекурсивно

Листинг m/gasket00.m

```
function gasket00(n)

% Функция gasket00
% gasket00 строит тесьму Серпинского 6-го порядка
% gasket00(n) строит тесьму Серпинского порядка n

if nargin < 1
    n = 6;
end

do_gasket([0 0], [1 0], [1/2 sqrt(3)/2], n)
```

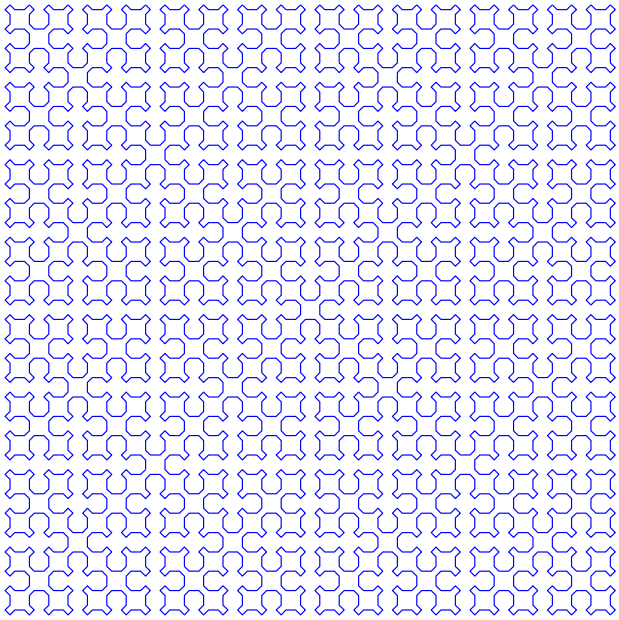


Рис. 1.4. Ковер Серпинского 5-го порядка

```
function do_gasket(A, B, C, n)
if n == 0 % На нулевом уровне рекурсии
    fill([A(1), B(1), C(1)],[A(2), B(2), C(2)], 'g', 'EdgeColor', 'None');
    hold on
    axis off
else % Рекурсивный вызов для трех «маленьких» треугольников
    do_gasket(A, (A + B)/2, (A + C)/2, n - 1)
    do_gasket(B, (B + A)/2, (B + C)/2, n - 1)
    do_gasket(C, (C + A)/2, (C + B)/2, n - 1)
end
```

Далее мы предложим намного более быстрый вариант этой функции.

1.5.4. Вложенные функции

Существует другой способ написания *m*-файлов, содержащих несколько функций. Функции можно вкладывать внутрь других функций. В этом случае каждая из функций, описанных в файле должна заканчиваться ключевым словом **end**. Все команды после заголовка функции (строка **function** ...) и до соответствующего ключевого

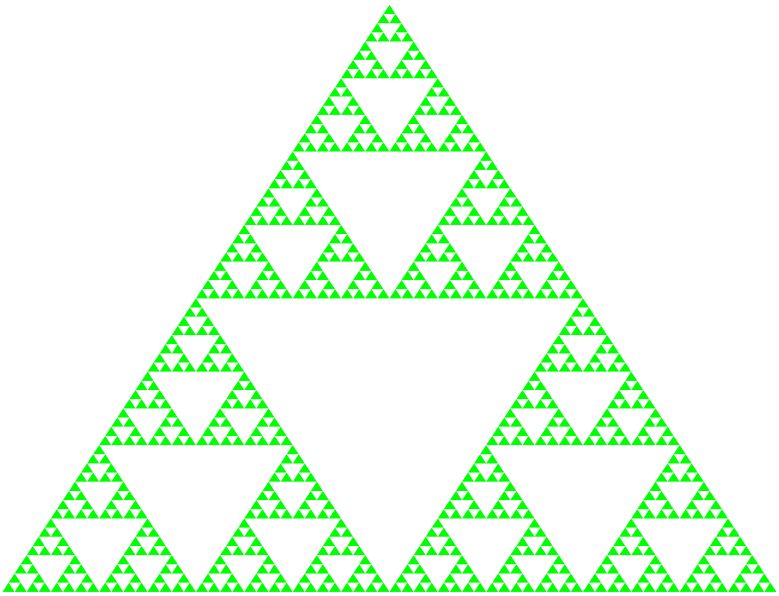


Рис. 1.5. Тесьма Серпинского 6-го порядка

слова **end** составляют *тело* функции. Чтобы определить *вложенную функцию* (nested function), нужно разместить ее тело внутри тела другой функции (в любом месте). Количество уровней вложенности не ограничено. Например:

```
function [...] = main(...)
    a = ...;
    b = ...;
    function [...] = sub(...)
        c = ...;
        function [...] = subsub(...)
        end;
    end;
    function [...] = sub2(...)
        c = ...;
    end;
end
```

Область видимости переменных функции распространяется на все вложенные в нее функции (а также функции, вложенные во вложенные функции, и т. д.). В примере выше во вложенных функциях *sub*, *subsub* и *sub2* есть доступ к переменным *a* и *b* из

основной функции *main*. Во вложенной функции *subsub* есть доступ к переменной c из *sub*. В то же время переменные с одинаковым именем c в функции *sub* и функции *sub2* различны (предполагается, что в основной функции *main* нигде не встречается переменная c).

В качестве примера рассмотрим известную задачу о ханойских башнях. Имеется 3 колышка и несколько дисков. Все диски имеют разный диаметр. Вначале все они расположены на колышке A, причем внизу лежит диск с максимальным диаметром и диаметры дисков убывают снизу вверх. Требуется переложить все диски с колышка A на колышек B, используя как вспомогательный колышек C. Разрешается надевать диски только на колышки, причем диск с большим диаметром не может лежать на диске с меньшим диаметром. Запрещено за один ход брать сразу несколько дисков. Алгоритм решения задачи хорошо описывается рекурсивно.

Если у нас нет дисков, то ничего перекладывать не нужно. Предположим теперь, что мы умеем решать задачу для $n-1$ дисков. Если у нас n дисков, то переложим верхние $n-1$ дисков с колышка A на колышек C, пользуясь как вспомогательным колышком B. Это мы сможем сделать, так как решать задачу для $n-1$ дисков мы умеем. При этом самый нижний диск на колышке A имеет максимальный диаметр, поэтому он не сможет нам помешать. Далее перемещаем этот большой диск с A на B и затем перекладываем все $n-1$ дисков с колышка C на колышек B, пользуясь как вспомогательным колышком A.

Данный алгоритм реализует следующая программа.

Листинг m/hanoi.m

```
function hanoi(n)

% Функция hanoi — задача о ханойский башнях
% hanoi — решение задачи с 4 дисками
% hanoi(n) — решение задачи с n дисками

if nargin < 1
    n = 4;
end

count = 0;
do_hanoi(n, 'A', 'B', 'C');
disp(['Суммарное число ходов: ' num2str(count)]);
```

```
    function do_hanoi(n, from, to, using)
        if n > 0
            do_hanoi(n - 1, from, using, to);
            disp(['Переместить диск с ' from ' на ' to]);
            count = count + 1;
            do_hanoi(n - 1, using, to, from);
        end
    end

end
```

Например для 4 дисков программа напечатает:

Переместить диск с A на C
Переместить диск с A на B
Переместить диск с C на B
Переместить диск с A на C
Переместить диск с B на A
Переместить диск с B на C
Переместить диск с A на C
Переместить диск с A на B
Переместить диск с C на B
Переместить диск с C на A
Переместить диск с B на A
Переместить диск с C на B
Переместить диск с A на C
Переместить диск с A на B
Переместить диск с C на B
Суммарное число ходов: 15

В программе для подсчета общего числа перемещений мы использовали переменную *count*. Она была проинициализирована (значением 0) в основной функции и поэтому доступна в подфункции *do_hanoi*.

Следующая программа реализует алгоритм поиска пути в лабиринте. Массив *visited* посещенных узлов лабиринта инициализирован в основной функции и при каждом обращении к вложенной функции *find_path* доступен. Инициализация этого массива внутри функции *find_path* привела бы к тому, что у нас был бы не один, а *много* массивов

(по количеству вызовов функции *find_path*: эта функция вызывает себя рекурсивно) и программа не правильно бы работала. Передача массива *visited* с помощью параметров (входного и выходного) функции *find_path* привела бы к большим накладным расходам.

Листинг m/hampton.m

```
function path = hampton

% Функция находит путь в лабиринте из точки
% (startx, starty) в (stopx, stopy)

    plan = [
        '8888888888888888888888888888888888'
        '8␣␣␣␣␣␣␣␣␣8␣␣␣␣8␣␣␣␣␣␣␣8␣␣␣␣␣␣␣␣␣␣␣␣8'
        '8␣88888888␣8␣8888␣8888␣8␣888888888␣8'
        '8␣8␣␣␣␣␣␣␣8␣8␣␣␣␣␣␣␣8␣8␣␣␣␣␣␣␣␣␣8␣8'
        '8␣8␣8888888␣8␣88888␣8␣888888888␣8␣8'
        '8␣8␣8␣␣␣␣␣␣8␣␣␣␣8␣␣␣␣8␣␣␣␣␣␣␣␣␣8␣8␣8'
        '8␣8␣8␣888888888␣8␣88888␣88888␣8␣8␣8'
        '8␣8␣8␣8␣␣␣␣8␣␣␣␣␣␣8␣␣␣␣␣␣8␣8␣8␣8␣8␣8'
        '8␣8␣8␣8␣8␣8␣8888888888␣8␣8␣8␣8␣8␣8␣8'
        '8␣8␣␣␣8␣8␣8␣␣␣␣␣␣␣␣8␣8␣8␣8␣8␣␣␣␣8␣8'
        '8␣888␣8␣8␣8␣8␣␣␣␣␣␣␣␣8␣8␣8␣8␣8␣888␣8'
        '8␣␣␣␣8␣8␣8␣8␣8␣␣␣␣␣␣␣␣8␣8␣8␣8␣8␣␣␣␣8'
        '888␣8␣8␣8␣8␣8␣␣␣␣␣␣␣␣8␣8␣8␣8␣8␣8␣888'
        '8␣␣␣␣8␣8␣8␣8␣8␣␣␣␣␣␣␣␣8␣8␣8␣8␣8␣␣␣␣8'
        '8␣888␣8␣8␣8␣8888␣8888␣8␣8␣8␣8␣8␣8␣8␣8'
        '8␣8␣␣␣8␣8␣8␣␣␣␣8␣8␣␣␣␣8␣8␣8␣␣␣␣8␣8␣8'
        '8␣8␣8␣8␣8␣8␣8888␣8␣8␣8888␣8␣88888␣8␣8␣8'
        '8␣8␣8␣␣␣␣8␣␣␣␣␣8␣8␣8␣␣␣␣␣␣␣␣8␣␣␣␣␣␣8␣8'
        '8␣8␣8888888888␣8␣8888888888888888␣8'
        '8␣8␣␣␣␣␣␣␣␣␣␣␣␣8␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣8'
        '8␣888888888888888888888888888888888888'
        '8␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣8'
        '88888888888888␣88888888888888888'];

    startx = 18;
```

```
    starty = 23;
    stopx = 17;
    stopy = 14;

    [m, n] = size(plan);

    visited = zeros(m, n);
    path = find_path(startx, starty);

    function path = find_path(x, y)

        if x > n || x < 1 || y > m || y < 1
            path = []; % Запрещается выходить за пределы поля
        else
            if plan(y, x) == '8' || visited(y, x)
                % На поле стоит препятствие или поле уже посещалось
                path = [];
            elseif x == stopx && y == stopy
                % Дошли до цели
                path = [x, y];
            else
                visited(y, x) = 1;
                for d = [1, 0; -1, 0; 0, 1; 0, -1]'
                    % Перебираем все возможные ходы
                    path = find_path(x + d(1), y + d(2));
                    if ~isempty(path)
                        path = [x, y; path];
                        return
                    end
                end
            end
        end
    end
end
```

В заключение раздела вернемся к задаче построения тесьмы Серпинского. Для ускорения мы не будем строить маленькие треугольники как отдельные графические объекты, а будем накапливать их координаты в массивах x и y, а потом нарисуем все сразу. Массивы x и y можно было передавать и возвращать как параметры (входные и выходные) рекурсивной функции do_gasket, однако намного удобнее (более того, это приведет еще к более быстрому коду) сделать функцию do_gasket вложенной и проинициализировать x и y в вызывающей функции, так, чтобы к x и y был постоянный доступ внутри do_gasket.

Листинг m/gasket.m

```
function gasket(n)

% Функция gasket — тесьма Серпинского.
% Более быстрая версия функции gaske00
% gasket строит тесьму Серпинского 6-го порядка
% gasket(n) строит тесьму Серпинского порядка n

if nargin < 1
    n = 6;
end

x = [];
y = [];
do_gasket([0 0], [1 0], [1/2 sqrt(3)/2], n)
fill(x', y', 'm', 'EdgeColor', 'None');
axis off

    function do_gasket(A, B, C, n)
        if n == 0
            x = [x; A(1), B(1), C(1)];
            y = [y; A(2), B(2), C(2)];
        else
            % Рекурсивный вызов для трех «маленьких» треугольников
            do_gasket(A, (A + B)/2, (A + C)/2, n - 1)
            do_gasket(B, (B + A)/2, (B + C)/2, n - 1)
            do_gasket(C, (C + A)/2, (C + B)/2, n - 1)
```

```
        end
    end
end
```

Итак, MATLAB допускает два способа описания нескольких функций внутри одного файла:

- основная функция и ее подфункции;
- основная функция и вложенные функции.

Пользоваться обеими этими способами в одном *m*-файле нельзя.

1.5.5. Частные функции

Частные функции — это функции, размещенные в папке с именем *private*. Частные функции доступны только из функций, расположенных в самой этой папке и в родительской папке. Папку *private* не следует указывать в путях доступа.

2. Объектно-ориентированное программирование на MATLAB'е

2.1. Объектно-ориентированное программирование

Объектно-ориентированным программированием (ООП) называют стиль разработки программ, при котором предметная область рассматривается как совокупность взаимодействующих друг с другом объектов. Объекты могут посылать друг другу сообщения и реагировать на них. Каждый объект может рассматриваться как черный (или серый) ящик, детали внутренней реализации которого скрыты. ООП иногда противопоставляется процедурному программированию, в котором программа представляется как некоторый набор инструкций компьютеру.

Любой объект обладает некоторыми *атрибутами* (называемыми также *полями* или *свойствами* объекта). Например, объект «окружность» (на экране компьютера) может иметь следующие атрибуты: координаты центра, радиус, цвет, толщину линии, флаг заполнения (заливки) внутренней части окружности и др. Совокупность всех атрибутов объекта, называется его *внутренней структурой*. Атрибуты объекта принимают некоторые значения. Эти значения могут меняться, но в определенный момент времени они точно определены. Помимо свойств у объекта есть *методы*, характеризующие его модель поведения. Методы описывают, что́ объект может делать, и как он реагирует на действия других объектов. У объекта «окружность» могут быть методы, которые ее рисуют, стирают, перемещают, меняют цвет и т.д. Даже если список и значения всех атрибутов у двух объектов совпадают, то это не означает, что совпадают сами объекты, так как помимо аттрибутов и методов каждый объект обладает своей *индивидуальностью*, которая не меняется в ходе работы программы. Как правило, индивидуальность обеспечивается наличием у объекта некоторого уникального *имени* (*идентификатора*). Имена нужны для того, чтобы отличать объекты друг от друга.

Класс — это множество объектов с одинаковой внутренней структурой и одинаковыми методами. Таким образом, объект – это конкретная реализация (экземпляр) того или иного класса. Например, класс «окружность», класс «прямоугольник» — это множества всех окружностей или прямоугольников соответственно, которые можно нарисовать, скажем, в графическом окне, а класс «окно» — это множество всех графических окон, которые можно изобразить на экране.

Сообщением называют передачу данных от одного объекта (отправителя) другому (получателю) или запрос одного объекта (отправителя) на запуск определенного ме-

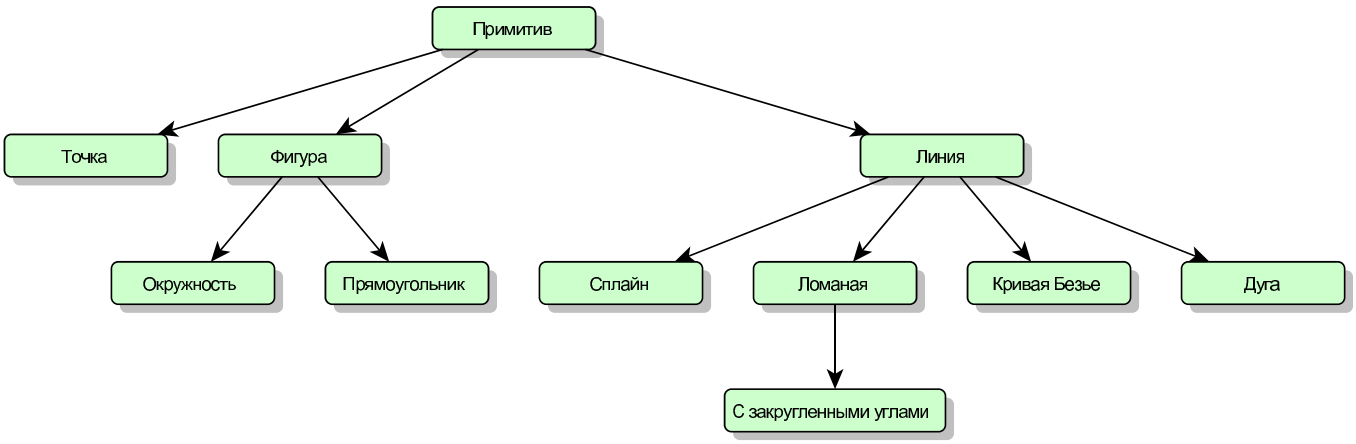


Рис. 2.1. Иерархия классов, реализующих некоторые графические примитивы

тода другого объекта (получателя). Например, объект класса «окно» может посылать запрос объекту класса «окружность» нарисовать себя. Обычно передача сообщения в языках программирования осуществляется вызовом соответсвующего метода объекта-получателя.

Среди методов специально выделяют *конструктор*, который создает экземпляр класса, и *деструктор*, уничтожающий его[1].

Остановимся еще на трех важных концепциях ООП: наследовании, инкапсуляции и полиморфизме.

Наследование — это механизм порождения новых классов из старых. Более специализированные версии класса называются *подклассами* (или *дочерними классами*). Подклассы наследуют атрибуты и поведение от родительских классов, но к спискам атрибутов и методов могут добавлять новые. Например, классы «окружность» и «прямоугольник» могут являться дочерними по отношению к классу «фигура».

В случае *множественного наследования* подклассы наследуют атрибуты и поведение двух или более родительских классов. Множественное наследование поддерживается, например, в C++. Иерархия классов может быть изображена в виде ориентированного графа, в котором вершинам соответствуют классы, а дугам — отношения наследования[2]. На рис. 2.1 приведена гипотетическая иерархия классов, реализующих некоторые графические примитивы.

[1] Формально конструктор не является методом, так как вызывается, когда объекта еще не существует.

[2] Этот граф не содержит орциклов и, если множественное наследование не допускается, является лесом.

Инкапсуляцией называют скрытие деталей внутреннего устройства объекта. Скрытию (полному или частичному) могут подлежать как некоторые свойства объекта, так и некоторые методы. Методы и свойства могут быть *открытыми* (public), *защищенными* (protected) или *частными* (private), в зависимости от этого они доступны соответственно объектам всех классов, порожденных классов или только объектам данного класса.

Полиморфизмом в объектно-ориентированном программировании называют способность объектов из разных классов обрабатывать одинаковые сообщения своим, индивидуальным для класса, образом. Рассмотрим следующие виды полиморфизма: полиморфизм наследования, перегрузку и параметрический полиморфизм.

Полиморфизм наследования заключается в том, что дочернии классы могут иметь новые реализации унаследованных от родителей методов. Предположим, что объект «ломаная с закругленными углами» является дочерним по отношению к классу «ломаная». От родительского класса он унаследовал метод «нарисовать себя», но он дожен его переопределить, чтобы получить правильный рисунок.

Перегрузка — это использование одинаковых имен для функций или одинаковых обозначений для операций над объектами разных классов. Этот вид полиморфизма давно используется: во многих языках программирования операции $+$, $-$, $*$, $/$ могут применяться как для целых чисел, так и для вещественных (а также эти операции разумно перегрузить для работы с матрицами, векторами и др.).

В *параметрическом полиморфизме* код пишется один раз сразу для объектов разных типов. Таким образом, параметром здесь выступает класс, для которого метод может быть использован. Параметрический полиморфизм естественным образом реализуется на языках программирования, поддерживающих шаблоны, например, на C++. Другой способ написать код в духе параметрического полиморфизма — использовать указатели вместо самих объектов, и передавать их в качестве параметров функций. Использование параметрического полиморфизма широко распространено в библиотеках общего назначения, например, STL, реализующей такие структуры данных, как список, вектор, дек, очередь и др. Каждая из этих структур представляет собой контейнер, содержащий данные любого (но одного для данного контейнера) класса. Реализация этих структур написана только один раз, но может быть использована для практически любык классов.

2.2. Объектно-ориентированные языки

Объектно-ориентированными языками программирования называются языки, поддерживающие стиль ООП. К таким относят C++, Java, Python, Ruby и др. В объектно-

ориентированных языках возможность конструирования классов предусмотрена уже с помощью синтаксических средств. Например, в C++ общий синтаксис описания класса имеет следующий вид:

```
class имя_класса : имя_родительского_класса
{
    private:
        частные_поля
        частные_методы
    protected:
        защищенные_поля
        защищенные_методы
    public:
        открытые_поля
        открытые_методы
};
```

Так, объявление класса *curve*, реализующего кривые на экране, могло выглядеть следующим образом:

```
class curve : public primitive /* Класс curve порождается из класса primitive */
{
    protected:
        uint n; /* Число характерных точек */
        double *x; /* (Указатель на) массив, содержащий координаты x характерных точек */
        double *y; /* (Указатель на) массив, содержащий координаты y характерных точек */
        double rc, gc, bc; /* r–g–b составляющие вектора цветности */
        double width; /* Толщина линии */
    public:
        curve() /* Конструктор по умолчанию */
        curve(double nn, double *xx, double *yy, double rcc,
            double gcc, double bcc, double w); /* Конструктор */
        ~curve(); /* Деструктор */
};
```

В рассматриваемом примере мы не собираемся создавать экземпляры класса *curve*. Таким образом, класс *curve* является *абстрактным*.

Класс *polyline*, реализующий ломаные линии, наследует от *curve* все указанные выше

поля и методы, а также добавляет еще один — *draw*(), отвечающий за прорисовку:

```
class polyline : public curve
{
    public:
        polyline(); /* Конструктор по умолчанию */
        polyline(uint nn, double *xx, double *yy, double rcc, double gcc,
            double bcc, double w); /* Конструктор */
        ~polyline(); /* Деструктор */
        draw(); /* Изображает ломаную на экране */
};
```

Класс *curve* — был абстрактным, поэтому и не содержал метода *draw*.

Класс *smooth_polyline*, реализующий ломаные линии с закругленными углами, порождается от *polyline*. Новый класс имеет новое поле *alpha*, отвечающее за степень гладкости, и реализует заново (перегружает) метод *draw*().

```
class smooth_polyline : public polyline
{
    protected:
        double alpha; /* Степень гладкости */
    public:
        smooth_polyline(); /* Конструктор по умолчанию */
        smooth_polyline(uint nn, double *xx, double *yy, double rcc, double gcc,
            double bcc, double w, double a); /* Конструктор */
        ~smooth_polyline(); /* Деструктор */
        draw(); /* Изображает ломаную на экране */
};
```

Как только класс объявлен, его имя можно использовать для объявления новых экземпляров класса. Например,

```
polyline a, b;
smooth_polyline c, d, e;
```

При объявлении переменных для них автоматически вызывается конструктор. В данном случае — конструктор по умолчанию, так как не были заданы параметры. В следующем примере будет вызван второй из предусмотренных конструкторов:

```
double x[4] = {0, 1, 2, 3};
double y[4] = {0, 1, 1, 0};
```

smooth_polyline a(4, *x*, *y*, 0, 0, 0, 0.5, 0.01);

Вызов метода осуществляется путем приписывания через точку имени метода к имени объекта, например:

a.draw();

Аналогично осуществляется доступ (если он открыт) к полям объекта:

$a.n = 4$;

2.3. Классы в Matlab'е

Сразу же отметим некоторые важные отличия ООП в Matlab'е от ООП в других языках, например, таких, как C++ или Java.

- Традиционно в объектно-ориентированных языках программирования обращение к методу некоторого объекта осуществляется путем приписывания через точку имени метода к имени этого объекта, например,

  obj.methodname(*arg1*, *arg2*, ..., *argn*)

  В Matlab'е для этого используется другой синтаксис: объект, к которому относится метод должен стоять первым в списке входных аргументов функции:

  methodname(*obj*, *arg1*, *arg2*, ..., *argn*)

  Например, если p — то объект класса *polyline*, а *draw* — метод этого класса, то вызов этого метода для объекта p осуществляется следующим образом:

  draw(*p*, *arg1*, *arg2*, ..., *argn*)

- В Matlab'е не предусмотрена передача параметров функции по ссылке и поэтому методы, меняющие внутренне состояние объекта обязательно должны возвращать сам этот объект в качестве одного из выходных параметров. Например, $G = addedge(G, u, v)$. Заметим, что это не всегда удобно. Частично эту проблему можно решить с помощью функции *assignin*.

- В Matlab'е нет деструкторов объектов. Для уничтожения объекта можно воспользоваться командой *clear*.

- Чтобы породить один объект на основе других необходимо в конструкторе дочернего объекта создать родительский объект (путем вызова его конструктора), а затем вызвать функцию *class*. Возможно множественное наследование.

- Обращение к полям объекта возможно только из его методов и не возможно из других функций, включая методы дочерних объектов. Таким образом, все поля объекта являются частными (private).

- Родительский объект содержится в дочернем как одно из полей, причем именем этого поля является имя родительского класса: *obj.parent_ class*. Мы не можем обращаться к полям родительского объекта, т.е. выражения вида *obj.parent_ class.fieldname* приведут к сообщению об ошибке. По той же причине запрещен доступ к родителю родителя: *obj.parent_ class.grand_ parent_ class*.

- В MATLAB'е нельзя переопределить оператор присваивания =. Однако присваивание возможно: *obj1 = obj2* означает, что *obj1* станет точной копией объекта *obj2* (в результате побайтового копирования).

- В MATLAB'е нет языковой поддержки абстрактных классов.

- В MATLAB'е нет аналога оператора :: из C++.

- В MATLAB'е не поддерживаются виртуальные функции и виртуальные классы. В этом нет необходимости, так как используется только динамическое связывание.

- В MATLAB'е нет аналога шаблонов (templates) из C++.

В MATLAB'е (как, впрочем, и во многих других языках) термины «тип данных» и «класс» являются синонимами. Диаграмма на рис.1.1, таким образом, является графическим представлением иерархии стандартных классов, поддерживаемых MATLAB'ом. Из диаграммы видно, что классы пользователя порождаются от класса *struct*.

2.4. Как пользоваться готовыми классами?

Концепцию ООП в MATLAB'е мы будем иллюстрировать на учебном примере *graph*. Класс *graph*, который мы реализуем, представляет простые (т.е. неориентированные без циклов и кратных ребер) графы (см. рис. 2.4). Он поддерживает операции создания (путем указания матрицы смежности или списка ребер), отрисовки, добавления/удаления ребер/вершин, объединения и произведения графов и др. Класс имеет три поля *AdjMatrix*, *X*, *Y*, где *AdjMatrix* — матрица смежности вершин графа — для ее представления будем использовать структуру данных *sparse* (разреженные матрицы). Векторы-столбцы X, Y содержат координаты вершин графа. Массивы X, Y либо имеют одинакову длину, равную числу вершин графа, либо пусты.

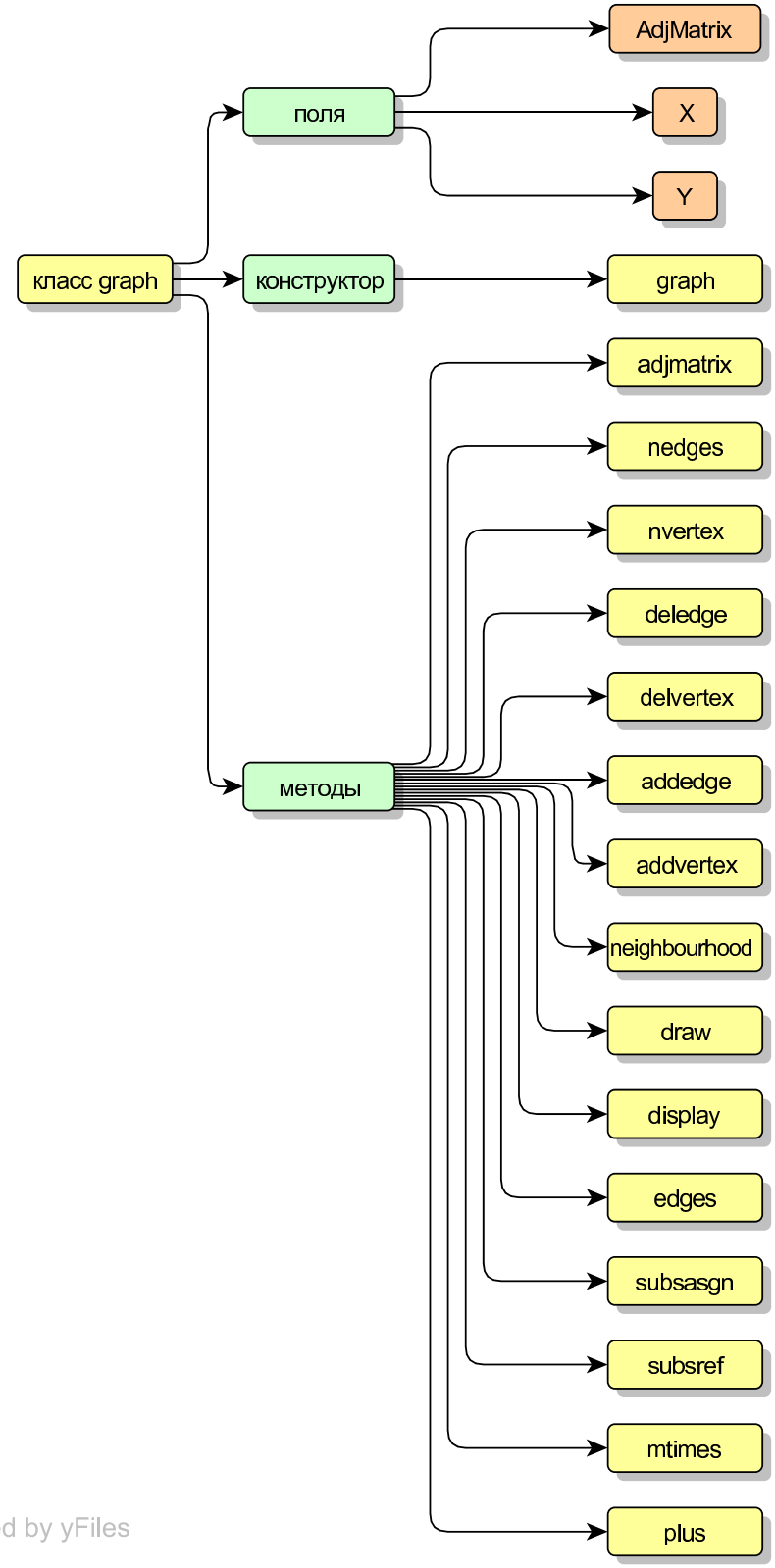


Рис. 2.2. Поля и методы класса *graph*.

Вначале приведем простую программу-сценарий, использующую некоторые методы класса.

Листинг oop/xgraph.m

```
% Создадим полный граф с 5 вершинами:
A = ones(5); % Матрица смежности
phi = 0:2*pi/5:8*pi/5;
[x, y] = pol2cart(phi, 1) % Координаты вершин
G = graph(A, x, y) % Создаем граф (вызов конструктора)
shg
draw(G) % Рисуем граф. См. рис. 2.3
pause

G = addvertex(G, 2, 0); % Добавляем новую вершину с координатами (2, 0)
G = addedge(G, 2, 6); % Новую вершину соединяем ребром со 2-й вершиной
G = addedge(G, 5, 6); % Новую вершину соединяем ребром с 5-й вершиной
draw(G) % Рисуем граф. См. рис. 2.4
pause

G = delvertex(G, 1); % Удаляем 1-ю вершину. Внимание: вершины изменили нумерацию
G = deledge(G, 2, 4); % Удаляем ребро (2, 4)
G = deledge(G, 1, 3); % Удаляем ребро (1, 3)
draw(G) % Рисуем граф. См. рис. 2.5
pause

% Теперь проиллюстрируем создание графа на основе списка ребер:
E = [1, 4; 1, 5; 1, 6; 2, 4; 2, 6] % Список ребер
x = [0, 0, 0, 1, 1, 1]; % Координаты вершин
y = [0, 1, 2, 0, 1, 2];
G = graph(E, x, y)
draw(G) % Рисуем граф. См. рис. 2.6

% Сумма и произведение графов:
G1 = graph([1, 2; 2, 3; 3, 4]) % Цепь длины 4
G2 = G1
draw(G1 + G2) % См. рис. 2.7
```

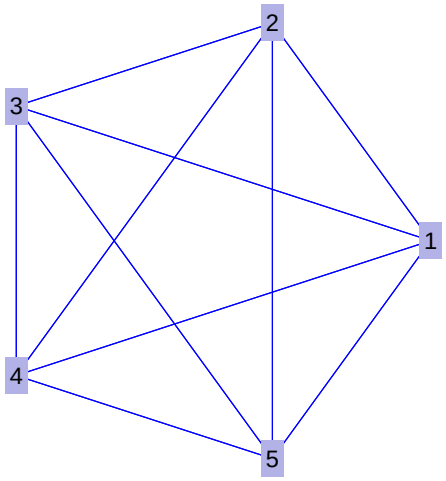


Рис. 2.3. K_5 — полный граф с 5 вершинами

```
pause
draw(G1*G2) % См. рис. 2.8
```

Результаты представлены на рис. 2.3–2.8.

Дадим некоторые комментарии к приведенной программе. В MATLAB'е имя конструктора совпадает с именем класса. Чтобы создать экземпляр класса необходимо вызвать этот конструктор. Например, команда

```
G = graph([1, 2; 2, 3; 3, 4; 4, 1; 1, 2])
```

создает граф, заданный списком ребер $(1,2)$, $(2,3)$, $(3,4)$, $(4,1)$, $(1,2)$.

Сказанное насчет конструктора относится также и к стандартным классам. Например, экземпляр класса *struct* создается вызовом конструктора этого класса, например,

```
S = struct('Name', 'Mick', Age, 40)
```

Методы класса представляют собой *m*-функции. Первым аргументом такой функции является объект, к которому этот метод относится:

```
[out1, out2, ...] = methodname(obj, arg1, arg2, ...)
```

Например, класс *graph* содержит метод *addedge*. Этот метод добавляет к заданному графу новое ребро. Метод вызывается следующим образом:

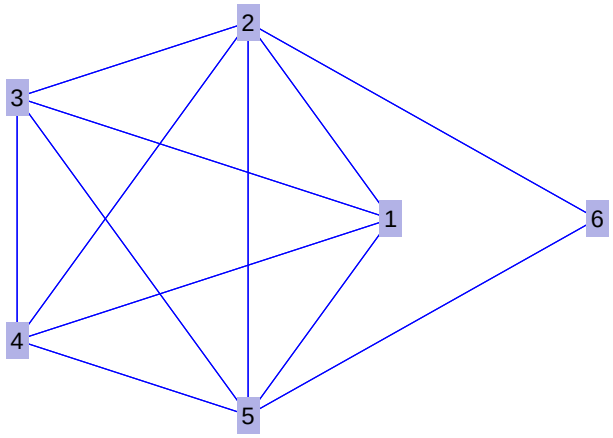


Рис. 2.4. Граф после добавления новой вершины и двух ребер

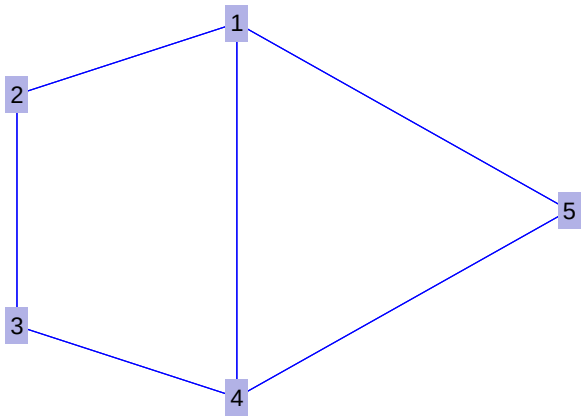


Рис. 2.5. Граф после удаления 1-й вершины и двух ребер

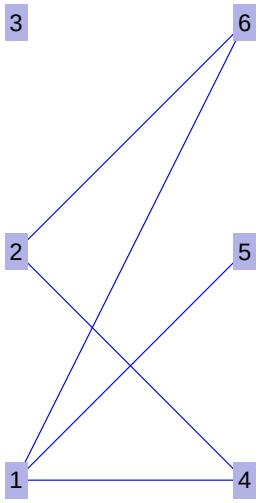


Рис. 2.6. Граф задан списком своих ребер

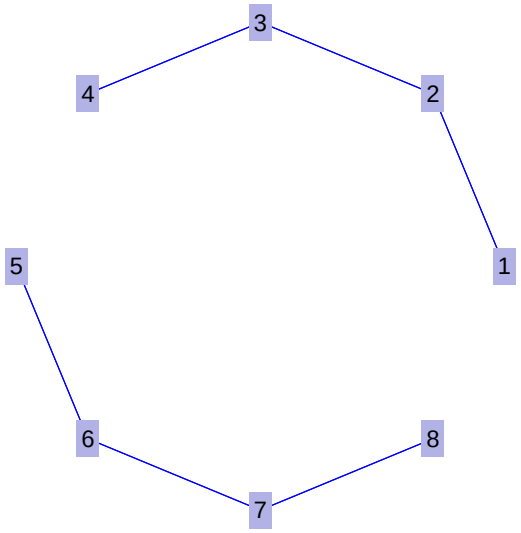


Рис. 2.7. Сумма (объединение) двух цепей длины 4

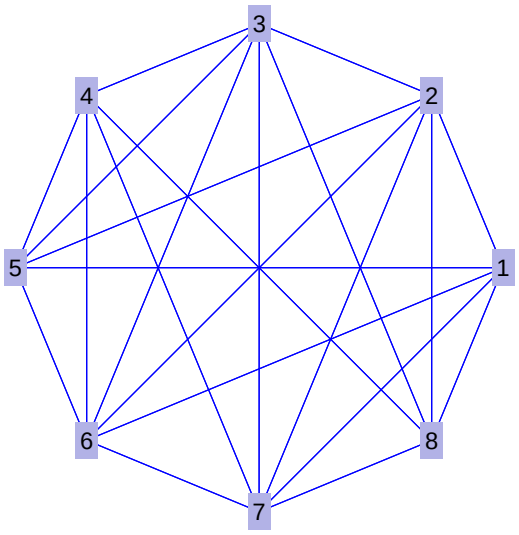


Рис. 2.8. Граф $K_{4,4}$ произведение двух цепей длины 4

$G = addedge(G,\ u,\ v)$

Здесь u, v — концевые вершины нового ребра.

Функцию *class* можно использовать для определения класса заданного объекта. Например,

$G = graph;$
$class(G)$

В результате получим

$ans =$
graph

Ниже мы познакомимся с другим вариантом функции *class*.

К функции *class* близка функция *isa*:

$b = isa(obj,$ `'class_name'`$)$

Она возвращает 1 (логическую истину), если объект *obj* относится к классу *classname* или к одному из его подклассов. В противном случае возвращается 0 (логическая ложь).

Функция *methods*(`'classname'`) и *methods*(*obj*) возвращают полный список методов указанного класса. Например,

methods(`'graph'`)

2.5. Как создать новый класс?

В MATLAB'е каждый метод класса записывается в своем *m*-файле. Все *m*-файлы, относящиеся к одному классу *classname*, должны находиться в папке *@classname*, имя которой получается приписыванием спереди к имени класса символа *@*. Например, реализации всех методов класса *graph* располагаются в папке *@graph*. Чтобы MATLAB мог найти описание класса, папка *@classname* должна находиться в некоторой другой папке, указанной в MATLAB'овских путях доступа, или располагаться в текущей папке. Саму папку *@classname* в путях доступа прописывать не нужно.

2.5.1. Конструктор

Каждый класс в MATLAB'е должен иметь конструктор. Конструктор создает экземпляры данного класса. Имя конструктора совпадает с именем классы. Как правило, конструктор должен обрабатывать несколько вариантов комбинаций входных аргументов:

- *Конструктор вызван без входных аргументов.* В этом случае необходимо проинициализировать поля некоторыми значениями по умолчанию, а затем вызвать функцию *class*. Конструктор без входных параметров вызывается MATLAB'ом автоматически в следующих случаях: при загрузке объекта из файла командной *load*, при создании массива объектов — в этом случае конструктор вызывается для каждого объекта.

- *Первым входным аргументом является объект того же класса.* Определить принадлжность объекта заданному классу можно с помощью функции *isa*. Как правило, в этом случае конструктор должен возвратить этот же объект (*конструктор копирования*). В MATLAB'е констуктор копирования никогда не вызывается автоматически.

- *Первый входной аргумент не является объектом того же класса.* В этом случае объект создается и его поля инициализируются некоторыми значениями на основе данных, переданных конструктору. В заключение должна быть вызвана функция *class*.

Синтаксис функции *class*:

```
obj = class(s, 'classname')
obj = class(s, 'classname', parent1, parent2, ...)
```

где s — структура с инициализированными полями, `'classname'` — имя класса, *parent1*, *parent2* и т.д. объекты родительских классов, чьи поля и методы наследует создаваемый объект. В качестве полей у объекта будут все поля структуры s. Поля можно создавать только в конструкторе. Все объекты одного класса должны иметь одинаковый список полей. Поля доступны из любого метода данного объекта: *obj.fieldname*, но ниоткуда больше, в том числе не доступны из методов дочерних классов. Как уже отмечалось, объект *obj* получит поля *parentc1ass1*, *parentclass2* и т.д., в которых будут содержаться родительские объекты. Обращаться с помощью двойного индексирования — *obj.parentc1ass1.fieldname* — к полям родительских объектов запрещено.

Следующие функции могут быть вызваны только в конструкторе: *class* (создание экземпляра класса), *superiorto* и *inferiorto* и др.

В качестве примера рассмотрим конструктор класса *graph*.

Листинг oop/@graph/graph.m

```
function G = graph(varargin)

% Конструктор класса graph
% G = graph создает пустой граф (без вершин и ребер)
% G = graph(G1) копирует граф
% G = graph(A) создает граф G по матрице смежности A, где A — квадратная матрица
% G = graph(E) создает граф G по списку ребер E, где E — матрица с двумя столбцами
% G = graph(..., x, y) позволяет также задать координаты вершин

% Объект класса graph имеет 3 поля, которые мы вначале инициализируем
% пустыми значениями:
G.AdjMatrix = sparse(0, 0); % Матрица смежности
G.X = []; % x-координата вершин графа
G.Y = []; % y-координата вершин графа

arg = varargin;

if nargin == 0 % Нет входных аргументов
    G.AdjMatrix = sparse([]);
```

```
    G = class(G, 'graph');
elseif isa(arg{1}, 'graph') % «Конструктор копирования»
    G = arg{1};
elseif isa(arg{1}, 'double') && all(arg{1}(:)) && size(arg{1}, 2) == 2
    % Первый аргумент — это список ребер
    n = max(arg{1}(:));
    m = size(arg{1}, 1);
    A = sparse(arg{1}(:, 1), arg{1}(:, 2), ones(m, 1), n, n);
    G.AdjMatrix = A;
    G = class(G, 'graph');
elseif isa(arg{1}, 'double') && size(arg{1}, 1) == size(arg{1}, 2)
    % Первый аргумент — это матрица смежности
    A = sparse(arg{1});
    n = size(A, 1);
    A = abs(A) + abs(A'); % «Симметризация» матрицы
    A(A ~= 0) = 1;
    A(1:n + 1:n^2) = 0; % На диагонали ставим нули
    G.AdjMatrix = A;
    G = class(G, 'graph');
else
    error('Неправильный набор параметров');
end
if nargin >= 3
    G.X = arg{2}(:);
    G.Y = arg{3}(:);
    if length(G.X) ~= n || length(G.Y) ~= n
        error('Неправильное число координат')
    end
end
superiorto('pointer', 'double')
```

2.5.2. Методы класса

Как уже отмечалось, каждый метод класса должен располагаться в своем m-файле. Все m-файлы, соответсвующие конкретному классу располагаются в папке $@classname$.

В качестве примера приведем реализацию следующих методов класса *graph*: *adjmatrix*,

edges, *nedges*, *nvertices*, *addedge*, *addvertex*, *deledge*, *delvertex*, *draw*. Остальные методы этого класса рассматриваются в следующем разделе.

Листинг oop/@graph/adjmatrix.m

```
function A = adjmatrix(G)

% Метод adjmatrix класса graph
% A = adjmatrix(G) возвращает матрицу смежности графа G

    A = full(G.AdjMatrix);
```

Листинг oop/@graph/edges.m

```
function edg = edges(G)

% Метод edges класса graph
% edg = edges(G) возвращает список ребер графа G

    [i, j] = find(G.AdjMatrix);
    edg = [i, j];
    edg = edg(i < j, :);
```

Листинг oop/@graph/nedges.m

```
function m = nedges(G)

% Метод nedges класса graph
% m = nedges(G) возвращает количество ребер графа G

    m = nnz(G.AdjMatrix)/2;
```

Листинг oop/@graph/nvertices.m

```
function n = nvertices(G)

% Метод nvertices класса graph
% m = nvertices(G) возвращает количество вершин графа G

    n = length(G.AdjMatrix);
```

Листинг oop/@graph/addedge.m

```
function G = addedge(G, u, v)

% Метод addedge класса graph
% G = addedge(G,u,v) добавляет ребро (u,v) к графу G

        G.AdjMatrix(u, v) = 1;
        G.AdjMatrix(v, u) = 1;
```

Листинг oop/@graph/addvertex.m

```
function G = addvertex(G, x, y)

% Метод addvertex
% G = addvertex(G) добавляет новую вершину к графу G
% G = addvertex(G,x,y) добавляет новую вершину к графу G и задает ее координаты

    G.AdjMatrix(end + 1, end + 1) = 0; % Порядок матрицы увеличивается на 1

    if nargin == 1
        G.X = [];
        G.Y = [];
    elseif nargin == 3 && ~isempty(G.X) && ~isempty(G.Y)
        G.X = [G.X; x];
        G.Y = [G.Y; y];
    end
```

Листинг oop/@graph/deledge.m

```
function G = deledge(G, u, v)

% Метод deledge класса graph
% G = deledge(G,u,v) удаляет ребро (u,v) в графе G

    G.AdjMatrix(u, v) = 0;
    G.AdjMatrix(v, u) = 0;
```

Листинг oop/@graph/delvertex.m

```
function G = delvertex(G, v)

% Метод delvertex класса graph
% G = delvertex(G,v) удаляет вершину v в графе G

        G.AdjMatrix(v, :) = [];
        G.AdjMatrix(:, v) = [];

        if ~isempty(G.X) && ~isempty(G.Y)
                G.X(v) = [];
                G.Y(v) = [];
    end
```

Листинг oop/@graph/draw.m

```
function draw(G)

% Метод draw класса graph
% draw(G) изображает граф G в графическом окне

    n = nvertices(G);

    if n == 0
        disp('Граф не содержит вершин')
        return
    end
    if n > 1000
        disp('Слишком много вершин, чтобы нарисовать граф')
        return
    end
    if nedges(G) > 1000
        disp('Слишком много ребер, чтобы нарисовать граф')
        return
    end
```

```
if isempty(G.X) || isempty(G.Y)
    % Если координаты вершин не заданы, то размещаем их по кругу
    phi = 0:2*pi/n:2*pi*(n - 1)/n;
    [x, y] = pol2cart(phi', 1);
else
    x = G.X;
    y = G.Y;
end

gplot(G.AdjMatrix, [x, y])

if n <= 50
    % Если вершин не много, то рисуем их метки
    for j = 1:n
        text(x(j), y(j), num2str(j), ...
            'HorizontalAlignment','center', ...
            'BackgroundColor',[.7 .7 .9], ...
            'FontSize', 10);
    end
end

% Вычисляем величину полей вокруг рисунка
xmin = min(x);
xmax = max(x);
ymin = min(y);
ymax = max(y);

blankx = (xmax - xmin)/n;
blanky = (ymax - ymin)/n;

blank = max([blankx, blanky]);

if blank == 0
    blank = 1;
end
```

axis([*xmin* − *blank*, *xmax* + *blank*, *ymin* − *blank*, *ymax* + *blank*])

axis equal

axis off

Частные (private) *методы* — это такие методы класса, которые могут быть вызваны только методами данного класса и никакими другими функциями и методами других классов, а также не могут быть вызваны из командного окна. Чтобы определить частный метод, необходимо разместить реализующий ее *m*-файл в папку *private*, которая в свою очередь должна находиться в папке *@classname*. Папку *private* не следует указывать в MATLAB'овских путях доступа.

Частные методы отличаются от частных функций тем, что в методах первым аргументом всегда является объект соответсвующего класса. Частные функции также размещаются в папке *@classname/private*. Как и частные методы, частные функции могут быть вызваны только из методов (общих или частных) заданного класса и из других функций, *m*-файл которых расположены в папках *@classname* и *@classname/private*.

2.5.3. Перегрузка операторов

В MATLAB'е можно перегрузить любой встроенный оператор (кроме оператора присваивания =). Каждому встроенному оператору соответсвует некоторая функция и вызов оператора эквивалентен вызову этой функции. Например, бинарному оператору + соответствует функция *plus*(*arg1*, *arg2*). Таким образом, чтобы перегрузить некоторый оператор, нужно создать метод класса с соответсвующим именем. Если, например, *a*, *b* — объекты класса *classname*, то выражение *a* + *b* приведет к вызову метода *classname/plus.m*. Если *a* и *b* — объекты разных классов, то вызываемый метод определяется специальными правилами (см. разделы 2.7).

В таблице 2.1 приведены имена функций, соответсвующие всем операторам MATLAB'а.

Ниже будут даны дополнительные пояснения к функциям *display*(*a*), *subsref*(*a*,*s*), *subsasgn*(*a*,*s*,*b*), *subsindex*(*a*).

В классе *graph* переопределяются следующие функции: *plus*, *mtimes*, *display*, *subsasgn*, *subsref*.

Вначале приведем реализации функций сложения *plus* и матричного умножения *mtimes*.

Листинг oop/@graph/plus.m

function *G* = *plus*(*G1*, *G2*)

Таблица 2.1. Операторы и соответсвующие им функции

Операция	*Функция*	*Описание*
$a + b$	*plus*(a, b)	сумма
$a - b$	*minus*(a, b)	разность
$+a$	*uplus*(a, b)	унарный плюс
$-a$	*uminus*(a, b)	унарный минус
$a.*b$	*times*(a, b)	покомпонентное умножение
$a*b$	*mtimes*(a, b)	матричное умножение
$a./b$	*rdivide*(a, b)	покомпонентное правое деление
$a.\backslash b$	*ldivide*(a, b)	покомпонентное левое деление
a/b	*mrdivide*(a, b)	матричное правое деление
$a\backslash b$	*mldivide*(a, b)	матричное левое деление
$a.\hat{}\, b$	*power*(a, b)	покомпонентное возведение в степень
$a\hat{}\, b$	*mpower*(a, b)	матричная степень
$a < b$	*lt*(a, b)	отношение «меньше»
$a > b$	*gt*(a, b)	отношение «больше»
$a <= b$	*le*(a, b)	отношение «не больше»
$a >= b$	*ge*(a, b)	отношение «не меньше»
$a == b$	*eq*(a, b)	отношение «равно»
$a \sim= b$	*ne*(a, b)	отношение «не равно»
$a \,\&\, b$	*and*(a, b)	логическое «и»
$a \mid b$	*or*(a, b)	логическое «или»
$\sim a$	*not*(a)	логическое «не»
$a{:}d{:}b$	*colon*(a, d, b)	формирование вектора
$a{:}b$	*colon*(a, b)	формирование вектора
$a.'$	*transpose*(a)	транспонирование
a'	*ctranspose*(a)	комплексное сопряжение
a	*display*(a)	печать в командном окне
$[a,\ b]$	*horzcat*(a, b)	горизонтальная конкатенация
$[a;\ b]$	*vertcat*(a, b)	вертикальная конкатенация
$a(s1,\ s2,\ldots,\ sn)$	*subsref*(a, s)	обращение к элементу массива или полю структуры
$a(s1,\ s2,\ldots,\ sn) = b$	*subsasgn*(a, s, b)	запись элемента массива или поля структуры
$b(a)$	*subsindex*(a)	обращение к элементу массива или полю структуры

```
% Метод plus (оператор +) класса graph
% G = plus(G1,G2) или G = G1 + G2 возвращает объединение графов G1 и G2

    n1 = nvertices(G1);
    n2 = nvertices(G2);

    if isempty(G1.X) || isempty(G1.Y) || isempty(G2.X) || isempty(G2.Y)
        G = graph([G1.AdjMatrix, sparse(n1, n2); sparse(n2, n1), G2.AdjMatrix]);
    else
        % Граф G2 будет расположен чуть правее графа G1
        G1xmin = min(G1.X);
        G1xmax = max(G1.X);
        G2xmin = min(G2.X);
        G1ymin = min(G1.Y);
        G2ymin = min(G2.Y);
        blank = (g1xmax - g1xmin)/nvertices(g1);
        G = graph([G1.AdjMatrix, sparse(n1, n2); sparse(n2, n1), G2.AdjMatrix], ...
            [G1.X; G2.X + G1xmax - G2xmin + blank], ...
            [G1.Y; G2.Y + G1ymin - G2ymin]);
    end
```

Листинг oop/@graph/mtimes.m

```
function G = mtimes(G1, G2)

% Метод mtimes (оператор *) класса graph
% G = mtimes(G1,G2) или G = G1 * G2 возвращает произведение графов G1 и G2

    n1 = nvertices(G1);
    n2 = nvertices(G2);

    if isempty(G1.X) || isempty(G1.Y) || isempty(G2.X) || isempty(G2.Y)
        G = graph([G1.AdjMatrix, ones(n1, n2); ones(n2, n1), G2.AdjMatrix]);
    else
        % Граф G2 будет расположен чуть правее графа G1
```

```
    G1xmin = min(G1.X);
    G1xmax = max(G1.X);
    G2xmin = min(G2.X);
    G2xmax = max(G2.X);
    blank = (g1xmax - g1xmin)/nvertices(g1);
    G = graph([g1.AdjMatrix, ones(n1, n2); ones(n2, n1), g2.AdjMatrix], ...
        [g1.X; g2.X + g1xmax - g2xmin + blank], [g1.Y; g2.Y]);
end
```

Метод display

Функция *display*(*a*) вызывается MATLAB'ом всякий раз, когда выполняемая MATLAB'ом команда производит некоторый результат, а в конце ее не стоит символа «;». В частности, *display* вызывается когда пользователь набирает в командной строке имя переменной заданного класса или выражение, не завершив его «;». Если функция *display* для класса не определена, то MATLAB вызывает стандартную встроенную функцию *display*.

Приведем реализацию метода *display* для класса *graph*.

Листинг oop/@graph/display.m

```
function display(G)

% Метод display класса graph
% display(G) отображает в командном окне информацию о графе G

    disp(' ');
    disp([inputname(1),' = ']);
    disp(' Граф');
    disp([' Число вершин = ' num2str(nvertices(G))]);
    disp([' Число ребер = ' num2str(nedges(G))]);
    disp(' ');
```

Методы для индексации

В этом разделе рассматривается перегрузка методов *subsref*(*a*,*s*), *subsasgn*(*a*,*s*,*b*), *subsindex*(*a*).

Перегрузка функции $b = subsref(a, s)$ позволяет переопределить операторы индексирования, такие, как $a(i)$, $a\{i\}$, *a.fieldname*, а также смешанное индексирование, например, *a.fieldname*(1, **end**, :){4:21, 3, 7:11}.*subfieldname.subsubfieldname* и т.п.

MATLAB вызывает функцию $b = subsref(a, s)$ всякий раз, когда встречает в программе выражение вида $A(i)$, $A\{i\}$, $A.i$ или более сложные индексные выражения (в правой части присваивания), где a — это объект некоторого класса. В этом случае в функции $b = subsref(a, s)$ параметром s будет являться массивом структур с двумя полями:

- $s.type$, содержащим одну из следующих строк: `'()'`, `'{}'` или `'.'`, соответствующую типу индексирования,
- $s.subs$ — массив ячеек или строка, содержащий индексы. Имена полей передаются в виде строк, а символ : — как строка в массиве ячеек.

Рассмотрим некоторые примеры:

Пусть встретилось выражение $a(5, 3{:}6, :)$. В этом случае MATLAB вызывает метод $b = subsref(a, s)$, в котором $s.type =$ `'()'`, $s.subs = \{5, 3{:}6,$ `':'`$\}$.

Пусть встретилось выражение $a\{7, 1{:}9\}$. В этом случае MATLAB вызывает метод $b = subsref(a, s)$, в котором $s.type =$ `'{}'`, $s.subs = \{7, 1{:}9\}$.

Пусть встретилось выражение $a\{:\}$ — MATLAB вызывает метод $b = subsref(a, s)$, в котором $s.type =$ `'{}'`, $s.subs =$ `{':'}`.

Пусть встретилось выражение $a.fieldname$. В этом случае MATLAB вызывает метод $b = subsref(a, s)$, в котором $s.type =$ `'{}'`, $s.subs =$ `'fieldname'`.

Пусть встретилось выражение $a\{1, 2\}.fieldname(3{:}4)$. В этом случае MATLAB вызывает метод $b = subsref(a, s)$, в котором

$s(1).type =$ `'{}'`, $s(1).subs = \{1, 2\}$
$s(2).type =$ `'.'`, $s(2).subs =$ `'fieldname'`
$s(3).type =$ `'()'`, $s(3).subs = \{3{:}4\}$

Функция $subsasgn$ аналогична, но вызывается MATLAB'ом всякий раз, когда индексное выражение находится в левой части от знака присваивания. Например пусть встретилось выражение $a(:) = b$. В этом случае MATLAB вызывает метод $a = subsasgn(a, s, b)$, в котором $s.type =$ `'()'`, $s.subs = \{b\}$.

Иногда необходимо самому пользователю вызывать напрямую методы $subsref$ и $subsasgn$. При этом построить структуру s может помочь функция

$s = substruct(type1, subs1, type2, subs2, ...)$

Например, для выражения $a\{1, 2\}.fieldname(3{:}4)$ нужно воспользоваться командой

$s =$ *substruct*(`'{}'`, {1, 2}, `'.'`, `'fieldname'`, `'()'`, {3:4}))

Функция $i = subsindex(a)$ вызывается, когда встречаются выражения вида $b(a)$, в котором a — объект. Функция $subsindex(a)$ должна возвращать значение в диапазоне от

0 до $prod(size(X)) - 1$, которое интерпретируется как индекс (отсчет ведется с нуля). Этот метод вызывается стандартными функциями *subsref* and *subsasgn*.

Таким образом, с помощью переопределения функций *subsref*(a,s), *subsasgn*(a,s,b), *subsindex*(a) можно эмулировать массивы, массивы ячеек и структуры. Напомним, что все поля любого класса являются частными и доступ к ним, как на чтение, так и на запись не из методов класса закрыт. Приведенные функции позволяют сэмитировать открытость тех или иных полей.

Приведем реализацию методов *subsasgn* и *subsref* для класса *graph*. Эти метода написаны так, что становятся возможными присваивания вида $G.adjmatrix = A$ и $G.edges = E$ и запросы *G.nvertices*, *G.nedges*, *G.adjmatrix*, *G.edges*. Например,

$G = graph(ones(4))$

Получим

$G =$

Граф

Число вершин $= 4$

Число ребер $= 6$

Далее наберем

$n = G.nvertices$

$m = G.nedges$

$A = G.adjmatrix$

$E = G.edges$

В результате будем иметь

$$n = 4, \quad m = 6, \quad A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 3 \\ 1 & 4 \\ 2 & 4 \\ 3 & 4 \end{bmatrix}.$$

Теперь наберем

$G.adjmatrix = ones(3)$

Получим

$G =$

Граф

Число вершин $= 3$

Число ребер $= 3$

Теперь наберем

$G.edges = [1, 2; 2, 3; 3, 4; 1, 4]$

Получим

$G =$

Граф

Число вершин $= 4$

Число ребер $= 2$

Листинг oop/@graph/subsasgn.m

```
function G = subsasgn(G, index, val)

% Метод subsasgn класса graph
% G.adjmatrix = A позволяет задать новую матрицу смежности для графа G
% G.edges = E позволяет задать новый список ребер для графа G

    switch index.type
        case '.'
            switch index.subs
                case 'adjmatrix'
                    G = graph(val);
                case 'edges'
                    G = graph(val);
                case 'nvertices'
                    error('graph.nvertices - возможно только чтение')
                case 'nedges'
                    error('graph.nedges - возможно только чтение')
                otherwise
                    error('Неверное имя поля')
            end
        otherwise
```

```
            error('Неправильный тип индекса')
    end
```

Листинг oop/@graph/subsref.m

```
function val = subsref(G, index)

% Метод subsref класса graph
% G.nvertices возвращает число вершин графа G
% G.nedges возвращает число ребер графа G
% G.adjmatrix возвращает матрицу инциденций графа G
% G.edges возвращает список ребер графа G

    switch index.type
        case '.'
            switch index.subs
                case 'adjmatrix'
                    val = adjmatrix(G);
                case 'edges'
                    val = edges(G);
                case 'nedges'
                    val = nedges(G);
                case 'nvertices'
                    val = nvertices(G);
                otherwise
                    error('Неверное имя поля')
            end
        otherwise
            error('Неправильный тип индекса')
    end
```

2.5.4. Перегрузка функций

Кроме операторов можно перегрузить любую функцию, создав ее новый вариант с тем же именем и разместив соответсвующий файл в папке *@classname*. Когда функция с таким именем вызывается, в первую очередь MATLAB ищет соответсвующий *m*-файл в этой папке.

2.6. Наследование

Напомним один из вариантов вызова функции *class*:

child_obj = *class*(*child_obj*, 'child_class', *parent_obj*);

Здесь мы создаем объект класса *child_class* на основе родительского объекта *parent_obj*. Методы родительского объекта имеют доступ ко всем полям, унаследованным от этого родителя, но не имеют доступа к полям и методам дочернего объекта. В свою очередь дочерний объект не имеет прямого доступа к полям родительского объекта (только через методы). Но методы родительского объекта в дочернем доступны, например, *display*(*c.parentClassname*).

Для иллюстрации наследования рассмотрим небольшой учебный пример. В иерархию будут входить 3 класса: класс *curve* (кривые), *polyline* (ломаные) и *smoothpolyline* (ломаные с закругленными углами). Класс *curve* является родителем *polyline*, который в свою очередь является родителем *smoothpolyline*. См. рис. 2.6. Можно создавать линии, рисовать их, задавать новые координаты характерных точек, новые толщину и цвет линий. Класс *curve* является абстрактным. Вначале приведем небольшую программу-сценарий, использующей эти классы.

Листинг oop/xcurve.m

```
shg
clf
axis([0, 6, 0, 5])
x = [1, 2, 3, 4, 5];
y = [1, 2, 1, 2, 1];
p = polyline(x, y, 1, 'b');
s = smoothpolyline(x, y + 2, 1, 'r');
p = draw(p);
s = draw(s); % См. рис. 2.10
pause


p = setwidth(p, 4);
p = setcolor(p, 'g');
x = [1, 2, 3, 4, 5];
y = [3, 4, 2, 4, 3];
s = setxy(s, x, y);
s = setcolor(s, 'm');
```

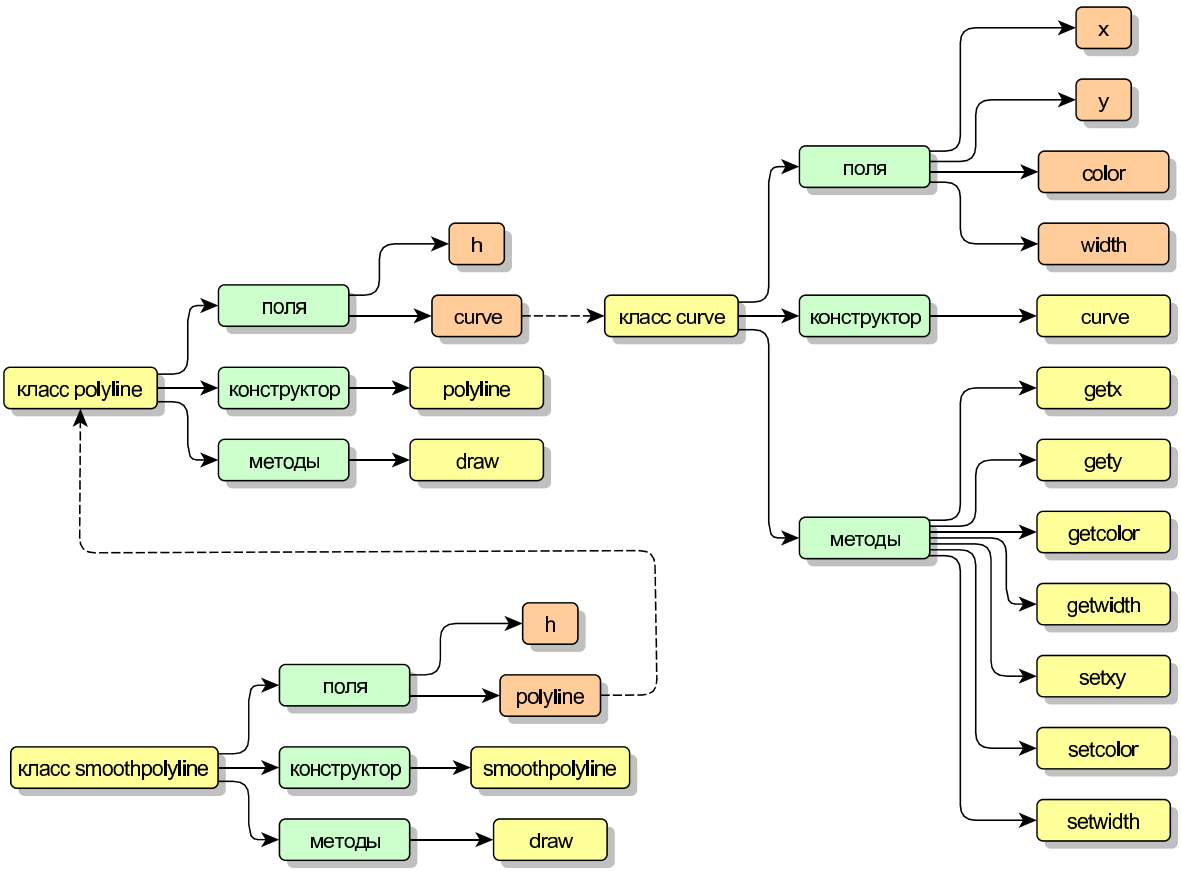


Рис. 2.9. Поля и методы классов *curve*, *polyline*, *smoothpolyline*.

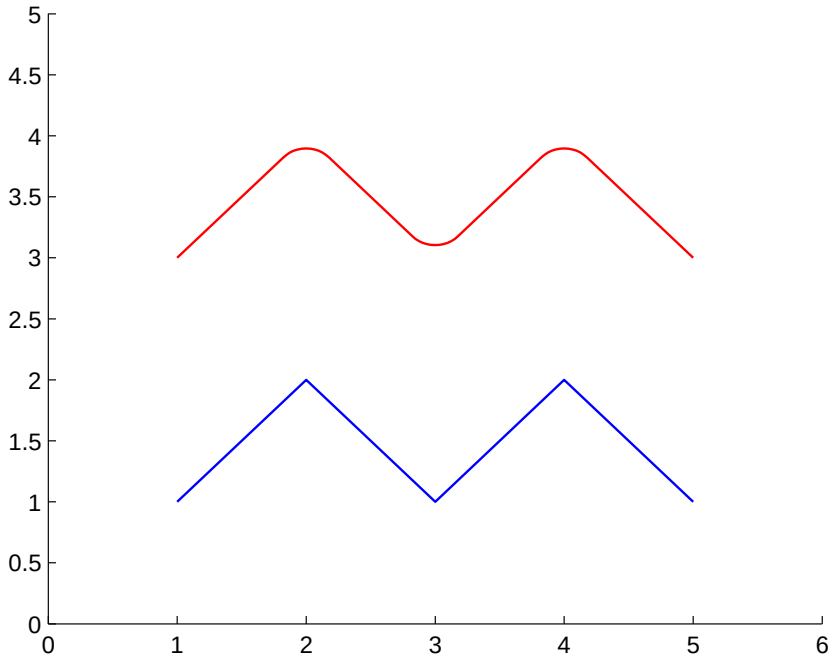


Рис. 2.10.

```
s = setwidth(s, 5);
p = draw(p);
s = draw(s); % См. рис. 2.11
```

Листинг oop/@curve/curve.m

```
function c = curve(varargin)

arg = varargin;

if nargin == 1 && isa(arg{1}, 'curve')
    c = varargin;
else
    if nargin < 1 || isempty(arg{1})
        arg{1} = [];
    end
    if nargin < 2 || isempty(arg{2})
        arg{2} = [];
    end
```

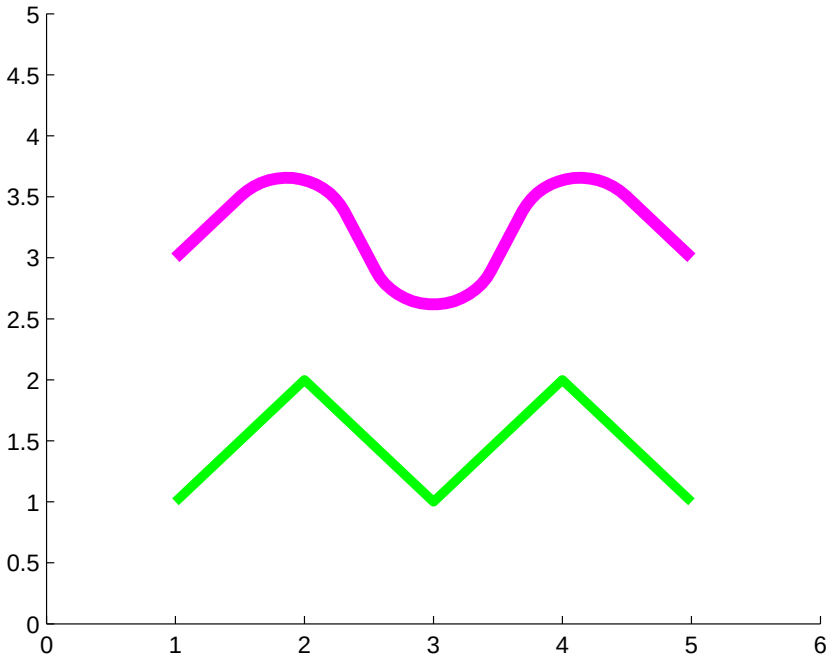


Рис. 2.11.

```
    if nargin < 3 || isempty(arg{3})
        arg{3} = 0.5;
    end
    if nargin < 4 || isempty(arg{4})
        arg{4} = 'b';
    end
    c.x = arg{1};
    c.y = arg{2};
    c.width = arg{3};
    c.color = arg{4};
    c = class(c, 'curve');
end
```

Листинг oop/@curve/getx.m

```
function x = getx(c)

    x = c.x;
```

Листинг oop/@curve/gety.m

```
function y = gety(c)

    y = c.y;
```

Листинг oop/@curve/getcolor.m

```
function col = getcolor(c)

    col = c.color;
```

Листинг oop/@curve/getwidth.m

```
function w = getwidth(c)

    w = c.width;
```

Листинг oop/@curve/setxy.m

```
function c = setxy(c, x, y)

c.x = x;
c.y = y;
draw(c);
```

Листинг oop/@curve/setcolor.m

```
function c = setcolor(c, col)

    c.color = col;
    draw(c);
```

Листинг oop/@curve/setwidth.m

```
function c = setwidth(c, w)

    c.width = w;
    draw(c);
```

Листинг oop/@polyline/polyline.m

```
function p = polyline(varargin)

if nargin == 1 && isa(varargin{1}, 'polyline')
    p = varargin;
else
    c = curve(varargin{:});
    p.h = [];
        % p.h — дескриптор графического объекта
    p = class(p, 'polyline', c);
        % Класс polyline порожден от класса curve
end
```

Листинг oop/@polyline/draw.m

```
function p = draw(p)

if isempty(p.h)
    p.h = line;
end
set(p.h, ...
    'XData', getx(p.curve), ...
    'YData', gety(p.curve), ...
    'Color', getcolor(p.curve), ...
    'LineWidth', getwidth(p.curve));
```

Листинг oop/@smoothpolyline/smoothpolyline.m

```
function s = smoothpolyline(varargin)

if nargin == 1 && isa(varargin{1}, 'smoothpolyline')
    s = varargin;
else
    p = polyline(varargin{:});
    s.h = [];
        % p.h — дескриптор графического объекта
        % Мы могли бы воспользоваться дескриптором родительского объекта,
        % но доступ к нему затруднен
```

```
    s = class(s, 'smoothpolyline', p);
        % Класс smoothpolyline порожден от класса polyline
end
```

Листинг oop/@smoothpolyline/draw.m

```
function s = draw(s)

x = getx(s); % Мы не можем просто написать x = s.polyline.curve.x
y = gety(s); % т.к. все поля частные, т.е. не доступны даже из потомков
[xx, yy] = smooth(x, y);

if isempty(s.h)
    s.h = line;
end
set(s.h, ...
    'XData', xx, ...
    'YData', yy, ...
    'Color', getcolor(s), ...
    'LineWidth', getwidth(s));
```

Обратите внимание, что, например, когда для объектов класса *polyline* или класса *smoothpolyline* вызываются методы, унаследованные от *curve* (например, *setxy*, *setcolor*, *setwidth*), то те в свою очередь вызывают соответствующие методы исходного объекта (метод *draw*).

В случае множественного наследования

```
obj = class(structure,'classname',parent1,parent2,...)
```

к объекту автоматически добавляются поля от каждого родительского класса. Обращение к полям и методам родительских классов происходит аналогично случаю с простым наследованием.

2.7. Где MATLAB ищет нужную функцию?

MATLAB'овские функции размещаются в *m*-файлах, которые в свою очередь, располагаются в различных папках на диске. Таким образом, уникальность имени функции соблюдается только внутри отдельной папки и даже в этом случае возможна ситуация, когда имеются m-файл, вложенная функция и частная функции с одним и тем же име-

нем. Кроме того, в одной папке могут находиться *m*-, *p*- и *mex*- и др. функции с одним и тем же именем. По каким правилам MATLAB определяет, какую функцию ему следует вызывать, если, например, в одном из *m*-файлов встретил обращение к функции *functionname*? Используются следующие правила «старшинства».

1) Подфункции. В первую очередь MATLAB проверяет, нет ли подфункции или вложенной функции с таким именем. Если такая функция имеется, то вызывается именно она.

2) Частные функции. Во вторую очередь, если подфункции или частной функции с заданным именем не найдено, то проверяется папка *private*.

3) Конструктор. Если ни подфункции, ни вложенной, ни частной функций не найдено, то производится попытка найти конструктор с указанным именем, т.е. файл *@functionname/functionname.m*, где папка *@functionname* располагается в папке, прописанной MATLAB'овских путях доступа.

4) Перегруженные методы. При неудачных попытках найти подфункцию, частную или вложенную функцию или конструктор, производится попытка найти метод класса с заданным именем. Поиск производится во всех папках, имеющих префикс @, расположенных в папках, прописанных в путях доступа.

5) Текущая папка. При неудачных предыдущих попытках, MATLAB ищет файл с заданным именем в текущей папке.

6) MATLAB'овские пути доступа. В последнюю очередь MATLAB предпринимает попытку найти указанную функцию в папках, прописанных в путях доступа.

Пути доступа всегда сканируются в обычном порядке. Заметим, что кроме *m*-файлов, функции могут быть представлены в *p*-файлах, *mex*-файлах и т.п. Приоритет здесь следующий:

1) встроенные (*.bi*) функции,

2) *mex*-файлы,

3) *mdl*-файлы (модели Simulink),

4) *p*-файлы,

5) *m*-файлы.

Определить, какая функция будет вызвана, можно с помощью команды

which functionname

которая возвращает полный путь к найденной функции.

При перегрузке методов и операторов, возникает вопрос, какой из версий того или иного метода или оператора будет вызываться в той или иной ситуации. Пусть, например, у нас есть класс *polynom*, представляющий многочлены и класс *rational*, представляющие дробно-рациональные функции. Метод какого класса будет вызван, если в программе встретится выражение вида $p + r$, где p и r — объекты классов *polynom* и *rational* соответсвенно?

Если вызван метод, для которого есть несколько версий с одинаковым именем, то MATLAB руководствуется следующими правилами:

1) среди аргументов находит аргумент, принадлежащий классу с наивысшим приоритетом.

2) производит поиск среди функций с таким именем.

Управляющим аргументом является аргумент с наивысшим приоритетом или, в случае равенства приоритетов, самый левый из них.

Пусть, например, p принадлежит к классу *polynom*, а r — классу *rational*. И пусть оператор + перегружен для обоих классов. Если, скажем, приоритет класса *rational* выше, чем приоритет *polynom*, то для обработки каждого из выражений $p + r$ или $r + p$ будет вызван метод *plus* класса *polynom*. Если приоритеты равны, то выражение $r + p$ поступит на обработку методу *plus* класса *rational*, а выражение $p + r$ — методу *plus* класса *polynom*.

Приоритет можно задать с помощью функций *superiorto* и *inferiorto*. Эти функции допустимо вызывать только в конструкторе. Команда

superiorto(`'class1'`, `'class2'`, ...)

означает, что данный класс (в конструкторе которого встретилась эта команда) имеет более высокий приоритет, чем классы *class1*, *class2* и т.д. Команда

inferiorto(`'class1'`, `'class2'`, ...)

означает, наоборот, что данный класс в иерархии приоритетов, располагается ниже классов, указанных в списке аргументов. Необходимо отличать эту иерархия от иерархии наследования классов.

3. Динамические структуры данных

В данной главе описывается библиотека Pointer (*http://code.google.com/p/pointer/*), позволяющая работать в MATLAB'е со сложными динамическими структурами данных. Ядром библиотеки является класс *pointer*, экземпляры которого суть аналоги ссылочных объектов в C++ и Java. Почти все методы этого класса являются *mex*-функциями. Алгоритмы для работы со структурами данных реализованы как *m*-функции. Библиотека Pointer является свободной и распространяется по лицензии GNU General Public License - version 2 (http://www.gnu.org/licenses/gpl.txt). Идея создания в Matlab'е ссылочного класса была почерпнута из библиотеки DSATBX [13] (автор - Yaron Keren, MathWorks Inc.). Эта библиотека содержит реализацию класса *pointer* и некоторые алгоритмы работы со структурами данных (списками, очередями, стеками, деревьями поиска). К сожалению, исходные коды *mex*-функций этой библиотеки не доступны, а имеющиеся *dll*-файлы работают только с вер-сиями Matlab'а 6.x. С другой стороны, сами алгоритмы для работы со структурами данных реализованы в DSATBX как *m*-функции. Класс *pointer* из библиотеки Pointer поддерживает всю функциональность аналогичного класса из DSATBX. Таким образом, возможно использование структур данных из DSATBX с классом *pointer* (в том числе в версиях MATLAB'а 7.x и более поздних). Заметим, что схожие идеи — по реализации в MATLAB'е ссылочного класса — реализованы в библиотеке Inplace [9].

3.1. Класс *pointer*

Объекты класса *pointer* ведут себя как ссылки (references) в C++ или Java и могут рассматриваться как разные наименования (псевдонимы) одной и той же порции данных. Таким образом, два (и более) объекта этого класса могут «указывать» на одно и то же содержимое. В текущей версии библиотеки само это содержимое может быть структурой или массивом ячеек (но не обычным массивом). Реализованы следующие операции.

- Создание («объявление») объекта класса pointer:

  $p = pointer$
  $p = malloc$

Данные функции (они эквивалентны) присваивают объекту p значение $NULL$, означающее, что p указывает на пустой объект. Чтобы объявить сразу несколько объектов, можно воспользоваться командой *pointers*:

$pointers\ p1\ p2\ \ldots\ pn$

Конвертирование структуры или массива ячеек в объект класса *pointer* осуществляется одноименной функцией с одним входным параметром:

$p = pointer(s)$

- Конвертирование объекта класса pointer в структуру и массив ячеек:

  $s = struct(p)$
  $c = cell(p)$

- Присваивание одного объекта класса pointer другому:

  $b = a$

  приводит к тому, что b и a указывают на одну и ту же порцию данных.

- В отличие от оператора присваивания функция *copy* приводит к копированию данных:

  $b = copy(a)$

  Данные, на которые указывает a, копируются. Затем b становится наименованием этой новой порции данных.

- Реализованы операции сравнения:

  $b == a$
  $b \tilde{}= a$

  При этом два объекта класса pointer считаются равными, если и только если они указывают на одни и те же данные. Сравнение с 0:

  $p == 0$
  $p \tilde{}= 0$

  означает проверку на совпадение/несовпадение p со значением $NULL$.

- Функция

  $free(p)$

освобождает память, на которую указывает p, и присваивает p значение $NULL$. Заметим, что обычная команда

$clear\ p$

не освобождает память, на которую указывает p. К сожалению, возможности языка системы MATLAB, по-видимому, не позволяют переопределить функцию clear, а также реализовать автоматический сборщик мусора.

- Обращение к полям и ячейкам осуществляется обычным образом, например:

$pointers\ a\ b$
$a.field1 = value1$
$a.field2 = value2$
$a.data(2, 4) = value3$
$b\{1\} = value1$
$b\{2{:}5, \mathbf{end}\} = value2$

Реализованный класс *pointer* полностью совместим с классом *pointer* из библиотеки DSATBX [13]. Более того, исправлены замеченные ошибки и реализованы дополнительные возможности. Например, сейчас возможна корректная работа с «многоуровневым» доступом к данным:

$a = malloc$
$a.next = malloc$
$a.next.next = malloc$
$a.next.next.next = malloc$

3.2. Примеры

В качестве простого примера рассмотрим создание однонаправленного списка:

Листинг pointer/xpointer.m

```
% Массив с данными:
data = {
    'Isaac Newton',
    'Carl F. Gauss',
    'Evariste Galois',
    'Blaise Pascal',
```

```
    'Georg F. L. Ph. Cantor',
    'Nikolai I. Lobachevski'};
n = length(data);

% Инициализация элементов списка:
prev = 0;
cur = 0;
for j = 1:n
    cur = malloc;
    cur.data = data{j};
    cur.next = prev;
    prev = cur;
end

% Печать элементов списка:
list = cur;
while (cur ~= 0)
    disp(cur.data);
    cur = cur.next;
end

% Удаление элементов списка:
cur = list;
while (cur ~= 0)
    prev = cur;
    cur = cur.next;
    free(prev);
end
```

Приведенная выше программа напечатает следующее:

Nikolai I. Lobachevski
Georg F. L. Ph. Cantor
Blaise Pascal
Evariste Galois
Carl F. Gauss
Isaac Newton

В качестве еще одного примера рассмотрим известную игру «Животные». Компьютер предлагает человеку загадать животное, а затем с помощью вопросов, допускающих ответ «да»/«нет», пытается его отгадать. Если животное угадано не правильно, то компьютер предлагает человеку ввести вопрос, ответы на который для животных, загаданного человеком и названного компьютером, различны.

Программа является обучающейся в том смысле, что этот вопрос запоминается в базе всех вопросов, а затем используется при угадывании других животных. База вопросов представляется корневым бинарным поисковым деревом, каждой неконцевой вершине которого приписаны вопросы, а двум исходящим из вершины дугам соответствуют ответы «да» и «нет». Концевым вершинам приписаны животные. Алгоритм, используемый компьютером, заключается в движении по этому дереву от корня к одной из концевых вершин. Если животное угадано не правильно, то компьютер предлагает ввести вопрос, после чего дерево достраивается следующим образом. К вершине, в которой мы оказались, приписывается вопрос, предложенный человеком, и добавляются две новые дуги. Двум новым концевым вершинам приписываются животные, загаданное человеком и предложенное компьютером.

Сначала приведем распечатку диалога человека и компьютера (часть несущественной информации опущена):

Задумайте животное!
Вы готовы? −> *да*
Оно домашнее? −> *да*
На нем ездят? −> *нет*
Дает молоко? −> *да*
Оно крупное? −> *нет*
Это КОЗА
Верно? −> *да*

Задумайте животное!
Вы готовы? −> *да*
Оно домашнее? −> *нет*
Оно летает? −> *нет*
Оно живет в море? −> *да*
Это КИТ
Верно? −> *нет*
Так, что же вы задумали? *Акула*

Введите утверждение, верное для КИТ и неверное для АКУЛА,
или наоборот: верное для АКУЛА, но неверное для КИТ.
Например: у него есть хобот
−> Это млекопитающее
Какой правильный ответ для АКУЛА? −> нет
Задумайте животное!
Вы готовы? −> нет
Сохранить изменения? −> да

Вершины дерева вопросов в программе представляются объектами класса *pointer* со следующими полями:

- *yes*, *no* указывают на два поддерева
- *question* содержит приписанный данной вершине вопрос

У концевых вершин поля *yes* и *no* отсутствуют, а поле *question* содержит имя животного.

Приведем полный код:

Листинг pointer/animals.m

```
function animals

% Игра «Животные»

    disp('**** Ж И В О Т Н Ы Е ****');
    if exist('animals.mat', 'file')
        load animals;
        Tree = readtable(Table);
    else
        disp('Файл animals.mat не найден');
        disp('База пуста');
        Tree = [];
    end

    Tree = game(Tree);

    if input_ans('Сохранить изменения')
        Table = writetable(Tree);
```

```
        save animals Table;
    end

    release(Tree);
end

% Основной алгоритм реализует функция root = game(root).
% На ее вход подается корень поискового дерева. На выходе — корень дерева после одного сеанса игры.

function root = game(root)

    if nargin < 1 || isempty(root)
        root = malloc;
        root.question = 'КОТ';
    end

    while 1
        cur = root;

        disp('');
        disp('Задумайте животное!');
        if ~input_ans('Вы готовы?')
            return
        end

        while isfield(struct(cur), 'yes') && isfield(struct(cur), 'no')
            if input_ans([cur.question '?'])
                cur = cur.yes;
            else
                cur = cur.no;
            end
        end

        animal = cur.question;
        disp(['Это ', animal]);
```

```
        if ~input_ans('Верно?')
            user_animal = upper(input_str('Так, что же вы задумали? '));
            disp(['Введите утверждение, верное для ' animal ...
                ' и неверное для ' user_animal ',']);
            disp(['или наоборот: верное для ' user_animal ...
                ' но неверное для ' animal '.']);
            disp('Например: у него есть хобот');
            question = input_str('-> ');

            cur.question = question;
            cur.yes = malloc;
            cur.no = malloc;

            if input_ans(['Какой правильный ответ для ' user_animal '?'])
                cur.yes.question = user_animal;
                cur.no.question = animal;
            else
                cur.yes.question = animal;
                cur.no.question = user_animal;
            end
        end
    end
end

% Следующие две функции используются при сохранении поискового дерева на
% диске и чтении его с диска. На диске дерево хранится как специальный
% массив структур с двумя полями yesno и question.
% Функция tbl = writetable(root) обходит дерево поиска, заданное корнем
% root, и запоминает всю информацию в массиве структур tbl.
% Реализован обычный рекурсивный алгоритм обхода (поиск в глубину).

function tbl = writetable(root)
    tbl(1).yesno = NaN;
    iterate(root)
```

```
    function iterate(cur)
        tbl(end).question = cur.question;

        if isfield(struct(cur), 'yes') && isfield(struct(cur), 'no')
            tbl(end + 1).yesno = 1;
            iterate(cur.yes)

            tbl(end + 1).yesno = 0;
            iterate(cur.no)
        end
    end
end

% Функция root = readtable(tbl) по таблице tbl строит соответствующее
% поисковое дерево и возвращает его корень root.

function root = readtable(tbl)

    root = malloc;
    i = 0;
    do_readtable(root);

    function do_readtable(cur)
        i = i + 1;
        cur.question = tbl(i).question;

        if i < length(tbl) && tbl(i + 1).yesno == 1
            cur.yes = malloc;
            do_readtable(cur.yes);
            cur.no = malloc;
            do_readtable(cur.no);
        end
    end
end
```

```
% Функция release(root) освобождает память, занимаемую
% поисковым деревом с корнем root.

function release(root)

    do_release(root)

    function do_release(cur)
        if isfield(struct(cur), 'yes') && isfield(struct(cur), 'no')
            do_release(cur.yes);
            do_release(cur.no);
        else
            free(cur);
        end
    end
end

% Следующие две функции являются вспомогательными и предназначены
% для организации диалога между компьютером и человеком.

function answer = input_ans(prompt)
    list = 'дн';
    default = 'д';
    while 1
        ch = input([prompt ' Д/Да/Н/Нет [Да]-> '], 's');
        if isempty(ch)
            ch = default;
        end
        ch = lower(ch(1));
        if ~isempty(find(list == ch), 1)
            break
        end
    end
    if ch == 'д'
        answer = 1;
```

```
    else
        answer = 0;
    end
end

function answer = input_str(prompt)
    while 1
        answer = input(prompt, 's');
        if ~isempty(answer)
            return
        end
    end
end
```

3.3. Замечания о производительности

Класс *pointer* показывает хорошие результаты при программировании динамических структур данных. Рассмотрим две простые реализации стека: на основе массивов и на основе односвязных списков. Очевидно, что вставка в стек элемента, в котором уже находится n элементов, в случае реализации с помощью массивов может потребовать времени $O(n)$ (необходимо перепаковать массив). Тогда как вставка в стек, реализованного на основе списка, требует времени $O(1)$. Если же необходимо вставить в первоначально пустой стек n элементов, то оценки нужно заменить соответственно на $O(n)$ и $O(n^2)$. Следующая программа осуществляет соответствующий эксперимент:

Листинг pointer/xstack.m

```
N = 100000;

timearr = zeros(N, 1);
timelst = zeros(N, 1);

tic
arr = [];
for n = 1:N
    arr(end + 1) = n;
    timearr(n) = toc;
```

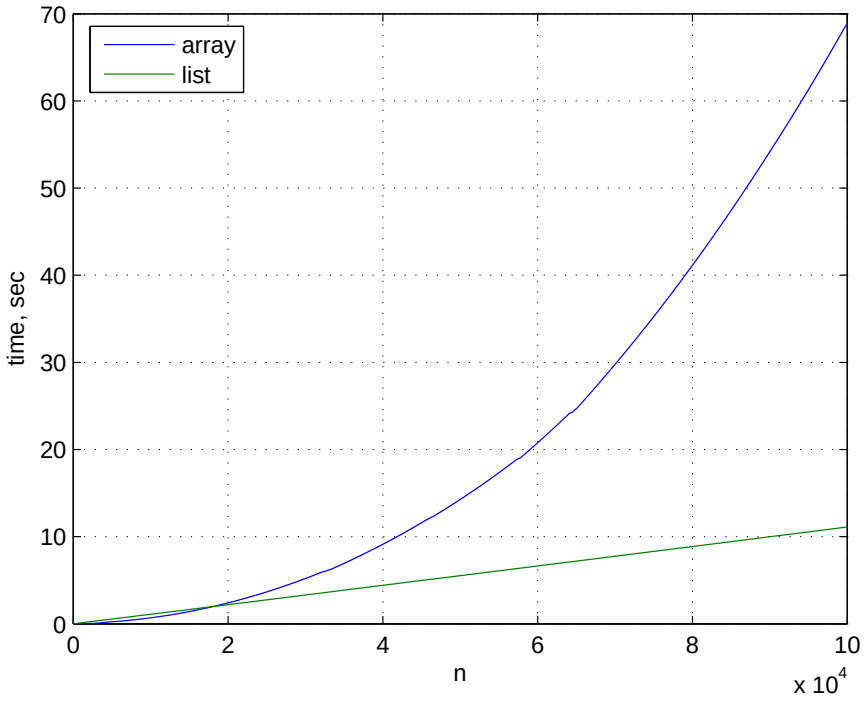


Рис. 3.1. Вставка n элементов в пустой стек. Реализации на основе массивов и списков

```
end

tic
lst = malloc;
cur = lst;
for n = 1:N
    cur = malloc;
    cur.data = n;
    cur.next = lst;
    lst = cur;
    timelst(n) = toc;
end

plot(1:N, [timearr, timelst])
grid on
legend('array', 'list', 2)
xlabel('n')
ylabel('time, sec')
```

Результирующие графики приведены на рис. 3.1. Эксперимент проводился на машине Intel Pentium M, 1.6 GHz, 512 M с установленной ОС Windows XP Home Edition. Использовался MATLAB 7 (R14). При $n > 20000$ выгоднее использовать списки. Аналогичные результаты можно получить для операции удаления элементов из стека.

В разобранном примере, несмотря на то, что общий объем всех данных был велик, каждый объект класса *pointer* ссылался только на единственное число класса *double*. Если же данные, на которые ссылается один объект, занимают большой объем, то работа, к сожалению, не столь эффективна. Для примера сравним время доступа к элементам массива 1000×1000 обычным образом и «через указатель»:

Листинг pointer/xpointer1000.m

```
n = 1000;
tic
A = zeros(n);
for i = 1:n
    A(i, i) = i;
end
toc

p = malloc;
tic
p.data = zeros(n);
for i = 1:n
  p.data(i, i) = i;
end
toc
free(p)
```

На машине с приведенными выше характеристиками время работы составило 0.10 с и 22.24 с соответственно. Заметим, что ранние версии библиотеки Pointer работали еще (в несколько десятков раз) медленнее. К сожалению, доступные возможности языка не позволяют, по-видимому, дальше ускорить такого сорта вычисления.

Таким образом, в настоящее время объекты класса Pointer разумно использовать при программировании структур данных со многими сложными связями, но с небольшой нагрузкой на узлах.

3.4. Списки и поисковые деревья (библиотека DSATBX)

Библиотека DSATBX [13] реализует операции для работы со следующими структурами данных:

- простые (односвязные) списки,
- двусвязные списки,
- стеки,
- очереди,
- бинарные деревья,
- красно-черные бинарные деревья,
- АВЛ-деревья.

Определения структур данных и алгоритмы на них см., например, в [1, 2, 3, 7].

Для работы с простыми списками реализованы следующие функции:

- $sl = sl\_new$ — создает новый список
- $sl = sl\_appnd(sl, data)$ — добавляет данные к концу списка
- $cnt = sl\_count(sl)$ — возвращает число узлов в списке
- $sl = sl\_del(sl)$ — удаляет первый элемент в списке
- $sl\_disp(sl)$ — печатает все элементы списка (используя sl_trav: $sl\_trav(sl,$ `'disp'`$)$)
- $b = sl\_empty(sl)$ — возвращает 1 (логическую истину), если список пустой, и 0 в противном случае
- $sl = sl\_free(sl)$ — освобождает память
- $data = sl\_get(sl)$ — возвращает первый элемент
- $sl = sl\_insrt(sl, data)$ — добавляет данные в начало списка
- $sl\_trav(sl, func, varargin)$ — итератор по всем узлам списка; к каждому элементу будет применена функция $func$

Для работы с двойными списками реализованы следующие функции:

- $sl = sl\_new$ — создает новый список
- $dl = dl\_del(dl, node)$ — удаляет узел $node$ в списке
- $sl = sl\_appnd(sl, data)$ — добавляет данные к концу списка
- $cnt = sl\_count(sl)$ — возвращает число узлов в списке
- $dl\_disp(dl)$ — печатает все элементы списка (используя dl_trav: $dl\_trav(dl, \texttt{'disp'})$)
- $b = dl\_empty(dl)$ — возвращает 1 (логическую истину), если список пустой, и 0 в противном случае
- $sl = sl\_free(sl)$ — освобождает память
- $data = sl\_get(sl)$ — возвращает первый элемент
- $sl = sl\_insrt(sl, data)$ — добавляет данные в начало списка
- $sl\_trav(sl, func, varargin)$ — итератор по всем узлам списка; к каждому элементу будет применена функция $func$

В качестве примера рассмотрим некоторые операции с простыми списками. Работа с двусвязными списками, стеками и очередями аналогична.

Следующая программа читает из файла свой текст строка за строкой и размещает эти строки в список. Затем строки печатаются (в обратном порядке) на экран.

Листинг pointer/xsl.m

```
sl = sl_new;

fid = fopen('xsl.m', 'r'); % Открываем файл на чтение
while ~feof(fid)
    line = fgetl(fid); % Читаем строку
    if isempty(line);
        line = ' ';
    end
    sl = sl_put(sl, line); % Размещаем строку в списке
end
fclose(fid);

disp('Печать содержимого файла (в обратном порядке)');
```

$sl\_\,disp(sl)$;

$sl = sl\_\,free(sl)$;

В примере использовались следующие функции:

- $sl = sl\_\,new$ — создает новый список
- $sl = sl\_\,put(sl, x)$ — размещаем элемент x в списке sl
- $sl\_\,disp(sl)$ — печать элементов списка
- $dsl = sl\_\,free(sl)$ — освобождает память, занимаемую списков sl

Теперь рассмотрим некоторые операции с АВЛ-деревьями. Работа с произвольными бинарными деревьями и красно-черными деревьями аналогична.

Узел каждого дерева представлен объектом типа *pointer* со следующими полями:

- *key* — ключ. Может иметь произвольный тип данных, для которого определена операция $<$. Как правило, целое число.
- *data* — данные (информационное поле). Может иметь произвольный тип данных.
- *left* — левое поддерево. Тип — *pointer*.
- *right* — правое поддерево. Тип — *pointer*.

Красно-черные и АВЛ-деревья имеют другие поля. Внимание: следует различать понятия «узел», «ключ» и «информационное поле» («данные»).

Функции для работы с бинарными деревьями начинаются с префикса $bt\_$, с красно-черными деревьями — с префикса $rb\_$, с АВЛ-деревьями — с префикса $avl\_$. Приведем список доступных операций на примере обычных бинарных деревьев. Для других типов деревьев доступны анлогичные функции.

- $bt = bt\_\,new$ создает новое дерево
- $bt = bt\_\,free(bt)$ освобождает память, занимаемую указанным деревом
- $bt\_\,graph(bt)$ — визуализация дерева
- $bt\_\,inord(bt, fn)$, $bt\_\,preor(bt, fn)$, $bt\_\,posto(bt, fn)$ — различные алгоритмы обхода дерева (во внутреннем, или концевом, прямом и обратном порядках — используя терминологию из [6, 2]); к данным, расположенным в каждом узле применяется функция fn

- $bt = bt\_put(bt, data, key)$ размещает в дереве новый узел с ключом key и данными $data$
- $node = bt\_find(bt, key)$ — поиск в дереве узла с ключом key и данными $data$. Функция возвращает соответсвующий узел. В случае неудачи возвращается 0.
- $bt = bt\_del(bt, node)$ их дерева bt удаляет узел $node$

Рассмотрим пример.

Листинг pointer/treedemo.m

```
Months = {
    'January', 'February', 'March', 'April', ...
    'May', 'June', 'July', 'August', ...
    'September', 'October', 'November', 'December'};

% Размещаем данные в обычном бинарном дереве:
bt = bt_new;
for i = 1:12
    bt = bt_put(bt, Months{i}, i);
end
node = bt_find(bt, 1); % Поиск элемента с ключом 1
node.data = 'January is the first month in a year'; % Меняем данные
bt_graph(bt); % Рисуем дерево. См. рис. 3.2
bt = bt_free(bt);
pause;

% Теперь запишем данные в дерево в случайном порядке:
bt = bt_new;
rand('state', 0)
p = randperm(12);
for i = 1:12
    bt = bt_put(bt, Months{p(i)}, p(i));
end
node = bt_find(bt, 1);
node.data = 'Jan';
bt_graph(bt); % Рисуем дерево. См. рис. 3.3
```

```
bt = bt_free(bt);
pause

% Размещаем данные в красно-черном дереве:
rb = rb_new;
for i = 1:12
    rb = rb_put(rb, Months{i}, i);
end
rb_graph(rb); % Рисуем дерево. См. рис. 3.4
rb = rb_free(rb);
pause;
rb = rb_del(rb, rb_find(rb, 3));
rb = rb_del(rb, rb_find(rb, 8));
rb_graph(rb); % Рисуем дерево. См. рис. 3.5
pause;

% Размещаем данные в АВЛ-дереве:
avl = avl_new;
for i = 1:12
    avl = avl_put(avl, Months{i}, i);
end
disp('Обход во внутреннем порядке:')
avl_inord(avl, 'disp');

disp('Обход в прямом порядке:')
avl_preor(avl, 'disp');

disp('Обход в обратном порядке:')
avl_posto(avl, 'disp');

avl_graph(avl); % Рисуем дерево. См. рис. 3.6
pause

% Удаляем некоторые узлы:
avl = avl_del(avl, avl_find(avl, 6));
```

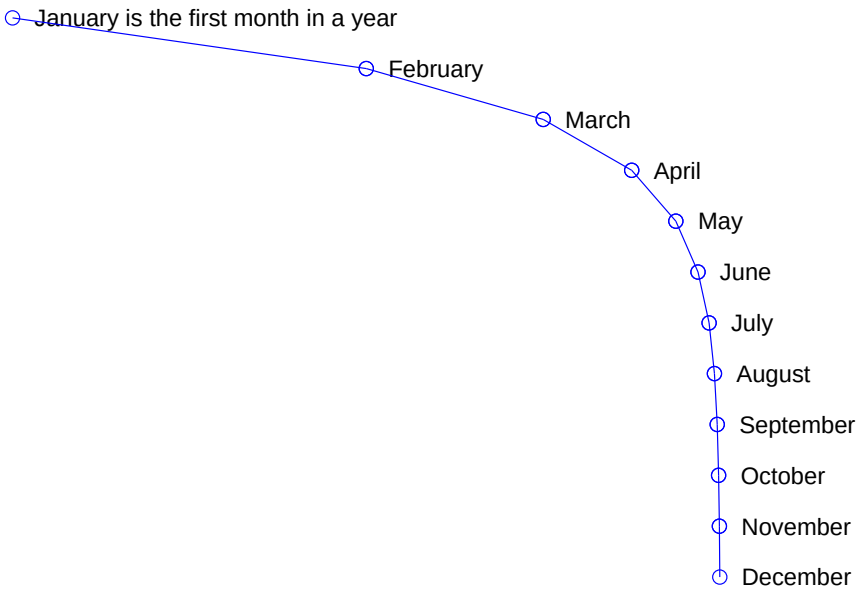


Рис. 3.2. (Крайне) не сбалансированное поисковое дерево

$avl = avl\_del(avl, avl\_find(avl, 7));$
$avl\_graph(avl);$ % Рисуем дерево. См. рис. 3.7
$pause$
$avl = avl\_del(avl, avl\_find(avl, 8));$
$avl\_graph(avl);$ % Рисуем дерево. См. рис. 3.8
$avl = avl\_free(avl);$

Обход во внутреннем порядке:
January
February
March
April
May
June
July
August
September
October
November

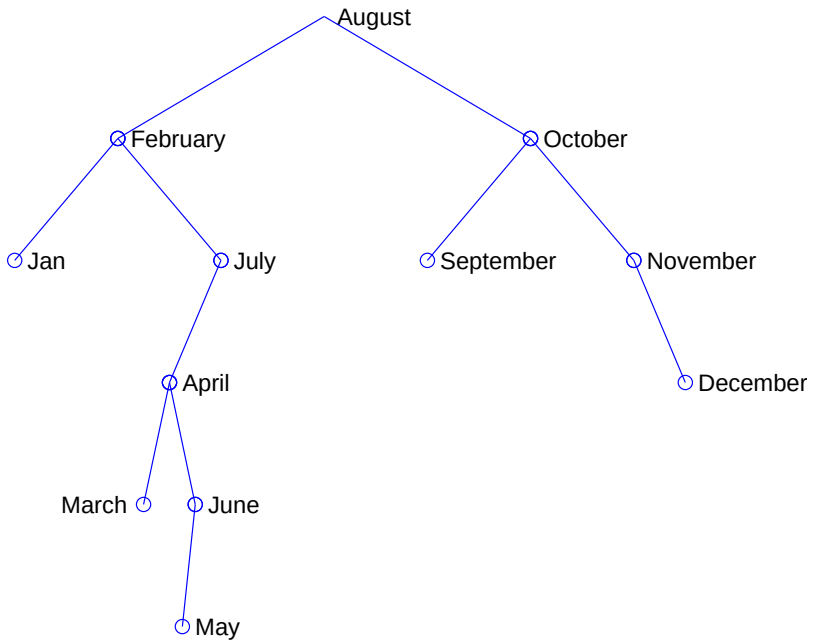


Рис. 3.3. Случайное поисковое дерево

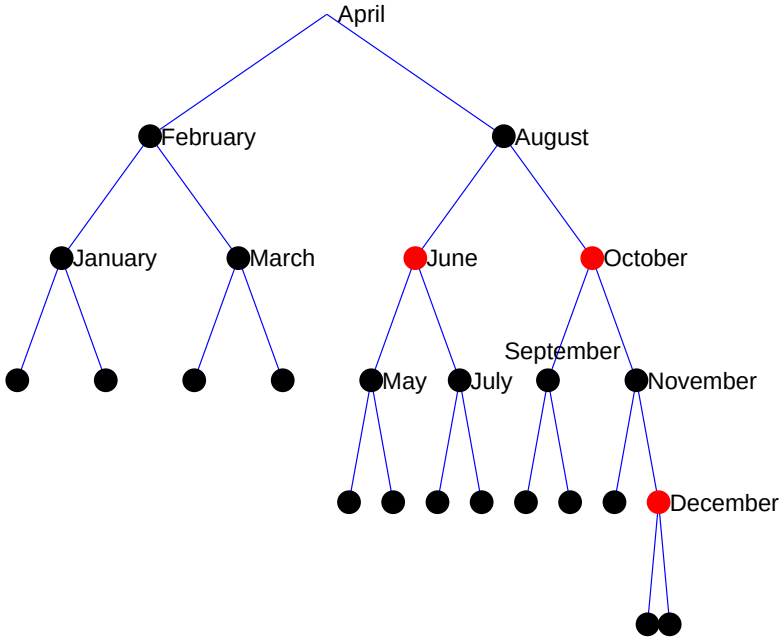


Рис. 3.4. Красно-черное поисковое дерево. Фиктивные (*nil*) ребра также отражены

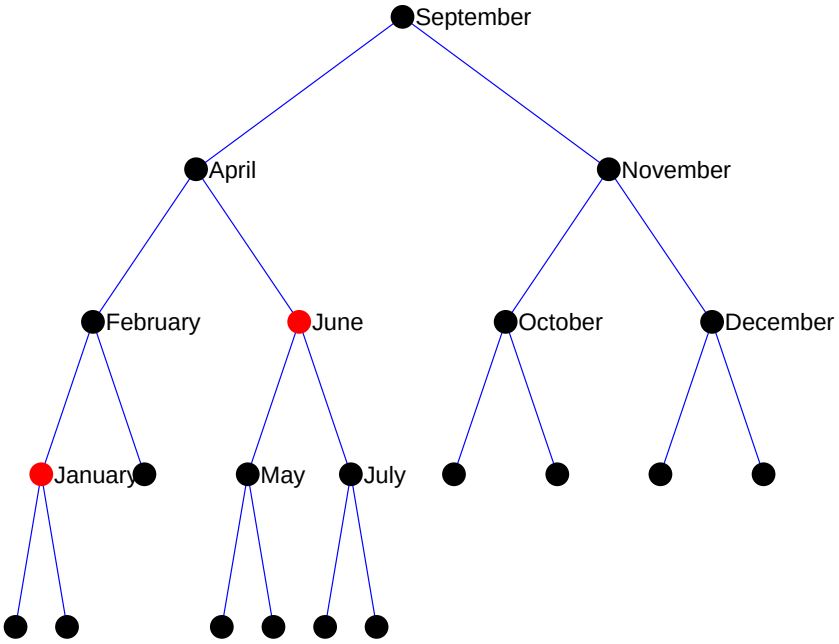


Рис. 3.5. Предыдущее дерево после удаления узлов *March*, *August*

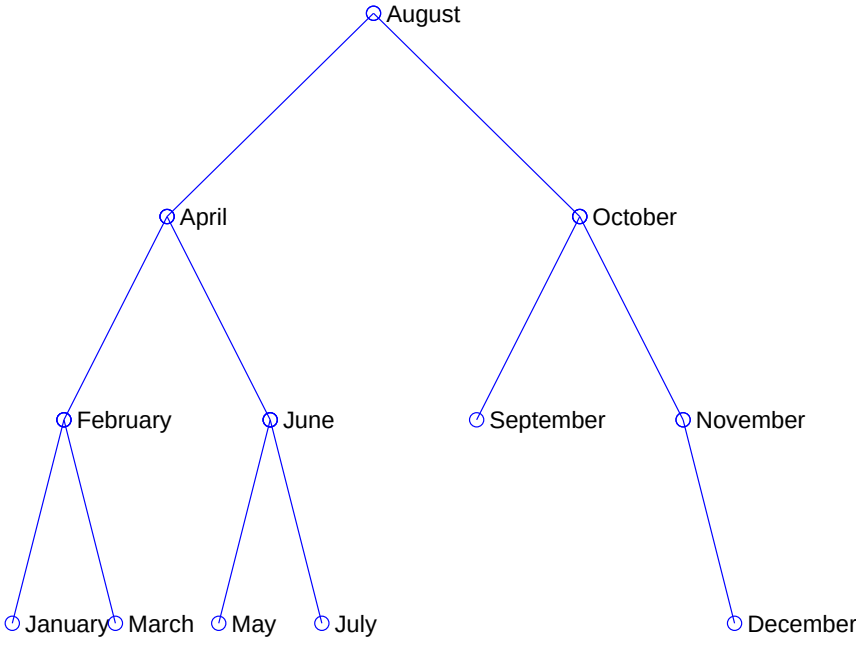


Рис. 3.6. АВЛ-дерево

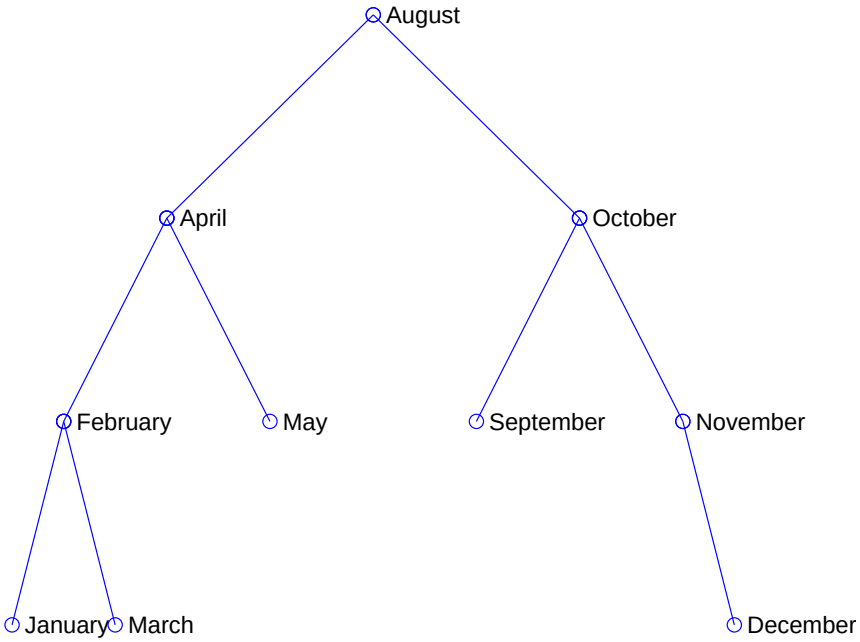


Рис. 3.7. Предыдущее дерево после удаления узлов *June*, *July*

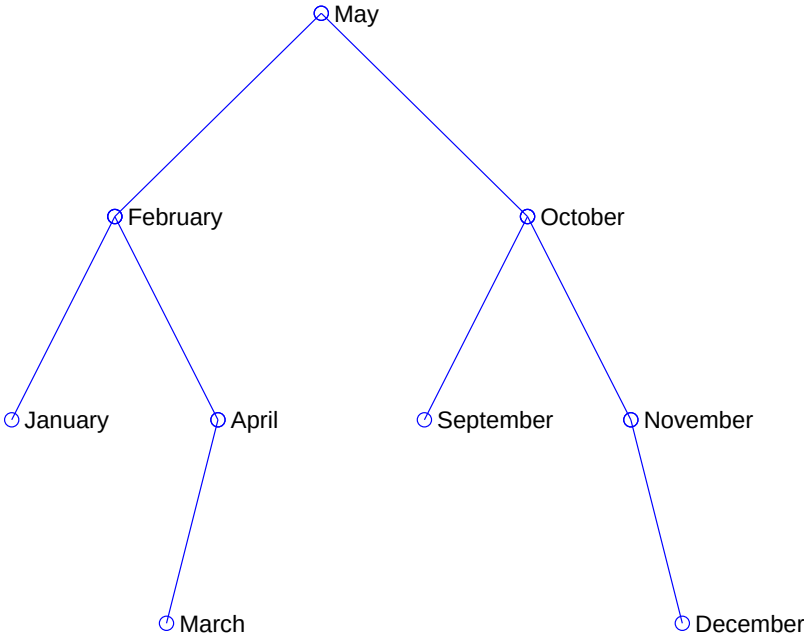


Рис. 3.8. Предыдущее дерево после удаления узла *August*

December

Обход в прямом порядке:
August
April
February
January
March
June
May
July
October
September
November
December

Обход в обратном порядке:
January
March
February
May
July
June
April
September
December
November
October
August

3.5. Разделенные множества

Система разделенных (или *непересекающихся*) *множеств* — это абстрактный тип данных для представления коллекции, состоящей из некоторого числа попарно непересекающихся подмножеств. Основные операции над разделенными множествами следу-

ющие:

- $create(x)$ — создание подмножества, содержащего единственный элемент x. Именем созданного подмножества будет считаться сам элемент x.
- $merge(x,\ y)$ — объединение подмножеств с именами x и y. При этом оба объединяемые подмножества удаляются из коллекции, а вновь созданное подмножество в качестве имени получает одно из имен x или y.
- $y = find(x)$ — определение имени y того подмножества, которому принадлежит элемент x.

В библиотеке POINTER реализованы разделенные множества на основе следующих структур хранения: массив, древовидная структура, древовидная структура с использованием рангов вершин, древовидная структура с использованием ранга вершин и сжатия путей. Эффективные реализации этих структур возможны на обычных массивах и не требуют использования указателей (см., например, в [1, 2, 8]).

В качестве примера рассмотрим задачу выделения компонент связности неориентированного графа.

Листинг pointer/graphcomp.m

```
function graphcomp

E = [1, 5; 5, 9; 9, 6; 6, 1; 1, 9; 3, 2; 7, 8; 8, 4; 4, 7];
x = [0, 2, 2, 3, 1, 0, 3, 4, 1];
y = [0, 0, 1, 1, 0, 1, 0, 0, 1];
G = graph(E, x, y)
draw(G) % Рисуем граф; см. рис. 3.9
pause

djs = components(E);
% Рисуем дерево, представляющее систему разделенных множеств
% см. рис. 3.10:
djs_graph(djs);

for j = 1:max(E(:))
    % Находим подмножество, которому принадлежит j:
    [name, djs] = djs_find(djs, j);
```

```
    A(j, :) = [j, name];
end
disp('Принадлежность вершин компонентам связности:')
disp(A)

function djs = components(E)

n = max(E(:)); % Число вершин
m = size(E, 1); % Число ребер
djs = djs_new;
for j = 1:n
    djs = djs_create(djs, j); % Создаем синглетоны
end

for i = 1:m
    u = E(i, 1);
    v = E(i, 2);
    [superu, djs] = djs_find(djs, u); % Нашли множество, содержащее u
    [superv, djs] = djs_find(djs, v); % Нашли множество, содержащее v
    if superu ~= superv
        djs = djs_merge(djs, superu, superv); % Объединяем подмножества
    end
end
```

Принадлежность вершин компонентам связности:

```
1 5
2 2
3 2
4 8
5 5
6 5
7 8
8 8
9 5
```

В примере использовались следующие функции:

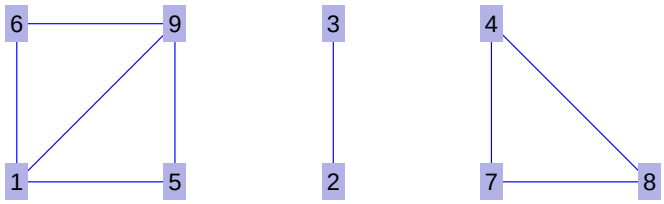


Рис. 3.9. Тестовый граф для выделения компонент связности

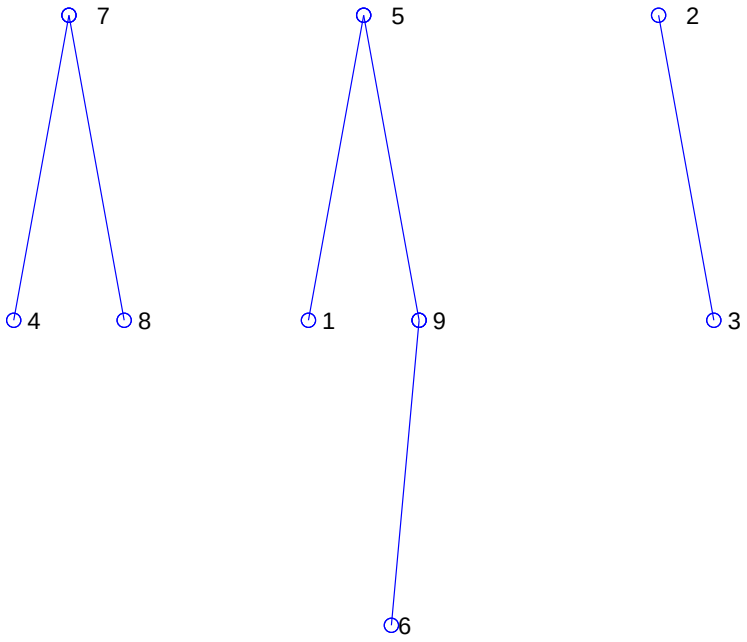


Рис. 3.10. Представление разделенного множества деревом

- $djs = djs\_new$ — создает новое множество
- $djs = djs\_create(djs, name)$ — создает в множестве djs одноэлементное подмножество с именем $name$
- $djs = djs\_put(djs, data, name)$ — добавляет в подмножество с именем name новый элемент $data$
- $[name, djs] = djs\_find(djs, data)$ — находит имя подмножества, которому принадлежит элемент $data$
- $djs = djs\_free(djs)$ — освобождает память, занимаемую множеством djs

3.6. Приоритетные очереди

Приоритетная очередь — это абстрактный тип данных, предназначенный для представления *взвешенных множеств*. Множество называется *взвешенным*, если каждому его элементу однозначно соответствует число, называемое *ключом* или *весом* (см., например, [1, 2, 8]). Следующие операции над приоритетными очередями являются основными:

- $put(x, key)$ — создание в множестве нового элемента x со своим ключом key.
- $y = find\_min$ — поиск в множестве элемента с минимальным ключом. Если элементов с минимальным ключом несколько, то находится один из них. Найденный элемент не удаляется из множества.
- $y = del\_min$ — удаление из множества элемента с минимальным ключом. Если элементов с минимальным ключом несколько, то удаляется один из них.

Также рассматриваются дополнительные операции:

- $Z = merge(X, Y)$ — объединение двух множеств в одно.
- $decrease\_key(x, newkey)$ — уменьшение ключа указанного элемента множества до заданного положительного числа $newkey$.

Чаще всего приоритетная очередь представляется с помощью корневого дерева или набора корневых деревьев с определенными свойствами. При этом узлам дерева ставятся во взаимно однозначное соответствие элементы рассматриваемого множества.

Соответствие между узлами дерева и элементами множества называется *кучеобразным*, если для каждого узла i соблюдается условие: ключ элемента, приписанного узлу i, не превосходит ключей элементов, приписанных его потомкам.

Такие представления взвешенных множеств называются кучами. Вид дерева и способ его представления в памяти компьютера подбирается в зависимости от тех операций, которые предполагается выполнять над множеством и от того, насколько эти операции сказываются на суммарной трудоемкости алгоритма.

В библиотеке POINTER реализованы следующие представления куч:

- d-куча,
- левосторонняя куча,
- ленивая левосторонняя куча,
- самоорганизующаяся куча.

Первое из них, d-куча, основано на использовании массивов, остальные используют класс *pointer*.

В качестве примера рассмотрим некоторые операции с самоорганизующейся кучей. Работа с остальными представлениями куч аналогична.

Листинг pointer/stones.m

```
BirthStones = {'Garnet', 'Amethyst', 'Aquamarine', ...
    'Diamond', 'Emerald', 'Pearl', 'Ruby', 'Peridot', ...
    'Sapphire', 'Opal', 'Topaz', 'Turqouze'};
OtherStones = {'Agate', 'Malachite', 'Jasper', 'Hematite'};

hp = soh_new; % Создаем новую кучу
for i = 1:length(BirthStones)
    hp = soh_put(hp, BirthStones{i}, i); % Помещаем данные
end
soh_graph(hp) % Графическое изображение. См. рис. 3.11
pause

hp = soh_delmin(hp); % Удаляем элемент с минимальным весом
soh_graph(hp) % Графическое изображение. См. рис. 3.12
pause
```

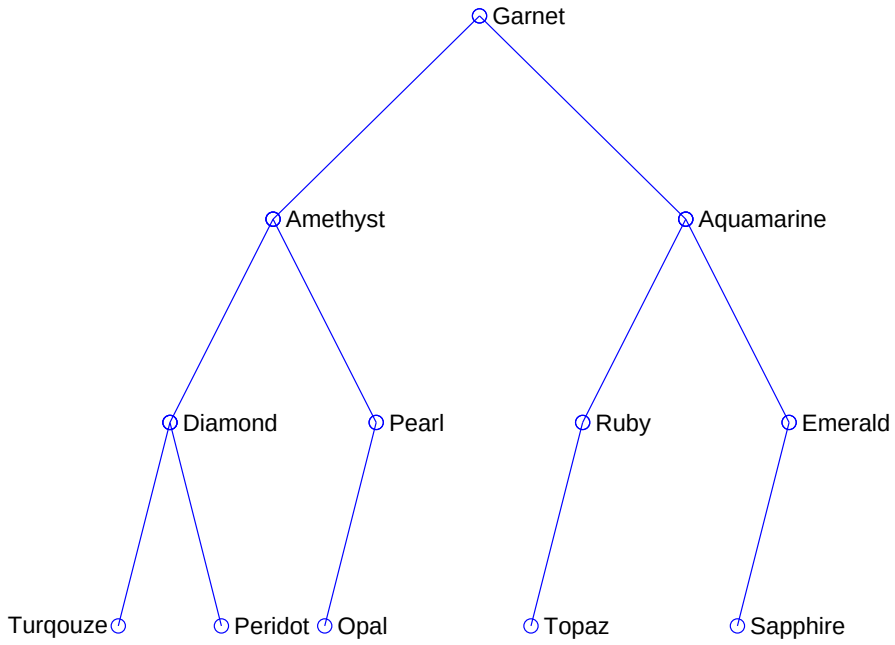


Рис. 3.11. Самоорганизующаяся куча

```
hp2 = soh_new; % Создаем новую кучу
for i = 1:length(OtherStones)
    hp2 = soh_put(hp2, OtherStones{i}, i); % Помещаем данные
end
soh_graph(hp2) % Графическое изображение. См. рис. 3.13
pause

hp = soh_merge(hp, hp2); % Сливаем кучи. После слияния hp2 пуста
soh_graph(hp) % Графическое изображение. См. рис. 3.14

hp = soh_free(hp); % Освобождаем память
```

В примере использовались следующие функции:

- $soh = soh\_new$ — создает новую кучу
- $soh = soh\_put(soh, data, key)$ — вставляет новый узел с данными $data$ и ключом key
- $soh\_graph(soh)$ — визуализирует дерево, представляющее кучу

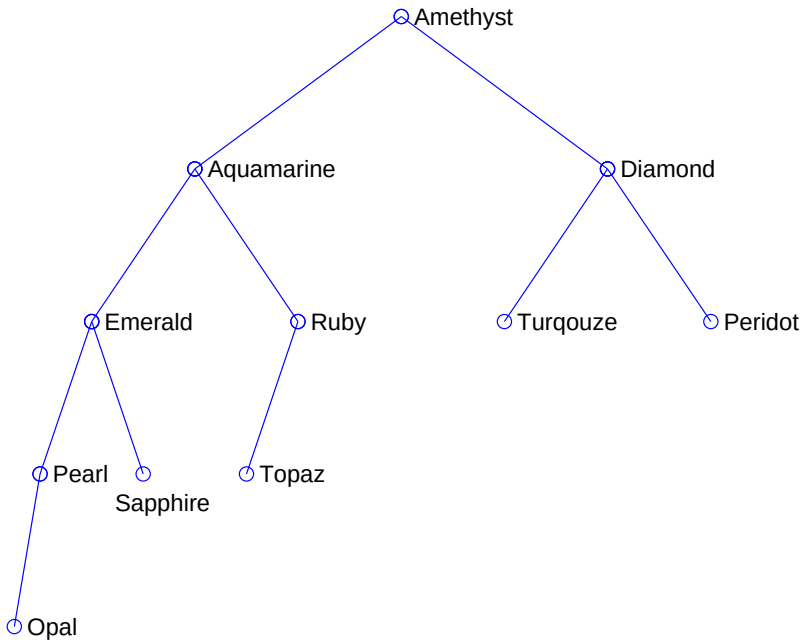


Рис. 3.12. Куча после удаления элемента с минимальным весом (*Garnet*)

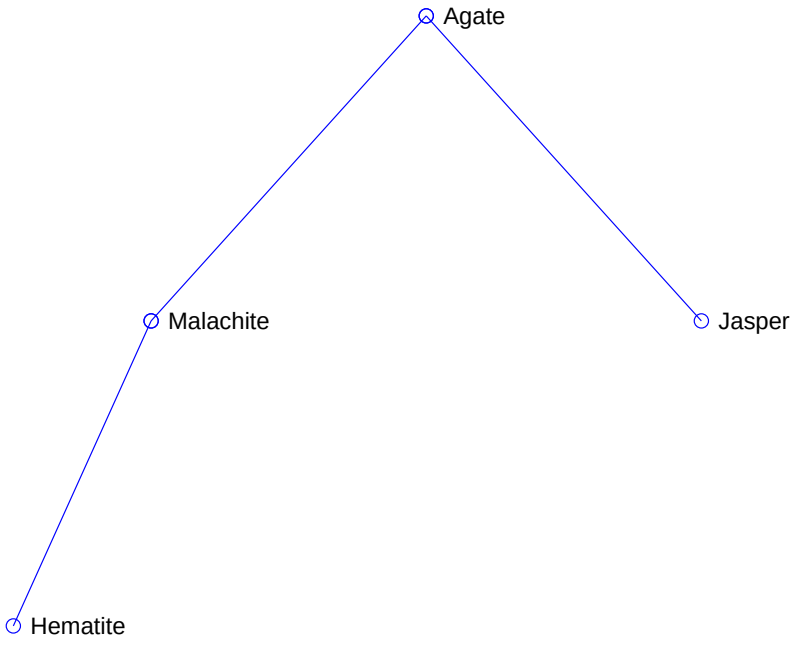


Рис. 3.13. Другая самоорганизующаяся куча

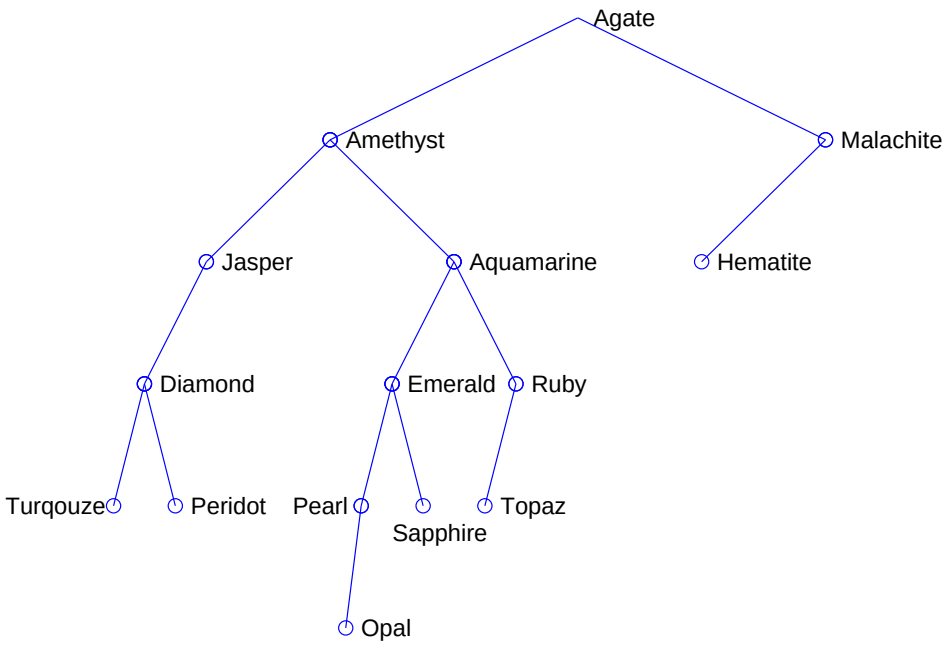


Рис. 3.14. Результат слияния куч

- $soh = soh\_delmin(soh)$ — удаляет элемент с минимальным ключом
- $soh\_merge(soh1, soh2)$ — сливает две кучи; после слияния куча $soh2$ не содержит данных
- $soh = soh\_free(soh)$ — освобождает память

4. Графы

В настоящей главе описывается класс *graph*, представляющий простые графы. Методы класса вызывают функции из библиотеки MATLAB GBL [10], которая, в свою очередь, представляет собой набор оберток для другой известной библиотеки — Graph Boost Library [11]. В настоящее время класс *graph* поддерживает работу только с простыми (т.е. неориентированными, без кратных ребер и петель) графами.

Учебный пример *graph* из главы 2 является упрощенной версией описываемого в настоящей главе класса *graph*. Сейчас мы не будем освещать вопросы реализации, укажем только, что графы представлены своими матрицами смежности, для хранения которых используется структура данных *sparse*.

4.1. Методы класса *graph*

4.1.1. Конструктор

Возможны следующие варианты вызова конструктора:

- *G1* = *graph*(*G*) — конструктор «копирования».
- *G* = *graph*(*A*) — создание графа по матрице смежности *A*. Матрица *A* должна быть квадратной. Если она не симметрична, то симметризация происходит внутри конструктора. Например, *graph*(*ones*(5)) создает полный граф с 5 вершинами — K_5 (см. рис. 4.1).
- *G* = *graph*(*E*) — создание графа по списку ребер *E*. Каждая строка матрицы *E* соответсвует ребру графа и содержит номера смежных вершин. Например, *E* = [1, 4; 1, 5; 1, 6; 2, 4; 2, 5; 2, 6; 3, 4; 3, 5; 3, 6] соответствует ребрам двудольного графа $K_{3,3}$.
- *G* = *graph*(..., *x*, *y*) — опционально мы можем задать координаты вершин графа. Векторы *x* и *y* должны иметь одинаковую длину, совпадающую с числом вершин графа.
- *G* = *graph*('name', *arg1*, *arg2*, ...) — создание графа по его «названию». В частности,

$G = graph(\texttt{'k'}, n)$ — полный граф с n вершинами — K_n (см. рис. 4.1, 4.4)

$G = graph(\texttt{'k'}, n1, n2)$ — полный двудольный граф $K_{n1,n2}$ (см. рис. 4.2)

$G = graph(\texttt{'c'}, n)$ — цикл из n вершин — C_n (см. рис. 4.5)

$G = graph(\texttt{'l'}, n)$ — цепь из n вершин — L_n (см. рис. 4.6)

$G = graph(\texttt{'p'}, n)$ — цепь из n вершин — L_n с расположением вершин по кругу (см. рис. 4.7)

$G = graph(\texttt{'tetrahedron'})$ — граф смежности вершин тетраэдра (см. рис. 4.8)

$G = graph(\texttt{'cube'})$ — граф смежности вершин куба (см. рис. 4.9)

$G = graph(\texttt{'octahedron'})$ — граф смежности вершин октаэдра (см. рис. 4.10)

$G = graph(\texttt{'dodecahedron'})$ — граф смежности вершин додекаэдра

$G = graph(\texttt{'icosahedron'})$ — граф смежности вершин икосаэдра

$G = graph(\texttt{'wheel'}, n)$ — «колесо» из $n + 1$ вершин (одна вершина в центре) (см. рис. 4.11)

$G = graph(\texttt{'petersen'})$ — граф Петерсена (см. рис. 4.12)

$G = graph(\texttt{'fulleren'})$ — «фуллерен» (см. рис. 4.13)

$G = graph(\texttt{'grid'}, m, n)$ — решетка $m \times n$ (см. рис. 4.14)

$G = graph(\texttt{'board'}, [i, j], m, n)$ — граф (i, j)-ходов фигуры на шахматной доске размера $m \times n$ (по умолчанию, 8×8). Под (i, j)-ходом понимается перемещение фигуры на i полей в одном направлении и j полей в ортогональном направлении. Например $(1, 2)$ соответствует ходу коня (см. рис. 4.15)

4.1.2. Другие методы

Доступны следующие методв:

- *addedges* — добавление новых ребер
- *addvertices* — добавление новых вершин
- *adjmatrix* — возвращает матрицу смежности
- *allshortestpaths* — кратчайшие расстояния для каждой пары вершин
- *bfs* — поиск в ширину

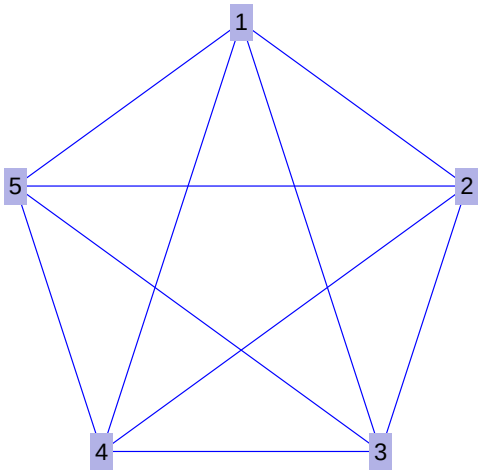


Рис. 4.1. Граф K_5: *graph*(`'k'`, 5)

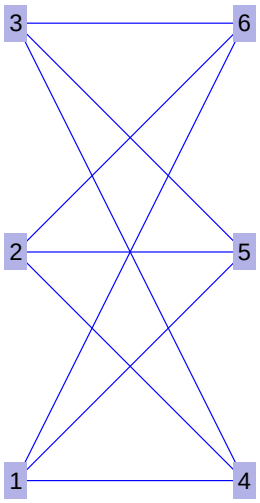


Рис. 4.2. Граф $K_{3,3}$: *graph*(`'k'`, 3, 3)

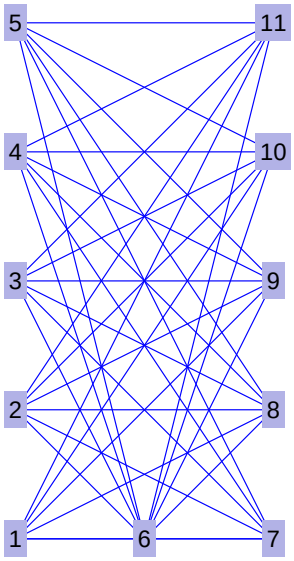


Рис. 4.3. Граф $K_{5,1,5}$: *graph*('k', 5, 1, 5)

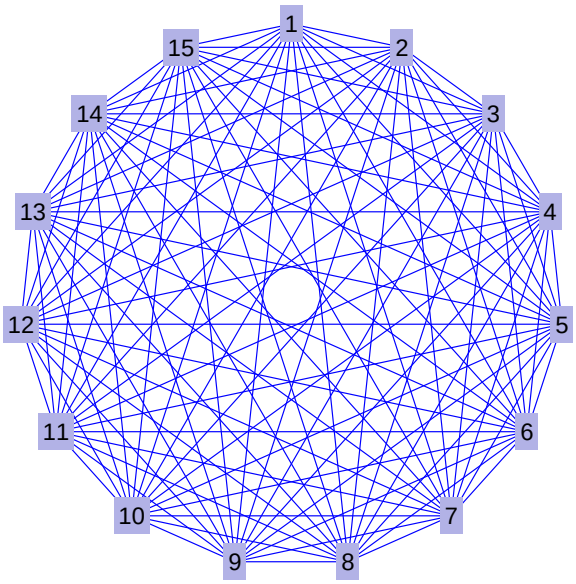
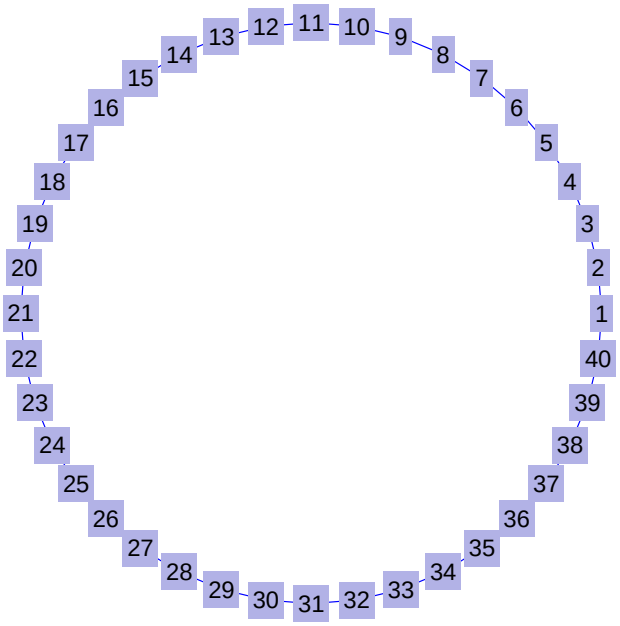


Рис. 4.4. Граф K_{15}: *graph*('k', 15)



Puc. 4.5. Граф C_{40}: *graph*('c', 40)

Puc. 4.6. Граф L_5: *graph*('l', 5)

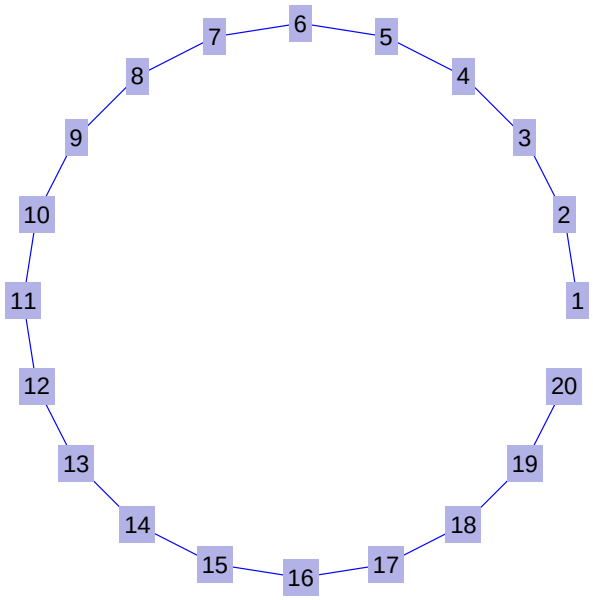


Рис. 4.7. Граф P_{20}: *graph*('p', 20)

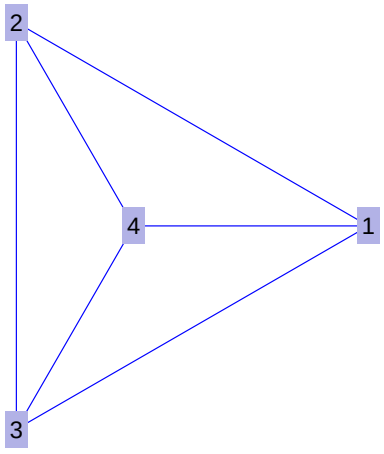


Рис. 4.8. Граф смежности вершин тетраэдра

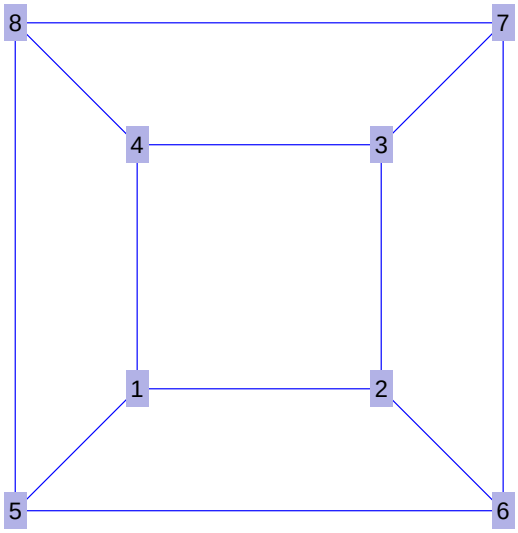


Рис. 4.9. Граф смежности вершин куба

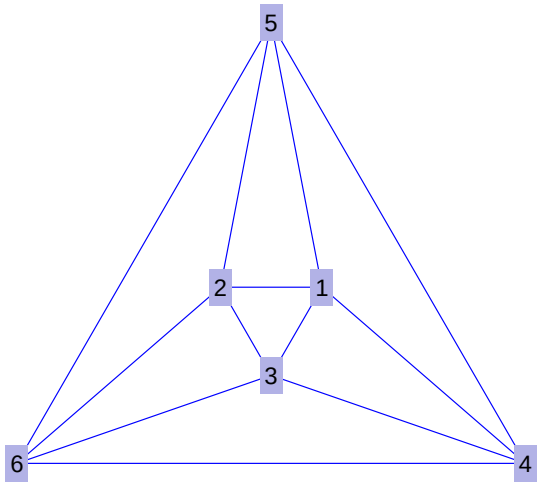


Рис. 4.10. Граф смежности вершин октаэдра

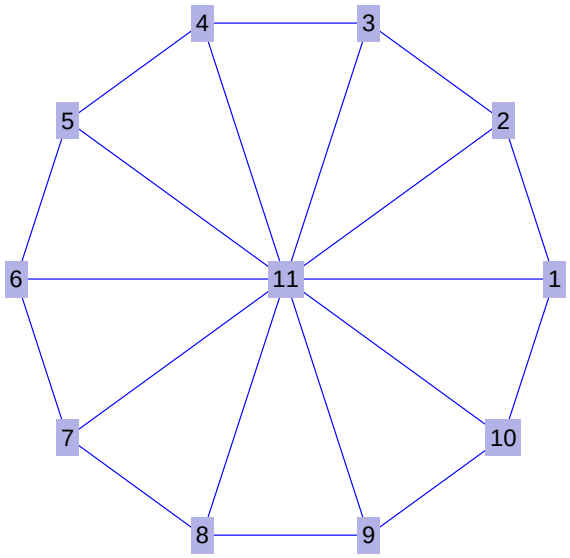


Рис. 4.11. «Колесо» с 11 вершинами. Одна вершина в центре и 10 — по кругу: *graph*(`'wheel'`,10)

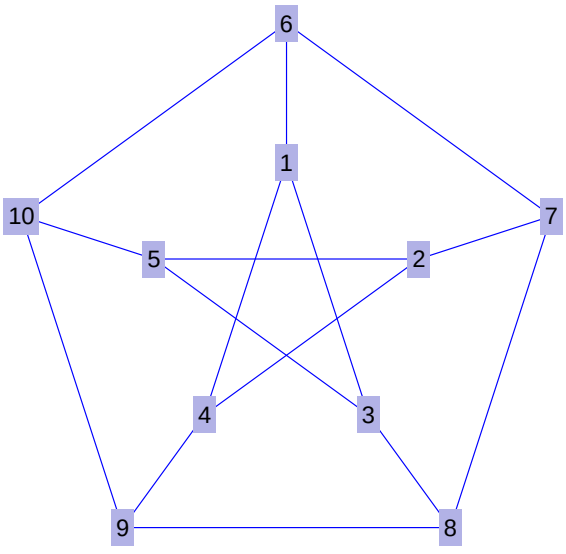


Рис. 4.12. Граф Петерсена: *graph*(`'petersen'`)

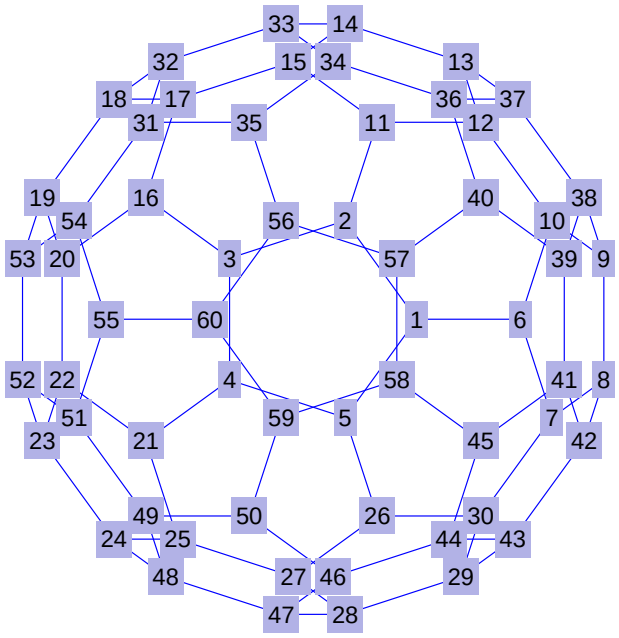


Рис. 4.13. Граф «фуллерен»: *graph*(`'fulleren'`)

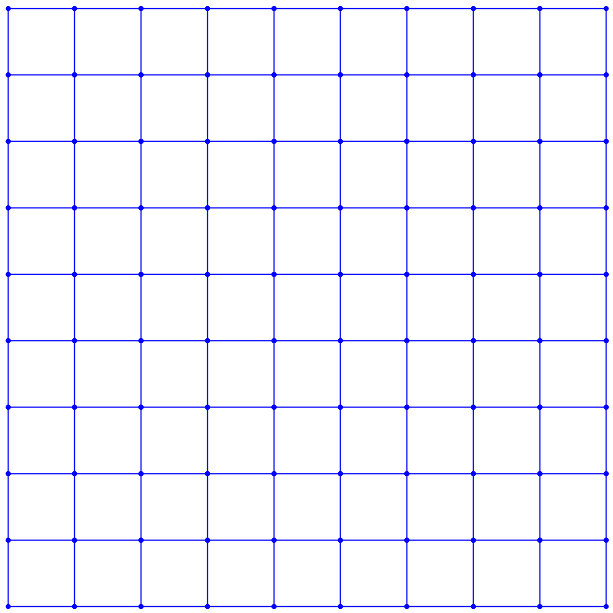


Рис. 4.14. Решетка 10 × 10: *graph*(`'grid'`, 10, 10)

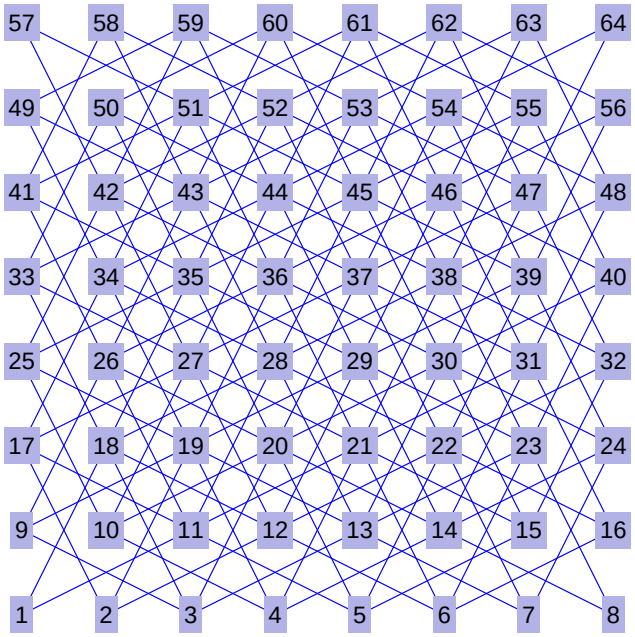


Рис. 4.15. Граф ходов коня: *graph*(`'board'`, [1, 2])

- *bicomponents* — выявление компонент 2-связности
- *complement* — дополнительный граф
- *components* — выявление компонент связности
- *coords* — возвращает координаты вершин
- *degree* — возвращает степень заданной вершины
- *degrees* — возвращает степени всех вершин
- *deledges* — удаляет указанные ребра
- *delvertices* — удаляет указанные вершины
- *dfs* — поиск в глубину
- *diameter* — находит диаметр графа
- *display* — отображает в командном окне краткую информацию о графе
- *draw* — изображает граф в графическом окне

- *edges* — возвращает список ребер
- *graph* — конструктор
- *mst* — находит остовное дерево
- *mtimes* (*G1*∗*G2*) — произведение графов (объединение вершин и ребер)
- *ncomponents* — возвращает число компонент связности
- *nedges* — возвращает число ребер
- *neighbourhood* — возвращает список вершин соседних с заданной
- *nvertices* — возвращает число вершин в графе
- *plus* (*G1* + *G2*) — объединение графов
- *setadjmatrix* — задать новую матрицу смежности
- *setcoords* — задать координаты вершин
- *setedgecolors* — задать цвет ребер
- *setvertexcolors* — задать цвет вершин
- *shortestpath* — находит кратчайший путь между двумя заданными вершинами
- *shortestpaths* — находит кратчайшие пути от заданной вершины до всех остальных
- *subgraph* — выделение порожденного подграфа
- *times* (*G1.*∗*G2*) — прямое произведение графов

4.2. Примеры

4.2.1. Создание графов и простые операции

В качестве первого примера рассмотрим создание графов из различных классов и простые операции над графами (объединение и произведение графов, удаление/добавление вершин и ребер графов и т.п.)

Листинг graph/examples/graphdemo.m

```
disp('In graph window press any key');
shg
wh = graph('wheel', 10); % «Колесо» с 11 вершинами
draw(wh) % См. рис. 4.11
pause
wh2 = wh + wh;
draw(wh2)
pause
[x, y] = coords(wh2);
x(12:end) = x(12:end) + 1;
wh2 = setcoords(wh2, x, y);
draw(wh2)
pause
wh2 = addedges(wh2, [3,15; 9,19]);
draw(wh2) % См. рис. 4.16
pause
wh2 = addvertices(wh2, 4);
draw(wh2)
pause
wh2 = delvertices(wh2, 23:26)
draw(wh2)
pause
wh2 = delvertices(wh2, [11, 22]);
draw(wh2) % См. рис. 4.17
pause;
wh2=addedges(wh2,[1,16]);
draw(wh2);
pause;
wh2=addedges(wh2,[23,25]);
draw(wh2);
pause;
wh2=delvertices(wh2,21:25);
draw(wh2);
pause;
draw(graph('tetrahedron')); % См. рис. 4.8
```

```
pause;
draw(graph('cube')); % См. рис. 4.9
pause;
draw(graph('octahedron')); % См. рис. 4.10
pause;
draw(graph('dodecahedron'));
pause;
draw(graph('icosahedron'));
pause;
draw(graph('k', 5)); % См. рис. 4.1
pause;
draw(graph('k', 3, 3)); % См. рис. 4.2
pause;
draw(graph('k', 5, 1, 5)); % См. рис. 4.3
pause;
g = graph('k', 15);
draw(g); % См. рис. 4.4
disp('K-15');
disp(['Max degree = ' num2str(max(degree(g)))]);
pause;
draw(graph('petersen')); % См. рис. 4.12
pause;
draw(graph('fulleren')); % См. рис. 4.13
pause;
draw(graph('fulleren', 30)); % См. рис. 4.18
pause;
g = graph('euclid');
draw(g);
disp('Euclidean graph');
disp(['Max degree = ' num2str(max(degree(g)))]);
pause;
draw(graph('grid', 10, 10)); % См. рис. 4.14
pause;
draw(graph('board', [1, 2])); % См. рис. 4.15
pause;
```

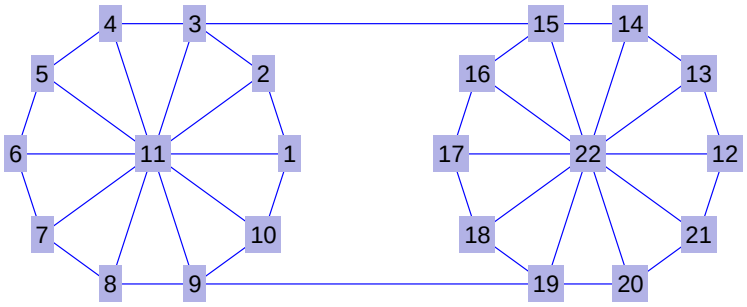


Рис. 4.16. Два колеса, соединенные двумя ребрами

```
draw(graph('c', 40)); % См. рис. 4.5
pause;
draw(graph('p', 20)); % См. рис. 4.7
pause;
draw(graph('l', 5)); % См. рис. 4.6
pause;
```

4.2.2. Минимальное остовное дерево

Напишем простую программу, которая ищет в графе остовное дерево (метод *mst*), раскрашивает ребра найденного дерева в цвет, оличный от цвета остальных ребер, и рисует граф.

Листинг graph/examples/xmst.m

```
G = graph('dodecahedron');
[u, v] = mst(G);
setedgecolors(G, [u, v], 'm')
draw(G) % См. рис. 4.19
```

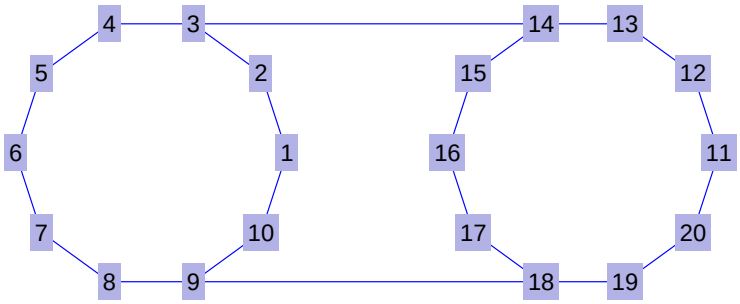


Рис. 4.17. Удалили центральные вершины

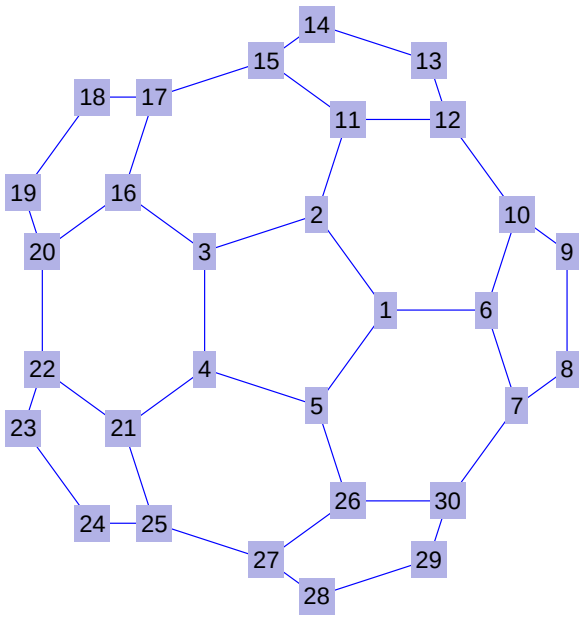


Рис. 4.18. «Половина» фуллерена

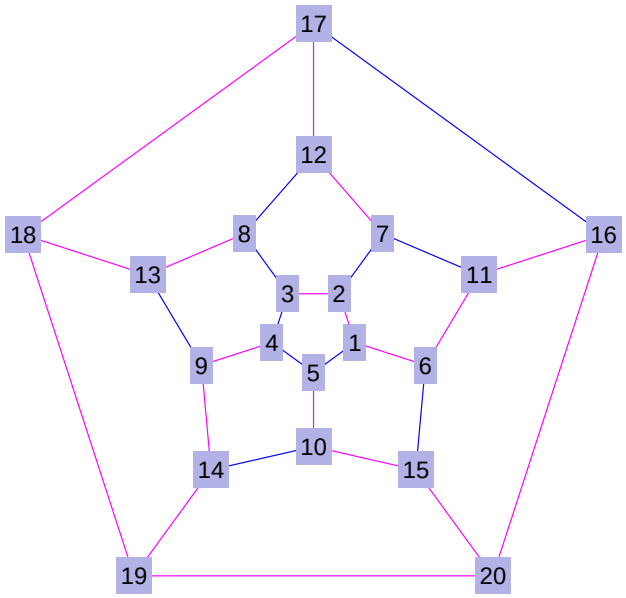


Рис. 4.19. Остовное дерево

4.2.3. Поиск в ширину и поиск в глубину

Рассмотрим применение поиска в ширину (*bfs*) для нахождения расстояния от вершины до всех остальных и задачи проверки графа на двудольность. Для решения первой задачи есть также метод *shortestpath*. Для решения второй задачи вместо поиска в ширину можно воспользоваться поиском в глубину (*dfs*).

Для определения расстояния от заданной вершины до всех остальных мы последовательно рассматриваем все вершины с помощью поиска в ширину и приписываем им метки. Начальной вершине приписывается метка 0. Как только встретилась новая вершина, то ее метка становится равной метке вершины, из которой мы пришли, плюс 1. В конце процедуры метки будут равны расстояшию от вершины до начальной.

Для проверки связного графа на двудольность мы раскрашиваем его вершины в два цвета. Первая вершина раскрашивается в некоторый цвет. Далее с помощью поиска в глубину или ширину просматриваются все ребра графа. По крайней мере одна из концевых вершин каждого из рассматриваемых ребер будет уже окрашена. Если другая из концевых вершин которого не окрашена, то окрашиваем ее цветом, отличным от цвета первой вершины. Если встретилось ребро, концевые вершины которого раскрашены в разные цвета, то граф не является двудольным.

Листинг graph/examples/xbfs.m

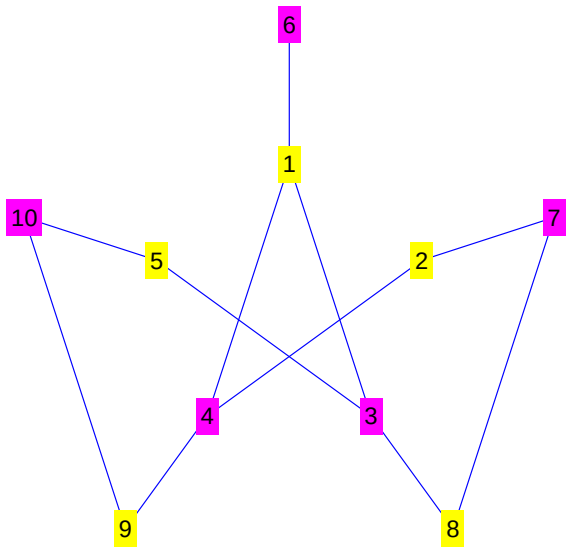


Рис. 4.20. Проверка графа на двудольность. Вершины разных долей раскрашены в разные цвета. Вершины одной доли — в один цвет

function *xbfs*

```
% xbfs — пример использования функции bfs (поиска в ширину)
% для нахождения расстояния от вершины до всех остальных.
% Также проверяем граф на двудольность (для этого можно использовать dfs)

shg
G = graph('petersen');
deledges(G, [7 6; 6 10; 9 8; 5 2]);
draw(G)
pause

dmap = zeros(nvertices(G), 1); % Расстояния от вершины 1 до всех остальных
colors = zeros(nvertices(G), 1); % «Цвет» вершин
colors(1) = 1; % 1-ю вершину раскрашиваем в 1-м цветом
bipartite = 1; % bipartite = 1, если граф двудольный, иначе bipartite = 0
```

```
bfs(G, 1, struct(...
    'tree_edge', @tree_edge, ...
    'examine_edge', @examine_edge, ...
    'gray_target', @gray_target, ...
    'black_target', @black_target));
disp('Расстояние от вершины 1 до всех остальных:')
disp(dmap)

colors
bipartite

setvertexcolors(G, find(colors == 1), 'y');
setvertexcolors(G, find(colors == 2), 'm');

draw(G); % См. рис. 4.20

    function tree_edge(ei, u, v)
        % [u, v] — ребро остовного дерева
        dmap(v) = dmap(u) + 1; %
    end

    function examine_edge(ei, u, v)
        % [u, v] — еще не посещенное ребро
        if colors(v) == 0
            colors(v) = 3 - colors(u); % Меняем цвет
        elseif colors(u) == colors(v)
            bipartite = 0; % Граф не двудольный
        end
    end

    function gray_target(ei, u, v)
        disp('Gray target')
        disp([u v])
    end
```

```
    function black_target(ei, u, v)
        disp('Black target')
        disp([u v])
    end
end
```

4.2.4. Гамильтонов путь

В качестве еще одного примера рассмотрим задачу поиска гамильтонова пути в графе из вершины v_0 в вершину v_{end}, т.е. пути, проходящего через все вершины по одному разу и не посещающего ни одного ребра более одного раза. Реализуем алгоритм поиска с возвратом. Начиная с вершины v_0, последовательно будем добавлять в путь новые вершины. Если дальнейшее продолжение не возможно (не существует гамильтонова пути с имеющимся началом), то отбрасываем последнюю добавленную к пути вершину и пробуем присоединить другую. Как определить, можно ли продолжить имеющийся путь до гамильтонова? Понятно, что если последняя посещенная вершина не инцидентна ни одной из непосещенных вершин, то, очевидно, гамильтонова пути с заданным началом не существует. В программе проверяется также другое условие: если граф, полученный удалением уже посещенных вершин, не связен, то гамильтонова пути с заданным началом нет.

Листинг graph/examples/hamilton.m

```
function path = hamilton(G, v0, vend)

% Поиск гамильтонова пути из вершины v0 в вершину vend

if nargin < 1 || isempty(G)
    G = graph('board', 6, [1, 2]);
end
if nargin < 2 || isempty(v0)
    v0 = 1;
end
if nargin < 3 || isempty(vend)
    vend = nvertices(G);
end
```

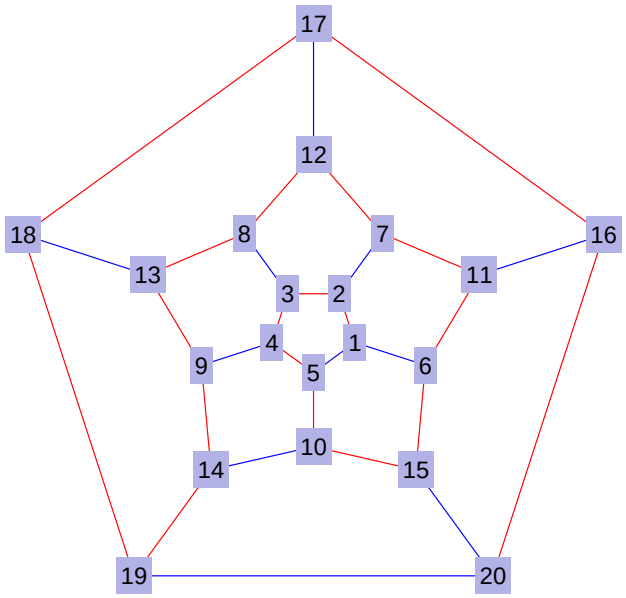


Рис. 4.21. Гамильтонов путь из вершины 1 в вершину 20

```
n = nvertices(G);
path = v0; % path — список посещенных вершин

success = findpath;

if ~success
    % Гамильтонова пути нет
    path = [];
else
    % Гамильтонов путь найден
    shg
    setedgecolors(G, [path(1:(end - 1)); path(2:end)]', 'r')
    draw(G) % См. рис. 4.21
end

    function success = findpath

        k = length(path);
```

```
if k == n
    % Посетили все вершины
    if path(n) == vend
        success = 1;
    else
        success = 0;
    end
    return
end

% Среди вершин, смежных текущей, находим еще не посещенные
if k < n - 1
    nbrh = setdiff(neighbourhood( G, path(end)), [path, vend]);
else
    nbrh = setdiff(neighbourhood( G, path(end)), path);
end

% Перебираем все непосещенные смежные вершины
for i = nbrh
    path = [path, i];
    % Проверяем число компонент графа с удаленными посщенными
    % вершинами
    if ncomponents(delvertices( G, path)) > 1
        % Граф не связен
        path(end) = [];
    elseif findpath
        % Граф связен. Рекурсивный вызов findpath.
        % Гамильтонов путь найден
        success = 1;
        return
    else
        % Гамильтонова пути нет
        path(end) = [];
    end
```

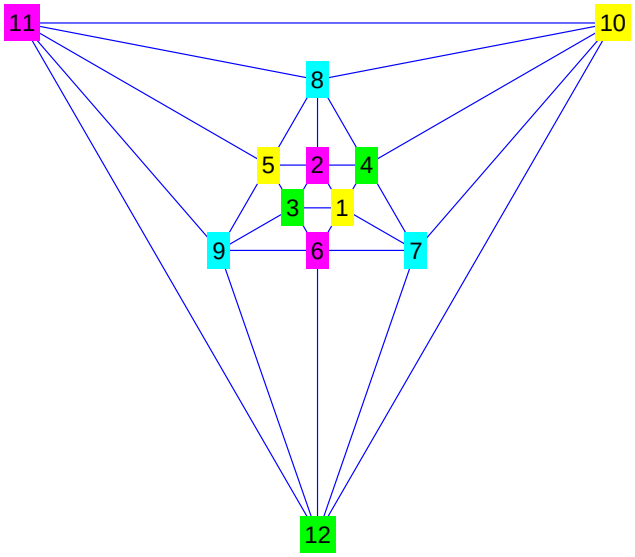


Рис. 4.22. Раскраска в 4 цвета вершин графа инциденций икосаэдра

```
        end
        success = 0;
    end
end
```

4.2.5. Вершинная раскраска

Рассмотрим задачу раскраски вершин графа в минимальное число цветов, так, чтобы никакие две смежные вершины не были окрашены в два одинаковых цвета. Известная теорема о 4 красках утверждает, что любой планарный граф допускает вершинную раскраску в 4 цвета. Напишем программу, которая будет искать раскраску вершин заданного графа в 4 цвета.

Реализуем алгоритм поиска с возвратом. Последовательно будем рассматривать все вершины и раскрашивать их. В качестве цвета очередной вершины буди испытывать все цвета, которые не использовались при раскраске смежных вершин. Если выбрать цвет не удается, то возвращаемся назад.

Листинг graph/examples/fourcolors.m

```
function fourcolors(G)
```

```
% Раскраска вершин графа в 4 цвета, так, чтобы
% никакие две смежные не были окрашены в один цвет

if nargin < 1
    G = graph('icosahedron');
end

n = nvertices(G);
% colors — номера цветов для каждой вершины
% Вначале вершины не раскрашены (цвет = 0)
colors = zeros(n, 1);

success = findfourcolors(1);
success
colors

setvertexcolors(G, find(colors == 1), 'y');
setvertexcolors(G, find(colors == 2), 'm');
setvertexcolors(G, find(colors == 3), 'g');
setvertexcolors(G, find(colors == 4), 'c');
shg
draw(G); % См. рис. 4.22

    function success = findfourcolors(v)
        % Раскрашиваем вершины графа, если
        % первые v - 1 вершин уже раскрашены

        if v > n
            success = 1;
            return
        end

        % Для вершины v рассматриваем все цвета,
        % которые не использовались при раскраске смежных с v вершин
```

```
        for c = setdiff(1:4, colors(neighbourhood(G, v)))
            colors(v) = c;
            % Раскрашиваем остальную часть графа
            if findfourcolors(v + 1)
                success = 1;
                return
            end
        end

        % Раскраска не найдена
        colors(v) = 0;
        success = 0;
    end
end
```

Литература

1. *Алексеев В.Е., Таланов В.А.* Графы и алгоритмы. Структуры данных. Модели вычислений. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006

2. *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Вильямс, 2000

3. *Вирт Н.* Алгоритмы + структуры данных = программы. М.: Мир, 1978

4. *Золотых Н.Ю.* Использование пакета Matlab в научной и учебной работе. Нижний Новгород: Нижегород. гос. ун-т им. Н.И. Лобачевского, 2006.

5. *Иглин* С.П. Математические расчеты на базе MATLAB. М.: BHV, 2005

6. *Кнут Д.* Искусство программирования для ЭВМ. Т. 1: Основные алгоритмы. М.: Мир, 1976

7. *Кнут Д.* Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. М.: Мир, 1978

8. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2001

9. *Gleich D.* Inplace
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=11290

10. *Gleich D.* MatlabBGL
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=10922

11. Boost Graph Library. Руководство программиста. М.: BHV, 2006

12. *Iglin S.* grTheory — Graph Theory Toolbox
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=4266

13. *Keren Y.* Data Structures & Algorithms Toolbox
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=212

14. *Miller J., de Plinval H., Hsiao K.* Mapping Contoured Terrain: A Comparison of SLAM Algorithms for Radio-Controlled Helicopters. MIT, 2005

15. *Murphy K.* Graph theory toolbox http://www.cs.ubc.ca/ murphyk/Software/graph.zip