```
        print(prompt); return inputLong();
    }

    public static float inputFloat() throws IOException,
        NumberFormatException {
        return Float.valueOf(inputString()).floatValue();
    }
    public static float inputFloat(String prompt) throws IOException,
        NumberFormatException {
        print(prompt); return inputFloat();
    }

    public static double inputDouble() throws IOException,
        NumberFormatException {
        return Double.valueOf(inputString()).doubleValue();
    }
    public static double inputDouble(String prompt)
        throws IOException, NumberFormatException {
        print(prompt); return inputDouble();
    }
    // Методы ввода строки, рассматриваемой как массив символов.
    public static char[] inputChars() throws IOException {
        return (inputString()).toCharArray();
    }
    public static char[] inputChars(String prompt)
        throws IOException {
        print(prompt);
        return (inputString()).toCharArray();
    }
}
```

§ 3. Высказывания и предикаты

Материал этого и некоторых из следующих параграфов в значительной мере основан на подходе к программированию, применяемом в книге [4], знакомство с которой весьма полезно. Что же касается собственно математической теории предикатов, то нам необходимы только ее основы, и поэтому обращение к какой-либо специальной дополнительной литературе по этой тематике не требуется.

1. Зачем программисту предикаты. Все знают, что в программах бывают *ошибки (bugs)*. Существуют специальные теории, посвященные тому, как лучше их находить и исправлять (*debugging* дословно означает «выведение клопов»). Зачастую нахождение ошибки — очень нетривиальная задача, так как ее последствия могут сказываться совершенно в другом месте программы и быть весьма неожиданными.

При этом часто забывается тот очевидный факт, что ошибку гораздо легче *предотвратить* при написании программы, нежели найти и исправить потом. Существуют методы проектирования программ (по крайней мере, небольших), позволяющие не только написать правильную программу, но и получить одновременно с этим совершенно строгое доказательство ее правильности. Изучению таких методов и будет посвящена в основном вторая глава данной книги.

Однако для того, чтобы изучить какую-либо теорию, необходимо выучить язык, на котором теория может быть изложена. *Язык предикатов* — это именно тот язык, на котором можно строго сформулировать постановку задачи и доказать правильность конкретной программы. Попробуйте решить следующую задачу.

Задача 3.1. *Задача о банке с кофейными зернами.* В банке имеется несколько черных и белых кофейных зерен. Следующий процесс надо повторять, пока это возможно:

- случайно выберите из банки два зерна и
  - если они одного цвета, отбросьте их, но положите в банку другое черное зерно (имеется достаточный запас черных зерен, чтобы делать это);
  - если они разного цвета, поместите белое зерно обратно в банку и отбросьте черное зерно.

Выполнение этого процесса уменьшает количество зерен в банке на единицу. Повторение процесса должно прекратиться, когда в банке останется всего одно зерно, так как тогда нельзя уже выбрать два зерна. Можно ли что-то сказать о цвете оставшегося зерна, если известно, сколько вначале в банке было черных и белых зерен?

Попытка решать эту задачу, используя тестовые примеры с небольшим начальным количеством зерен в банке, не приводит ни к какому результату в течение значительного промежутка времени. Этот подход к решению является в каком-то смысле аналогом тестирования программы с целью выявления ошибок в ней.

Знание теории позволяет получить ответ на вопрос задачи почти мгновенно, однако объяснить это решение практически невозможно без привлечения такого понятия, как *инвариант цикла*, речь о котором пойдет в нашем курсе значительно позже. Инвариант цикла — это предикат, обладающий некоторыми специальными свойствами. Интересно, что при этом даже пятиклассник может справиться с рассматриваемой задачей, решив ее *как-то*. Под этим понимается по существу правильное решение, правильность которого доказать абсолютно невозможно.

Таблица 1. Вероятность правильной работы программы, содержащей n ветвлений

n	10	100	1000
P	0.904	0.366	0.00004

Теперь рассмотрим вопрос о том, насколько сложно протестировать *уже написанную* программу. При этом мы будем предполагать для простоты, что для линейной программы (без ветвлений) достаточно всего одного теста, для программы с одним оператором `if` — двух (чтобы протестировать правильность каждой из его ветвей) и так далее.

Реальные большие программы, конечно, не сводятся к совокупности вложенных друг в друга условных операторов. Однако они не проще, а сложнее — ведь в них есть и циклы и вызовы функций, подпрограмм или методов, обработка исключительных ситуаций и многое другое.

Вернемся к нашей модели. Если в программе 10 операторов `if`, то нужно выполнить $2^{10} = 1024$ теста, а если их 20, то уже $2^{20} \approx 10^6$! Можно ли надеяться на то, что в процессе тестирования реальной большой программы удастся проверить все возможные варианты ее работы? Нет, конечно. Именно поэтому так важно *не делать* ошибок, так как потом их скорее всего просто не обнаружить.

В заключение поговорим о *правильности большой программы*, состоящей из многих небольших частей. Предположим, что для каждой из этих частей мы можем гарантировать правильность ее работы с вероятностью 99%. Это значит, что в среднем только в одном случае из ста что-то может быть не так, а в 99 случаях каждая часть будет работать абсолютно правильно. Пусть всего таких частей n. Какова вероятность правильной работы программы в целом?

Это — простейшая задача по курсу теории вероятностей и ответ у нее такой: $P = p^n$. Здесь p — вероятность правильной работы каждой из частей, а P — вероятность правильной работы программы в целом. При $p = 0.99$ получаем результаты, приведенные в таблице 1.

При $n = 100$ программа будет работать правильно чуть более, чем в одной трети всех ситуаций, а при $n = 1000$ увидеть ее работающей вообще вряд ли удастся. Этот пример показывает, что нужно всеми силами стараться избегать написания *почти правильных* программ!

2. Синтаксис языка предикатов. Так как нам предикаты нужны прежде всего для описания программ, введем следующее определение.

Определение 3.1. *Высказывание* или *предикат* — это функция, действующая из некоторого множества значений переменных программы (идентификаторов) в множество из двух значений $\{T, F\}$ (*Да* и *Нет*).

В соответствии с ним предикатами будут следующие фразы:

- значение переменной `i` равно двум;
- переменная `k` положительна, а значение переменной `m` при этом не превосходит 100;
- значения всех целочисленных переменных программы являются нулевыми;
- неправда, что значение переменной `i` неположительно.

Чуть менее понятно, можно ли считать предикатами такие фразы:

- если предположить, что значение переменной `i` равно двум, то значения всех остальных целочисленных переменных программы будут неотрицательными;
- данная программа является правильной;
- данное высказывание ложно.

Особенно показательным является последний пример. Если мы согласимся с тем, что он представляет из себя предикат, то возникает естественный вопрос о его истинности. Предположение о том, что он истинен, заставляет считать его ложным, и наоборот. Получаемое противоречие говорит о том, что определение 3.1 не является достаточно корректным.

Ситуация с предикатами напоминает уже обсуждавшуюся нами ситуацию с языками для записи алгоритмов — необходим какой-то четкий критерий, позволяющий однозначно определить, что является предикатом, а что нет.

В теории формальных языков, более детальное знакомство с которой состоится позже, принято задавать язык с помощью грамматики. Грамматику же достаточно часто определяют с помощью так называемой нормальной формы Бэкуса-Наура (НФБН). Вот все необходимые общие определения.

Определение 3.2. *Алфавит* Σ — произвольное непустое множество.

Мы будем иметь дело только с конечными алфавитами, примерами которых являются алфавит русского языка, английский алфавит, множество цифр, алфавит всех символов, имеющихся на клавиатуре компьютера.

Определение 3.3. *Символом алфавита* Σ называют любой его элемент, а *цепочкой над алфавитом* — произвольную последовательность символов ω.

Цепочки часто называют также словами, фразами и предложениями. Пустая цепочка обозначается специальным символом ε, а множество всех цепочек над алфавитом Σ принято обозначать Σ^*. Если в качестве Σ взять множество букв русского алфавита, дополненное символом пробела и знаками пунктуации, то в Σ^* будут содержаться все фразы русского языка.

Определение 3.4. *Длиной* $|\omega|$ цепочки $\omega \in \Sigma^*$ называется количество входящих в нее символов.

Длина пустой цепочки равна нулю, а длины всех остальных цепочек над любым алфавитом положительны.

Определение 3.5. Операция $\circ\colon \Sigma^* \times \Sigma^* \to \Sigma^*$ *конкатенации* двух цепочек определена следующем образом. Пусть $\omega_1 = a_1 a_2 \dots a_n$, $\omega_2 = b_1 b_2 \dots b_m$, тогда $\omega_1 \circ w_2 = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$.

Операция конкатенации, называемая также операцией сцепления или дописывания, обладает следующими свойствами.

Предложение 3.1. *Для любых цепочек* ω, ω_1, ω_2 *и* ω_3 *справедливы следующие равенства:*
1) $\varepsilon \circ \omega = \omega \circ \varepsilon = \omega$,
2) $(\omega_1 \circ \omega_2) \circ \omega_3 = \omega_1 \circ (\omega_2 \circ \omega_3)$.

Имея все эти определения, уже можно дать формальное определения *языка над алфавитом* Σ.

Определение 3.6. *Язык* L — это произвольное подмножество множества цепочек Σ^*.

Если язык конечен, то есть состоит из конечного множества входящих в него цепочек, то его можно задать, просто перечислив все его элементы. Для бесконечных языков, которые чаще всего и представляют наибольший интерес, такой способ не годится. Достаточно часто для задания языка используют грамматику, несколько упрощенное определение которой мы сейчас рассмотрим.

Определение 3.7. Пусть Σ — некоторый алфавит, N — метаалфавит, т.е. какой-то другой алфавит, не пересекающийся с Σ ($\Sigma \cap N = \varnothing$). Элементы метаалфавита N называются *метасимволами. Грамматикой* G называется набор (Σ, N, P, S), где Σ — множество символов, N — множество метасимволов, P — множество правил вывода вида: $\alpha \to \beta$, где $\alpha \in N$ — какой-то метасимвол, $\beta \in (\Sigma \cup N)^*$ — произвольная цепочка над объединением двух алфавитов, и для каждого $\alpha \in N$ встречается хотя бы одно правило с α в левой части (до стрелочки), а $S \in N$ — так называемый *стартовый метасимвол.*

Содержательно каждое правило грамматики имеет смысл подстановки. Например, строка $\alpha \to \alpha\gamma\alpha$ означает возможность замены метасимвола α на цепочку $\alpha\gamma\alpha$. Начав со стартового символа и пользуясь различными правилами грамматики, мы можем получать различные цепочки из символов, которые называются *выводимыми цепочками*.

Заметим, что если в цепочке встречается метасимвол, то ее можно преобразовать дальше, применив одно из правил грамматики с этим метасимволом в левой части. Если же метасимволов в цепочке не осталось, то процесс ее преобразования закончен и больше с цепочкой ничего сделать нельзя. По этой причине обычные символы (из алфавита Σ) часто называют *терминалами*, а метасимволы (из N) — *нетерминалами*.

Определение 3.8. *Языком* $L(G)$, порожденным грамматикой G, называется множество всех терминальных выводимых цепочек.

Для задания грамматики часто используют очень наглядную форму представления, называемую *нормальной формой Бэкуса-Наура (НФБН)*. Набор правил P задают при этом в виде совокупности правил со стрелочками, перечисляющими все возможные цепочки, на которые может быть заменен каждый из метасимволов грамматики в процессе вывода, а стартовым метасимволом считается тот, который присутствует в левой части самого первого правила.

В качестве примера дадим строгое определение языка предикатов, или, как принято еще говорить, зададим *синтаксис* этого языка.

Определение 3.9. *Множество предикатов* — это язык, порожденный следующей грамматикой:

$$
\begin{array}{rll}
e \to & T & \text{(1-е правило: истина)} \\
\mid & F & \text{(2-е правило: ложь)} \\
\mid & id & \text{(3-е правило: идентификатор)} \\
\mid & (!e) & \text{(4-е правило: отрицание)} \\
\mid & (e \vee e) & \text{(5-е правило: дизъюнкция)} \\
\mid & (e \| e) & \text{(6-е правило: условное } \textit{Или}\text{)} \\
\mid & (e \wedge e) & \text{(7-е правило: конъюнкция)} \\
\mid & (e \&\& e) & \text{(8-е правило: условное } \textit{И}\text{)} \\
\mid & (e \Rightarrow e) & \text{(9-е правило: импликация)} \\
\mid & (e = e) & \text{(10-е правило: эквивалентность)}
\end{array}
$$

Единственным метасимволом данной грамматики является e, а алфавит $\Sigma = \{T, F, !, \vee, \|, \wedge, \&\&, \Rightarrow, =, (,)\} \cup M_{id}$, где множество M_{id} состоит из всех возможных идентификаторов (имен) переменных программ логического типа.

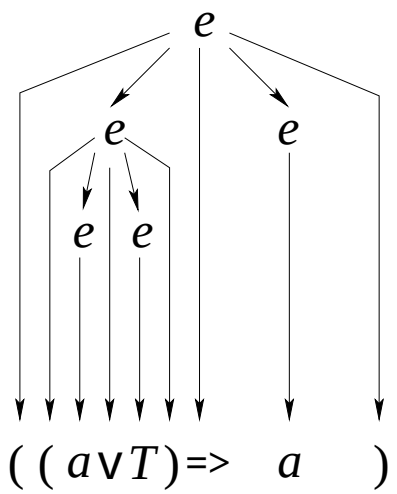

Рис. 1. Дерево вывода предиката $((a \vee T) \Rightarrow a)$

Приведем пример цепочки вывода в данной грамматике: $e \to (e \Rightarrow e) \to ((e \vee e) \Rightarrow e) \to ((a \vee e) \Rightarrow e) \to ((a \vee T) \Rightarrow e) \to ((a \vee T) \Rightarrow a)$. Это показывает, что выражение $((a \vee T) \Rightarrow a)$ является предикатом. Легко построить другой вывод этого же предиката: $e \to (e \Rightarrow e) \to ((e \vee e) \Rightarrow e) \to ((e \vee T) \Rightarrow e) \to ((a \vee T) \Rightarrow e) \to ((a \vee T) \Rightarrow a)$. Существует и еще несколько других цепочек вывода для предиката $((a \vee T) \Rightarrow a)$, отличающихся порядком замены метасимвола e на идентификатор a и значение T. Ясно, что в определенном смысле, который мы не будем сейчас уточнять, все эти цепочки эквивалентны. Говорят, что множество эквивалентных цепочек задает *дерево вывода* данного предиката, изображенное на рисунке 1.

Рассмотрим следующую задачу.

Задача 3.2. Докажите, что выражение $((a \wedge b) = (b \vee a))$ является предикатом.

Решение. Для доказательства достаточно предъявить вывод в грамматике 3.9 предложенного выражения, что не слишком трудно: $e \to (e = e) \to ((e \wedge e) = e) \to ((e \wedge e) = (e \vee e)) \to ((a \wedge e) = (e \vee e)) \to ((a \wedge b) = (e \vee e)) \to ((a \wedge b) = (b \vee e)) \to ((a \wedge b) = (b \vee a))$.

Покажем, как можно доказать, что выражение не является предикатом.

Задача 3.3. Докажите, что выражение $a \vee a$ — не предикат.

Решение. В самом деле, для того, чтобы в предикате появился символ $\vee$, необходимо применить правило $e \to (e \vee e)$, а его применение вызывает появление пары скобок, которых нет в выражении $a \vee a$. Ни один из

Таблица 2. Таблица истинности

b	c	$!b$	$b \vee c$	$b \| \| c$	$b \wedge c$	$b \& \& c$	$b \Rightarrow c$	$b = c$
T	T	F	T	T	T	T	T	T
T	F	F	T	T	F	F	F	F
F	T	T	T	T	F	F	T	F
F	F	T	F	F	F	F	T	T

терминальных символов, появившись в процессе вывода, не может измениться в дальнейшем (или исчезнуть). Таким образом, если 5-е правило грамматики 3.9 не применять, то мы не сумеем получить в итоговой цепочке символ $\vee$, а если его применить хотя бы раз, то в цепочке будут присутствовать скобки. Полученное противоречие и показывает, что выражение $a \vee a$ предикатом не является.

3. Семантика предикатов. В предыдущей секции мы разобрались с *синтаксисом* языка предикатов. Теперь нужно обсудить *семантику* этого языка, то есть придать смысл каждой из цепочек, ему принадлежащих. Иными словами, нам необходимо придать смысл каждому из предикатов. Сделаем это в три этапа, рассмотрев сначала простейшие предикаты T и F, затем константные предикаты, а уж затем определим значение произвольного высказывания.

Определение 3.10. Предикат F имеет значение $False$ (*Ложь*), а предикат T — значение $True$ (*Истина*).

Определение 3.11. Назовем предикат *константным*, если в нем не содержится ни одного идентификатора. Значение любого такого предиката находится с помощью *таблицы истинности* 2 и дерева вывода для него.

Рассмотрим, например, константный предикат $((F \vee T) \Rightarrow F)$. Изобразим дерево вывода для него и будем, двигаясь снизу вверх, заменять результаты операций в соответствии с таблицей истинности, пока не дойдем до самого корня дерева. Это последнее значение и будет считаться значением предиката (см. рис. 2).

Для задания семантики предиката, содержащего переменные, нам необходимо следующее определение состояния.

Определение 3.12. *Состояние* s — это отображение из множества идентификаторов в множество $\{T, F\}$. Пространство состояний переменных программы — это прямое произведение множеств состояний всех переменных программы.

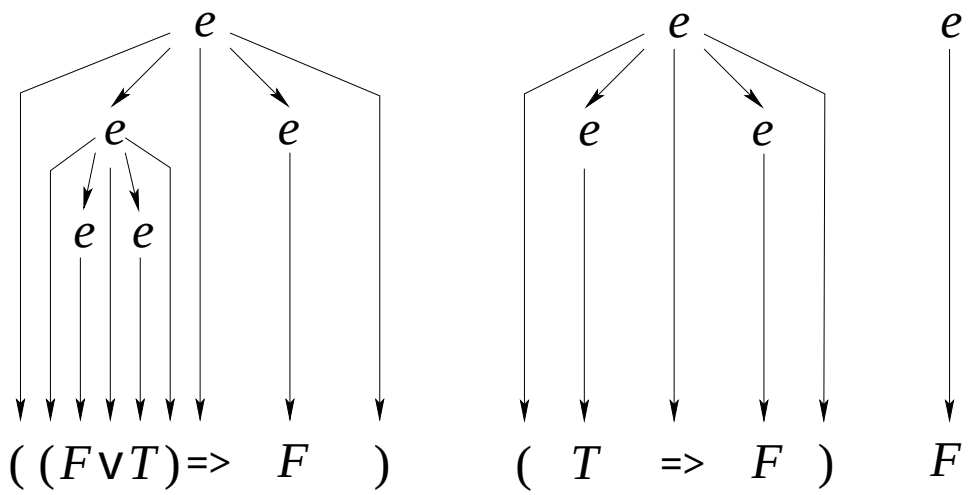

Рис. 2. Определение истинности предиката $((F \vee T) \Rightarrow F)$

Теперь можно дать определение значения любого предиката в заданном состоянии.

Определение 3.13. *Значение предиката e в состоянии s* (записывается как $e(s)$ или $s(e)$) совпадает со значением константного предиката, который получается из предиката e заменой всех идентификаторов на их значения в состоянии s.

Предикат $((a \vee T) \Rightarrow b)$, например, в состоянии $s = \{(a, T), (b, F)\}$ имеет значение F, ибо именно таково значение константного предиката $((T \vee T) \Rightarrow F)$.

Из таблицы истинности следует, что операции дизъюнкции $\vee$ и условного *Или* $\|$, а также конъюнкции $\wedge$ и условного *И* $\&\&$ попарно эквивалентны. Это действительно так, но только до тех пор, пока мы остаемся в рамках действия определения 3.12. Для нужд описания реального мира, однако, полезно его несколько ослабить и разрешить некоторым переменным оставаться неопределенными.

Определение 3.14. *Состояние s в расширенном смысле* — отображение из множества идентификаторов в множество $\{T, F, U\}$, где символ U означает неопределенное (undefined) значение.

Большинство предикатов не имеют определенного значения в таком состоянии, в котором не определены некоторые из переменных, входящих в него. В частности, операции $!$, $\vee$, $\wedge$, $\Rightarrow$ и $=$ дают результат U, если хотя бы один из операндов имеет значение U. Совершенно другая ситуация с условными операторами. Для них справедлива таблица истинности 3.

Среди огромного множества всех предикатов особую роль играют те из них, которые всегда являются истинными, как, например $((!(!a)) = a)$.

Таблица 3. Таблица истинности для условных операторов

b	T	T	F	F	T	F	U	U	U
c	T	F	T	F	U	U	T	F	U
$b\|\|c$	T	T	T	F	T	U	U	U	U
$b\&\&c$	T	F	F	F	U	F	U	U	U

Таблица 4. Таблица истинности предиката $((a \vee b) = (b \vee a))$

a	b	$(a \vee b)$	$(b \vee a)$	$((a \vee b) = (b \vee a))$
F	F	F	F	T
T	F	T	T	T
F	T	T	T	T
T	T	T	T	T

Определение 3.15. Предикат называется *тавтологией*, если он истинен во всех состояниях, в которых он определен.

Один из простейших способов доказать, что предикат является тавтологией, — это вычислить его значения во всех возможных состояниях. Таблица 4 является доказательством того факта, что предикат $((a \vee b) = (b \vee a))$ — тавтология.

Определение 3.16. Высказывания $e1$ и $e2$ *эквивалентны*, если предикат $e1 = e2$ является тавтологией.

Использование *законов эквивалентности* дает возможность упрощать предикаты, что часто бывает весьма полезно. Докажите самостоятельно, что все перечисленные ниже законы эквивалентности действительно являются тавтологиями.

Предложение 3.2. *Законы коммутативности:*
$((e_1 \wedge e_2) = (e_2 \wedge e_1))$;
$((e_1 \vee e_2) = (e_2 \vee e_1))$;
$((e_1 = e_2) = (e_2 = e_1))$.

Предложение 3.3. *Законы ассоциативности:*
$((e_1 \wedge (e_2 \wedge e_3)) = ((e_1 \wedge e_2) \wedge e_3))$;
$((e_1 \vee (e_2 \vee e_3)) = ((e_1 \vee e_2) \vee e_3))$;
$(e_1 \&\& (e_2 \&\& e_3)) = ((e_1 \&\& e_2) \&\& e_3))$;
$((e_1 \| (e_2 \| e_3)) = ((e_1 \| e_2) \| e_3))$.

Предложение 3.4. *Законы дистрибутивности:*
$((e_1 \wedge (e_2 \vee e_3)) = ((e_1 \wedge e_2) \vee (e_1 \wedge e_3)))$;

$((e_1 \vee (e_2 \wedge e_3)) = ((e_1 \vee e_2) \wedge (e_1 \vee e_3)))$;
$((e_1 \&\& (e_2 \| e_3)) = ((e_1 \&\& e_2) \| (e_1 \&\& e_3)))$;
$((e_1 \| (e_2 \&\& e_3)) = ((e_1 \| e_2) \&\& (e_1 \| e_3)))$.

Предложение 3.5. *Законы де Моргана:*
$((!(e_1 \wedge e_2)) = ((!e_1) \vee (!e_2)))$;
$((!(e_1 \vee e_2)) = ((!e_1) \wedge (!e_2)))$;
$((!(e_1 \&\& e_2)) = ((!e_1) \| (!e_2)))$;
$((!(e_1 \| e_2)) = ((!e_1) \&\& (!e_2)))$.

Предложение 3.6. *Закон отрицания:*
$((!(!e)) = e)$.

Предложение 3.7. *Законы исключенного третьего:*
$(e \vee (!e) = T)$;
$(e \| (!e) = T)$, *если* e *определено.*

Предложение 3.8. *Законы противоречия:*
$(e \wedge (!e) = F)$;
$(e \&\& (!e) = F)$, *если* e *определено.*

Предложение 3.9. *Закон импликации:*
$((e_1 \Rightarrow e_2) = ((!e_1) \vee e_2))$.

Предложение 3.10. *Закон равенства:*
$((e_1 = e_2) = ((e_1 \Rightarrow e_2) \wedge (e_2 \Rightarrow e_1)))$.

Предложение 3.11. *Законы упрощения дизъюнкции:*
$((e \vee e) = e)$;
$((e \vee T) = T)$;
$((e \vee F) = e)$;
$((e_1 \vee (e_1 \wedge e_2)) = e_1)$.

Предложение 3.12. *Законы упрощения конъюнкции:*
$((e \wedge e) = e)$;
$((e \wedge T) = e)$;
$((e \wedge F) = F)$;
$((e_1 \wedge (e_1 \vee e_2)) = e_1)$.

Предложение 3.13. *Законы упрощения условного Или:*
$((e \| e) = e)$;
$((e \| T) = T)$, *если* e *определено;*
$((e \| F) = e)$;
$((e_1 \| (e_1 \&\& e_2)) = e_1)$.

Предложение 3.14. *Законы упрощения условного И:*
$((e \&\& e) = e)$;

$((e\&\&T) = e)$;
$((e\&\&F) = F)$, *если* e *определено;*
$((e_1\&\&(e_1 \| e_2)) = e_1)$.

Предложение 3.15. *Закон тождества:*
$(e = e)$.

4. Расширение понятия предиката. В реальных программах далеко не все переменные имеют логический тип, что означает невозможность их описания с помощью предикатов, введенных определением 3.9. Так, например, выражение $i < 3$ для целочисленной переменной i предикатом в смысле упомянутого определения заведомо не является, что плохо. Эти и некоторые другие причины побуждают расширить множество предикатов, что и будет сейчас сделано в три этапа.

Определение 3.17. Будем называть *предикатом с переменными любых типов* выражение, которое может быть получено из предиката P (в смысле определения 3.9) заменой произвольного идентификатора id на любое заключенное в скобки выражение логического типа с переменными различных типов. Вычисление значения получившегося предиката в произвольном состоянии немедленно сводится к вычислению значения исходного предиката P.

Начиная с этого момента выражение $((i < 3) \vee (j = 0))$ — предикат, так как для него можно указать следующую цепочку вывода: $e \to (e \vee e) \to (id_1 \vee e) \to (id_1 \vee id_2) \to ((i < 3) \vee id_2) \to ((i < 3) \vee (j = 0))$. Множеством состояний этого предиката является пространство $\mathbb{Z}^2$, так как каждая из двух входящих в него целых переменных может изменяться независимо от другой. Если i и j — программные переменные, то пространство $\mathbb{Z}^2$ следует заменить на $\mathbb{Z}^2_M$.

Примером предиката, который определен в состоянии, в котором одна из входящих в него переменных не является определенной, может служить выражение $P = ((a = 0)||(b/a > 0))$. В состоянии $s = \{(a, 0), (b, U)\}$ он истинен: $P(s) = T$. Подобные предикаты возникают при описании фрагментов программ типа

```
if (a==0 || b/a > 0) ...
```

Следующим шагом в направлении расширения понятия предиката будет использование кванторов существования $\exists$ и всеобщности $\forall$.

Если $P(x)$ — предикат в смысле определения 3.17, зависящий от переменной x произвольного типа, а X — некоторое множество, то будем считать предикатами выражения $(\exists x(x \in X \wedge P(x)))$ и $(\forall x(x \in X \Rightarrow P(x)))$. Первое из них означает, что *существует хотя бы одно* $x \in X$ *для которого выполнено* $P(x)$, а второе — что *для всех* $x \in X$ *справедливо* $P(x)$.

Часто вместо приведенных выше достаточно громоздких форм записи используют следующие более удобные, которые мы тоже будем считать допустимыми выражениями для предикатов.

$$((\exists x \in X)P(x)) = (\exists x(x \in X \wedge P(x))),$$

$$((\forall x \in X)P(x)) = (\forall x(x \in X \Rightarrow P(x))).$$

Когда используемое множество X понятно из контекста, его опускают, что приводит к выражениям $(\exists x\ P(x))$ и $(\forall x\ P(x))$.

Кроме кванторов существования и всеобщности иногда используют еще один квантор — квантор $\exists!$ существования и единственности, который может быть определен с помощью двух основных кванторов следующим образом:

$$((\exists! x)P(x)) = (\exists x(P(x) \wedge \forall y(P(y) \Rightarrow (y = x)))).$$

Использование кванторов в предикатах должно удовлетворять некоторым дополнительным ограничением. *Связанным идентификатором* будем называть идентификатор, непосредственно следующий в предикате за квантором, а *свободным идентификатор* — идентификатор, не являющийся связанным. *Ограничение на использование кванторов в предикатах* таково: в предикате один и тот же идентификатор не может быть как связанным, так и свободным, и, кроме того, идентификатор не может быть связан двумя различными кванторами.

В предикате $R = (\forall i(m \leqslant i < n \Rightarrow x * i > 0))$ идентификатор i является связанным (квантором $\forall$), а идентификаторы m, n и x — свободными. Выражение $(i > 0 \wedge (\forall i(m \leqslant i < n \Rightarrow x * i > 0)))$ предикатом мы считать не можем, ибо i в нем является одновременно и свободным идентификатором и связанным, что недопустимо. Его легко слегка изменить так, чтобы оно стало предикатом: $(i > 0 \wedge (\forall k(m \leqslant k < n \Rightarrow x * k > 0)))$ — уже предикат.

Третьим шагом в направлении расширения множества языка предикатов будет ослабление требования на наличие в предикате всех тех скобок, которые возникают в процессе его вывода. Напомним, что с точки зрения определения 3.9 выражение $a \vee b$, например, предикатом не является, что не слишком удобно с практической точки зрения.

Разрешим удалять из предиката все те пары скобок, которые можно опустить без потери его однозначного толкования. При этом семантика полученного предиката определяется приоритетом операций: сначала вычисляются выражения внутри скобок, затем — логические выражения, заменившие логические идентификаторы id в смысле определения 3.17, после этого — кванторы существования и всеобщности, а затем — логические операции !, && и $\wedge$, || и $\vee$, $\Rightarrow$ и, наконец, $=$.

Таким образом, мы принимаем следующее определение.

Определение 3.18. Будем называть *предикатом в расширенном смысле* предикат с переменными любых типов (см. определение 3.17), который может содержать кванторы и не иметь скобок, не являющимися необходимыми для его однозначного толкования.

Подробное описание приоритетов операторов в программах на языке Java приведено в следующей секции текущего параграфа, а пока совет — *в случае сомнения всегда применяйте скобки для достижения нужного порядка вычисления выражения*. Этот совет актуален и для программ и для предикатов.

Рассмотрим, как решаются типичные задачи на вычисление значения предикатов в расширенном смысле в различных состояниях.

Задача 3.4. Вычислите значения предикатов $P_1 = (x = 0 \wedge x/(y-2) = 0)$ и $P_2 = (x = 0 \;\&\&\; x/(y-2) = 0)$ в состоянии $s = \{(x, 7), (y, 2)\}$.

Решение. Если восстановить в этих предикатах все недостающие скобки, то мы получим предикаты $P_3 = ((x = 0) \wedge ((x/(y-2)) = 0))$ и $P_4 = ((x = 0)\&\&((x/(y-2)) = 0))$ соответственно. В состоянии s выражение $(x = 0)$ является ложным, ибо $(7 = 0) = F$, а $x/(y-2)$ имеет значение U (не определено). В соответствии с таблицами истинности для операций $\wedge$ и $\&\&$ можно сделать вывод, что $P_1(s) = U$, а $P_2(s) = F$.

Задача 3.5. Вычислите значения предиката $P = (\exists i \; 0 \leqslant i \leqslant 9 \; (i^2 \leqslant 0)) \wedge (\forall j \; j^2 \geqslant k)$ в состоянии $s = \{(k, 0)\}$.

Решение. Предикат P представляет из себя конъюнкцию двух предикатов, первый из которых — это $\exists i \; 0 \leqslant i \leqslant 9 \; i^2 \leqslant 0$, а второй в состоянии s совпадает с $\forall j \; j^2 \geqslant 0$. Так как при $i = 0$ выполнено $i^2 \leqslant 0$, то первый из них истинен, а истинность второго предиката не вызывает сомнений. Таким образом, предикат $P(s)$, являющийся конъюнкцией двух истинных выражений, сам является истинным, — $P(s) = T$.

Для дальнейшего нам полезно формализовать понятие подстановки, которое мы фактически уже неоднократно использовали для вычисления значения предиката в заданном состоянии.

Определение 3.19. *Подстановкой* E_e^x называется выражение, получающееся одновременной подстановкой e вместо всех свободных вхождений x в E.

Вот несколько простых примеров: $(x + y)_z^x = (z + y)$; для $E = x < y \wedge (\forall i \; i < n \; f(i) < y)$ имеем $E_{x+y}^y = x < x + y \wedge (\forall i \; i < n \; f(i) < x + y))$, но $E_k^i = E$, так как i не свободно в E.

Для предикатов с кванторами справедливы дополнительные законы эквивалентности, называемые также правилами построения отрицания.

Предложение 3.16. *Законы построения отрицания:*
$!(\exists x P(x)) = \forall x (!P(x))$;
$!(\forall x P(x)) = \exists x (!P(x))$.

5. Приоритеты и ассоциативность операторов языка Java. При вычислении значения выражения в языке Java важны не только приоритеты, но и ассоциативность операторов.

Определение 3.20. Оператор `@` является левоассоциативным, если выражение `a @ b @ c` вычисляется, как `(a @ b) @ c`; правоассоциативным, если оно эквивалентно `a @ (b @ c)`; неассоциативным — если запись `a @ b @ c` не имеет смысла.

В этом определении символ `@` означает любой из бинарных операторов языка.

Использование круглых скобок для группировки всегда позволяет изменить порядок вычислений, так как выражения в скобках вычисляются в первую очередь. Иногда даже формально лишние скобки в выражении полезны — они облегчают правильное восприятие.

В таблице 5 все операторы языка Java разбиты на группы с одинаковым приоритетом (операторы с приоритетом 1 выполняются в первую очередь), левоассоциативность обозначена символом $\longrightarrow$, а правоассоциативность — символом $\longleftarrow$.

Таблица 5. Приоритеты и ассоциативность операторов

Пр-т	Оператор	Тип опер.	Асс-ть	Операция
1	`++`	числовой	$\longleftarrow$	пре- и постинкремент
	`--`	числовой	$\longleftarrow$	пре- и постдекремент
	`+, -`	числовой	$\longleftarrow$	унарные плюс и минус
	`~`	целый	$\longleftarrow$	побитовое дополнение
	`!`	логический	$\longleftarrow$	логическое отрицание
	`(type)`	любой	$\longleftarrow$	преобразование типа
2	`*, /, %`	числовой	$\longrightarrow$	умножение, деление и остаток
3	`+, -`	числовой	$\longrightarrow$	сложение и вычитание
	`+`	строковый	$\longrightarrow$	конкатенация строк
4	`<<`	целый	$\longrightarrow$	сдвиг влево
	`>>`	целый	$\longrightarrow$	сдвиг вправо с размножением знакового бита
	`>>>`	целый	$\longrightarrow$	сдвиг вправо с размножением нуля

Таблица 5. Приоритеты и ассоциативность операторов

Пр-т	Оператор	Тип опер.	Асс-ть	Операция
5	`<`, `<=`	числовой	$\longrightarrow$	меньше, меньше или равно
	`>`, `>=`	числовой	$\longrightarrow$	больше, больше или равно
	`instanceof`	объект, тип	$\longrightarrow$	сравнение типов
6	`==`	простой	$\longrightarrow$	равенство значений простых типов
	`!=`	простой	$\longrightarrow$	неравенство значений простых типов
	`==`	объект	$\longrightarrow$	равенство ссылок на объекты
	`!=`	объект	$\longrightarrow$	неравенство ссылок на объекты
7	`&`	целый	$\longrightarrow$	побитовое *И*
	`&`	логический	$\longrightarrow$	логическое *И*
8	`^`	целый	$\longrightarrow$	побитовое *исключающее Или*
	`^`	логический	$\longrightarrow$	логическое *исключающее Или*
9	`|`	целый	$\longrightarrow$	побитовое *Или*
	`|`	логический	$\longrightarrow$	логическое *Или*
10	`&&`	логический	$\longrightarrow$	условное *И*
11	`||`	логический	$\longrightarrow$	условное *Или*
12	`?:`	логический, любой, любой	$\longleftarrow$	оператор условия
13	`=`	любой	$\longleftarrow$	присваивание, присваивание с операцией
	`*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`			

С логическими и условными операторами мы уже знакомы, семантика арифметических операторов объясняется в следующем параграфе, а с оператором `instanceof` мы разберемся в третьей главе книги.

6. Задачи для самостоятельного решения.

Задача 3.6. Докажите, что выражение $((e_1 \wedge (e_2 \vee e_3)) = ((e_1 \wedge e_2) \vee (e_1 \wedge e_3)))$ является предикатом.

Задача 3.7. Докажите, что выражение $((a \vee a)$ — не предикат.

Задача 3.8. Изобразите деревья вывода для каждого из законов эквивалентности (см. страницу 45).

Задача 3.9. Покажите, что все законы эквивалентности (см. стр. 45) являются тавтологиями.

Задача 3.10. Решите задачу о банке с кофейными зернами (см. стр. 37)

§ 4. Особенности представления чисел в ЭВМ

Как уже отмечалось ранее, множествам целых $\mathbb{Z}$ и действительных $\mathbb{R}$ чисел в большинстве языков программирования соответствуют их *машинные аналоги*. В случае языка Java используемые в программах переменные величины и константы типов `int` и `double` принимают значения из множеств $\mathbb{Z}_M$ и $\mathbb{R}_M$ соответственно. В этом параграфе мы разберемся с тем, как именно устроены эти множества, и каковы последствия того, что программы оперируют не с настоящими числами, а с элементами указанных множеств. Однако сначала — некоторые напоминания об информации вообще и ее представлении в ЭВМ.

1. Представление информации в компьютере. Любая информация (числовая, текстовая, звуковая, графическая и т.д.) в компьютере представляется (кодируется) в так называемой *двоичной форме*. Как *оперативная*, так и *внешняя память*, где и хранится вся информация, могут рассматриваться, как достаточно длинные последовательности из нулей и единиц. Под внешней памятью подразумеваются такие носители информации, как магнитные и оптические диски, ленты и т.п.

Единицей измерения информации является *бит (BInary digiT)* — именно такое количество информации содержится в ответе на вопрос: нуль или один? Более крупными единицами измерения информации являются байт, килобайт (Kbyte), мегабайт (Mbyte), гигабайт (Gbyte) и терабайт (Tbyte). Один *байт (byte)* состоит из восьми бит, а каждая последующая величина больше предыдущей в 1024 раза.

Байта достаточно для хранения 256 различных значений, что позволяет размещать в нем любой из алфавитно-цифровых символов, если только мы можем ограничиться языками с небольшими алфавитами типа русского или английского. Первые 128 символов (занимающие семь младших бит) стандартизированы с помощью кодировки ASCII (American Standart

Code for Information Interchange). Хуже обстоит дело с кодировками русского текста (символы русского алфавита расположены во второй половине таблицы из 256 символов) — их несколько, а наиболее распространенные из них сейчас две — Windows-1251 и KOI8-R.

Для кодирования всех возможных символов, используемых народами мира, одного байта мало — необходимо использовать два последовательных (стандарт Unicode). Именно так и поступают при хранении символьных (`char`) значений в языке Java.

Полезно знать, что $2^{10} = 1024 \approx 1000 = 10^3$. Учитывая, что в книге среднего размера около 300000 букв, легко подсчитать, что, даже не используя никаких средств сжатия информации, на жестком диске современного персонального компьютера емкостью в 20 гигабайт можно разместить большую библиотеку из почти 70000 книг.

2. Целые числа. К целочисленным типам в языке Java относятся `byte`, `short`, `int` и `long`. Для хранения значений этих типов на любом компьютере отводится один, два, четыре и восемь байт соответственно. При этом применяется представление чисел в так называемом *двоичном дополнительном коде*.

Напомним, что используемая нами обычная система счисления является *позиционной* с основанием 10. Это значит, что в ней все натуральные числа представляются с помощью десяти цифр (от нуля до девяти включительно), а значение каждой из цифр числа зависит от позиции: самая правая цифра означает число единиц (10^0), вторая — десятков (10^1), третья — сотен (10^2) и так далее.

В p-ичной системе счисления все точно также, только число 10 в предыдущем абзаце нужно всюду заменить на p. Наряду с *двоичной* системой, в которой только две цифры (0 и 1), в информатике часто применяются *восьмеричная* с цифрами от нуля до 7 и *шестнадцатеричная*. В последнем случае в качестве цифр от десяти до пятнадцати используются буквы от A до F соответственно.

При записи положительных целых чисел в системе счисления с основанием p (на компьютере $p = 2$) все их множество оказывается состоящим из элементов вида

$$d = \alpha_{n-1}p^{n-1} + \alpha_{n-2}p^{n-2} + \ldots + \alpha_0,$$

где величины α_i для всех i из диапазона от $n-1$ до нуля — это цифры n-значного числа d в p-ичной системе счисления.

Перейдем теперь к вопросу представления отрицательных чисел. Для определенности рассмотрим тип `byte`, в котором любое число занимает ровно восемь бит. Из записи в двоичной системе счисления равенства

$(-1)+1=0$ легко найти, какой вид должно иметь неизвестное нам пока двоичное представление `xxxxxxxx` числа -1:

```
xxxxxxxx + 00000001 = 00000000
```

Ясно, что на месте символов `xxxxxxxx` должно быть расположено число `11111111`. Правильным результатом при этом, конечно, следовало бы считать `100000000`, а не `00000000`, но ведь мы имеем дело с типом `byte` и, так как результат обязан разместиться в байте, единица «исчезает».

Итак, число -1 должно кодироваться как `11111111`. Дальнейшее уже совсем просто: для получения -2 нужно -1 уменьшить на единицу, что даст `11111110`; число -3 представляется как `11111101` и т.д.

Отрицательные числа всегда имеют в своем двоичном представлении единицу в самом старшем разряде, который поэтому называют *знаковым*, а абсолютная величина кодируемого числа получается как двоичное дополнение остальных бит (нули нужно заменить на единицы и наоборот), увеличенное на один.

Легко видеть, что при этом самым маленьким отрицательным числом, которое принадлежит типу `byte`, является число -128 (двоичное представление `10000000`), а самым большим — число 127 (представление `01111111`). Все представимые числа (а их 256) в данном случае могут быть получены как пересечение двух множеств: множества $\mathbb{Z}$ всех целых чисел и отрезка $[-128, 127]$. Интересным является следующее наблюдение: если число `01111111` увеличить на единицу, то получится `10000000`, что означает следующее: $127+1=-128$!

Итак, множество элементов типа `byte` можно представлять себе в виде свернутого в кольцо отрезка $[-128, 127]$. Принципиально ничего не меняется и для типов `short`, `int` и `long` — увеличивается только длина отрезка, который вырезается из действительной прямой перед сворачиванием его в кольцо. Минимальные и максимальные представимые значения для каждого из этих типов в языке Java определены, как значения констант `MIN_VALUE` и `MAX_VALUE` в классах `java.lang.Short`, `java.lang.Integer` и `java.lang.Long` соответственно.

То, что для элементов множества $\mathbb{Z}_M$, являющегося машинным аналогом $\mathbb{Z}$, нарушено фундаментальное свойство целых чисел $x+1>x$, способно привести к различным невероятным на первый взгляд результатам, однако гораздо более странные вещи происходят при работе с вещественными числами.

3. Вещественные числа. Обсуждаемые в данной секции вопросы значительно более полно рассмотрены в четвертой главе классической книги Кнута [**7**].

Для представления вещественных чисел в языке Java используют переменные и константы типов `float` и `double`. Величины первого из них занимают четыре байта, а второго — восемь.

В отличие от множества целых чисел $\mathbb{Z}$ вещественные числа $\mathbb{R}$ обладают свойством *полноты*: между любыми двумя различными числами всегда найдется отличное от них третье. Понятно, что любой из способов представления бесконечного множества вещественных чисел с помощью некоторого конечного множества $\mathbb{R}_M$ не даст возможности сохранить свойство полноты. Наиболее распространенным способом реализации вещественных чисел на ЭВМ является использование *чисел с плавающей точкой*. При этом множество $\mathbb{R}_M$ оказывается состоящим из элементов вида

$$f = \pm(\frac{\alpha_1}{p} + \frac{\alpha_2}{p^2} + \ldots + \frac{\alpha_n}{p^n})p^e,$$

где целые числа α_i для всех i из диапазона от 1 до n удовлетворяют соотношению $0 \leqslant \alpha_i \leqslant p-1$, а величина e лежит в диапазоне от L до U ($L \leqslant e \leqslant U$). Число p является при этом основанием системы счисления (чаще всего это 2), константа n задает точность представления чисел (для типа `double` она больше, чем для `float`), а диапазон $[L, U]$ определяет область значений экспоненты (для типа `double` он также больше, чем для `float`).

Число $\pm(\frac{\alpha_1}{p} + \frac{\alpha_2}{p^2} + \ldots + \frac{\alpha_n}{p^n})$ принято называть *мантиссой* числа f с плавающей точкой. Часто требуют, чтобы для всех чисел $f \neq 0$ величина α_1 была ненулевой. Такие числа с плавающей точкой называют *нормализованными*.

В языке Java для кодирования величин типов `float` и `double` также используют числа с плавающей точкой. При этом часть из имеющегося множества бит используют для размещения *экспоненты* e, а остальные биты — для размещения мантиссы.

Для того чтобы хорошо понять, что же представляет из себя множество $\mathbb{R}_M$ нормализованных чисел с плавающей точкой, полезно изобразить его на числовой прямой для случая небольших n, L и U. Подобная задача приведена в конце параграфа. Сейчас же нам будет достаточно весьма качественного описания этого множества.

Так как $\mathbb{R}_M$ симметрично относительно начала координат, то можно разобраться только с неотрицательными числами. Нуль, конечно же, принадлежит искомому множеству. Ближайшая к нулю следующая точка получается при $\alpha_1 = 1$, $\alpha_2 = \alpha_3 = \ldots = \alpha_n = 0$ и $e = L$. Это число для чисел типов `float` и `double` определено, как `MIN_VALUE` в классах `java.lang.Float` и `java.lang.Double` соответственно.

Правее располагается множество точек, следующих друг за другом с шагом 2^{L-n}. Затем шаг увеличивается. Потом еще. И еще. При $e = U$ расстояние между двумя соседними точками множества $\mathbb{R}_M$ достигает 2^{U-n}. Самая правая точка множества получается при $\alpha_1 = \alpha_2 = \ldots = \alpha_n = 1$ и $e = U$. Для типов `float` и `double` это число определено, как `MAX_VALUE` в классах `java.lang.Float` и `java.lang.Double`.

Кроме перечисленных (и симметричных им отрицательных) значений в результате выполнения некоторых операций могут получиться также следующие особые значения: *плюс бесконечность* (`POSITIVE_INFINITY`), *минус бесконечность* (`NEGATIVE_INFINITY`), *минус ноль* и *не число* (`NaN`). Например, при делении единицы на *минус ноль* получается *минус бесконечность*.

Описанные особенности множества машинных вещественных чисел $\mathbb{R}_M$ приводят к тому, что не для всех его элементов верны следующие соотношения, всегда справедливые для множества настоящих вещественных чисел $\mathbb{R}$:

- $x + 1 > x$;
- $x + x$ существует;
- $(x \cdot 2)/2 = x$;
- $(x/2) \cdot 2 = x$;
- $a + (b + c) = (a + b) + c$;
- $a + b + c = c + b + a$.

Приведем несколько примеров, иллюстрирующих эти особенности множества $\mathbb{R}_M$.

Задача 4.1. Предъявите действительное (типа `double`) число x такое, что $x + 1 = x$, а $(x \cdot 2)/2 \neq x$. Воспользуйтесь тем, что класс `java.lang.Double` определяет константу `MAX_VALUE`.

Текст программы.

```
public class DblMaxVal {
    public static void main(String[] args) {
        double x = Double.MAX_VALUE;
        double y = x + 1.0;

        if (x == y) {
            Xterm.print("Для числа ");
            Xterm.print("x = " + x, Xterm.Blue);
            Xterm.print(" величины x и (x+1) ");
            Xterm.print("равны!\n", Xterm.Red);
        }
```

```
        y = 2.0 * x;
        double z = y / 2.0;

        if (x != z) {
            Xterm.print("Для числа ");
            Xterm.print("x = " + x, Xterm.Blue);
            Xterm.print(" величины x и (2.0*x)/2.0 ");
            Xterm.print("различны\n", Xterm.Red);
            Xterm.print("и равны соответственно");
            Xterm.print(" " + x, Xterm.Red);
            Xterm.print(" и");
            Xterm.print(" " + z + "\n", Xterm.Red);
        }
    }
}
```

Задача 4.2. Предъявите такие действительные (типа `double`) числа a, b и c такие, что $a + (b + c) \neq (a + b) + c$.

Достаточно вспомнить, что точки множества $\mathbb{R}_M$ расположены на числовой прямой неравномерно.

Текст программы.

```
public class DblNoAssociative {
    public static void main(String[] args) {
        double x = 1.0e-16;

        double y =  1.  + (x  +  x);
        double z = (1.  +  x) +  x ;

        if (y != z) {
            Xterm.print("Для числа ");
            Xterm.print("x = " + x, Xterm.Blue);
            Xterm.print(" величины 1.+(x+x) и (1.+x)+x ");
            Xterm.print("различны\n", Xterm.Red);
            Xterm.print("и равны соответственно");
            Xterm.print(" " + y, Xterm.Red);
            Xterm.print(" и");
            Xterm.print(" " + z + "\n", Xterm.Red);
        }
    }
}
```

Иногда даже работа с казалось бы не слишком маленькими или огромными по абсолютной величине числами может привести к удивительным результатам. Рассмотрим задачу о решении квадратного уравнения.

Задача. Напишите программу, вводящую действительные коэффициенты a, b и c квадратного уравнения $ax^2 + bx + c = 0$ с положительным дискриминантом, находящую оба корня этого уравнения.

Вот вполне естественное решение этой задачи, которое может быть написано любым человеком, обладающим минимальными знаниями языка Java.

Текст программы.

```
public class SquareEquation {
    public static void main(String[] args) throws Exception {
        double a = Xterm.inputDouble("Введите число a -> ");
        double b = Xterm.inputDouble("Введите число b -> ");
        double c = Xterm.inputDouble("Введите число c -> ");

        if (a == 0.) {
            Xterm.println("Уравнение не квадратное", Xterm.Red);
            System.exit(0);
        }

        if (b*b - 4.*a*c <= 0.) {
            Xterm.println("Дискриминант неположителен", Xterm.Red);
            return;
        }

        double x1 = ( -b - Math.sqrt(b*b - 4.*a*c) ) / (2.*a);
        double x2 = ( -b + Math.sqrt(b*b - 4.*a*c) ) / (2.*a);

        Xterm.println("Корни уравнения:");
        Xterm.println("x1 = " + x1, Xterm.Blue);
        Xterm.println("x2 = " + x2, Xterm.Blue);
    }
}
```

Для извлечения квадратного корня здесь используется метод `sqrt` класса `Math`. Вызов метода `System.exit` и применение оператора `return`, которые в теле метода `main` почти эквивалентны и приводят к немедленному завершению выполнения программы, позволяет обойтись без ветви `else` оператора `if-else`. Остальная часть программы комментариев не требует.

Попробуйте, однако, выполнить эту программу при `a=0.2E-16` и `b=c=1.0` — первый корень уравнения `x1` окажется равным `-5.0E16`, а второй `x2` — нулю.

Если подставить эти значения в выражение $ax^2 + bx + c$, то и для `x1` и для `x2` его значение окажется равным единице, а не нулю. Подобная

ошибка для первого, весьма большого по величине корня, представляется вполне естественной, но вот соглашаться с тем, что нуль является корнем данного уравнения, совсем не хочется. Этой проблеме посвящена одна из приведенных ниже задач.

4. Арифметические и побитовые операторы языка Java. К арифметическим операторам языка Java, которые определены для всех числовых типов, относятся: `+` (сложение и унарный плюс), `+=` (сложение с присваиванием), `-` (вычитание и унарный минус), `-=` (вычитание с присваиванием), `*` (умножение), `*=` (умножение с присваиванием), `/` (деление), `/=` (деление с присваиванием), `%` (остаток от деления или деление по модулю), `%=` (остаток от деления с присваиванием), `++` (инкремент) и `--` (декремент).

Любой оператор с присваиванием с точки зрения получаемого результата эквивалентен выполнению соответствующей операции с последующим присваиванием, но работает обычно быстрее: `a @= b` эквивалентно `a = a @ b` (здесь символ `@` означает любую из бинарных арифметических операций).

Результат выполнения операций деления и деления по модулю (нахождения остатка) для целочисленных операндов является целым числом. Так, например, `5/3` равняется 1.

Наконец, операции инкремента и декремента соответственно увеличивают и уменьшают свой операнд на единицу. Они бывают префиксными и тогда изменение их операнда происходит *до* того, как выполняются другие действия в выражении, и постфиксными, когда изменение происходит *после* вычисления выражения:

```
int i = 2, j = 3;   // i = 2, j = 3
int k = i + j++;    // k = 5, j = 4
int m = --i;        // i = 1, m = 1
```

С величинами целых типов можно выполнять дополнительные действия. К побитовым (или поразрядным) операторам языка Java относятся следующие: `~` (дополнение), `&` (побитовое *И*), `&=` (побитовое *И* с присваиванием), `|` (побитовое *Или*), `|=` (побитовое *Или* с присваиванием), `^` (исключающее *Или*), `^=` (исключающее *Или* с присваиванием), `<<` (сдвиг влево), `<<=` (сдвиг влево с присваиванием), `>>` (сдвиг вправо), `>>=` (сдвиг вправо с присваиванием), `>>>` (сдвиг вправо с размножением нуля) и `>>>=` (сдвиг вправо с присваиванием с размножением нуля).

Результаты операций побитового дополнения, *Или*, исключающего *Или* и *И* получаются путем выполнения над каждым из битов операнда (операндов) действий в соответствии с таблицей 6.

Таблица 6. Битовые операции

a	b	~a	a|b	a^b	a&b
0	0	1	0	0	0
1	0	0	1	1	0
0	1	1	1	1	0
1	1	0	1	0	1

Например, `4^5` равняется 1 (ибо двоичные представления этих чисел есть соответственно `100` и `101`).

У операторов сдвига второй аргумент показывает количество разрядов, на которое надо осуществить сдвиг в двоичном представлении первого операнда влево (для `<<`) или вправо (для `>>` и `>>>`).

```
int i = 3, j = 100; // i = 3, j = 100
int k = i << 4;     // k = 48
int m = j >> 2;     // m = 25
```

Легко понять, что сдвиг влево на n разрядов эквивалентен умножению на 2^n, а сдвиг вправо — делению на то же число.

5. Задачи для самостоятельного решения.

Задача 4.3. Предъявите целое число x такое, что $x + 1 < x$.

Задача 4.4. Явно перечислите и изобразите на числовой прямой все точки множества $\mathbb{R}_M$, сделав следующие допущения: числа хранятся в нормализованной форме с плавающей точкой; для хранения как мантиссы, так и порядка числа отводится по три бита (из которых в обоих случаях один является знаковым); никаких особых значений нет.

Задача 4.5. Предъявите действительное (типа `double`) число x такое, что $(x/2) \cdot 2 \neq x$. Воспользуйтесь тем, что класс `java.lang.Double` определяет константу `MIN_VALUE`.

Задача 4.6. Определите (приближенно) $MACHEPS$ (машинное эпсилон) для типов `double` и `float`. Машинным эпсилоном называется наибольшее число x данного типа, удовлетворяющее соотношению $1 + x = 1$.

Задача 4.7. Предъявите последовательность чисел (типа `float`), при суммировании которой в прямом и обратном порядке результаты будут отличаться не менее, чем вдвое.

Задача 4.8. Напишите программу, вводящую действительные коэффициенты a, b и c квадратного уравнения $ax^2 + bx + c = 0$ с положительным дискриминантом, находящую оба корня этого уравнения достаточно точно во всех случаях.

6. Числа произвольной длины и точности. При необходимости на любом алгоритмическом языке можно реализовать так называемую *длинную арифметику*, позволяющую работать с целыми числами произвольной длины. При этом числа размещаются в совокупности последовательных байтов необходимой длины. Естественно, что эффективность любых операций над такими числами значительно ниже, чем над величинами типа `int`.

В случае языка Java пакет `java.math` дает возможность программисту работать с целыми и вещественными числами произвольной длины и точности, не требуя при этом никакого дополнительного кодирования, однако обсуждение данного пакета в нашем курсе не предполагается.

Некоторые задачи, требующие реализовать арифметические операции над целыми числами достаточно большой величины, будут рассмотрены в курсе далее, а вот проблемы, связанные с особенностями представления действительных чисел, нас интересовать не будут. О них придется вспомнить позже, при изучении курса вычислительной математики.

§ 5. Рекурсия, итерация и оценки сложности алгоритмов

Более подробное изложение материала, рассматриваемого в данном параграфе, может быть найдено в книгах [**9**] и [**4**]. Вопросы, связанные с асимптотическими оценками сложности алгоритмов, подробно изложены в классической книге Кнута [**6**] и, охватывающем тематику двух первых курсов цикла программистских дисциплин, прекрасном издании [**8**].

1. Рекурсия и итерация. До сих пор мы рассматривали либо линейные программы, либо программы с ветвлениями, не требовавшими многократного повторения одних и тех же операций. Гораздо чаще, однако, приходится иметь дело именно с такими ситуациями. Существует два основных подхода к решению задач подобного типа: рекурсия и итерация.

Определение 5.1. *Рекурсия* — это такой способ организации обработки данных, при котором программа вызывает сама себя непосредственно, либо с помощью других программ.

Определение 5.2. *Итерация* — способ организации обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ.

Математическая модель рекурсии заключается в вычислении рекурсивно определенной функции на множестве программных переменных. Примерами таких функций могут служить факториал числа и числа Фибоначчи. В каждом из этих случаев значение функции для всех значений аргумента, начиная с некоторого, определяется через предыдущие значения.

Факториал $n!$ целого неотрицательного числа n задается следующими соотношениями:

$$0! = 1,$$
$$n! = n \cdot (n-1)! \quad \text{для} \quad n > 0.$$

Числами Фибоначчи f_n называют последовательность величин 0, 1, 1, 2, 3, 5, 8, ..., определяемую равенствами:

$$f_0 = 0,$$
$$f_1 = 1,$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{для} \quad n > 1.$$

Математическая модель итерации сводится к повторению некоторого преобразования (отображения) $T\colon X \to X$ на множестве переменных программы X (прямом произведении множеств значений отдельных переменных). Программной реализацией итерации является обычно некоторый цикл, тело которого осуществляет преобразование T.

В качестве примера можно рассмотреть схему вычисления факториала натурального числа в соответствии с его другим определением: $n! = 1 \cdot 2 \cdot \ldots \cdot n$. При написании программы в соответствии с ним нужно работать с двумя величинами целого типа $\mathbb{Z}_M$: числом i, которое будет играть роль счетчика и изменяться от 1 до n включительно, и величиной k, в которой будет накапливаться произведение чисел от 1 до i.

Пространством X в данном случае будет $\mathbb{Z}_M^2$, в качестве начальной точки в этом пространстве возьмем точку $(1, 1)$ (что соответствует $i = k = 1$), а преобразование $T\colon X \to X$ будет иметь вид $T(i, k) = (i + 1, k * i)$. В случае, например, трехкратного применения преобразования T получим $T(T(T(1, 1))) = T(T(2, 1)) = T(3, 2) = (4, 6)$, что обеспечит вычисление факториала числа 3.

Рекурсия и итерация взаимозаменяемы. Более точно, справедливо следующее утверждение.

Теорема 5.1. *Любой алгоритм, реализованный в рекурсивной форме, может быть переписан в итерационном виде, и наоборот.*

Данная теорема не означает, что временна́я и емкостна́я эффективности получающихся программ обязаны совпадать. Однако *наилучшие* рекурсивный и итерационный алгоритм имеют совпадающую с точностью до постоянного множителя временну́ю сложность.

2. Особенности рекурсивных программ. Решая некоторую задачу, рекурсивный алгоритм вызывает сам себя для решения ее подзадач. Вот классический пример.

Задача 5.1. Напишите рекурсивную программу, вычисляющую факториал введенного натурального числа.

Текст программы.

```
public class FactR {
    static int f(int x) {
        return (x == 0) ? 1 : x * f(x-1);
    }
    public static void main(String[] args) throws Exception {
        Xterm.println("n!=" + f(Xterm.inputInt("n=")));
    }
}
```

Организация рекурсивных вычислений на языке Java не требует использования никаких специальных конструкций — достаточно известного нам вызова метода. В приведенной программе метод `f` получает на вход целое число `x` и рекурсивно вычисляет его факториал. Рассмотрим процесс выполнения данной программы для $n = 3$ более подробно.

Вызов метода `f` с аргументом 3 представим в виде листа бумаги, на котором указано входное значение (число 3). Универсальный исполнитель должен выполнить предписанные действия на этом листе бумаги и получить итоговый результат. Так как число 3 отлично от нуля, универсальный исполнитель попытается умножить число 3 на `f(2)`, но последняя величина для ее вычисления требует вызова метода `f` с аргументом 2.

В результате выполнение метода `f` с аргументом 3 приостановится и Универсальный исполнитель приступит к выполнению этого же метода с меньшим на единицу значением аргумента, что можно себе представить в виде еще одного листа бумаги, положенного на первый.

Вычисление `f(2)` потребует нахождения `f(1)`, что вызовет появление еще одного листа бумаги — третьего экземпляра метода `f`. Он, в свою

очередь, вызовет `f(0)`. В этот момент накопится уже пачка листов (ее называют *стеком вызовов*) — целых четыре штуки. При этом вычисления на всех нижних листах приостановлены до завершения работы с верхними.

Далее события будут развиваться следующим образом. Метод `f`, вызванный с нулевым аргументом, самостоятельно вычисляет и возвращает с помощью оператора `return` результат — число 1. Верхний элемент из стека вызовов методов после этого удаляется и возобновляются вычисления величины `f(1)`. Этот процесс продолжается до тех пор, пока стек вызовов не станет пустым, что произойдет по завершению вычисления значения `f(3)`. Итоговое значение 6 будет возвращено в метод `main`, который его и напечатает.

При написании рекурсивных программ обычно необходимо исследовать два основных вопроса:
а) всегда ли и почему программа заканчивает работу?
б) почему после окончания работы программы будет получен требуемый результат?

Подобное исследование для рассмотренной программы провести достаточно просто. Докажем по индукции, что эта программа заведомо завершит свою работу для всех целых неотрицательных чисел.

Рассмотрим следующее утверждение P, зависящее от числа n: *программа завершает свою работу для входного значения n за конечное время*. Для того чтобы доказать истинность утверждения $P(n)$ для всех целых неотрицательных чисел с помощью метода математической индукции достаточно проверить его истинность при нулевом значении n (база индукции) и убедиться в справедливости индуктивного перехода, что требует проверки истинности предиката $P(n) \Rightarrow P(n+1)$.

При нулевом значении аргумента программа завершает свою работу немедленно, поэтому утверждение $P(0)$ истинно и база индукции проверена.

Пусть утверждение $P(n)$ истинно при некотором значении n. Покажем, что и $P(n+1)$ тогда истинно. Для вычисления `f(n+1)` в соответствии с текстом программы нужно перемножить `n+1` и `f(n)`. Истинность $P(n)$ гарантирует нам вычисление второго множителя за конечное время, а так как перемножение двух чисел тоже требует конечного времени, то и $P(n+1)$ истинно, что и завершает доказательство завершения работы программы при всех целых неотрицательных значениях аргумента.

Заметим, что для отрицательных значений $n \in \mathbb{Z}$ данная программа должна была бы работать бесконечно долго (как принято говорить, должна *зациклиться*). Однако из-за того, что в реальности мы имеем дело с машинным заменителем множества целых чисел — множеством

$\mathbb{Z}_M$, выполнение всегда завершится за конечное время, хотя полученный результат и не будет иметь никакого смысла.

Доказательство правильности вычисляемого приведенной программой значения проводится совершенно аналогично. Рассмотрим утверждение $P(n) = (\textit{для входного значения } n \textit{ результатом является } n!)$ и докажем его истинность по индукции.

Истинность $P(0)$ (база индукции) проверяется непосредственно, а для проверки корректности индуктивного перехода напишем следующую цепочку равенств: $P(n+1) = (n+1) \cdot P(n) = (n+1) \cdot n! = (n+1)!$, первое из которых следует из текста программы, второе — из предположения индукции, а третье — из определения факториала. Это завершает доказательство *теоретической* правильности написанной программы.

В реальности, однако, быстрый рост функции $n!$ и ограниченность множества $\mathbb{Z}_M$ приводят к тому, что эта программа позволяет получить правильные результаты только при очень небольших значениях n. Уже при $n = 13$ печатаемое ей значение 1932053504 отличается от правильного 6227020800, а при $n = 17$ программа выдает даже отрицательный результат!

Рассмотрим еще одну задачу.

Задача 5.2. Напишите рекурсивную программу, печатающую n-ое число Фибоначчи.

Текст программы.

```
public class FibR {
    static int fib(int x) {
        return (x > 1) ? fib(x-2) + fib(x-1) : (x == 1) ? 1 : 0;
    }
    public static void main(String[] args) throws Exception {
        int n = Xterm.inputInt("Введите n -> ");
        Xterm.println("fib(" + n + ") = " + fib(n));
    }
}
```

Обратите внимание, что при вычислении f_5 в соответствии с этой программой понадобится найти f_4 и f_3. Определение f_4, в свою очередь, потребует вычисления f_3 и f_2, и так далее. Внимательно изучение содержимого стека вызовов для этой задачи показывает, что для нахождения каждого следующего числа Фибоначчи требуется примерно *вдвое* большее время, чем для определения предыдущего. Для того, чтобы убедиться в этом, нарисуйте дерево, изображающее процесс вычисления f_7 с помощью данной программы.

3. Java и циклические конструкции. Математическая модель итерации, описанная в первой секции данного параграфа не дает никаких практических рекомендаций, как именно реализовать вычисления. Во второй главе книги будут рассмотрены три более частных схемы обработки информации с помощью метода итерации — проектирование цикла при помощи *инварианта* и схемы вычислений *инвариантной* и *индуктивной* функций. Пока же ограничимся описанием управляющих конструкций языка Java, которые используют для организации циклов, и рассмотрением нескольких примеров программ.

Язык Java предусматривает три различных оператора цикла: `while`, `do-while` и `for`. Первый из них обычно используют в ситуации, когда тело цикла нужно выполнить нуль или более раз, а второй применяется, если его выполнение хотя бы раз обязательно. Третий из операторов является наиболее универсальным и используется в различных ситуациях.

Дополнительными средствами, используемыми при организации циклов, являются операторы `break` и `continue`. Первый из них прерывает выполнение цикла, а второй позволяет досрочно перейти к выполнению следующей итерации, проигнорировав часть операторов тела цикла, еще не выполненных в текущей итерации. Заметим, что в языке Java *отсутствует* оператор `goto`, — в тех редких ситуациях, где он мог бы оказаться полезным, можно использовать операторы `break` или `continue` с метками.

В качестве первого примера рассмотрим опять задачу вычисления факториала и программу, реализующую предложенную выше схему применения преобразования $T(i,k) = (i+1, k*i)$.

Задача 5.3. Напишите итерационную программу, вычисляющую факториал введенного натурального числа.

Текст программы.

```
public class FactIv1 {
    public static void main(String[] args) throws Exception {
        int n, i, k;
        n = Xterm.inputInt("Введите n -> ");
        i = k = 1;
        while (i <= n) {
            k *= i;
            i += 1;
        }
        Xterm.println("" + n + "! = " + k);
    }
}
```

Конструкцию `while` в ней можно заменить на `do-while`:

Фрагмент программы (FactIv2.java).

```
do {
    k *= i;
    i += 1;
} while (i <= n);
```

А еще лучше воспользоваться циклом `for`:

Фрагмент программы (FactIv3.java).

```
int i, k, n = Xterm.inputInt("Введите n -> ");
for (i = k = 1; i <= n; i++)
    k *= i;
Xterm.println("" + n + "! = " + k);
```

4. Основы оценок сложности алгоритмов. Нам уже известно, что правильность — далеко не единственное качество, которым должна обладать хорошая программа. Одним из важнейших является эффективность, характеризующая прежде всего время выполнения программы $T(n)$ для различных входных данных (параметра n).

Нахождение точной зависимости $T(n)$ для конкретной программы — задача достаточно сложная. По этой причине обычно ограничиваются *асимптотическими оценками* этой функции, то есть описанием ее примерного поведения при больших значениях параметра n. Иногда для асимптотических оценок используют традиционное отношение O (читается «О большое») между двумя функциями $f(n) = O(g(n))$, определение которого можно найти в любом учебнике по математическому анализу, хотя чаще применяют отношение эквивалентности Θ (читается «тэта большое»). Его формальное определение есть, например, в книге [**8**], хотя нам пока достаточно будет понимания данного вопроса в общих чертах.

В качестве первого примера вернемся к только что рассмотренным программам нахождения факториала числа. Легко видеть, что количество операций, которые должны быть выполнены для нахождения факториала $n!$ числа n в первом приближении прямо пропорционально этому числу, ибо количество повторений цикла (итераций) в данной программе равно n. В подобной ситуации принято говорить, что программа (или алгоритм) имеет *линейную сложность* (сложность $O(n)$ или $\Theta(n)$).

Можно ли вычислить факториал быстрее? Оказывается, да. Можно написать такую программу, которая будет давать правильный результат для тех же значений n, для которых это делают все приведенные выше программы, не используя при этом ни итерации, ни рекурсии. Ее сложность будет $\Theta(1)$, что фактически означает организацию вычислений по некоторой формуле без применения циклов и рекурсивных вызовов!

Не менее интересен и пример вычисления n-го числа Фибоначчи. В процессе ее исследования фактически уже было выяснено, что ее сложность является *экспоненциальной* и равна $\Theta(2^n)$. Подобные программы практически не применимы на практике. В этом очень легко убедиться, попробовав вычислить с ее помощью 40-е число Фибоначчи. По этой причине вполне актуальна следующая задача.

Задача 5.4. Напишите программу, печатающую n-ое число Фибоначчи, которая имела бы линейную сложность.

Вот решение этой задачи, в котором переменные `j` и `k` содержат значения двух последовательных чисел Фибоначчи.

Текст программы.

```
public class FibIv1 {
    public static void main(String[] args) throws Exception {
        int n = Xterm.inputInt("Введите n -> ");
        Xterm.print("f(" + n + ")");
        if (n < 0) {
            Xterm.print(" не определено\n");
        } else if (n < 2) {
            Xterm.println(" = " + n);
        } else {
            long i = 0;
            long j = 1;
            long k;
            int  m = n;
            while (--m > 0) {
                k  = j;
                j += i;
                i  = k;
            }
            Xterm.println(" = " + j);
        }
    }
}
```

Следующий вопрос вполне естественен — а можно ли находить числа Фибоначчи еще быстрее?

После изучения определенных разделов математики совсем просто вывести следующую формулу для n-ого числа Фибоначчи f_n, которую легко проверить для небольших значений n:

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right].$$

Может показаться, что основываясь на ней, легко написать программу со сложностью $\Theta(1)$, не использующую итерации или рекурсии.

Текст программы.

```
public class FibIv2 {
    public static void main(String[] args) throws Exception {
        int    n = Xterm.inputInt("Введите n -> ");
        double f = ( 1.0 + Math.sqrt(5.) ) / 2.0;
        int    j = (int)( Math.pow(f,n) / Math.sqrt(5.) + 0.5 );

        Xterm.println("f(" + n + ") = " + j);
    }
}
```

На самом деле эта программа использует вызов функции возведения в степень `Math.pow(f,n)`, которая не может быть реализована быстрее, чем за логарифмическое время $(\log_2 n)$. Про алгоритмы, в которых количество операций примерно пропорционально $\log n$ (в информатике принято не указывать основание двоичного логарифма) говорят, что они имеет *логарифмическую сложность* $(\Theta(\log n))$.

Для вычисления n-го числа Фибоначчи существует такой алгоритм, программную реализацию которого мы приведем без дополнительных комментариев, — иначе нужно объяснять слишком много (связь чисел Фибоначчи со степенями некоторой матрицы порядка два, использование классов для работы с матрицами, алгоритм быстрого возведения матрицы в степень).

Задача 5.5. Напишите программу, печатающую n-ое число Фибоначчи, которая имела бы логарифмическую сложность.

Текст программы.

```
public class FibIv3 {
    public static void main(String[] args) throws Exception {
        int n = Xterm.inputInt("Введите n -> ");
        Xterm.print("f(" + n + ")");
        if (n < 0) {
            Xterm.println(" не определено");
        } else if (n < 2) {
            Xterm.println(" = " + n);
        } else {
            Matrix b = new Matrix(1, 0, 0, 1);
            Matrix c = new Matrix(1, 1, 1, 0);
            while (n>0) {
                if ((n&1) == 0) {
```

```
                    n >>>= 1; c.square();
                } else {
                    n  -=  1; b.mul(c);
                }
            }
            Xterm.println(" = " + b.fib());
        }
    }
}
class Matrix {
    private long a, b, c, d;

    public Matrix(long a, long b, long c, long d) {
        this.a = a; this.b = b; this.c = c; this.d = d;
    }
    public void mul(Matrix m) {
        long a1 = a*m.a+b*m.c;  long b1 = a*m.b+b*m.d;
        long c1 = c*m.a+d*m.c;  long d1 = c*m.b+d*m.d;
        a = a1; b = b1; c = c1; d = d1;
    }
    public void square() {
        mul(this);
    }
    public long fib() {
        return b;
    }
}
```

Если попробовать посчитать десятимиллионное число Фибоначчи с помощью этой и предыдущей программ, то разница во времени счета будет вполне очевидной. К сожалению, результат будет неверным (в обоих случаях) в силу ограниченности диапазона чисел типа `long`.

В заключение приведем сравнительную таблицу времен выполнения алгоритмов с различной сложностью и объясним, почему с увеличением быстродействия компьютеров важность использования быстрых алгоритмов значительно возрастает.

Рассмотрим четыре алгоритма решения одной и той же задачи, имеющие сложности $\log n$, n, n^2 и 2^n соответственно. Предположим, что второй из этих алгоритмов требует для своего выполнения на некотором компьютере при значении параметра $n = 10^3$ ровно одну минуту времени. Тогда

Таблица 7. Сравнительная таблица времен выполнения алгоритмов

Сложность алгоритма	$n = 10$	$n = 10^3$	$n = 10^6$
$\log n$	0.2 сек.	0.6 сек.	1.2 сек.
n	0.6 сек.	1 мин.	16.6 час.
n^2	6 сек.	16.6 час.	1902 года
2^n	1 мин.	10^{295} лет	10^{300000} лет

времена выполнения всех этих четырех алгоритмов на том же компьютере при различных значениях параметра будут примерно такими, как в таблице 7.

Когда начинающие программисты тестируют свои программы, то значения параметров, от которых они зависят, обычно невелики. Поэтому даже если при написании программы был применен неэффективный алгоритм, это может остаться незамеченным. Однако, если подобную программу попытаться применить в реальных условиях, то ее практическая непригодность проявится незамедлительно.

С увеличением быстродействия компьютеров возрастают и значения параметров, для которых работа того или иного алгоритма завершается за приемлемое время. Таким образом, увеличивается среднее значение величины n, и, следовательно, возрастает величина отношения времен выполнения быстрого и медленного алгоритмов. *Чем быстрее компьютер, тем больше относительный проигрыш при использовании плохого алгоритма!*

5. Массивы в языке Java. Язык Java, так же как и многие другие языки, позволяет работать с массивами элементов различных типов. *Массив (array)* используют, когда возникает необходимость хранить несколько однотипных значений, например, 50 целых чисел или 100 наименований книг, так как было бы неудобно использовать в таких случаях различные имена для всех этих переменных. В ближайшее время мы будем работать только с одномерными массивами и не будем пользоваться тем, что массивы — *динамические структуры данных*. Об этом речь пойдет в третьей главе книги.

Рассмотрим следующую простейшую задачу на работу с массивом целых чисел.

Задача 5.6. Напишите программу, которая вводит с клавиатуры непустой массив целых чисел, печатает его, затем инвертирует (то есть меняет местами первый элемент с последним, второй — с предпоследним и т.д.), и вновь печатает.

Текст программы.

```
public class InvArr {
    public static void main(String[] args) throws Exception {
        int n, i, a[];
        n = Xterm.inputInt("Введите длину массива n -> ");
        a = new int[n];

        for (i=0; i<n; i++)
            a[i] = Xterm.inputInt("Введите a[" + i + "] -> ");
        Xterm.print("Введенный массив a =");
        for (i=0; i<n; i++)
            Xterm.print(" " + a[i]);

        Xterm.print("\nИнвертированный массив a =");
        for (i=0; i<n/2; i++) {
            int j = a[i]; a[i] = a[n-1-i]; a[n-1-i] = j;
        }
        for (i=0; i<n; i++)
            Xterm.print(" " + a[i]);
        Xterm.print("\n");
    }
}
```

Как видно из текста этой программы, для того чтобы завести массив, необходимо объявить его и зарезервировать затем память для его элементов с помощью оператора `new`. В этом случае все они будут автоматически проинициализированы нулевыми значениями. Массив из трех целых чисел, содержащий величины 3, 7 и 11, можно задать так:

```
int a[] = {3, 7, 11};
```

Элементы любого массива нумеруются с *нуля*, а для доступа к i-му его элементу используется выражение `a[i]`. Язык Java отслеживает все попытки обратиться к несуществующему элементу массива — при попытке работать, скажем с `a[n]`, возникает исключительная ситуация (*выход индекса за границу массива*), так как последний элемент имеет индекс $n-1$, а не n.

В качестве более содержательного комментария к программе инвертирования массива заметим, что выполнение цикла, переставляющего элементы, завершается, как только будет достигнута середина массива. Если условие продолжения цикла `i<n/2` заменить на `i<n`, то процесс обмена не завершится после того, как массив уже будет инвертирован, и обмен элементов продолжится, что приведет к двукратному инвертированию массива, что эквивалентно тождественному преобразованию.

Рассмотрим еще одну задачу.

Задача 5.7. Напишите программу, печатающую максимальный элемент непустого массива.

Текст программы.

```
public class MaxArr {
    public static void main(String[] args) throws Exception {
        int n, i, a[];
        n = Xterm.inputInt("Введите длину массива n -> ");
        a = new int[n];
        for (i=0; i<n; i++)
            a[i] = Xterm.inputInt("Введите a[" + i + "] -> ");
        int max = a[0];
        for (i=1; i<n; i++)
            if (a[i] > max)
                max = a[i];
        Xterm.println("Максимальный элемент массива = " + max);
    }
}
```

В заключение рассмотрим задачу, включающую дополнительное требование на структуру программы.

Задача 5.8. Напишите программу, печатающую количество максимальных элементов непустого массива, в которой используется только один цикл.

Текст программы.

```
public class NumMaxArr {
    public static void main(String[] args) throws Exception {
        int n, i, a[];
        n = Xterm.inputInt("Введите длину массива n -> ");
        a = new int[n];
        int max  = Integer.MIN_VALUE;
        int nMax = 0;
        for (i=0; i<n; i++) {
            a[i] = Xterm.inputInt("Введите a[" + i + "] -> ");
            if (a[i] > max) {
                max  = a[i];
                nMax = 1;
            } else if (a[i] == max)
                nMax += 1;
        }
        Xterm.println("Количество макс. элементов = " + nMax);
    }
}
```

Использование константы `Integer.MIN_VALUE` позволяет избежать необходимости присваивания переменной `max` значения нулевого элемента массива до начала циклического просмотра остальных элементов.

Можно заметить, что при решении этой задачи массив по существу не используется. Если в этой программе удалить описание массива `a`, заменив его на описание целой переменной `a`, и произвести замену всех вхождений выражения `a[i]` на `a`, то программа останется работающей.

Задача сортировки является классической задачей на массивы.

Задача 5.9. Напишите программу, которая вводит с клавиатуры непустой массив целых чисел, печатает его, затем сортирует (то есть переставляет его элементы так, чтобы они располагались в неубывающем порядке), и вновь печатает.

Приведенный ниже алгоритм, который последовательно обменивает все соседние пары элементов, расположенных в неправильном порядке, является *одним из самых медленных* алгоритмов сортировки среди всех широко известных — его асимптотическая сложность является квадратичной ($\Theta(n^2)$).

Текст программы.

```
public class Sort {
    private static void sort(int[] a, int n) {
        int i, j, t;
        for (i=0; i<n-1; i+=1)
            for (j=n-1; j>i; j-=1)
                if (a[j] < a[j-1]) {
                    t=a[j]; a[j]=a[j-1]; a[j-1]=t;
                }
    }
    public static void main(String[] args) throws Exception {
        int n, i, a[];
        n = Xterm.inputInt("Введите длину массива n -> ");
        a = new int[n];
        for (i=0; i<n; i++)
            a[i] = Xterm.inputInt("Введите a[" + i + "] -> ");

        Xterm.print("Исходный массив\n");
        for (i=0; i<n; i++)
            Xterm.print(" " + a[i]);
        Xterm.print("\n");

        sort(a, n);
        Xterm.print("Отсортированный массив\n");
```

```
        for (i=0; i<n; i++)
            Xterm.print(" " + a[i]);
        Xterm.print("\n");
    }
}
```

6. Исключительные ситуации и работа с последовательностями. Часто при решении задачи, связанной с обработкой последовательно поступающих элементов, нет необходимости запоминать их, складывая в массив. Это позволяет экономить и память и время. Последняя задача предыдущей секции может быть слегка переформулирована.

Задача 5.10. Напишите программу, вводящую последовательность целых чисел, и печатающую количество ее максимальных элементов.

Подобная формулировка накладывает определенные ограничения на то, как должна быть устроена программа. Во-первых, не разрешается использовать массив для хранения элементов последовательности. Во-вторых, после ввода каждого очередного элемента (а часто и до ввода самого первого) программа обязана в принципе уметь напечатать ту характеристику последовательности, вычислению которой она посвящена. В-третьих, программа должна уметь работать со сколь угодно длинными последовательностями.

Текст программы.

```
public class NumMaxSeqv1 {
    public static void main(String[] args) {
        int max  = Integer.MIN_VALUE;
        int nMax = 0;
        try {
            while (true) {
                int x = Xterm.inputInt("Очередной элемент x -> ");
                if (x > max) {
                    max  = x;
                    nMax = 1;
                } else if (x == max)
                    nMax += 1;
                Xterm.println("Количество макс. элементов = "
                            + nMax);
            }
        } catch(Exception e) {
            ;
        }
    }
}
```

Эта программа использует конструкцию `try-catch`, предназначенную для обработки исключительных ситуаций (называемых и просто исключениями). Интерпретатор байт-кода пытается выполнить все действия, указанные в блоке `try` (*try* переводится именно так — *пытаться*). Если это удается, то содержимое блока `catch` не оказывает никакого влияния на ход выполнения программы. Однако, если в процессе выполнения блока `try` возникает *исключительная ситуация* — нечто препятствующее продолжению нормального выполнения программы, то управление немедленно передается блоку `catch` (*catch* означает *поймать*).

В приведенной выше программе содержимое блока `try` помещено в бесконечный цикл, что гарантирует выход из него только при возникновении исключительной ситуации. В данном случае причина возникновения такой ситуации может быть только одна — ошибка при вводе очередного числа. Если это случается, то управление передается в блок `catch`, который в данном случае ничего не делает, и программа завершает работу.

Приведенная программа печатает количество максимальных элементов после ввода каждого очередного элемента. Если это не желательно, то оператор вывода результата перемещают в блок `catch`:

Фрагмент программы (`NumMaxSeqv2.java`).

```
try {
    while (true) {
        int x = Xterm.inputInt("Очередной элемент x -> ");
        if (x > max) {
            max  = x;
            nMax = 1;
        } else if (x == max)
            nMax += 1;
    }
} catch(Exception e) {
    Xterm.println("\nКоличество макс. элементов = "
                + nMax);
}
```

Заметьте, что для программ, предусматривающих обработку исключительных ситуаций, теряет свою актуальность совет о необходимости дописывания фразы `throws Exception` в строку, начинающую определение метода `main`.

7. Задачи для самостоятельного решения.

Задача 5.11. Напишите программу, вычисляющую факториал введенного натурального числа, не использующую ни итерации, ни рекурсии (имеющую сложность $\Theta(1)$).

Указание. Воспользуйтесь тем, что факториал — очень быстро растущая функция, а множество $\mathbb{Z}_M$ — ограничено, и поэтому любая программа, работающая с величинами типа `int`, способна вычислить факториал только очень небольших чисел.

Задача 5.12. Напишите программу, перемножающую два натуральных числа, которая не использует операции умножения.

Задача 5.13. Напишите программу, перемножающую два натуральных числа, которая не использует операции умножения, и имеет при этом логарифмическую сложность.

Указание. Можете попробовать разобраться в программе, решающей задачу 5.5. В ней выполняется более сложное действие — квадратная матрица возводится в степень n за логарифмическое время.

Задача 5.14. Напишите программу, вводящую целое число a и натуральное n, вычисляющую и печатающую степень a^n без использования вызова функции возведения в степень.

Задача 5.15. Напишите программу (быстрое возведение в степень), возводящую целое число a в целую неотрицательную степень b без вызова функции возведения в степень с временно́й сложностью $\Theta(\log b)$.

Задача 5.16. Напишите программу, печатающую сумму квадратов всех натуральных чисел от 1 до введенного натурального n, которая имела бы константную сложность, т.е. не использовала бы ни итерации, ни рекурсии.

Задача 5.17. Напишите программу, вводящую натуральное число, и печатающую `Yes`, если оно является простым и `No` иначе.

Задача 5.18. Напишите программу, печатающую n-ое простое число.

Задача 5.19. Напишите рекурсивную программу, печатающую биноминальный коэффициент C_n^k для целых n и k, где $0 \leqslant k \leqslant n$. Для неотрицательных n и k имеют место следующие соотношения: $C_n^0 = C_n^n = 1$, $C_{n+1}^{k+1} = C_n^{k+1} + C_n^k$.

Задача 5.20. Напишите программу, вводящую имя пользователя (с применением метода `inputChars`), которая затем приветствует его.

Задача 5.21. Напишите программу, печатающую количество нулевых элементов в заданном целочисленном массиве.

Задача 5.22. Напишите программу, которая вводит с клавиатуры непустой массив целых чисел, и печатает `Yes`, если массив симметричен, и `No` иначе.

Задача 5.23. Напишите программу, которая вводит с клавиатуры непустой массив целых чисел, циклически сдвигает элементы массива вправо на одну позицию, и печатает результат. Цикличность означает, что последний элемент массива становится самым первым его элементом.

Задача 5.24. Напишите программу, которая вводит с клавиатуры непустой массив целых чисел, циклически сдвигает элементы массива вправо на k позиций, и печатает результат. Число k вводится с клавиатуры, а сложность программы должна быть $\Theta(n)$.

Задача 5.25. Напишите программу, которая вводит с клавиатуры непустой массив целых чисел, заменяет все элементы массива, кроме крайних, на полусумму соседей, и печатает результат.

Задача 5.26. Напишите программу (линейный поиск), определяющую первое вхождение заданного целого числа x в массив целых чисел, заведомо содержащий это число.

Задача 5.27. Напишите программу, вводящую последовательность целых чисел, и печатающую количество различных значений квадратов ее элементов.

Задача 5.28. Напишите программу, вводящую последовательность вещественных чисел, и печатающую среднее арифметическое ее элементов (для непустой последовательности).

Задача 5.29. Напишите программу, вводящую последовательность целых чисел, и печатающую максимальное число идущих подряд одинаковых элементов.

Задача 5.30. Напишите программу, вводящую последовательность целых чисел, и печатающую номера первого и последнего ее максимальных элементов.

Задача 5.31. Напишите программу, вводящую последовательность целых чисел, и печатающую номер первого элемента, равного нулю, и нуль при отсутствии такого элемента в последовательности.

Задача 5.32. Напишите программу, вводящую последовательность целых чисел, и печатающую число элементов, больших предыдущего (первый элемент последовательности тоже считается таким).

Задача 5.33. Напишите программу, вводящую фразу русского языка (с использованием метода `inputChars`), которая определяет, является ли введенная фраза палиндромом.

Указание. Палиндром — эта фраза, инвертирование которой не изменяет ее. При этом все пробелы во фразе игнорируются.

§ 6. Спецификация программ и преобразователь предикатов

Важнейший для дальнейшего изучения материал этого параграфа в более подробном изложении можно найти в книге [**4**]. Полезным будет также и знакомство с подходом учебного пособия [**9**].

1. Предикаты и документирование программ. Как уже отмечалось выше, предикаты нам нужны прежде всего для того, чтобы иметь возможность ясно и недвусмысленно формулировать определенные утверждения о программах и программных переменных. Сейчас мы рассмотрим ряд примеров подобного типа.

Задача 6.1. Запишите предикат, утверждающий, что если $i < j$, а $m > n$, то $u = v$.

Решение. В данном случае все очевидно: $i < j \wedge m > n \Rightarrow (u = v)$.

Задача 6.2. Запишите предикат, утверждающий, что ни одно из следующих утверждений не является истинным: $a < b$, $b < c$ и $x = y$.

Решение. Это задание тоже не является сложным. Вот несколько эквивалентных между собой решений: $P_1 = ((a < b) = F) \wedge ((b < c) = F) \wedge ((x = y) = F)$, $P_2 = !(a < b) \wedge !(b < c) \wedge !(x = y)$ и $P_3 = a \geqslant b \wedge b \geqslant c \wedge x \neq y$. Так как обычно нужно предъявить максимально простой предикат, то будем считать ответом последний из них — P_3.

В ряде последующих задач нам придется иметь дело с массивами, поэтому договоримся о следующих обозначениях: будем обозначать *вырезку* из массива $b[0..m-1]$, в которой содержатся все элементы данного массива с индексами от j до k включительно, символом $b[j..k]$. В случаях, когда $j > k$, $k < 0$ или $j \geqslant m$, вырезка представляет из себя пустое множество.

Задача 6.3. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ все элементы вырезки $b[j..k]$ являются нулевыми.

Решение. Используем квантор всеобщности: $\forall i \; j \leqslant i < k + 1 \; b[i] = 0$.

Задача 6.4. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ ни один из элементов вырезки $b[j..k]$ не нулевой.

Решение. Переформулировав данное высказывание (все элементы вырезки не нулевые), запишем ответ в следующем виде: $\forall i \; j \leqslant i < k + 1 \; b[i] \neq 0$.

Задача перевода высказывания с естественного языка на язык предикатов не всегда является однозначной, но именно это и является одной из причин его использования. Вот пример.

Задача 6.5. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ некоторые из элементов вырезки $b[j..k]$ нулевые.

Решение. Данное высказывание можно понимать двояко: оно может означать, что в вырезке существует хотя бы один нулевой элемент, а может значить и то, что в ней как минимум два нулевых элемента. Вот ответы для каждого из этих вариантов трактовки задачи: $P_1 = \exists i\ j \leqslant i < k+1\ b[i] = 0$ и $P_2 = \exists i\ j \leqslant i < k+1\ \exists m\ (j \leqslant m < k+1 \wedge m \neq i)(b[i] = 0) \wedge (b[m] = 0)$.

Вот аналогичные задачи из области математики.

Задача 6.6. Запишите предикат, который утверждает, что функция $f\colon \{1,2,3,4,5\} \to \{1,2,3,4,5\}$ является сюръективной и отрицание этого факта. Упростите получившиеся предикаты, если это возможно.

Решение. В соответствии с определением сюръективности имеем $P = (\forall y \in \{1,2,3,4,5\}\ \exists x \in \{1,2,3,4,5\}\ y = f(x))$. Построим отрицание и упростим его в соответствии с законами эквивалентности. $!P = (!(\forall y \in \{1,2,3,4,5\}\ \exists x \in \{1,2,3,4,5\}\ y = f(x))) = (\exists y \in \{1,2,3,4,5\}\ !(\exists x \in \{1,2,3,4,5\}\ y = f(x))) = (\exists y \in \{1,2,3,4,5\}\ \forall x \in \{1,2,3,4,5\}\ !(y = f(x))) = (\exists y \in \{1,2,3,4,5\}\ \forall x \in \{1,2,3,4,5\}\ y \neq f(x))$.

Задача 6.7. Запишите предикат, который утверждает, что функция $f\colon \{1,2,3,4,5\} \to \{1,2,3,4,5\}$ все элементы, не превосходящие трех, не увеличивает, и отрицание этого факта. Упростите получившиеся предикаты, если это возможно.

Решение. $P = (\forall x \in \{1,2,3\}\ f(x) \leqslant x)$. Его отрицание можно упростить: $!P = !(\forall x \in \{1,2,3\}\ f(x) \leqslant x) = (\exists x \in \{1,2,3\}\ f(x) > x)$.

В текст программы зачастую полезно включать предикаты, которые позволяли бы человеку или компьютеру проверять корректность хода выполнения программы. В тех случаях, когда подобные утверждения предназначены только для человека, они оформляются в виде комментариев.

Часто, однако, полезно поручить проверку истинности предикатов в процессе выполнения программы непосредственно компьютеру. В различных языках программирования эта возможность реализована поразному. Например, в программах на языке C можно использовать макрос `assert(P)`, который в случае ложности его аргумента (предиката `P`) немедленно прекращает выполнение программы, сообщая о причине этого.

Механизм работы с исключениями и оператор `if` позволяют легко реализовать аналогичную конструкцию в языке Java. Вот простейший пример.

Задача 6.8. Напишите программу, содержащую проверку истинности утверждения о положительности введенного с клавиатуры целого числа.

Текст программы.

```
public class Assert {
    public static void main(String[] args) throws Exception {
        int n = Xterm.inputInt("Введите n -> ");
        if (n <= 0)
            throw new Exception("n <= 0");
        Xterm.println("n = " + n);
    }
}
```

При вводе неположительного числа программа прекращает свое выполнение и печатает следующий текст:

```
java.lang.Exception: n<=0
        at Assert.main(Assert.java:4)
```

Богатые возможности работы с исключениями, имеющиеся в языке, позволяют реализовать гораздо более изощренные способы проверки истинности предикатов, нежели использованный в программе простейший вариант возбуждения исключения типа `Exception` без попыток его последующей обработки. Эти возможности, однако, не будут рассматриваться в нашем курсе.

2. Спецификация программы и преобразователь предикатов *wp*. Для того чтобы доказывать правильность программ необходимо прежде всего дать строгое определение понятию *правильная программа*. Ясно, что оно зависит не только от результата, который должен быть получен в процессе выполнения программы, но и от того, в каких условиях начинается ее выполнение.

Определение 6.1. *Спецификацией* $\{Q\}\ S\ \{R\}$ программы S, где Q и R — предикаты, называется *предикат*, означающий, что *если выполнение S началось в состоянии, удовлетворяющем Q, то имеется гарантия, что оно завершится через конечное время в состоянии, удовлетворяющем R.*

Под программой S в данном определении может пониматься один или несколько отдельных операторов или же действительно целая большая программа.

Определение 6.2. Предикат Q называется *предусловием* или входным утверждением S; R — *постусловием* или выходным утверждением программы S.

Так как спецификация программы является предикатом, то она может быть истинной, а может быть и ложной. Возможна и такая ситуация, когда в некоторых состояниях она истинна, а в других — ложна. Вот соответствующие примеры.

Спецификация $\{i = 0\}$ `"i++;"` $\{i = 1\}$ является тавтологией, спецификация $\{i = 0\}$ `"i++;"` $\{i = 0\}$ ложна во всех состояниях, а спецификация $\{i = 0\}$ `"i++;"` $\{i = j\}$ истинна при $j = 1$ и ложна в остальных состояниях.

Спецификация программы является единственным корректным способом постановки задачи. Только четко сформулировав пред- и постусловия, можно обсуждать затем правильность программы.

Определение 6.3. Программа S является *правильной* при заданных Q и R, если спецификация $\{Q\}\ S\ \{R\}$ является тавтологией.

С практической точки зрения особый интерес представляют программы, которые позволяют получить нужный результат при минимальных требованиях к начальным условиям, а также программы, позволяющие достичь как можно большего при фиксированном предусловии.

Слабейшее предусловие — предикат, описывающий максимально широкое множество в пространстве состояний переменных программы S, на котором гарантируется получение постусловия R. *Сильнейшее постусловие* — предикат, описывающий максимально сильные ограничения на состояние переменных программы S, которые могут быть получены при данном предусловии Q.

Для целей доказательства правильности программ особенно важен следующий предикат.

Определение 6.4. *Слабейшее предусловие* $wp(S, R)$ — предикат, представляющий множество всех состояний переменных программы S, для которых выполнение команды S, начавшееся в таком состоянии, обязательно закончится через конечное время в состоянии, удовлетворяющем R.

Проиллюстрируем введенное понятие на нескольких примерах.

$wp($`"i = i + 1;"`$, i \leqslant 1) = (i \leqslant 0)$, так как если переменная i удовлетворяла условию $i \leqslant 0$, то после выполнения программы `"i = i + 1;"` она действительно будет удовлетворять неравенству $i \leqslant 1$.

$wp($`"if (x >= y) z=x; else z=y;"`$, z = max(x, y)) = T$, ибо выполнение программы `"if (x >= y) z=x; else z=y;"` при *любых* начальных

условиях приведет к тому, что переменная z станет равной максимальному значению из величин x и y.

$wp($`"if (x >= y) z=x; else z=y;"`$, z = y) = (y \geqslant x)$, потому что y будет равно максимуму из чисел x и y (а именно таково будет z после выполнения программы `"if (x >= y) z=x; else z=y;"`) тогда и только тогда, если именно переменная y имеет бóльшее значение.

$wp($`"if (x >= y) z=x; else z=y;"`$, z = y - 1) = F$. Это (пустое множество состояний) означает, что *ни при каких* начальных условиях программа `"if (x >= y) z=x; else z=y;"` не сможет сделать величину z меньше, чем y.

$wp($`"if (x >= y) z=x; else z=y;"`$, z = y+1) = (x = y+1)$, ибо только при таком начальном условии после выполнения приведенной программы переменная z станет равной $y + 1$.

Заметим, что из определений спецификации программы и ее слабейшего предусловия вытекает следующее утверждение.

Предложение 6.1. $\{Q\}\ S\ \{R\} = (Q \Rightarrow wp(S, R))$.

Определение 6.5. *Преобразователем предикатов* (обозначаемый через $wp_S(R)$) называют $wp(S, R)$ когда фиксируют программу S и рассматривают $wp(S, R)$ как функцию одной переменной R.

Предложение 6.2. *Преобразователь предикатов $wp(S, R)$ обладает следующими свойствами:*
1) $wp(S, F) = F$ (закон исключенного чуда);
2) $wp(S, Q) \wedge wp(S, R) = wp(S, Q \wedge R)$ (дистрибутивность конъюнкции);
3) $(Q \Rightarrow R) \Rightarrow (wp(S, Q) \Rightarrow wp(S, R))$ (закон монотонности);
4) $wp(S, Q) \vee wp(S, R) = wp(S, Q \vee R)$ (дистрибутивность дизъюнкции).

Величина $wp(S, F)$ описывает такое множество начальных условий, при которых выполнение программы S завершится через конечное время в состояний, удовлетворяющем F, то есть ни в каком состоянии. Этого, конечно, быть не может, что и поясняет название свойства — закон *исключенного чуда*.

Докажем аккуратно *дистрибутивность конъюнкции*. Для доказательства эквивалентности достаточно показать, что из условия $wp(S, Q) \wedge wp(S, R)$, стоящего в левой части, вытекает условие $wp(S, Q \wedge R)$, размещенное в правой, и наоборот. Для доказательства импликации $wp(S, Q) \wedge wp(S, R) \Rightarrow wp(S, Q \wedge R)$ рассмотрим произвольное состояние s, удовлетворяющее условию $wp(S, Q) \wedge wp(S, R)$. Так как выполнение программы S, начавшееся в s, завершится при истинных Q и R, то истинным будет и предикат $Q \wedge R$.

Для доказательства обратной импликации $wp(S, Q \wedge R) \Rightarrow wp(S, Q) \wedge wp(S, R)$ рассмотрим состояние s, удовлетворяющее условию $wp(S, Q \wedge R)$. Тогда выполнение S, начавшееся в s, обязательно завершится в некотором состоянии s', удовлетворяющем $Q \wedge R$. Но любое такое s' обязательно удовлетворяет и Q и R, так что s удовлетворяет и $wp(S, Q)$ и $wp(S, R)$, что и завершает доказательство.

Закон *монотонности* докажите самостоятельно, а вот по поводу последнего свойства преобразователя предикатов — *дистрибутивности дизъюнкции* — надо сделать некоторые замечания. Дело в том, что если в качестве S рассмотреть операцию бросания монеты, которая может завершиться либо выпадением герба (G), либо решки (R), то $wp(S, G) = wp(S, R) = F$, ибо нельзя гарантированно предсказать результат бросания ни при каких начальных условиях. С другой стороны, $wp(S, G \vee R) = T$, так как всегда выпадет либо герб, либо решка.

Если S является недетерминированной, то эквивалентность в законе дистрибутивности дизъюнкции превращается в импликацию. Однако для программ S, реализованных с помощью большинства языков программирования, подобная ситуация невозможна.

3. Определение простейших операторов языка Java. До сих пор все наши манипуляции с wp основывались на том, что мы считали известным, как именно выполняются те или иные команды, из которых состоит программа S.

Сейчас мы полностью изменим точку зрения. Будем считать первичным предикат wp и условия, которым он удовлетворяет. Это позволит нам определить в терминах wp все команды языка, а затем доказывать всевозможные утверждения о программах.

Первым оператором, который мы определим, будет пустой оператор.

Определение 6.6. $wp(\texttt{";"}, R) = R$.

Определение не требует доказательства, однако полезно убедиться, что оно не противоречит нашему внутреннему пониманию того, как именно работает пустой оператор. Проверьте это сами.

Определение 6.7. $wp(\texttt{"System.exit(0);"}, R) = F$.

Выполнение вызова метода `"System.exit(0)"` приводит к немедленному завершению выполнения программы. Поэтому вполне естественно, что ни при каком начальном состоянии после его выполнения предикат R истинным не будет.

Следующее определение связано с последовательным выполнением двух операторов одного за другим.

Определение 6.8. $wp(\texttt{"S1; S2;"}, R) = wp(\texttt{"S1;"}, wp(\texttt{"S2;"}, R))$.

В случае последовательного выполнения нескольких операторов данным определением нужно воспользоваться многократно.

Более сложным и интересным является определение оператора присваивания, которое будет рассмотрено нами в двух вариантах. Сначала рассмотрим случай присваивания простой переменной.

Определение 6.9. $wp(\texttt{"x = e;"}, R) = domain(e)\&\&R^x_e$, где $domain(e)$ — предикат, описывающий множество всех состояний, в которых может быть вычислено значение e (т.е., где e определено), а R^x_e обозначает подстановку в предикат R выражения e вместо всех свободных вхождений переменной x.

Это определение может вызвать определенные вопросы, так как на первый взгляд кажется, что оно не соответствует нашему интуитивному пониманию того, что делает оператор присваивания. Поэтому рассмотрим ряд примеров, которые развеют эти сомнения.

$wp(\texttt{"x = 5;"}, x = 5) = (T\&\&(5 = 5)) = T$, что вполне правильно, так как присваивание переменной x числа 5 всегда делает $x = 5$.

$wp(\texttt{"x = 5;"}, x \neq 5) = (T\&\&(5 \neq 5)) = F$, что тоже правильно, ибо присваивание переменной x числа 5 никогда не может сделать $x \neq 5$.

Часто можно позволить себе опустить предикат $domain(e)$ и считать, что $wp(\texttt{"x = e;"}, R) = R^x_e$, так как присваивание всегда должно выполняться только в тех ситуациях, когда e может быть вычислено. В случае сомнений, однако, лучше воспользоваться исходным определением.

$wp(\texttt{"x = 1/y;"}, x \leqslant 0) = (y \neq 0)\&\&(1/y \leqslant 0) = (y < 0)$.

Случай присваивания элементу массива несколько сложнее, так как выражение $b[i]$ само по себе может оказаться неопределенным из-за некорректного значения индекса i. Введем предикат $inrange(b, i)$, который будет определять *множество допустимых значений индекса*.

Определение 6.10. Для присваивания элементу массива слабейшее предусловие $wp(\texttt{"b[i] = e;"}, R) = inrange(b, i)\&\&domain(e)\&\&R^{b[i]}_e$.

В качестве примера рассмотрим массив $b[0..9]$ и вычислим слабейшее предусловие $wp(\texttt{"b[i] = i;"}, b[i] = i) = (inrange(b, i)\&\&domain(i)\&\&i = i) = ((0 \leqslant i < 10)\&\&T\&\&T) = (0 \leqslant i < 10)$.

4. Оператор `if` и слабейшее предусловие. Перед тем, как дать формальное определение оператора `if-else` в терминах wp, заметим, что управляющая конструкция `if` (без `else` части) эквивалентна использованию пустого оператора в `else`-ветви. Оператор `switch` также может быть заменен несколькими вложенными друг в друга операторами `if-else`.

Таким образом, для того чтобы задать все основные конструкции выбора языка Java, достаточно дать определение только одной из них, — конструкции `"if (e) S1; else S2;"`.

Определение 6.11. $wp(\texttt{"if (e) S1; else S2;"}, R) = domain(e)\&\&(e \Rightarrow wp(\texttt{"S1;"}, R)) \wedge (!e \Rightarrow wp(\texttt{"S2;"}, R))$.

В качестве примера использования этого определения убедимся в том, что $wp(\texttt{"if (x >= y) z=x; else z=y;"}, z = max(x,y)) = T$.

В самом деле, $wp(\texttt{"if (x >= y) z=x; else z=y;"}, z = max(x,y)) = (((x \geqslant y) \Rightarrow wp(\texttt{"z=x;"}, z = max(x,y))) \wedge !(x \geqslant y) \Rightarrow \texttt{"z=y;"}, z = max(x,y))$
$= (((x \geqslant y) \Rightarrow (x = max(x,y))) \wedge !(x \geqslant y) \Rightarrow (y = max(x,y))) = ((!(x \geqslant y) \vee (x = max(x,y))) \wedge !(!(x \geqslant y)) \vee (y = max(x,y))) = (((x < y) \vee (x = max(x,y))) \wedge (x \geqslant y) \vee (y = max(x,y)))$.

Покажем, что каждый из двух членов получившейся конъюнкции является тавтологией. Рассмотрим, например, первый из них — $((x < y) \vee (x = max(x,y))$. Если $x < y$, то истинен первый член дизъюнкции. В противном случае $x \geqslant y$ и поэтому истинен ее второй член, что и доказывает требуемое. Аналогичные рассуждения можно провести и для выражения $(x \geqslant y) \vee (y = max(x,y)))$, что и завершает доказательство.

Для практических приложений часто необходимо не вычисление слабейшего предусловия, а лишь проверка того факта, что некоторое другое известное предусловие обеспечивает его выполнение. По этой причине полезна следующая теорема.

Теорема 6.1. *Пусть предикат Q удовлетворяет условию*

$$((Q \wedge e) \Rightarrow wp(S1, R)) \wedge ((Q \wedge !e) \Rightarrow wp(S2, R)).$$

Тогда (и только тогда)

$$Q \Rightarrow wp(\texttt{"if (e) S1; else S2;"}, R).$$

Доказательство. $((Q \wedge e) \Rightarrow wp(S1,R)) = (!(Q \wedge e) \vee wp(S1,R)) = (!Q \vee !e \vee wp(S1,R)) = (!Q \vee (e \Rightarrow wp(S1,R))) = (Q \Rightarrow (e \Rightarrow wp(S1,R)))$.

Аналогично получаем $((Q \wedge !e) \Rightarrow wp(S2,R)) = (Q \Rightarrow (!e \Rightarrow wp(S2,R)))$, и, следовательно, $(((Q \wedge e) \Rightarrow wp(S1,R)) \wedge ((Q \wedge !e) \Rightarrow wp(S2,R))) = ((Q \Rightarrow (e \Rightarrow wp(S1,R))) \wedge (Q \Rightarrow (!e \Rightarrow wp(S2,R)))) = (Q \Rightarrow ((e \Rightarrow wp(S1,R)) \wedge (!e \Rightarrow wp(S2,R)))) = (Q \Rightarrow wp(\texttt{"if (e) S1; else S2;"}, R))$. □

5. Циклы в терминах wp**.** В терминах слабейшего предусловия можно определить все оставшиеся еще неопределенными конструкции языка, но мы ограничимся только определением цикла `while`. Дело в том, что

уже даже это определение оказывается достаточно бесполезным с точки зрения практики. Именно оно, однако, позволяет предложить несколько вполне конструктивных подходов к построению цикла, которые будут изучаться в следующей главе.

Для определения слабейшего предусловия $wp(\texttt{"while(e)S;"}, R)$ нам потребуются следующие вспомогательные определения:

Определение 6.12. $H_0(R) = domain(e)\&\&(!e \wedge R)$, $H_k(R) = H_{k-1}(R) \vee (domain(e)\&\&wp(S, H_{k-1}(R)))$, где предикат $H_k(R)$ описывает множество всех состояний, в которых выполнение цикла `"while(e)S;"` заканчивается *не более, чем за* k *итераций*, в состоянии, удовлетворяющем R.

Теперь можно дать основное определение.

Определение 6.13. $wp(\texttt{"while(e)S;"}, R) = (\exists k \geqslant 0 H_k(R))$.

Для цикла `"while (i<1) i=i+1;"` и постусловия $R = (i = 1)$ имеем $H_0(R) = (i \geqslant 1 \wedge i = 1) = (i = 1)$. Действительно, именно при таком предусловии выполнение цикла завершится за ноль итераций и даст результат R.

Далее, легко посчитать $wp(\texttt{"i=i+1;"}, H_0(R)) = (i + 1 = 1) = (i = 0)$ и $H_1(R) = (i = 1) \vee (i < 1 \wedge i = 0) = (i = 1 \vee i = 0)$.

Аналогично находим $H_2(R) = (i = 1 \vee i = 0 \vee i = -1)$ и т.д.

6. Вычисление слабейшего предусловия. Покажем на ряде примеров, как решаются задачи на вычисление слабейшего предусловия.

Задача 6.9. Вычислите и упростите $wp(\texttt{"i=i+2; j=j-2;"}, i + j = 0)$.

Решение. Для вычисления воспользуемся определениями слабейшего предусловия для последовательного выполнения операторов и оператора присваивания. Так как в данном случае все выражения заведомо являются определенными, то истинный предикат $domain(e)$ будет опущен изначально.

$wp(\texttt{"i=i+2; j=j-2;"}, i + j = 0) = wp(\texttt{"i=i+2;"}, wp(\texttt{"j=j-2;"}, i + j = 0)) = wp(\texttt{"i=i+2;"}, i + j - 2 = 0) = (i + 2 + j - 2 = 0) = (i = -j)$.

Задача 6.10. Вычислите и упростите $wp(\texttt{"x=(x+y)*(x-y);"}, x + y^2 \neq 0)$.

Решение. $wp(\texttt{"x=(x+y)*(x-y);"}, x + y^2 \neq 0) = (x^2 - y^2 + y^2 \neq 0) = (x^2 \neq 0) = (x \neq 0)$.

Задача 6.11. Вычислите и упростите $wp(\texttt{"x=a/b;"}, x^2 \geqslant 0)$.

Решение. В данном случае ответ T является ошибочным, так как $wp(\texttt{"x=a/b;"}, x^2 \geqslant 0) = (domain(a/b)\&\&(a/b)^2 \geqslant 0) = (b \neq 0)$.

Задача 6.12. Вычислите и упростите $wp(\texttt{"i=1; s=b[0];"}, 1 \leqslant i < n \ \wedge \ s = b[0] + \ldots + b[i-1])$.

Решение. $wp(\texttt{"i=1; s=b[0];"}, 1 \leqslant i < n \ \wedge \ s = b[0] + \ldots + b[i-1]) = wp(\texttt{"i=1;"}, wp(\texttt{"s=b[0];"}, 1 \leqslant i < n \ \wedge \ s = b[0] + \ldots + b[i-1])) = wp(\texttt{"i=1;"}, 1 \leqslant i < n \ \wedge \ b[0] = b[0] + \ldots + b[i-1]) = (1 \leqslant 1 < n \wedge (b[0] = b[0])) = (1 < n \wedge (b[0] = b[0])) = (n > 1)$.

Задача 6.13. Вычислите и упростите $wp(\texttt{"if(true);"}, R)$ для произвольного предиката R.

Решение. $wp(\texttt{"if(true);"}, R) = wp(\texttt{"if(true); else ;"}, R) = ((T \Rightarrow wp(\texttt{";"}, R)) \wedge (F \Rightarrow wp(\texttt{";"}, R))) = ((F \vee R) \wedge (T \vee R)) = (R \wedge T) = R$.

Рассмотрим в заключение задачу на решение уравнения, связанного со слабейшим предусловием.

Задача 6.14. Найдите такое значение выражения x, включающее другие переменные, для которого спецификация $\{Q\}\ S\ \{R\}$ становится тавтологией: $\{T\}$ `"a=a+1; b=x;"` $\{b = a + 1\}$.

Решение. Вспомним, что имеет место эквивалентность $\{Q\}\ S\ \{R\} = (Q \Rightarrow wp(S, R))$. Таким образом, нам необходимо подобрать такое x, для которого $(T \Rightarrow wp(\texttt{"a=a+1; b=x;"}, b = a + 1)) = T$. Вычислим сначала слабейшее предусловие, входящее в этот предикат: $wp(\texttt{"a=a+1; b=x;"}, b = a + 1) = wp(\texttt{"a=a+1;"}, wp(\texttt{"b=x;"}, b = a + 1)) = wp(\texttt{"a=a+1;"}, x = a + 1) = (x = a + 2)$.

Легко убедиться, однако, что мы получили неверный результат! И все дело в том, что переменная x *зависит* от a. Проведем вычисления повторно, заменив x на $x(a)$: $wp(\texttt{"a=a+1; b=x(a);"}, b = a + 1) = wp(\texttt{"a=a+1;"}, wp(\texttt{"b=x(a);"}, b = a + 1)) = wp(\texttt{"a=a+1;"}, x(a) = a + 1) = (x(a + 1) = a + 2)$.

Вернемся к исходной задаче. Нам нужно выяснить, при каких значениях x выражение $(T \Rightarrow (x(a + 1) = a + 2)) = T$ окажется тавтологией. Упростим данное выражение: $((T \Rightarrow (x(a + 1) = a + 2)) = T) = (T \Rightarrow (x(a + 1) = a + 2)) = (F \vee (x(a + 1) = a + 2)) = (x(a + 1) = a + 2)$.

Теперь ответ очевиден: $x(a) = a + 1$ или просто $x = a + 1$.

7. Задачи для самостоятельного решения.

Задача 6.15. Запишите предикат, утверждающий, что самое большее одно из следующих утверждений истинно: $a < b$, $b < c$.

Задача 6.16. Запишите предикат, утверждающий, что следующие утверждения не являются истинными одновременно: $a < b$, $b < c$ и $x = y$.

Задача 6.17. Запишите предикат, утверждающий следующее: когда $x < y$, $y < z$ означает, что $v = w$, но если $x \geqslant y$, то $y < z$ не может выполняться; однако если $v = w$, то $x < y$.

Задача 6.18. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ все нули массива находятся в вырезке $b[j..k]$.

Задача 6.19. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ некоторые нули массива находятся в вырезке $b[j..k]$.

Задача 6.20. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ справедливо высказывание: неверно, что все нули массива находятся в вырезке $b[j..k]$.

Задача 6.21. Запишите предикат, утверждающий, что для массива $b[0..n-1]$ длины $n > 0$ справедливо высказывание: неверно, что не все нули массива находятся в вырезке $b[j..k]$.

Задача 6.22. Запишите предикат, который утверждает, что функция $f\colon \{1,2,3,4,5\} \to \{1,2,3,4,5\}$ является инъективной и отрицание этого факта. Упростите получившиеся предикаты, если это возможно.

Задача 6.23. Запишите предикат, который утверждает, что функция $f\colon \{1,2,3,4,5\} \to \{1,2,3,4,5\}$ является биективной и отрицание этого факта. Упростите получившиеся предикаты, если это возможно.

Задача 6.24. Запишите предикат, который утверждает, что функция $f\colon \{1,2,3,4,5\} \to \{1,2,3,4,5\}$ все существует единственный элемент $x \in \{1,2,3,4,5\}$, который функция f уменьшает, и отрицание этого факта. Не используйте при этом квантора $\exists!$.

Задача 6.25. Основываясь на определении 6.4 и спецификации программы 6.1, докажите истинность эквивалентности $\{Q\}\ S\ \{R\} = (Q \Rightarrow wp(S,R))$.

Задача 6.26. Основываясь на определении 6.4, докажите закон монотонности $(Q \Rightarrow R) \Rightarrow (wp(S,Q) \Rightarrow wp(S,R))$.

Задача 6.27. Основываясь на определении 6.4, докажите закон дистрибутивности дизъюнкции $wp(S,Q) \vee wp(S,R) = wp(S, Q \vee R)$.

Задача 6.28. Вычислите и упростите $wp(\texttt{"i=i+1; j=j-1;"}, i \cdot j = 0)$.

Задача 6.29. Вычислите и упростите $wp(\texttt{"x=x+y;"}, x < 2y)$.

Задача 6.30. Вычислите и упростите $wp(\texttt{"i=i+1; j=j+1;"}, i = j)$.

Задача 6.31. Вычислите и упростите $wp(\texttt{"a=0; n=1;"}, a^2 < n \ \wedge \ (a+1)^2 \geqslant n)$.

Задача 6.32. Вычислите и упростите $wp(\texttt{"s=s+b[i]; i=i+1;"}, 0 < i < n \ \wedge \ s = b[0] + \ldots + b[i-1])$.

Задача 6.33. Вычислите и упростите следующее слабейшее предусловие $wp(\texttt{"if (a > b) a=a-b; else b=b-a;"}, a > 0 \ \wedge \ b > 0)$.

Задача 6.34. Найдите такое значение выражения x, включающее другие переменные, для которого спецификация $\{Q\}\ S\ \{R\}$ становится тавтологией: $\{T\}$ `"b=x; a=a+1;"` $\{b = a + 1\}$.

Задача 6.35. Найдите такое значение выражения x, включающее другие переменные, для которого спецификация $\{Q\}\ S\ \{R\}$ становится тавтологией: $\{i = j\}$ `"i=i+1; j=x;"` $\{i = j\}$.

§ 7. Все задачи главы

Некоторые из сформулированных ниже задач уже были разобраны ранее, другие — предложены для самостоятельного решения, решения третьих будут приведены в последующих главах. Часть из них войдет в список зачетных задач, умение справиться с которыми является необходимым условием успешного завершения изучения курса.

Бóльшая часть задач позаимствована из различной литературы, среди которой хочется отметить книги [**9**] и [**14**].

1. Задачи на составление алгоритмов.

Задача 7.1. Придумайте алгоритм, вводящий натуральное число, большее единицы, который находит наименьший простой делитель этого числа.

Задача 7.2. Придумайте алгоритм, вводящий три целых числа и определяющий, есть ли среди введенных чисел одинаковые или нет.

Задача 7.3. Придумайте алгоритм, вводящий три целых числа, который находит второе по величине число, если оно существует.

Задача 7.4. Придумайте алгоритм, вводящий три целых числа, определяющий количество максимальных чисел среди введенных.

Задача 7.5. Придумайте алгоритм, вводящий действительное число, который рассматривает это число, как координаты точки на прямой, и находит расстояние от этой точки до отрезка $[0, 1]$.

Задача 7.6. Придумайте алгоритм, находящий n-ое простое число.