

Пермский Государственный Технический Университет

А. Е. СОЛОВЬЕВ

СПЕЦИАЛЬНАЯ МАТЕМАТИКА

конспект лекций

для студентов специальности АСУ

Пермь, 2001г.

Оглавление

Введение.....	5
1. Теория множеств.....	6
1.1 Понятие множества.....	6
1.2. Операции над множествами.....	7
1.3. Диаграммы Эйлера - Венна.....	7
U.....	8
1.4. Алгебра множеств.....	8
1.5. Кортж. График.....	9
1.6. Соответствия.....	1
1.7. Отношения.....	13
1.7.1 Отношение эквивалентности.....	13
1.7.2. Отношения порядка.....	14
1.7.3. Морфизмы.....	14
1.8. Решетки.....	15
1.8.1. Диаграммы Хассе.....	15
1.8.2. Понятие решетки.....	15
1.8.3. Алгебраическое представление решеток.....	16
Булевы решетки.....	16
1.8.4. Подрешетки.....	18
1.8.5. Морфизмы решеток.....	18
1.9. Мощность множества.....	18
1.9.1. Понятие мощности.....	18
1.9.2. Аксиоматика Пеано.....	18
1.9.3. Сравнение мощностей.....	19
1.9.4. Мощность множества R.....	20
Теорема Кантора.....	20
1.9.5. Арифметика бесконечного.....	20
1.9.6. Противопоставление системного и.....	21
теоретико-множественного подходов.....	21
2. Математическая логика.....	21
2.1. Логика высказываний.....	21
2.1.1. Операции над высказываниями.....	21
2.1.2. Построение и анализ сложных высказываний.....	22
2.1.3. Алгебра высказываний.....	23
2.1.4. Формы представления высказываний.....	24
2.1.5. Преобразование высказываний.....	25
2.1.6. Минимизация высказываний методом Квайна.....	26
2.1.7. Минимизация с помощью карт Вейча.....	28
2.1.8. Функциональная полнота.....	29
2.2. Логика предикатов.....	29
2.2.1. Основные равносильности для предикатов.....	30
2.2.2. Получение дизъюнктов.....	31
2.3. Аксиоматические теории.....	32
2.3.1. Аксиоматическая теория исчисления высказываний.....	32
2.3.2. Непротиворечивость и полнота аксиоматической теории исчисления высказываний.....	33
2.4. Аксиоматические теории первого порядка.....	34
2.5. Метод резолюций.....	35
2.6. Система Генцена.....	37
2.7. Система Аристотеля.....	38
2.8. Примеры неклассических логик.....	40

3. Теория Автоматов.....	42
3.1. Понятие автомата.....	42
3.2. Примеры автоматов.....	43
3.3. Минимизация автоматов.....	45
3.4. Особенности минимизации автомата Мура.....	46
3.5. Переход от автомата Мура к автомату Мили и наоборот.....	47
4. Теория графов.....	48
4.1. Понятие графа.....	48
4.2. Теорема Эйлера.....	51
4.3. Полные графы и деревья.....	53
4.4. Деревья.....	54
4.5. Алгоритм Краскала.....	55
4.6. Планарные графы.....	55
4.7. Задача о 4 красках.....	56
4.8. Определение путей в графе.....	58
4.9. Приведение графа к ярусно-параллельной форме.....	59
4.10. Внутренняя устойчивость графа.....	60
4.11. Множество внешней устойчивости.....	61
Ядро графа.....	61
4.12. Клика.....	62
5. Теория групп.....	63
5.1. Понятие группы.....	63
5.2. Морфизмы групп.....	63
5.3. Инвариантные (нормальные) подгруппы.....	64
5.4. Группа Диэдра (D3).....	65
5.5. Смежные классы.....	67
5.6. Фактор-группы.....	67
5.7. Группа Клейна четвертой степени.....	68
6. Теория алгоритмов.....	68
6.1. Понятие алгоритма.....	68
6.2. Конкретизация понятия алгоритма.....	69
6.3. Сложность вычислений.....	69
6.4. Машины Тьюринга.....	70
6.5. Нормальные алгорифмы Маркова.....	71
6.6. Рекурсивные функции.....	72
6.7. λ -исчисление.....	74
7. Формальные грамматики.....	75
7.1. Понятие формальной грамматики.....	75
7.2. Деревья вывода.....	77
7.3. Классификация языков по Хомскому.....	78
7.4. Распознающие автоматы.....	79
7.5. Понятие транслятора.....	80
7.6. Основные функции компилятора.....	81
Лексический анализ.....	81
7.7. Переход от недетерминированного распознающего автомата к.....	81
детерминированному.....	81
7.8. Переход от праволинейной грамматики к автоматной.....	82
7.9. LEX.....	83
7.10. Детерминированные автоматы с магазинной памятью.....	85
(МП-автоматы).....	85
7.11. Транслирующие грамматики.....	86
7.12. s и q - грамматики.....	87
7.13. LL(1) - грамматики.....	88

(left - leftmost).....	88
7.14. Метод рекурсивного спуска.....	89
7.15. LR - грамматики.....	90
(left - rightmost).....	90
7.16. Функции предшествования.....	93
7.17. Атрибутные грамматики.....	94
7.18. YACC.....	95
7.19. Область действия и передача параметров.....	96
7.20. Генерация выходного текста.....	97
Польская инверсная запись.....	97
7.21. Оптимизация программ.....	99
8. Функциональное программирование.....	100
9. Логическое программирование.....	104
Язык Пролог.....	104
10. Объектно-ориентированное программирование.....	105
Заключение.....	108
Литература.....	109

Введение

Специальная математика – это некоторые разделы современной математики. Речь идет о математическом аппарате, который помогает расширить возможности математического описания или, выражаясь изящнее – *математического моделирования*, сложных систем. Далеко не все задачи, которые возникают в сложных системах, включающих человека, можно свести к задачам механики или математического анализа, традиционно называемого в технических вузах «высшей математикой».

Самостоятельное значение имеют математические проблемы теоретического и практического программирования.

Последние сто лет интенсивно развивались разделы математики, многие из которых часто объединяют общим названием *дискретная математика*, хотя деление на дискретную и непрерывную математику более чем условно. (Возьмите множество всех подмножеств эталонного дискретного множества – множества натуральных чисел, и вы получите мощность базового для традиционного математического анализа множества – множества действительных чисел).

Так что чисто формально нет непреодолимой пропасти и антагонизма между дискретной и непрерывной математикой. Всякий инструмент хорош для решения задач, на которые он ориентирован. Вопрос удобства, эффективности использования и адекватности того или иного математического аппарата вообще до определенной степени вопрос субъективный.

А что касается классификации, то относить ли, например, теорию графов к дискретной математике или к топологии – тоже вопрос. Отнесение к дискретной математике теории групп еще более условно.

Задача данного курса состоит в выработке навыков формализации физических сущностей с помощью различных «диалектов» современного математического языка. И наоборот, интерпретации полученных математических результатов.

Содержательный аспект обычно предшествует формализации и имеет для нас значение при осмыслении результатов математических манипуляций.

Так что акцент в большей степени сделан на понятийной, а не вычислительной стороне ряда разделов математики.

1. Теория множеств

1.1 Понятие множества

Множество - фундаментальное неопределяемое понятие. Множество понимается как объединение в одно целое объектов, хорошо различимых нашей интуицией или мыслью.

Теорию множеств можно подразделить на аксиоматическую и интуитивную (наивную).

Аксиоматическая теория исходит из того, что множество определяется совокупностью аксиом, записанных обычно на языке логики (предикатов). Интуитивная теория множеств апеллирует к интуиции, к базовому понятию *принадлежности* элемента множеству, то есть к интуитивной понятности отношения принадлежности \in (*$a \in A$ - элемент a принадлежит множеству A*).

Для интуитивного понятия множества существенны два момента, следующие из "определения":

1. Различимость элементов.
 2. Возможность мыслить их как нечто единое.
- Студенты образуют группу. Деревья составляют лес.*
Целые числа составляют множество целых чисел.
Жители Марса - множество марсиан.

Множество, не содержащее элементов, называется **пустым множеством** и обозначается \emptyset или $\{\}$. Обычно именно фигурные скобки используются для выделения множества (отсутствие элементов в скобках и говорит о том, что это пустое множество).

Множество, *заведомо* содержащее все *рассматриваемые* элементы, называется **универсальным** или **универсумом** - U .

Было бы опрометчиво говорить просто, что U содержит *все* элементы. К сожалению, имеют место так называемые парадоксы теории множеств. Самый знаменитый – **парадокс Рассела**, который показывает невозможность построить множество *всех* подмножеств, не содержащих себя в качестве элемента. Более прост в понимании *парадокс бороды*, которому приказано брить в тридевятом государстве *всех тех и только тех*, кто не бреется сам. Перед бородыреем неразрешимый вопрос:

Включать ли самого себя в множество тех, кого он обязан брить?!

Способы задания множеств:

$A = \{a, b, c, d\}$ - задание множества явным перечислением элементов.

Например, *гвардия* = $\{\text{Иванов, Петров, Сидоров}\}$

$B = \{x \mid C(x)\}$ - задание множества (характеристическим) свойством $C(x)$.

Например, *студенчество* = $\{x \mid x - \text{студент}\}$ - множество таких x , что x - студент.

Отношение включения \subseteq . Множество A включено в множество B ($A \subseteq B$) или A есть **подмножество** множества B , если из $x \in A$ следует $x \in B$.

Например, *студенческая группа* \subseteq *студенты данной специальности*

Отношение строгого включения \subset : Если $A \subseteq B$ и $A \neq B$, то можно написать $A \subset B$.

Например: $\emptyset \subset$ *множество отличников*

Кстати, на что намекает это отношение?

Свойства отношения включения:

1. **Рефлексивность**: $A \subseteq A$

2. **Принцип объемности:** $A \subseteq B$ и $B \subseteq A$ следует $B = A$ (на основе этого принципа и доказывается равенство двух множеств).

3. **Транзитивность:** $A \subseteq B$ и $B \subseteq C$ следует $A \subseteq C$

Полезные соотношения:

$$\{\} = \emptyset; \quad 1 \neq \{1\}; \quad \{\{1\}\} \neq \{1\}; \quad \{a, b\} = \{b, a\}$$

1.2. Операции над множествами

1. **Объединение** множеств A и B

$$A \cup B = \{x \mid x \in A \text{ или } x \in B\} \text{ (или - неисключающее)}$$

2. **Пересечение** множеств A и B

$$A \cap B = \{x \mid x \in A \text{ и } x \in B\}$$

3. **Разность** множеств A и B

$$A \setminus B = \{x \mid x \in A \text{ и } x \notin B\}$$

4. **Симметрическая разность** множеств A и B

$$A \Delta B = \{x \mid (x \in A \text{ и } x \notin B) \text{ или } (x \notin A \text{ и } x \in B)\} = (A \setminus B) \cup (B \setminus A)$$

5. **Дополнение** множества \overline{A}

$$\overline{A} = \{x \mid x \notin A\}$$

Пример.

Пусть $A = \{1, 2, 3\}$ и $B = \{3, 4\}$, тогда

$$A \cup B = \{1, 2, 3, 4\}$$

$$A \cap B = \{3\}$$

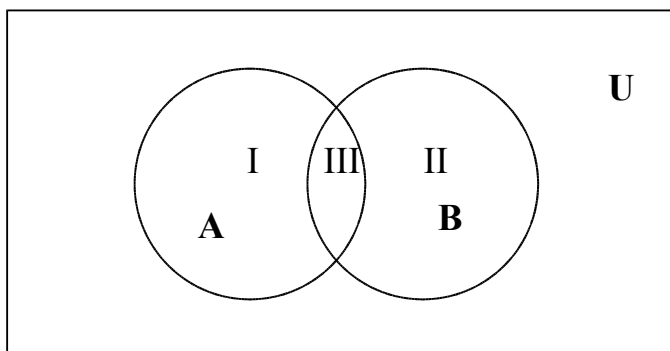
$$A \setminus B = \{1, 2\}$$

$$A \Delta B = \{1, 2, 4\}$$

\overline{A} = множество чисел кроме 1, 2, 3.

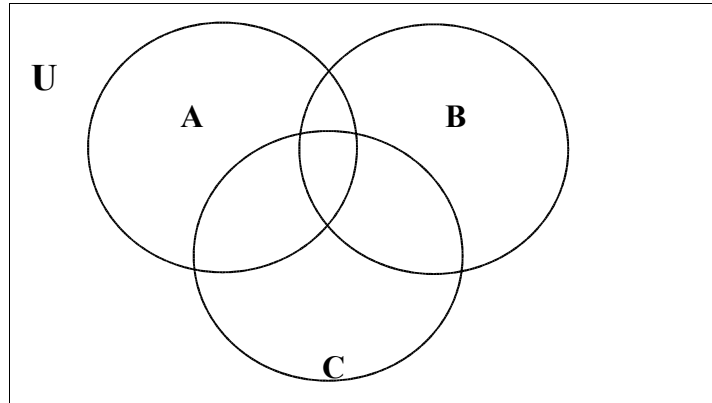
1.3. Диаграммы Эйлера - Венна

Диаграммы Эйлера-Венна позволяют представить множества, как множества точек на плоскости, ограниченные замкнутыми кривыми круглой или овальной формы. Прямоугольная рамка ограничивает универсум. Обычно, если не требуется иное, рисуют так называемый общий случай: когда каждое из множеств имеет свои собственные точки и точки, общие с другими множествами.



$A \cup B$ – зоны I, II, III.
 $A \cap B$ – зона III.
 $A \setminus B$ – зона I.
 \bar{A} – все, кроме круга A.
 $A \Delta B$ – зоны I, III.

Диаграмма для общего случая с тремя множествами будет иметь вид:



Построение диаграммы Эйлера-Венна для общего случая с четырьмя и более множествами можно предложить для самостоятельных развлечений.

1.4. Алгебра множеств

Операции над множествами дают в результате новые множества.

Для операций справедлив ряд законов. Приведем наиболее часто используемые.

Для упрощения записи, уменьшения числа скобок, определяющих последовательность операций, можно использовать соглашение о "силе" операций (в порядке убывания): дополнение, пересечение, объединение.

Остальные операции можно выразить через эти три.

Законы:

1. Коммутативный:

$$\overline{A \cup B} = \bar{B} \cap \bar{A}$$

$$\overline{A \cap B} = \bar{B} \cup \bar{A}$$

2. Ассоциативный:

$$A \cup (B \cap C) = (A \cup B) \cap C = A \cup B \cap C \quad A \cap (B \cup C) = (A \cap B) \cup C = A \cap B \cup C$$

3. Дистрибутивный:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

4. Поглощения:

$$A \cup (A \cap B) = A$$

$$A \cap (A \cup B) = A$$

5. Идемпотентности:

$$A \cup A = A$$

$$A \cap A = A$$

6. Исключенного третьего:

$$A \cup \bar{A} = U$$

Противоречия:

$$A \cap \bar{A} = \emptyset$$

7. $A \cup \emptyset = A$

$$A \cap \emptyset = \emptyset$$

8. $A \cup U = U$

$$A \cap U = A$$

9. Де Моргана:

$$\overline{A \cup B} = \bar{A} \cap \bar{B}$$

$$\overline{A \cap B} = \bar{A} \cup \bar{B}$$

$$10. \quad \overline{\emptyset} = U$$

$$\overline{U} = \emptyset$$

$$11. \text{ Двойного отрицания: } A = \overline{\overline{A}}$$

$$12. \quad A \setminus B = \overline{A} \cap B$$

$$13. \quad A \Delta B = A \cap \overline{B} \cup \overline{A} \cap B$$

Пример доказательства варианта дистрибутивного закона:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

I. Докажем, что левая часть включена в правую:

$$A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$$

Пусть $x \in A \cup (B \cap C)$, тогда у x есть две возможности

1. $x \in A$. Тогда $x \in A \cup B$ и $x \in A \cup C \Rightarrow x \in (A \cup B) \cap (A \cup C)$.

2. $x \in B \cap C$. Тогда $x \in B$ и $x \in C \Rightarrow x \in A \cup B$ и $x \in A \cup C$,
то есть $x \in (A \cup B) \cap (A \cup C)$.

II. Докажем, что правая часть включена в левую:

$$(A \cup B) \cap (A \cup C) \subseteq A \cup B \cap C.$$

Пусть $x \in A \cup B$ и $x \in A \cup C$. Тогда возможны два варианта:

1. $x \in A \Rightarrow x \in A \cup B \cap C$

2. $x \in B$ и $x \in C \Rightarrow x \in B \cap C \Rightarrow x \in A \cup B \cap C$.

То есть левое и правое множества равны.

1.5. Кортеж. График

Кортеж - фундаментальное неопределяемое понятие.

В кортеже существенны не только элементы, но и порядок, в котором они располагаются. Следовательно, кортеж может содержать одинаковые элементы.

Примерами кортежей могут служить *очередь*, *свадебный кортеж*. Кортежем является *вектор*, заданный проекциями на оси.

Кортеж заключается в угловые скобки.

$\langle a_1, a_2, a_3, \dots, a_n \rangle$ - кортеж длиной n или *упорядоченная n -ка*.

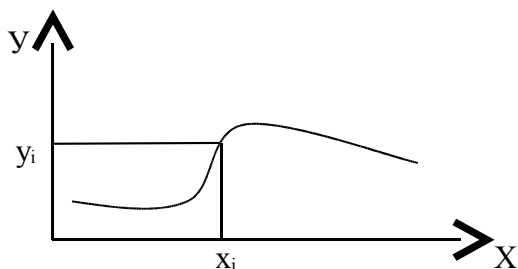
$\langle 1, 1, 1 \rangle$ - упорядоченная тройка – единичный вектор.

$\langle a, b \rangle$ - упорядоченная двойка или *пара*. Пару (и не только ее) можно представить и в традиционном виде, как множество: $\{a, \{a, b\}\}$. Однако использование угловых скобок упрощает представление.

График - множество пар. Можно дать и более общее определение графика в n -мерном пространстве, как множества n -ок). Однако в дальнейшем будут рассматриваться только двумерные графики.

Примеры: $G = \{ \langle a, b \rangle, \langle c, a \rangle, \langle d, b \rangle \}$ - график.

Несколько эпатажно звучит слово *график* применительно к аналитической записи. Но это лишь подчеркивает его универсальность. Для множеств действительных чисел X и Y приведем *графический пример графика*.



Декартово (прямое) произведение множеств A и B :

$$A \times B = \{ \langle a, b \rangle \mid a \in A, b \in B \}$$

В общем случае : $A_1 \times A_2 \times A_3 \times \dots \times A_n = \{ \langle a_1, a_2, \dots, a_n \rangle \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n \}$

Пример : Для $A = \{ 1, 2 \}$ и $B = \{ 1, 2, 3 \}$ декартово произведение
 $A \times B = \{ \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle \}$

График является **полым**, если он совпадает с декартовым произведением.

Композицией графиков P и Q называется график $R = P \bullet Q$, если он состоит из таких пар $\langle x, y \rangle \in R$, что для каждой пары найдется свое z , такое, что $\langle x, z \rangle \in P$, $\langle z, y \rangle \in Q$. Очевидно, что это некоммутативная операция.

Пример :

$$P = \{ \langle a, b \rangle, \langle 1, r \rangle, \langle c, 3 \rangle, \langle a, 4 \rangle \}$$

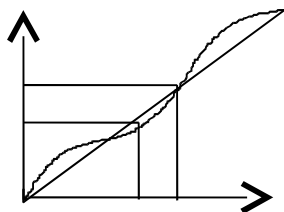
$$Q = \{ \langle 2, 3 \rangle, \langle 4, 5 \rangle, \langle a, c \rangle, \langle b, d \rangle \}$$

$$R = P \bullet Q = \{ \langle a, d \rangle, \langle a, 5 \rangle \}$$

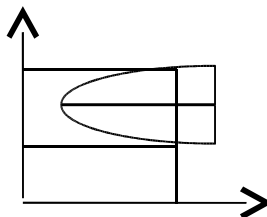
Свойства графиков

1. График называется **функциональным**, если он не содержит пар с одинаковой первой и различными вторыми компонентами.
2. График называется **инъективным**, если он не содержит пар с одинаковой второй и различными первыми компонентами.
3. График называется **симметричным**, если он равен своей инверсии.
4. График называется **диагональю** множества M , если он состоит из пар вида $\langle x, x \rangle$: $\Delta_M = \{ \langle x, x \rangle \mid x \in M \}$

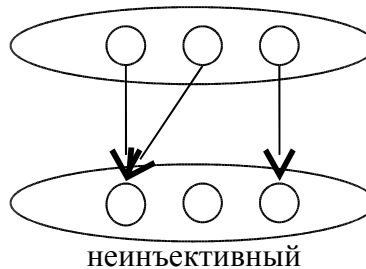
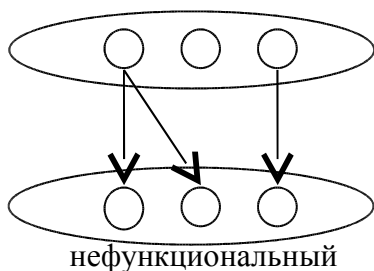
Примеры



функциональный



нефункциональный



Пара $\langle a, b \rangle$ называется **инверсией пары** $\langle c, d \rangle$, если $a = d, b = c$.

График P^{-1} - **инверсия** графика P , если он состоит из инверсий пар графика P .

Пример

$P = \{\langle a, b \rangle, \langle b, e \rangle, \langle k, s \rangle\}$

$P^{-1} = \{\langle b, a \rangle, \langle e, b \rangle, \langle s, k \rangle\}$

Проекция кортежа на заданные оси - есть кортеж, составленный из соответствующих компонент исходных кортежей. Рассматриваются только проекции на возрастающий (по номеру) список осей.

Пример

$V = \langle 2, 5, 6, 4, 2, 6 \rangle$

$\text{пр}.V_{1,2,4} = \langle 2, 5, 4 \rangle$

Проекция некоторого множества M на множество осей дает множество проекций кортежей, составляющих множество. Исходное множество должно состоять из кортежей одинаковой длины.

Пример

$M = \{\langle a, b, c \rangle, \langle a, c, d \rangle, \langle k, l, m \rangle, \langle o, p, r \rangle\}$

$\text{пр}.M_{1,3} = \{\langle a, c \rangle, \langle a, d \rangle, \langle k, m \rangle, \langle o, r \rangle\}$

1.6. Соответствия

$\Gamma = \langle G, X, Y \rangle$

Соответствие - тройка, такая, что $G \subseteq X * Y$ - подмножество произведения второго компонента на третий.

Первый компонент (G) - график.

Второй компонент (X) - область отправления (определения).

Третий компонент (Y) - область прибытия (значений).

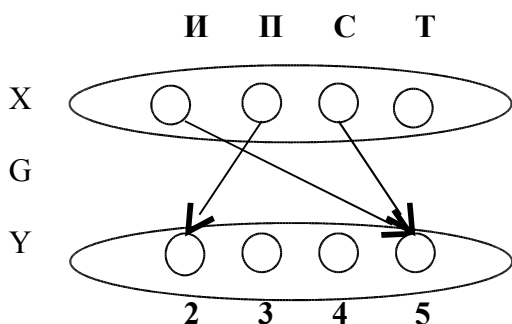
Соответствие называется **полным**, если $G = X \times Y$.

Свойства соответствий

1. Соответствие называется **функциональным**, если его график функционален.
2. Соответствие называется **инъективным**, если его график инъективен.
3. Соответствие называется **всюдуопределенным**, если проекция графика на первую ось совпадает с областью отправления. $\text{пр}.G_1 = X$.
4. Соответствие называется **сюръективным**, если проекция графика на вторую ось совпадает с областью прибытия. $\text{пр}.G_2 = Y$.

5. Соответствие называется **биективным** (взаимно-однозначным), если оно **функционально, инъективно, всюдуопределено и сюръективно**.

Пример : Соответствие «студенты сдавали экзамен». (Трифонов не пришел).



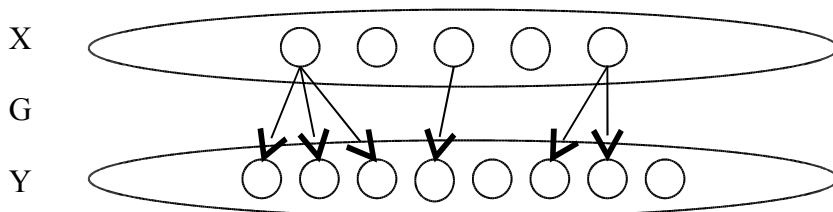
$X = \{\text{Иванов, Петров, Сидоров, Трифонов}\}$ – множество студентов.

$Y = \{2, 3, 4, 5\}$ – множество возможных оценок.

$G = \{<\text{И}, 5>, <\text{П}, 2>, <\text{С}, 5>\}$ – результаты сдачи экзамена.

Соответствие функционально, неинъективно, не всюдуопределено, несюръективно, небиективно.

Пример : Соответствие «покупателей и купленных товаров».



Типовая ситуация для такого соответствия: не функционально, инъективно, не всюду определено, несюръективно, небиективно.

1.7. Отношения

Отношение, это пара

$$\rho = \langle R, M \rangle$$

$$R \subseteq M * M = M^2$$

Первый компонент (R) - график отношения.

Второй компонент (M) - множество, на котором отношение определено.

Более традиционная запись отношения $x \rho y$ для $x \in M, y \in M$.

Свойства отношений

1. **Рефлексивность:** $x \rho x$ (например, $x = x$)

2. **Антирефлексивность:** $\neg x \rho x$ (например, $x < x$)

3. **Симметричность:** $x \rho y \Rightarrow y \rho x$ (например, $x = y \Rightarrow y = x$)

4. **Антисимметричность:** $x \neq y, x \rho y \Rightarrow \neg y \rho x$ (например, $x \neq y; y \leq x \Rightarrow \neg y \geq x$)

4'. **Асимметричность:** $x \rho y \Rightarrow \neg y \rho x$ (например, $x < y \Rightarrow \neg y < x$)

5. **Связность (полнота)**: $x \neq y \Rightarrow x \rho y$ или $y \rho x$ (например, для любых двух различных натуральных чисел: либо $x < y$, либо $y < x$)
6. **Транзитивность**: $x \rho y, y \rho z \Rightarrow x \rho z$ (например, $x = y$ и $y = z \Rightarrow x = z$)
7. **Антитранзитивность**: $x \rho y, y \rho z \Rightarrow \neg x \rho z$ (например, отношение перпендикулярности прямых).

1.7.1 Отношение эквивалентности

Отношение, обладающее одновременно свойствами рефлексивности, симметричности и транзитивности, называется **отношением эквивалентности**.

\sim - символ отношения эквивалентности.

$[x]$ - множество элементов, эквивалентных x (**класс эквивалентности x**).

Свойства отношения эквивалентности:

1. $x \sim x$

2. Если $x \sim y \Rightarrow [x] = [y]$

Доказательство 1-го свойства: Следует из свойства рефлексивности.

Доказательство 2-го свойства: 1. $z \in [x] \Rightarrow z \sim x, x \sim y \Rightarrow z \sim y \Rightarrow z \in [y]$, т.е. $[x] \subseteq [y]$

2. $z \in [y] \Rightarrow z \sim y, x \sim y \Rightarrow z \sim x \Rightarrow z \in [x]$, т.е. $[y] \subseteq [x]$.

Следовательно $[x] = [y]$

$P(M)$ - **множество-степень** множества M есть множество всех подмножеств множества M .

Пример:

$M = \{1, 2, 3\}$

$P(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

$P(M)$ - **покрытием** множества M будем называть любое подмножество множества $P(M)$, такое, что объединение входящих в него элементов совпадает с M .

$P(M) = \{\{1, 2\}, \{2\}, \{2, 3\}\}$

так как $\{1, 2\} \cup \{2\} \cup \{2, 3\} = \{1, 2, 3\}$

$R(M)$ - **разбиением** множества M называется такое покрытие множества M , в котором элементы не пересекаются.

Пример разбиения: $R = \{\{1, 2\}, \{3\}\}$

Свойства :

1. Каждый элемент исходного множества M принадлежит какому-либо из множеств, составляющих разбиение.

2. Каждый элемент исходного множества принадлежит строго одному из множеств, составляющих разбиение.

Теорема: Отношение эквивалентности разбивает множество, на котором оно определено на классы эквивалентности.

Доказательство: 1. Очевидно. $x \sim [x]$

2. Предположим, что $z \in [x]$ и $z \in [y]$. Тогда из $x \sim y$ и $z \sim y$ следует $x \sim y$ и по второму свойству отношения эквивалентности $[x] = [y]$.

1.7.2. Отношения порядка

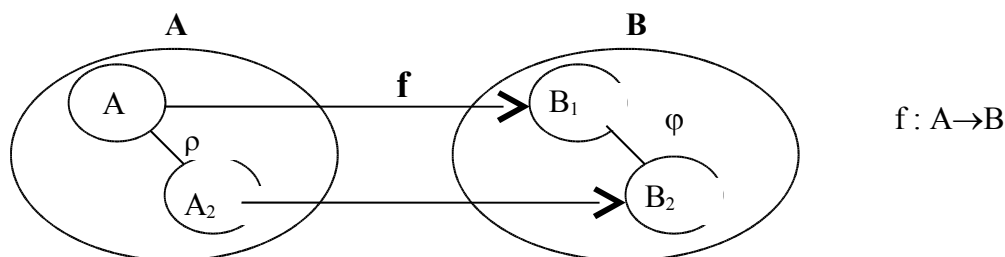
Четыре определения отношений порядка можно свести в таблицу.

Свойства Порядки	Рефлексивность	Антирефлексивность	Антисимметричность	Полнота	Транзитивность
нестрогий (частичный)	+		+		+
совершенный нестрогий	+		+	+	+
строгий		+	(+)		+
совершенный строгий		+	(+)	+	+

То есть, например, нестрогий (частичный) порядок - отношение, обладающее свойствами , рефлексивности, антисимметричности и транзитивности.

1.7.3. Морфизмы

Всюду-определенное функциональное соответствие называется **отображением**.



Отображение f называется **отображением гомоморфизма** или **гомоморфным отображением**, или просто **морфизмом**, если для элементов множества A выполняется $A_1 \rho A_2$, а для образов выполняется $B_1 \varphi B_2$. То есть $f(A_1 \rho A_2) = f(A_1) \varphi f(A_2)$, где $f(A_1) = B_1$, $f(A_2) = B_2$.

Содержательный пример морфизма – высота земной поверхности над уровнем моря и более темный коричневый цвет на географической карте.

Эндоморфизм - гомоморфизм "в себя".

Мономорфизм - инъективный гомоморфизм.

Эпиморфизм - сюръективный гомоморфизм.

Изоморфизм - биективный гомоморфизм

Аutomорфизм - изоморфизм в себя.

1.8. Решетки

Решетки - это частично-упорядоченные множества, отношения порядка на которых, удовлетворяют ряду дополнительных требований.

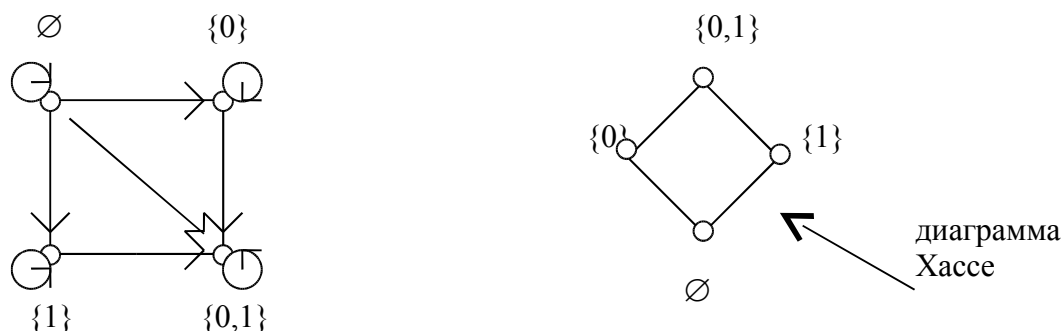
чум - частично-упорядоченное множество, т.е. множество с определенным на нем частичным порядком.

1.8.1. Диаграммы Хассе

Диаграммы Хассе используются для того, чтобы за счет принятых по умолчанию соглашений облегчить графическое представление частично-упорядоченных множеств.

Пример изображения частичного порядка (устанавливаемого отношением включения) для множества

$\{\emptyset, \{0\}, \{1\}, \{0,1\}\}$



По умолчанию на диаграмме Хассе:

«Стрелки» направлены снизу вверх.

Не отображается рефлексивность.

Не отображаются транзитивные замыкания.

1.8.2. Понятие решетки

Пусть рассматриваемые далее множества A и B - чум.

Наибольшим (наименьшим) элементом $a \in A$ называется элемент a , если $a \geq (\leq) x$, где $x \in A$.

Теорема: Если в множестве A существует наибольший элемент, то он единственный.

Доказательство: Предположим, что существуют два наибольших элемента a_1 и a_2 , тогда :

$$\left. \begin{array}{l} a_1 \geq a_2 \\ a_2 \geq a_1 \end{array} \right\} a_1 = a_2;$$

Максимальным (минимальным) элементом множества A называется элемент $a \in A$, когда неверно, что $a \leq (\geq) x$, где $x \in A$.

Мажорантой (минорантой) множества B (такого что $\emptyset \subset B \subseteq A$) является элемент $a \in A$, такой что элемент a является наибольшим (наименьшим) элементом для множества B .

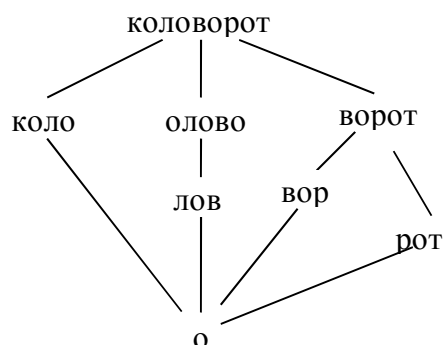
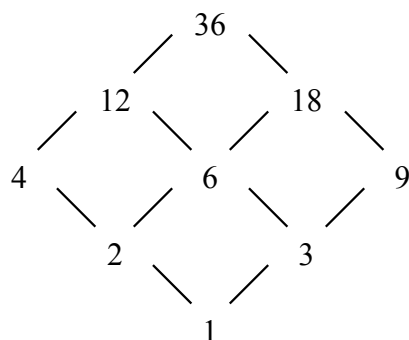
Множество мажорант (минорант) множества B образует **верхнюю (нижнюю) грань** множества B .

Наименьший элемент верхней грани называется **точной верхней гранью** или **Supremum (Sup)**.

Наибольший элемент нижней грани называется **точной нижней гранью** или **Infimum (Inf)**.

Частично-упорядоченное множество, в котором любая пара элементов имеет Sup и Inf называется **решеткой**.

Примеры решеток.



1.8.3. Алгебраическое представление решеток.

Булевы решетки

Введем обозначения $\text{Sup}(a, b) = a \cup b$, $\text{Inf}(a, b) = a \cap b$,

Будем считать традиционно используемые здесь значки \cup , \cap не имеющими никакого отношения к теоретико-множественным операциям объединения и пересечения.

Если выполняются законы :

$$1. a \cup b = b \cup a$$

$$2. (a \cup b) \cup c = (b \cup c) \cup a = a \cup b \cup c$$

$$3. a \cup (a \cap b) = a$$

$$4. a \cup a = a$$

$$1'. a \cap b = b \cap a$$

$$2'. (a \cap b) \cap c = (b \cap c) \cap a = a \cap b \cap c$$

$$3'. a \cap (b \cup a) = a$$

$$4'. a \cap a = a$$

то имеет место **решетка**.

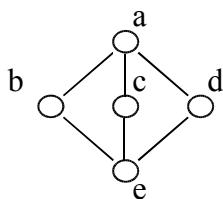
То есть решетка можно определить как алгебру $Z = \langle L, \cap, \cup \rangle$, для операций которой выполняются вышеперечисленные законы.

Решетка называется **дистрибутивной**, если дополнительно к вышеперечисленным выполняется дистрибутивный закон:

$$5. a \cup b \cap c = (a \cup b) \cap (a \cup c)$$

$$5'. a \cap (b \cup c) = a \cap b \cup a \cap c$$

Пример : Недистрибутивная решетка:



$$a \cup b \cap c = (a \cup b) \cap (a \cup c)$$

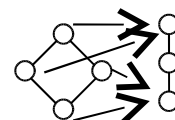
$$a \cup e = a \cap a$$

$$a = a$$

$$b \cup c \cap d = b \cap c \cup b \cap d$$

$$b \cup e = a \cup a$$

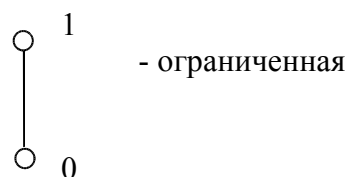
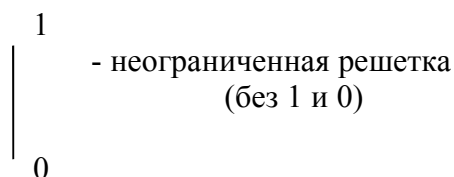
$$b \neq a \text{ недистрибутивность}$$



Эта решетка недистрибутивная.

Решетка называется **ограниченной**, если она имеет максимальный и минимальный элементы.

Например, если взять отрезок действительной оси от 0 до 1 (вместе с конечными точками) и отношение "меньше", то это будет ограниченная решетка. Убрав крайние точки, получаем неограниченную решетку.



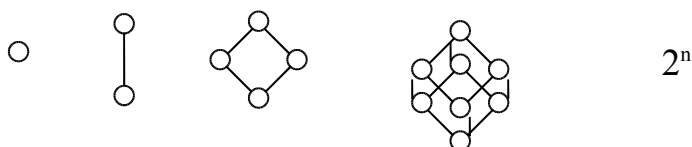
Обычно минимальный элемент решетки обозначают как 0, а максимальный как 1.

\bar{a} - **дополнение** a , если $a \cup \bar{a} = 1$ и $a \cap \bar{a} = 0$

Решетка является **решеткой с дополнением**, если каждый элемент имеет хотя бы одно дополнение.

Ограниченная дистрибутивная решетка с дополнением является **булевой**.

Примеры булевых решеток:



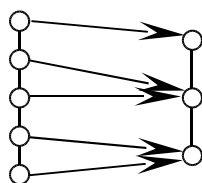
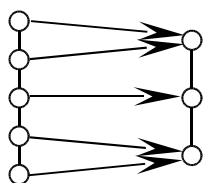
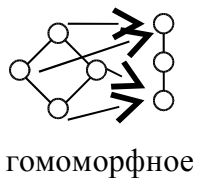
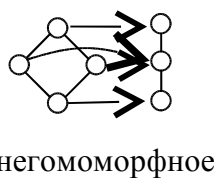
1.8.4. Подрешетки

Пусть даны две решетки: $\mu = \langle L, \cap, \cup \rangle$ и $\lambda = \langle N, \cap, \cup \rangle$, тогда λ - **подрешетка** решетки μ , если $N \subseteq L$ и $n_1 \in N, n_2 \in N$, то $n_1 \cap n_2 \in N$ и $n_1 \cup n_2 \in N$.

Если $\chi = \langle I, \cap, \cup \rangle$ - подрешетка решетки μ , и из $i \in I, l \in L$ следует $i \cap l \in I$, то χ называется **идеалом**.

Если $v = \langle F, \cap, \cup \rangle$ - подрешетка решетки μ , и из $f \in F, l \in L$ следует $f \cup l \in F$, то v называется **фильтром**.

1.8.5. Морфизмы решеток



гомоморфные

1.9. Мощность множества

Обозначения:

\mathbb{N} - множество натуральных чисел.

\mathbb{Z} - множество целых чисел.

\mathbb{Q} - множество рациональных чисел.

\mathbb{R} - множество действительных чисел.

\mathbb{C} - множество комплексных чисел.

1.9.1. Понятие мощности

Г. Кантор понимал мощность, как двойную абстракцию. Мы абстрагируемся от конкретных элементов множества и от порядка, в котором они расположены. То, что в результате остается и есть **мощность**. Мощности можно сравнивать на больше, меньше, равно.

\bar{N} - мощность множества N .

1.9.2. Аксиоматика Пеано

Наименьшей бесконечной мощностью является **счетная мощность** - мощность множества натуральных чисел. Это бесконечное множество можно задать с помощью системы аксиом:

$$1. 0 \in \mathbb{N}$$

$$2. n \in \mathbb{N} \Rightarrow n' \in \mathbb{N}$$

$$3. n \in \mathbb{N} \Rightarrow n' \neq 0$$

$$4. n \in \mathbb{N}, m \in \mathbb{N}, n' = m' \Rightarrow n = m$$

$$\left. \begin{array}{l} 5. 0 \in A \subseteq \mathbb{N} \\ n \in A \Rightarrow n' \in A \end{array} \right\} A = \mathbb{N}$$

где n' - элемент, следующий за n .

$\bar{N} = \aleph_0$ (алеф-нуль) - счетная мощность.

1.9.3. Сравнение мощностей

1. Сравним мощность множества \mathbb{N} и мощность множества целых четных положительных чисел:

1 2 3 4 5 ...

2 4 6 8 10 ...

то есть можно между этими множествами установить взаимно-однозначное соответствие. Это будет множество пар вида $\langle n, 2n \rangle$.

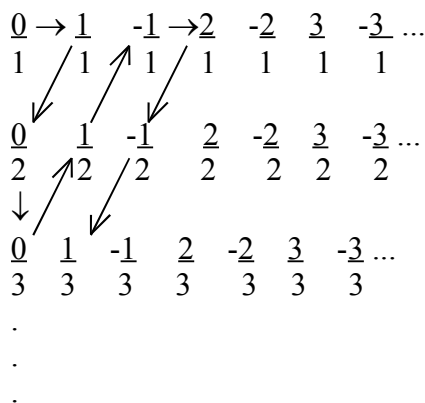
2. Сравним мощность множества \mathbb{N} и множества \mathbb{Z} .

1 2 3 4 5 6 ...

0 1 -1 2 -2 3 ...

Здесь также имеет место взаимно-однозначное соответствие. То есть эти множества равномощны.

3. Сравним мощность множества N и множества Q .



В эту сетку попадут все рациональные числа.

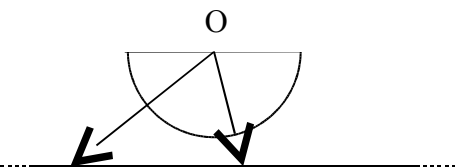
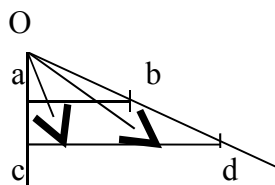
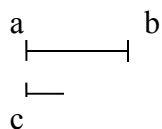
Мощность Q также равна мощности N .

1.9.4. Мощность множества R .

Теорема Кантора

Аналогом мощности действительных (вещественных) чисел служит множество точек на отрезке действительной оси или на всей действительной оси.

Равномощность различных отрезков, а также отрезка и всей прямой показаны на рисунках.



Теорема Кантора.

$$N = N_0 < R \quad (\aleph_0 < \aleph_1)$$

Доказательство.

1. Поскольку множество R имеет такую же мощность, как и любой отрезок R , то будем рассматривать отрезок между 0 и 1. Числа будут представляться в виде бесконечных десятичных дробей. Конечные дроби для однозначности будут заменяться своими бесконечными аналогами. Например, $0.45 = 0.4499999\ldots$

Допустим, что каким-то образом установлено взаимно-однозначное соответствие между числами отрезка от 0 до 1 и множеством N .

$$\begin{aligned} &0, a_1^1, a_2^1, a_3^1 \dots\dots \\ &0, a_1^2, a_2^2, a_3^2 \dots\dots \\ &0, a_1^3, a_2^3, a_3^3 \dots \end{aligned}$$

Но здесь отсутствует число $0, b_1, b_2, b_3 \dots$ где $a_1^1 \neq b_1, b_2 \neq a_2^2 \dots b_n \neq a_n^n$
 Следовательно, предположение о возможности «пересчитать» множество действительных чисел на отрезке от 0 до 1 неверно. Действительных чисел больше.
 Мощность множества действительных чисел \aleph_1 называется *мощностью континуума*.

1.9.5. Арифметика бесконечного

Бесконечных мощностей бесконечно много: $\aleph_0 < \aleph_1 < \aleph_2 < \aleph_3 < \dots$
 \aleph_0 - самая маленькая бесконечная мощность.

$\aleph_0 + \underline{A} = \aleph_0$	$\aleph_1 - \aleph_0 = \aleph_1$
$\aleph_0 + \aleph_0 = \aleph_0$	$\aleph_0 - A = \aleph_0$
$\aleph_1 + \aleph_1 = \aleph_1$	$\aleph_0 - \aleph_0 = \aleph_0$
$\aleph_1 + \aleph_1 = \aleph_1$	$\aleph_0 - \aleph_1 = \aleph_1$

1.9.6. Противопоставление системного и теоретико-множественного подходов

1. Системы, как и множества, состоят из элементов.
Теория систем исходит из первичности системы, в то время как теоретико-множественный подход считает, что первичен элемент.
2. Естественность системы (в ней нет случайных элементов) и "неразборчивость" множества.
3. Абстракция отождествления для множеств и априорная организация систем.
4. Системам присуща внутренняя организация, множествам - внешняя.

2. Математическая логика

2.1. Логика высказываний

Под **высказыванием** будем понимать повествовательное предложение, относительно которого можно сказать - истинно оно или ложно.

Высказываниями не являются определения, восклицательные и вопросительные предложения, а также логические парадоксы.

Определение: Угол в 90 градусов называется прямым углом.

Восклицание: Смирно!

Вопрос: Кто сказал "мяу"?

Парадокс лжеца: "Я лгу".

Если это высказывание ложь, то я говорю правду.

Но если я говорю правду, то я действительно лгу.

Высказывания будем обозначать отдельными буквами.

Более строго их можно называть **элементарными высказываниями**.

Главный содержательный парадокс логики высказываний состоит в том, что она не интересуется *смыслом* высказываний. По образному сравнению логика Клини в математической логике на высказывания смотрит через «рентген», который отбрасывает их содержательный смысл и оставляет только "скелет" высказывания - его истинность.

Истинность может принимать два значения

истинно	ложно
и	л
true	false
t	f

но самые популярные обозначения

1 0

которые не следует путать с числами двоичной арифметики.

2.1.1. Операции над высказываниями

1. Дизъюнкция (логическое “или”, “логическое сложение”). Наиболее популярные обозначения: \vee и $+$.
2. Конъюнкция (логическое “и” “логическое умножение”). Наиболее популярные обозначения: \cdot , \wedge и $\&$.
3. Отрицание (логическое “не”). Наиболее популярные обозначения: \neg и $\overline{}$.
4. Импликация (логическое “если ... , то”, “влечет”) \rightarrow .
5. Эквивалентность (логическое “если и только если”) \leftrightarrow .
6. Неравнозначность (или “сумма по модулю 2”, или “исключающее или”) \oplus .
7. Штрих Шеффера (логическое “и-не”) $|$.
8. Стрелка Пирса (логическое “или-не”) \downarrow .

Операции сведены в таблицу:

A	B	\vee	\wedge	\bar{A}	\rightarrow	\leftrightarrow	\oplus	$ $	\downarrow
0	0	0	0	1	1	1	0	1	1
0	1	1	0	1	1	0	1	1	0
1	0	1	0	0	0	0	1	1	0
1	1	1	1	0	1	1	0	0	0

Соглашение о старшинстве некоторых операций (по силе связывания):

\neg , $\&$, \vee , \rightarrow , \leftrightarrow .

2.1.2. Построение и анализ сложных высказываний

В качестве примера возьмем отрывок из Шолом-Алейхема (заимствованный, однако, у Д.А. Пospelова).

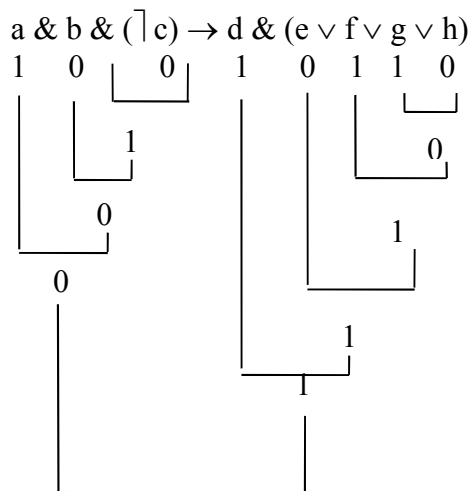
“... - Вот, что. Если вы хотите остаться у нас, если вы хотите, чтобы мы стали друзьями.

- Если вы не хотите, чтобы вам пришлось уезжать отсюда, то

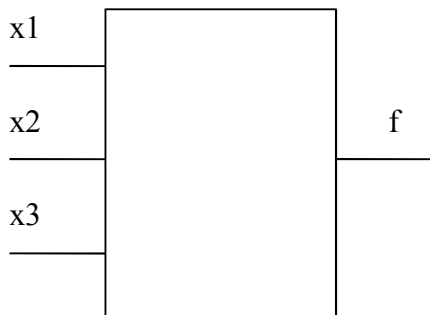
- Забросьте книги под стол. Будем играть в шашки, в бб или будем валяться на кровати и плевать в потолок...”

Придав конкретные значения отдельным элементарным высказываниям, можем определить истинность всего сложного высказывания для этого набора значений.

a - хочешь остаться в доме	1
b - хочешь остаться другом	0
c - захотеть уехать	0
d - забросить под стол книги	1
e - играть в шашки	0
f - играть в бб	1
g - валяться на кровати	1
h - плевать в потолок	0



Другой пример. Пусть на три входа "черного ящика" (x_1 , x_2 , x_3) подаются (1) или не подаются (0) импульсы во всевозможных сочетаниях. На выходе (f) импульс либо появляется (1), либо отсутствует (0).



Результаты замеров заносятся в «журнал исследований» - таблицу истинности. Пусть она имеет следующий вид:

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1

x1	x2	x3	f
1	1	0	1
1	1	1	1

Что также можно записать в виде формулы:

$$f = \overline{x_1} \cdot x_2 \cdot x_3 \vee x_1 \cdot \overline{x_2} \cdot x_3 \vee x_1 \cdot x_2 \cdot \overline{x_3} \vee x_1 \cdot x_2 \cdot x_3$$

2.1.3. Алгебра высказываний

Сложные высказывания называются **равносильными** ($f \equiv g$), если на одинаковых наборах значений элементарных высказываний они принимают одинаковые значения.

Законы :

1. Коммутативный.

$$A \vee B \equiv B \vee A \quad A \cdot B \equiv B \cdot A$$

2. Ассоциативный.

$$A \vee (B \vee C) \equiv A \vee B \vee C \quad A \cdot (B \cdot C) \equiv A \cdot B \cdot C$$

3. Дистрибутивный.

$$A \vee B \cdot C \equiv (A \vee B) \cdot (A \vee C)$$

$$A \cdot (B \vee C) \equiv A \cdot B \vee A \cdot C$$

4. Де Моргана.

$$\overline{A \vee B} \equiv \overline{A} \cdot \overline{B} \quad \overline{A \cdot B} \equiv \overline{A} \vee \overline{B}$$

5. Идемпотентности.

$$A \vee A \equiv A \quad A \cdot A \equiv A$$

6. Поглощения.

$$A \vee (A \cdot B) \equiv A \quad A \cdot (A \vee B) \equiv A$$

7. Искл. третьего. Противоречия.

$$\overline{A} \vee A \equiv 1 \quad \overline{A} \cdot A \equiv 0$$

8. $A \vee 1 \equiv 1$

$$A \cdot 1 \equiv A$$

9. $A \vee 0 \equiv A$

$$A \cdot 0 \equiv 0$$

10. $\overline{0} \equiv 1$

$$\overline{1} \equiv 0$$

11. $\overline{\overline{A}} \equiv A$

12. $A \rightarrow B \equiv \overline{A} \vee B$

13. $A \leftrightarrow B \equiv A \cdot B \vee \overline{A} \cdot \overline{B}$

14. $A \oplus B \equiv A \cdot \overline{B} \vee \overline{A} \cdot B$

15. $A | B \equiv \overline{A \cdot B} \equiv \overline{A} \vee \overline{B}$

16. $A \downarrow B \equiv \overline{A \vee B} \equiv \overline{A} \cdot \overline{B}$

17. Операция склеивания.

$$A \cdot B \vee A \cdot \overline{B} \equiv A$$

2.1.4. Формы представления высказываний

1. Форма $A_1 \vee A_2 \vee \dots \vee A_n$, где A_i - элементарное высказывание или отрицание элементарного высказывания (литерал), называется **элементарной дизъюнкцией**.

2. Форма $B_1 \cdot B_2 \cdot \dots \cdot B_n$, где B_i - литерал, называется **элементарной конъюнкцией**.

3. Форма $D_1 \cdot D_2 \cdot \dots \cdot D_n$, где D_j - элементарная дизъюнкция, называется **конъюнктивной нормальной формой (КНФ)**.
4. Форма $K_1 \vee K_2 \vee \dots \vee K_n$, где K_j - элементарная конъюнкция, называется **дизъюнктивной нормальной формой (ДНФ)**.

Всегда истинное (на любых наборах значений входящих в него элементарных высказываний) сложное высказывание называется **тавтологией**.

Всегда ложное (на любых наборах значений входящих в него элементарных высказываний) высказывание называется **противоречием**.

Совершенной КНФ (СКНФ) называется такая КНФ, что каждая входящая в нее элементарная дизъюнкция содержит все элементарные высказывания прямо или с инверсией строго по одному разу. Нет повторяющихся дизъюнкций. Любое сложное высказывание, кроме тавтологии, имеет единственную СКНФ.

Совершенной ДНФ (СДНФ) называется такая ДНФ, что каждая входящая в нее элементарная конъюнкция содержит все элементарные высказывания прямо или с инверсией строго по одному разу. Нет повторяющихся конъюнкций. Любое сложное высказывание, кроме противоречия, имеет единственную СДНФ.

2.1.5. Преобразование высказываний

Сложное высказывание, представленное в произвольном виде с помощью равносильностей с 11 по 16, а также с использованием законов Де Моргана могут быть преобразованы к нормальной форме.

Преобразование КНФ в СКНФ.

Схематично основную идею преобразования можно представить так:

$$X \vee Y \equiv X \vee Y \vee 0 \equiv X \vee Y \vee Z \cdot \bar{Z} \equiv (X \vee Y \vee Z) \cdot (X \vee Y \vee \bar{Z})$$

Преобразование ДНФ в СДНФ.

Схематично основную идею преобразования можно представить так:

$$X \cdot Y \equiv X \cdot Y \cdot 1 \equiv X \cdot Y \cdot (Z \vee \bar{Z}) \equiv X \cdot Y \cdot Z \vee X \cdot Y \cdot \bar{Z}$$

Преобразование СДНФ в СКНФ и наоборот.

Рассмотрим на примере:

Возьмем логическую функцию f (сложное высказывание) в СДНФ и построим отрицание этой функции, т.е. функцию \bar{f} , путем выписывания всех конstituент единицы, не входящих в f .

Примеры:

Пусть f имеет вид

$$f = \bar{X}_1 \cdot X_2 \cdot X_3 \vee X_1 \cdot \bar{X}_2 \cdot X_3 \vee X_1 \cdot X_2 \cdot \bar{X}_3 \vee X_1 \cdot X_2 \cdot X_3$$

3
5
6
7

(мнемонический прием – приписать конstituентам числа, которые получаются, если посмотреть на конstituенты как на двоичные числа)

Отрицание функции f получим выписыванием недостающих конституент (недостающих двоичных чисел).

$$\bar{f} = \underbrace{\bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3}_{0} \vee \underbrace{X_1 \cdot \bar{X}_2 \cdot \bar{X}_3}_{1} \vee \underbrace{\bar{X}_1 \cdot X_2 \cdot \bar{X}_3}_{2} \vee \underbrace{X_1 \cdot \bar{X}_2 \cdot X_3}_{4}$$

А теперь применим отрицание к функции \bar{f} .

$$\begin{aligned} \bar{\bar{f}} &= \overline{\bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3 \vee \bar{X}_1 \cdot \bar{X}_2 \cdot X_3 \vee \bar{X}_1 \cdot X_2 \cdot \bar{X}_3 \vee X_1 \cdot \bar{X}_2 \cdot X_3} = \\ &= (X_1 \vee X_2 \vee X_3) \cdot (X_1 \vee X_2 \vee \bar{X}_3) \cdot (X_1 \vee \bar{X}_2 \vee X_3) \cdot (\bar{X}_1 \vee X_2 \vee X_3) - \text{СКНФ (функции } f). \end{aligned}$$

Пример 2:

$$f = \underbrace{\bar{X} \cdot Y \cdot \bar{Z}}_2 \vee \underbrace{X \cdot Y \cdot Z}_7 \vee \underbrace{\bar{X} \cdot \bar{Y} \cdot \bar{Z}}_0 \vee \underbrace{X \cdot \bar{Y} \cdot Z}_5 \vee \underbrace{X \cdot \bar{Y} \cdot \bar{Z}}_4 \vee \underbrace{\bar{X} \cdot Y \cdot Z}_3$$

$$\bar{\bar{f}} = \overline{\underbrace{X \cdot Y \cdot \bar{Z}}_6 \vee \underbrace{\bar{X} \cdot \bar{Y} \cdot Z}_1} = (\bar{X} \vee \bar{Y} \vee Z) \cdot (X \vee Y \vee \bar{Z})$$

Переход от СКНФ к СДНФ.

Возьмем логическую функцию f в СКНФ и построим отрицание этой функции, т.е. функцию \bar{f} , путем выписывания всех конституент нуля, не входящих в f .

Пусть f имеет вид

$$f = (\bar{X} \vee \bar{Y} \vee Z) \cdot (X \vee Y \vee \bar{Z})$$

$$\begin{aligned} \bar{\bar{f}} &= \overline{(\bar{X} \vee \bar{Y} \vee Z) \cdot (\bar{X} \vee \bar{Y} \vee \bar{Z}) \cdot (\bar{X} \vee Y \vee Z) \cdot (X \vee \bar{Y} \vee \bar{Z}) \cdot (X \vee \bar{Y} \vee Z) \cdot (X \vee Y \vee Z)} = \\ &= X \cdot Y \cdot Z \vee X \cdot \bar{Y} \cdot Z \vee X \cdot \bar{Y} \cdot \bar{Z} \vee \bar{X} \cdot Y \cdot Z \vee \bar{X} \cdot Y \cdot \bar{Z} \vee \bar{X} \cdot \bar{Y} \cdot Z \end{aligned}$$

2.1.6. Минимизация высказываний методом Квайна

1. Выражение из произвольной формы приводится к СДНФ.
2. Выполнив в СДНФ все возможные неполные склеивания, а затем все возможные поглощения мы получим **Сокращенную ДНФ (С_кДНФ)**. Конъюнкции в С_кДНФ называются **импликантами**.

Примечание: Склеивание: $X \cdot Y \vee X \cdot \bar{Y} \equiv X$

Неполное склеивание: $X \cdot Y \vee X \cdot \bar{Y} \equiv X \vee X \cdot Y \vee X \cdot \bar{Y}$

3. На основании С_кДНФ и СДНФ строим *импликантную матрицу* и путем нахождения минимального покрытия этой матрицы получаем **минимальную дизъюнктивную нормальную форму (МДНФ)**.

Пример 1:

$$f = \underbrace{\bar{X} \cdot \bar{Y} \cdot \bar{Z}}_{(I)} \vee \underbrace{\bar{X} \cdot \bar{Y} \cdot Z}_{(II)} \vee \underbrace{X \cdot \bar{Y} \cdot \bar{Z}}_{(III)} \vee \underbrace{X \cdot Y \cdot \bar{Z}}_{(IV)} \vee \underbrace{\bar{X} \cdot Y \cdot \bar{Z}}_{(V)}$$

I-II : $\bar{X} \cdot \bar{Y}$ (VI)
 I-III : $\bar{Y} \cdot \bar{Z}$ (VII)
 I-V : $\bar{X} \cdot \bar{Z}$ (VIII)
 III-IV : $X \cdot Z$ (IX)
 IV-V : $Y \cdot \bar{Z}$ (X)
 VII-X : Z
 VIII-IX : \bar{Z}

Импликантная матрица.

	$\bar{X} \cdot \bar{Y} \cdot \bar{Z}$	$\bar{X} \cdot \bar{Y} \cdot Z$	$X \cdot \bar{Y} \cdot \bar{Z}$	$X \cdot Y \cdot \bar{Z}$	$\bar{X} \cdot Y \cdot \bar{Z}$
$\bar{X} \cdot \bar{Y}$	+	+			
\bar{Z}	+		+	+	+

$C_{\kappa} \text{ДНФ}(f) = \bar{X} \cdot \bar{Y} \vee \bar{Z} = \text{МДНФ}.$

Пример 2:

$X \cdot Y \cdot Z \vee X \cdot Y \cdot \bar{Z} \vee X \cdot \bar{Y} \cdot \bar{Z} \vee \bar{X} \cdot Y \cdot Z \vee \bar{X} \cdot \bar{Y} \cdot Z \vee \bar{X} \cdot \bar{Y} \cdot \bar{Z}$
 1 2 3 4 5 6

1-2 : $X \cdot Y$ $C_{\kappa} \text{ДНФ} = XY \vee Y \cdot Z \vee \bar{X} \cdot \bar{Z} \vee \bar{Y} \cdot Z \vee X \cdot \bar{Z} \vee \bar{X} \cdot Y$
 1-4 : $Y \cdot Z$
 2-3 : $X \cdot \bar{Z}$
 3-6 : $\bar{Y} \cdot \bar{Z}$
 4-5 : $X \cdot Z$
 5-6 : $X \cdot Y$

Импликантная матрица.

	$X \cdot Y \cdot Z$	$X \cdot Y \cdot \bar{Z}$	$X \cdot \bar{Y} \cdot \bar{Z}$	$\bar{X} \cdot Y \cdot Z$	$\bar{X} \cdot \bar{Y} \cdot Z$	$\bar{X} \cdot \bar{Y} \cdot \bar{Z}$
$X \cdot Y$	* +	* +				
$Y \cdot Z$	# +			# +		
$X \cdot \bar{Z}$		# +	# +			
$\bar{Y} \cdot \bar{Z}$			* +			* +
$\bar{X} \cdot Z$				* +	* +	
$\bar{X} \cdot \bar{Y}$					# +	# +

$$\text{МДНФ}_1 = X \cdot Y \vee \bar{Y} \cdot \bar{Z} \vee \bar{X} \cdot Z$$

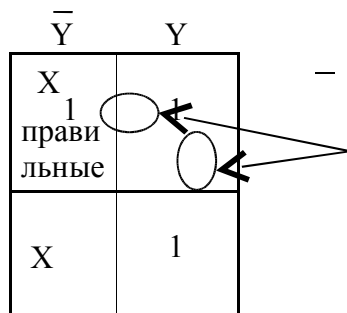
$$\text{МДНФ}_2 = Y \cdot Z \vee \bar{X} \cdot \bar{Y} \vee X \cdot \bar{Z}$$

2.1.7. Минимизация с помощью карт Вейча

Смысл минимизации состоит в том, что специальным образом размечаются *карты*, где каждая клеточка – возможная комбинация значений аргументов. В эту карту заносятся единицы, соответствующие конstituентам единицы минимизируемой функции. А затем выделяются максимальные правильные **подкубы**, что соответствует операциям склеивания и поглощения.

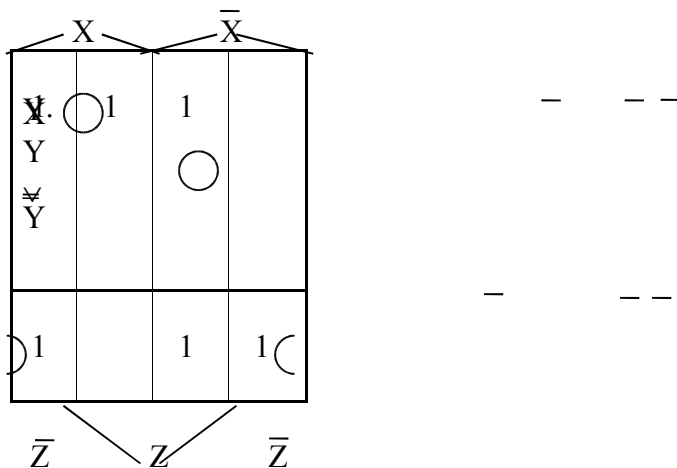
Примеры.

Пусть дана СДНФ импликации: $\bar{X}\bar{Y} \vee \bar{X}Y \vee XY$

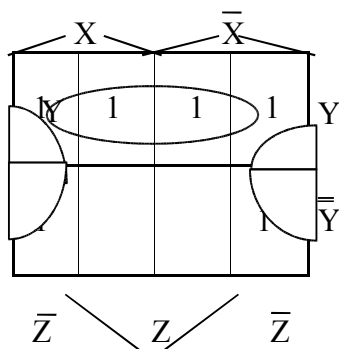


МДНФ для импликации, в соответствии с двумя выделенными подкубами, будет:
 $\bar{X} \vee Y$

Для СДНФ $XYZ \vee XY\bar{Z} \vee \bar{X}YZ \vee \bar{X}\bar{Y}Z \vee \bar{X}\bar{Y}\bar{Z}$



Для СДНФ $XYZ \vee XY\bar{Z} \vee \bar{X}YZ \vee \bar{X}\bar{Y}Z \vee \bar{X}\bar{Y}\bar{Z}$



2.1.8. Функциональная полнота

Совокупность логических операций *функционально полна*, когда какие-либо из операций совокупности обладают нижеперечисленными свойствами:

1. Несохранение 0 ($f(0, 0, \dots, 0) = 1$)
2. Несохранение 1 ($f(1, 1, \dots, 1) = 0$)
3. Не самодвойственность.

$$f(\overline{X_1}, \overline{X_2}, \dots, \overline{X_n}) \neq f(X_1, X_2, \dots, X_n)$$

4. Немонотонность.

$$\alpha_1 \cdot \alpha_2 \cdot \dots \cdot \alpha_n \geq \beta_1 \cdot \beta_2 \cdot \dots \cdot \beta_n$$

$$f(\alpha_1, \alpha_2, \dots, \alpha_n) < f(\beta_1, \beta_2, \dots, \beta_n)$$

5. Нелинейность.

Функция называется нелинейной, если она не может быть представлена в виде :

$$a_0 \oplus a_1 X_1 \oplus a_2 X_2 \oplus \dots,$$

где $a_i = 1$ или 0

Примеры линейных функций:

$$1 \oplus X = \overline{X}$$

$$a_0 = 1$$

$$a_1 = 1$$

$$a_{2..n} = 0$$

$X \oplus Y$ - неравнозначность.

$$a_0 = 0$$

$$a_1 = 1$$

$$a_2 = 1$$

$$a_{3..n} = 0$$

Функционально полные наборы создают, например:

\neg и $\&$; \neg и \vee ; \neg и \rightarrow . Операции штрих Шеффера $|$ и стрелка Пира \downarrow каждая в отдельности образуют функционально полный набор.

2.2. Логика предикатов

Предикат - логическая функция, аргументы которой могут принимать значения из некоторой предметной функции, а сама функция может принимать значение истина либо ложь.

Если переменная одна, то предикат одноместный, две - двухместный и т.д.

Нульместный предикат, то есть предикат, не содержащий переменных - высказывание.

Операции:

Из элементарных (атомарных) предикатов с помощью логических операций можно получить сложные предикаты.

Здесь уместно сделать важное содержательное замечание:

Язык предикатов - наиболее приближенный к естественным языкам формальный математический (логический) язык.

В логике предикатов к операциям, имеющим место в логике высказываний, добавляются операции навешивания кванторов.

\forall - *квантор общности*. $\forall x P(x)$ - "для всех x - $P(x)$ ".

\exists - *квантор существования*. $\exists x P(x)$ - "есть такие x , что $P(x)$ ".

($\exists!$ или \exists_1 - существует и притом единственный).

Кванторы связывают соответствующие переменные. Связанные переменные можно воспринимать как константы, а несвязанные переменные - *свободные переменные* - как собственно переменные.

Содержательные примеры предикатов :

$R(x)$ - x любит кашу (одноместный предикат).

$\forall x R(x)$ - все любят кашу (нульместный предикат - высказывание).

$\exists x R(x)$ - некоторые (есть такие) x любят кашу.

$L(x, y)$ - x любит y (двухместный предикат).

$\exists x \forall y L(x, y)$ - Существует x , который любит всех y .

$\forall x (C(x) \rightarrow O(x))$ - Все студенты $C(x)$ отличники $O(x)$.

$\exists x (C(x) \& O(x))$ - Некоторые студенты $C(x)$ отличники $O(x)$.

Здесь есть повод поразмышлять об использовании операций \rightarrow и $\&$ в двух последних высказываниях.

Для конечных областей можно операции навешивания кванторов выразить через конъюнкцию и дизъюнкцию:

Пусть $x \in \{a_1, a_2, \dots, a_n\}$

$\forall x P(x) = P(a_1) \& P(a_2) \& \dots \& P(a_n)$.

$\exists x P(x) = P(a_1) \vee P(a_2) \vee \dots \vee P(a_n)$.

2.2.1. Основные равносильности для предикатов

Для нас имеют смысл и значение только *интерпретированные* предикаты. То есть предикаты, которым поставлены в соответствии некоторые отношения (одномерным предикатам – свойства). В результате, предикаты дают некоторые содержательные высказывания относительно объектов рассматриваемых областей. Если соответствующее высказывание истинно, то говорят, что оно *выполняется в данной интерпретации*.

Предикат называется *общезначимым*, если он истинен в любой интерпретации.

1. $\neg \forall x P(x) \equiv \exists x \neg P(x)$

2. $\neg \exists x P(x) \equiv \forall x \neg P(x)$

3. $\neg \forall x \neg P(x) \equiv \exists x P(x)$

4. $\neg \exists x \neg P(x) \equiv \forall x P(x)$

5. $\forall x P(x) \vee Q$ (предикат Q не зависит от x .)

6. $\forall x P(x) \& Q \equiv \forall x (P(x) \& Q)$
7. $\exists x P(x) \vee Q \equiv \exists x (P(x) \vee Q)$
8. $\exists x P(x) \& Q \equiv \exists x (P(x) \& Q)$
9. $\forall x Q \equiv Q$
10. $\exists x Q \equiv Q$
11. $\forall x P(x) \& \forall x R(x) \equiv \forall x (P(x) \& R(x))$
12. $\exists x P(x) \vee \exists x R(x) \equiv \exists x (P(x) \vee R(x))$
13. $\forall x P(x) \vee \forall x R(x) \rightarrow \forall x (P(x) \vee R(x))$
14. $\exists x (P(x) \& R(x)) \rightarrow \exists x P(x) \& \exists x R(x)$
15. $\forall x P(x) \equiv \forall y P(y)$ (x, y - из одной предметной области)
16. $\exists x P(x) \equiv \exists y P(y)$
17. $\forall x \exists y P(x, y) \not\equiv \exists x \forall y P(x, y)$
18. $\forall x \forall y P(x, y) \equiv \forall x \forall y P(x, y)$
19. $\exists x \exists y P(x, y) \equiv \exists x \exists y P(x, y)$

2.2.2. Получение дизъюнктов

Важное замечание. Рассматриваем только **замкнутые предикаты**, то есть предикаты, **не содержащие** свободных вхождений переменных.

В общем случае необходимо пройти три этапа:

1. Получить предваренную нормальную форму.
2. Произвести сколемизацию.
3. Получить дизъюнкты.

Предваренная нормальная форма - такая форма представления предиката, когда все кванторы вынесены в начало за скобки (кванторная приставка), а в скобках есть только операции дизъюнкции, конъюнкции и отрицания. При этом символы отрицания, если таковые имеются, стоят непосредственно перед символами предикатов.

Сколемизация (от фамилии математика - Skölem) позволяет получать запись замкнутого предиката в форме без кванторов.

Избавляемся от кванторов существования:

- 1) Если левее нет кванторов общности, то соответствующая переменная заменяется константой сколема;
- 2) Иначе переменная заменяется функцией сколема от переменных, на которые навешаны кванторы общности, стоящие левее данного квантора существования.

После чего кванторы общности просто отбрасываются.

Пример:

$$\begin{aligned}
 & \neg \forall x (\forall y P(x, y) \vee \neg \exists z R(z) \rightarrow Q(x) \& \forall y M(x, y)) \equiv \\
 & \equiv \neg \forall x (\exists y \neg P(x, y) \& \exists z R(z) \vee Q(x) \& \forall y M(x, y)) \equiv \\
 & \equiv \exists x ((\forall y P(x, y) \vee \forall z \neg R(z)) \& (\neg Q(x) \vee \exists y \neg M(x, y))) \equiv \\
 & \equiv \exists x (\forall yz (P(x, y) \vee \neg R(z)) \& \exists y (\neg Q(x) \vee \neg M(x, y))) \equiv \\
 & \equiv \exists x \forall y \forall z \exists h ((P(x, y) \vee \neg R(z)) \& (\neg Q(x) \vee \neg M(x, h))) \equiv \\
 & \equiv (P(a^c, y) \vee \neg R(z)) \& (\neg Q(a^c) \vee \neg M(a^c, f(y, z)))
 \end{aligned}$$

Каждая элементарная дизъюнкция в полученном выражении является дизъюнктом.

2.3. Аксиоматические теории

2.3.1. Аксиоматическая теория исчисления высказываний

Для того чтобы задать аксиоматическую теорию необходимо задать язык, аксиомы и правила вывода данной теории.

1. Язык:

а) Символы теории, это

- буквы (для определенности, заглавные латинские): A, B, C, \dots, Z
- специальные символы: $(,), \rightarrow, \neg$

б) Последовательности символов образуют выражения.

Например, выражениями будут $AB \neg \rightarrow (B \neg$ или другое, более приятное глазу,

$(A \rightarrow B) \rightarrow (\neg B)$

Формулами будем называть выражения, задаваемые индуктивно следующим образом:

а) Любая буква ($A \dots Z$) есть формула.

б) Если A, B - формулы, то (A) , $\neg A$, $A \rightarrow B$ - также формулы.

2. Аксиомы зададим тремя схемами аксиом:

$$A \rightarrow (A \rightarrow B)$$

$$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

$$(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$$

В схемы аксиом вместо A, B, C могут быть подставлены любые формулы. В результате конкретных подстановок на основе схем аксиом будут появляться конкретные аксиомы.

3. Правила вывода: В данной конкретной версии аксиоматической теории используется всего одно (но самое известное) правило вывода *modus ponens*

(*модус утверждающий*) или кратко - **mp**. Это правило, учитывая особенность его работы, еще называют *правилом отсечения*.

$$A, A \rightarrow B \mid - B$$

Символ $\mid -$ читается как "выводимо". То есть в данной теории из формул

A и $A \rightarrow B$ выводима формула B или формула B есть *теорема* данной теории.

Выводом (в данной теории) называется последовательность формул $\Phi_1, \Phi_2, \dots, \Phi_n$, где каждая следующая формула является аксиомой, или следует по правилу вывода из предыдущих. Последняя формула вывода называется *теоремой*.

Важное замечание. При описании теории, в том числе и ее языка, использовались средства, не принадлежащие определяемому (целевому) языку: запятые, точки, слова русского языка и т.д. Совокупность средств, используемых при описании целевого языка, называется **метаязыком**.

Пример:

Лемма: $\mid - A \rightarrow A$

Ф1: Возьмем схему аксиом 2 и подставим $A = A, C = A, B = A \rightarrow A$, в результате получим:

$$(A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$$

Ф2 : Из схемы аксиом 1, при $A = A, B = A \rightarrow A$, получим :

$$(A \rightarrow ((A \rightarrow A) \rightarrow A))$$

из Ф1, Ф2 по **m.p.** получаем Ф3: $(A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)$

Ф4 : Из схемы аксиом 1, при $A = A, B = A$, получим:

$(A \rightarrow (A \rightarrow A))$

из Ф3, Ф4 по **м.р.** получаем Ф5: $A \rightarrow A$

2.3.2. Непротиворечивость и полнота аксиоматической теории исчисления высказываний

Нет ничего проще создания аксиоматических теорий! Как сказал один известный математик: "Аксиоматизация сродни воровству!".

Определив свой язык, придумав свои аксиомы и правила вывода, вы получаете свою аксиоматическую теорию.

Например, в качестве языка возьмем любые последовательности символов @, единственной аксиомой объявим один символ @, а правило вывода будет

@ |— @@.

Тогда в данной теории будет выводима любая последовательность из одного или более символов @.

Одно плохо, толку в таких теориях обычно никакого нет...

А вот рассмотренная ранее аксиоматическая теория исчисления высказываний имеет ряд важных (интересных, замечательных) свойств. Формулы этой теории можно интерпретировать как формулы алгебры высказываний, записанные с использованием (*функционально полного набора!*) операций: \neg и \rightarrow (отрицания и импликации).

Для этой теории доказано, что она **полна**. То есть в этой теории могут быть выведены все тавтологии логики высказываний (которые могут быть записаны с помощью \neg и \rightarrow).

Более того, данная **теория непротиворечива**. То есть в этой теории не могут быть выведены какая-то формула Φ и ее отрицание ($\neg\Phi$).

Докажем непротиворечивость этой теории.

Прямой проверкой доказываем, что все аксиомы, получаемые из схем аксиом, являются тавтологиями. Например, для первой схемы аксиом:

$A \rightarrow (B \rightarrow A)$

A	B	Φ
0	0	1
0	1	1
1	0	1
1	1	1

А из тавтологий с помощью **м.р.** ($A, A \rightarrow B \mid\text{—} B$) можно получить только тавтологии. А поскольку любая полученная в этой теории формула Φ есть тавтология, то ее отрицание $\neg\Phi$ было бы противоречием, которое не выводимо.

Полнота и непротиворечивость очень важные свойства. Увы, большинство более сложных аксиоматических теорий не может похвастаться полнотой (открытый Геделем принцип неполноты). В них могут существовать формулы, для которых невозможно доказать как выводимость, так и невыводимость...

Что же касается непротиворечивости, то это очень жесткое требование.

Стоит допустить в теории возможность хотя бы одного противоречия (для одной формулы Φ допустить возможность вывода и $\neg\Phi$), как теория становится бессмысленной, так как тогда в ней можно вывести любую формулу. (Из ложной посылки может следовать что угодно).

2.4. Аксиоматические теории первого порядка

Зададим аксиоматическую теорию, используя расширенный язык предикатов:

1. Язык :

- 1). Символы: *служебные*: $\rightarrow, \neg, (,), \forall, \exists$
предметные переменные: z, y, x, \dots
вещественные переменные: a, b, c, \dots
функциональные символы: f, g, h, \dots
символы предикатов: P, Q, R, \dots

2). Терм:

Константа или переменная есть **терм**.

Если t_1, t_2, \dots, t_n - термы, то $f(t_1, t_2, \dots, t_n)$ - тоже терм.

3). Формула:

Если t_1, t_2, \dots, t_n - термы, то $P(t_1, t_2, \dots, t_n)$ - формула.

Если P, Q - формулы, то

$(P), P \rightarrow Q, \neg P, \forall x P, \exists x P$ - также формулы.

2. Аксиомы:

1)-3) - соответствуют схемам аксиом логики высказываний.

4) $\forall x A(x) \rightarrow A(t)$

5) $A(t) \rightarrow \exists x A(x)$

где терм t свободен для x .

Терм t называется **свободным для переменной x** , если никакое свободное вхождение x в A не лежит в области действия никакого квантора $\forall y$, где y переменная, входящая в t .

Например, терм y свободен для переменной x в $A(x)$, но не свободен для переменной x в $\forall y A(x)$. Терм $f(x, z)$ свободен для x в $\forall y A(x, y) \rightarrow B(x)$, но не свободен для x в $\exists z \forall y A(x, y) \rightarrow B(x)$.

3. Правила вывода:

1). $A, A \rightarrow B \mid \text{---} B$ (**m.p.**)

2) $B \rightarrow A(x) \mid \text{---} B \rightarrow \forall x A(x)$

3) $A(x) \rightarrow B \mid \text{---} \exists x A(x) \rightarrow B$

Вышеприведенное исчисление называют еще **исчислением предикатов**.

На практике, как правило, к этим аксиомам, называемым *логическими аксиомами*

(коль скоро они описывают логическую составляющую рассматриваемого "мира"), добавляют еще аксиомы, описывающие конкретную "предметную область". Например, законы управления автоматическим регулятором или роботом.

Такие аксиомы называются **собственными аксиомами теории**, а сами теории - **аксиоматическими теориями первого порядка** или короче, **теориями первого порядка**.

Теории первого порядка неполны, но, как правило, непротиворечивы. (Хотя специалистов по искусственному интеллекту больше интересуют "локально-противоречивые системы". Однако, чем больше такой интерес, тем дальше они отходят от "классической математической логики" со всеми вытекающими последствиями).

Говоря о теориях первого порядка нельзя хотя бы не намекнуть на существование теорий более высоких порядков. Так, например, формула $\forall P \forall x P(x)$ — уже не принадлежит к языку теорий первого порядка из-за квантора $\forall P$.

2.5. Метод резолюций

Метод предложен Дж. Робинсоном в 1965 году.

Метод резолюций - аксиоматическая теория первого порядка, которая использует доказательство от противного, и, следовательно, не использует аксиоматику исчисления предикатов (которая находится в области тождественно истинных формул).

1. Язык метода резолюции - язык дизъюнктов :
2. Аксиомы только *собственные*.
3. Правило вывода - *резолюция*

Важное замечание. Доказательство корректности метода резолюции Дж. Робинсон выполнил с привлечением теории моделей, раздела математической логики, который в данном курсе не рассматривается. Поэтому мы воспользуемся "правдоподобными" рассуждениями, которыми изобиловали первые книги по языку Пролог (Prolog). А язык Пролог (Программирование с помощью Логик) - язык , основной представитель класс языков *логического программирования*, базируется как раз на методе резолюции!

Правило вывода резолюция использует расширенный принцип силлогизма и унификацию.

Традиционный силлогизм: $A \rightarrow B, B \rightarrow C \mid \text{---} A \rightarrow C$

Применительно к дизъюнктивной записи можно представить как

$\neg A \vee B$		$\neg A \vee B \vee \neg D$
$\neg B \vee C$	или "обобщенный" вариант	$\neg B \vee C \vee E$
<hr/>		<hr/>
$\neg A \vee C$		$\neg A \vee C \vee \neg D \vee E$

Унификация:

Унификация также не противоречит здравому смыслу. Она позволяет заменить переменную x на терм t . То есть вместо переменной могут быть подставлены константа или другая переменная (из той же области), или функция, область значений которой совпадает с областью определения x .

$$\begin{array}{c} a(\text{const}) \rightarrow x \leftarrow y (\text{из той же области}) \\ \uparrow \\ f(z) \end{array}$$

Вывод здесь заключается в том, что в систему добавляется отрицание формулы (дизъюнкта!), которую необходимо вывести. Вывод состоит в последовательном применении резолюции до получения пустого дизъюнкта . Это будет, с точки зрения интерпретации, означать, что не существует никакой модели («мира»), в которой бы была справедлива исходная система законов (дизъюнктов). А коль скоро доказательство выполняется методом от противного, то значит первоначальная формула (дизъюнкт) действительно выводима (и, значит, справедлива) в данной теории.

Пример 1 : Можно сказать, что это прообраз или предельно упрощенный вариант «системы искусственного интеллекта».

Пусть мир описывается двумя аксиомами:

Миша повсюду ходит за Леной $A1. \forall x (B(L, x) \rightarrow B(M, x))$

Лена в школе $A2. B(L, Ш)$

Требуется доказать (ответить на вопрос)

Где Миша?

A3. $\exists x V(M, x)$?

Вопрос (доказываемую формулу с добавленным знаком вопроса) $\exists x V(M, x)$? преобразуем в $\neg \exists x V(M, x)$ (отрицание вопроса). Далее задвигаем отрицание за квантор, производим сколемизацию и добавляем специальный «предикат ответа», который будет аккумулировать процесс унификации). В результате получаем дизъюнкт:

$\neg V(M, x) \vee \text{Отв}(M, x)$

Вся система (две аксиомы и вопрос) будет состоять из трех дизъюнктов:

Д1: $\neg V(J, x) \vee V(M, x)$

Д2: $V(J, Ш)$

Д3: $\neg V(M, x) \vee \text{Отв}(M, x)$

Вывод:

Резолюция Д1-Д2 дает Д4: $V(M, Ш)$

Резолюция Д4-Д3 дает Д5: $\text{Отв}(M, Ш)$

То есть, предикат ответа (при получении пустого дизъюнкта) можно интерпретировать как «Миша в школе». (Интерпретация ответа в системе искусственного интеллекта остается за человеком).

Пример 2:

1. Если x является родителем y и y является родителем z , то x является прародителем z .

A1. $\forall x y z (P(x, y) \& P(y, z) \rightarrow \Pi(x, z))$

2. Каждый человек имеет своего родителя.

A2. $\forall y \exists x P(x, y)$

3. Существуют ли такие x и y , что x является прародителем y ?

$\exists x y \Pi(x, y)$?

Преобразуем аксиомы в дизъюнкты.

Д1. $\neg P(x, y) \vee \neg P(y, z) \vee \Pi(x, z)$

Д2. $P(f(y), y)$

Д3. $\neg \Pi(x, y) \vee \text{Отв}(x, y)$

Д1 - Д2: Д4: $\neg P(y, z) \vee \Pi(f(y), z)$

Д4 - Д2: Д5: $\Pi(f(f(y)), y)$

Д5 - Д3: Д6: $\vee \Pi(f(f(x)), x)$

Заметим, что каждая переменная имеет уникальное имя в пределах одного дизъюнкта. Переменные, названные одинаково в разных дизъюнктах - это разные переменные. Интерпретация результата лежит на человеке. Будем интерпретировать

$f(x)$ - как *быть родителем* x . То есть $f(f(x))$ - *родитель родителя* x . Следовательно, $\Pi(f(f(x)), x)$ - *прародитель* x - это *родитель родителя* x .

2.6. Система Генцена

В ее основе лежит понятие **секвенции**.

Секвенции имеют вид

антецедент - $A_1, A_2, \dots A_n \mid \vdash B_1, B_2, \dots B_n$ - сукцедент

↑

знак секвенции

Содержательно это равносильно выражению:

$$A_1 \& A_2 \& \dots \& A_n \rightarrow B_1 \vee B_2 \vee \dots \vee B_n$$

Аксиома (схема аксиом) в системе Генцена единственная и она имеет вид:

$$A \mid \text{---} A$$

Правила вывода:

(Из секвенций над чертой выводимы секвенции под чертой, а Γ обозначает какое-то множество формул).

$$\begin{array}{l} 1) \quad \frac{A, \Gamma \mid \text{---} B}{\Gamma \mid \text{---} A \rightarrow B} \quad 1)' \quad \frac{\Gamma \mid \text{---} A; \Gamma \mid \text{---} A \rightarrow B}{\Gamma \mid \text{---} B} \end{array}$$

$$\begin{array}{l} 2) \quad \frac{\Gamma \mid \text{---} A; \Gamma \mid \text{---} B}{\Gamma \mid \text{---} A \& B} \quad 2)' \quad \frac{\Gamma \mid \text{---} A \& B}{\Gamma \mid \text{---} A} \end{array}$$

$$\begin{array}{l} 3) \quad \frac{\Gamma \mid \text{---} A}{\Gamma \mid \text{---} A \vee B} \quad 3)' \quad \frac{\Gamma, A \mid \text{---} B; \Gamma, C \mid \text{---} B; \Gamma \mid \text{---} A \vee B}{\Gamma \mid \text{---} B} \end{array}$$

$$\begin{array}{l} 4) \quad \frac{\Gamma, A \mid \text{---}}{\Gamma \mid \text{---} \neg A} \quad 4)' \quad \frac{\Gamma \mid \text{---} A; \Gamma \mid \text{---} \neg A}{\Gamma \mid \text{---}} \end{array}$$

$$5) \quad \frac{\Gamma, A, B \mid \text{---} C}{\Gamma, B, A \mid \text{---} C}$$

$$\begin{array}{l} 6) \quad \frac{A, A \mid \text{---} B}{A \mid \text{---} B} \quad 6)' \quad \frac{A \mid \text{---} B, B}{A \mid \text{---} B} \end{array}$$

$$\begin{array}{l} 7) \quad \frac{\Gamma \mid \text{---} B}{\Gamma, A \mid \text{---} B} \quad 7)' \quad \frac{\Gamma \mid \text{---} A}{\Gamma \mid \text{---} A, B} \end{array}$$

Докажем $\mid \text{---} A \rightarrow A$:

- 1) Из первой аксиомы, при $\Gamma = \emptyset$ и $B = A$:
- $$A \mid \text{---} A$$

$$\frac{}{\vdash A \rightarrow A}$$

Теорема доказана.

Докажем $\vdash \neg A \vee A$

$$\frac{\frac{\frac{}{\vdash A}}{\vdash \neg A, A}}{\vdash \neg A \vee A, \neg A}}{\vdash \neg A \vee A}$$

2.7. Система Аристотеля

Древнейшей аксиоматической системой является система Аристотеля. Она не может быть полностью интерпретирована с помощью логики предикатов. Тому ряд причин и одна из существенных – то, что при интерпретации сущностей аристотелевой логики могут использоваться только *непустые множества*.

В связи с этим прямой перевод на язык предикатов может приводить к парадоксальным ситуациям. Например,

пусть $P(x)$ - x выше двух метров

На множестве людей имеет место: $\forall x P(x) = 0, \exists x P(x) = 1$.

Но на множестве марсиан $\forall x P(x) = 1, \exists x P(x) = 0$.

т.е. $\forall x P(x) \not\rightarrow \exists x P(x)$

Рассуждения в аристотелевой логике базируются на том, что если некоторые высказывания верны, то и некоторое новое предложение обязано быть верным в силу правильности логической конструкции (силлогизма).

Пример.

Интерпретация множествами:



То есть из «Все животные смертны» и «Все люди – животные» следует «Все люди смертны» или

$Ж \rightarrow С, Л \rightarrow Ж \vdash Л \rightarrow С$

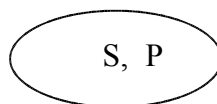
Категорические высказывания.

Имеется четыре типа так называемых **категорических высказываний**.

1) Общеутвердительные A_{sp} (A_{xy}):

Всякое S есть P .

Аналог на языке предикатов $\forall x (S(x) \rightarrow P(x))$



$S \cap \bar{P} = 0$ - интерпретация на множествах

2) Общеотрицательные Esp (Exy):

Не одно S не есть P.

Аналог на языке предикатов $\forall x (S(x) \rightarrow \neg P(x))$

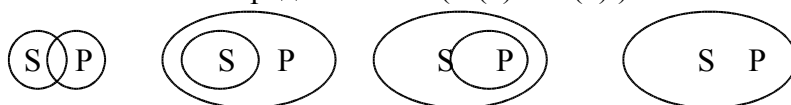


$S \cap P = 0$ - интерпретация на множествах

3) Частично-утвердительные Isp (Ixy):

Некоторые S есть P.

Аналог на языке предикатов $\exists x (S(x) \& P(x))$



$S \cap P \neq 0$ - интерпретация на множествах

4) Частное отрицание Osp (Oxy)

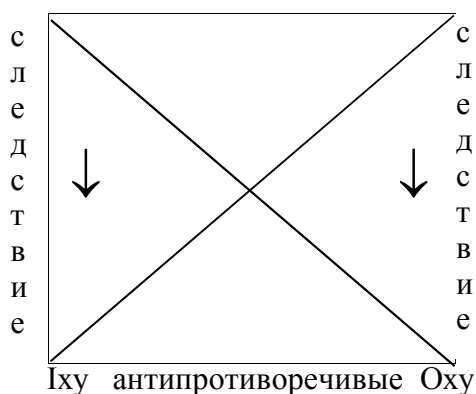
Некоторые S не есть P.

Аналог на языке предикатов $\exists x (S(x) \& \neg P(x))$

$S \cap \bar{P} \neq 0$ - интерпретация на множествах

Соотношения высказываний можно представить в виде **логического квадрата**.

Axy противоречивые Exy



Модус - структура умозаключения, которая определяет его истинность.

Модусы непосредственного заключения

Всего таких модусов 32. Вот некоторые из них.

Axy \rightarrow Axy истинно Axy \rightarrow Ayx ложно

Axy \rightarrow Exy ложно

Axy \rightarrow Ixy истинно

Противоположные

↙

$Axy \rightarrow Oxy$ ложно

$Exy \rightarrow Oxy$ истинно $Exy \rightarrow Oyx$ истинно
 $Oxy \rightarrow Oxy$ истинно $Oxy \rightarrow Oyx$ ложно

Категорические силлогизмы.

Всего категорических силлогизмов - 256.

$Axy \cdot Azy \rightarrow Azx$

$Exy \cdot Ayz \rightarrow Ozx$

.....

2.8. Примеры неклассических логик

Различных видов логик уже создано очень много, начиная с древнеиндийской логики *Навья-Ньяя* и вышеупомянутой системы Аристотеля. Всякая логика ограничена. Невозможно создать универсальную логику, исчерпывающую все возможные потребности. Поэтому и создаются все новые логики.

Одна из наиболее популярных неклассических логик последние двадцать лет - это **нечеткая (fuzzy) логика**. Нечеткая математика базируется на нечетком отношении принадлежности \in . Например: *Доцент Сидоров к множеству лысых, можно сказать, практически и не принадлежит!*

А также на понятии лингвистической переменной. Например, **лингвистическая переменная** *возраст* может иметь **лингвистические значения**: *очень молодой, молодой, средних лет, пожилой, старый, ...*

Рассуждения в нечеткой логике могут быть типа:

*Если **немного** добавить соли, то будет **гораздо** вкуснее.*

Разумеется, для машинной обработки необходимо отобразить эти нечеткие понятия на "числовые оси", что осуществляется с помощью подбираемых функций принадлежности μ

В нечеткой логике не выполняются закон исключенного третьего и закон противоречия.

Модальные логики.

Модальность - дополнительная характеристика, приписываемая высказыванию.

Пусть A - высказывание.

A - необходимо A .

$\Diamond A$ - возможно A .

Если. $A = 0$, то $A = 0$

Если $A = 1$, то $\Diamond A = 1$.

Но если $A = 1^*$, то A может быть истинно или ложно.

Скажем, «Вася ловит рыбу» - истинно, но «Необходимо, что Вася ловит рыбу» – ложно, поскольку Вася это делает только по настроению.

Например, «Летом выпал снег» – может быть ложным высказыванием, а «Возможен случай, что летом выпадет снег» - истинным.

Но если

$$A = 0, \text{ то } \Diamond A = \begin{cases} 0, & \text{если противоречит (физическим) законам.} \\ 1, & \text{иначе.} \end{cases}$$

Например, « $2 + 2 = 5$ » - ложно и «возможно, что $2 + 2 = 5$ » – также ложно, но

«Вася стреляет» – ложно, но «Вася может и пострелять» – истинно (особенно, когда деньги кончились, а курить охота).

Существует достаточно большое количество разновидностей модальных логик. Некоторые возможные соотношения в модальных логиках:

$$\begin{aligned}\neg P &= \Diamond \neg P & P \vee \neg P &= 1 \\ \neg \Diamond P &= \neg P & \Diamond P \& \neg \Diamond P &= 1 \\ \neg \neg P &= \Diamond P \\ \neg \Diamond \neg P &= P\end{aligned}$$

Некоторые возможные аксиомы:

$$A \rightarrow A$$

$$A \rightarrow A$$

$$(A \rightarrow B) \rightarrow (A \rightarrow B) \dots$$

Немонотонные логики. Кратко суть таких логик формулируется следующим образом: добавление в систему новых аксиом может привести к изменению уже существовавших... Они хороши тем, что часто следуют принципу: *Если не нравится полученный в процессе вывода результат - можно изменить исходные посылки.*

Существует много весьма разноплановых немонотонных логик. Например, вывод

$$C(x, y) \& \Gamma(y)=z : \Diamond \Gamma(x) = z \vdash \Gamma(x) = z$$

То есть из фактов, что $C(x, y)$ - "х,у - супруги", $\Gamma(y)=z$ - "город, где проживает у. называется z". Символ « : » отделяет условие от предложения.

и $\Diamond \Gamma(x) = z$ - "х живет в городе z" не противоречит существующим аксиомам, то.

в этой системе выводимо $\Gamma(x) = z$, что "х живет в городе z".

Индуктивные логики. Это логики правдоподобных рассуждений "от частного к общему". Когда Шерлок Холмс по отдельным уликам восстанавливал картину преступления, то его *дедуктивный метод* был чистой воды *индуктивным методом*.

Это элементарно, Ватсон!

Поскольку правдоподобные рассуждения не гарантируют стопроцентно правильность логического заключения (у Холмса результат был близок к 100%, хотя и у него бывали ошибки), то можно говорить о большей или меньшей правдоподобности результата.

Пусть $A \uparrow$ означает "А более правдоподобна", тогда можно предложить, например, такие индуктивные правила вывода:

$$A \rightarrow B, B \vdash A \uparrow$$

$$A \rightarrow B, C \rightarrow B \vdash A \sim C \uparrow$$

$$A \& B \vdash D \quad A \vdash D \uparrow$$

$$A \& C \vdash D$$

Эротетические логики. Так названы логики *вопросов и ответов*.

Правильно поставленный вопрос - это такой вопрос у которого предпосылка истинна и не противоречива.

В рамках этой логики доказана полезная для повседневной практики теорема:

На глупый вопрос нельзя дать умный прямой ответ.

Два основных типа вопросов:

уточняющие (типа *ли*),

воспроизводящие (типа *что*).

Вопросы могут быть простыми и сложными.

А ответы могут быть:

- истинные и ложные,

- прямые и косвенные,

- краткие и развернутые,
- полные и неполные.

Временные логики :

Высказывание A ,
 P_A - "было A "
 F_A - "будет A "
 $A \supset B$ - "если A , то после этого B ".

3. Теория Автоматов

3.1. Понятие автомата

Автомат - дискретный преобразователь информации, на вход которого поступают входные последовательности сигналов (входные слова). Он формирует выходные последовательности сигналов на основании своих внутренних состояний и входной последовательности сигналов.

В курсе рассматривается абстрактная теория автоматов.

Нас будет интересовать их *поведенческий аспект*. Автомат для нас – математическая модель, а не физическое устройство. Автоматы фактически позволяют реализовать логику, зависящую от времени.

Не рассматриваемая здесь *структурная теория автоматов* занимается реализацией абстрактного автомата с помощью физических сущностей, вроде элементов памяти (например, триггеров) и комбинационных (логических) схем...

Будем иметь в виду две ключевые абстракции:

1. Автомат функционирует в абстрактном времени.
2. Все переходы происходят мгновенно.

Автомат есть система шести объектов:

$$\alpha = \langle X, Y, Q, f, \varphi, q_0 \rangle$$

$X = \{x_1, \dots, x_n\}$ - конечный входной алфавит (множество входных сигналов).

$Y = \{y_1, \dots, y_m\}$ - конечный выходной алфавит (множество выходных сигналов).

$Q = \{q_0, q_1, \dots, q_k\}$ – множество состояния автомата.

Если множество конечно автомат называется *конечным*.

$f(q, x)$ - функция переходов.

$\varphi(q, x)$ - функция выходов.

$q_0 \in Q$ - начальное состояние.

Законы функционирования автоматов

$$\left. \begin{aligned} q(t) &= f(q(t-1), x(t)) \\ y(t) &= \varphi(q(t-1), x(t)) \end{aligned} \right\} \text{ Автомат I-го рода (автомат Мили)}$$

$$\left. \begin{aligned} q(t) &= f(q(t-1), x(t)) \\ y(t) &= \varphi(q(t), x(t)) \end{aligned} \right\} \text{ Автомат II-го рода}$$

$$\left. \begin{aligned} q(t) &= f(q(t-1), x(t)) \\ y(t) &= \varphi(q(t)) \end{aligned} \right\} \text{ Правильный автомат II-го рода (автомат Мура)}$$

3.2. Примеры автоматов

Замечание. Для удобства восприятия и сокращения описания будем говорить об автоматах как об автоматических роботоподобных устройствах, хотя на самом деле это, как уже было

сказано, лишь математические модели, преобразующие входные слова в выходные и не имеющие дела с физическими сущностями, вроде монет, билетов и т.п.

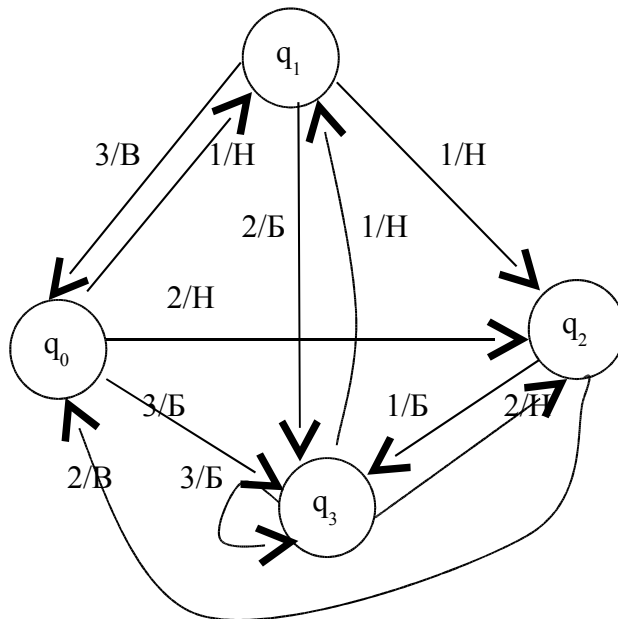
Пример 1 (автомат Мили):

Построить (синтезировать) автомат, на вход которого могут поступать в любой последовательности и, возможно, с повторениями монеты (как в добрые старые времена) 1; 2 и 3 копейки. Автомат продает билет, если сумма опущенных монет равна 3. В случае превышения суммы автомат возвращает деньги.

Входной алфавит в описании задан явно: $X = \{1, 2, 3\}$.

Выходной алфавит будет содержать буквы (сигналы): Б – выдает билет, В – возвращает деньги, Н – ничего не выдает (это такой специфический выходной сигнал). То есть $Y = \{Б, В, Н\}$.

Можно представить автомат в виде графа, где вершины представляют состояния, а к каждой стрелке приписана пара *входной сигнал/выходной сигнал*. То есть размеченные стрелки отражают функции переходов и выходов.



От представления автомата в виде графа можно очевидным образом перейти к его табличному представлению, которое также однозначно определяет автомат. Табличное представление предпочтительно для автоматов с большим числом состояний и при представлении автоматов в компьютере.

Можно построить для данного автомата таблицы

переходов

Т.П.	q ₀	q ₁	q ₂	q ₃
1 ф	q ₁	q ₂	q ₃	q ₁
2	q ₂	q ₃	q ₀	q ₂
3	q ₃	q ₀	q ₀	q ₃

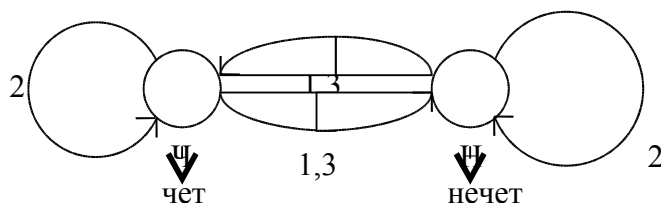
и

ВЫХОДОВ

Т.В.	q ₀	q ₁	q ₂	q ₃
1 ф	Н	Н	Б	Н
2	Н	Б	В	Н
3	Б	В	В	Б

Пример 2 (автомат Мура).

Построить автомат, на вход которого могут поступать монеты 1, 2, 3 коп. Автомат выдает сигнал “чет”, если поступившая сумма в данный момент четная и “нечет”, если наоборот.



Это автомат Мура. Поэтому выходные сигналы приписаны не стрелкам, а к состояниям, которыми они однозначно определяются. Табличное представление сводится к одной таблице – **расширенной таблице переходов**. В ней добавляется верхняя строка, позволяющая приписать выходные сигналы состояниям.

-	Чет	Нечет
В		
	Ч	Н
1	Н	Ч
2	Ч	Н
3	Н	Ч

3.3. Минимизация автоматов

Автоматы различной конфигурации могут реализовывать одну и ту же функцию. Для практических целей важно уметь находить автомат с минимальным количеством состояний, реализующий заданное поведение. Число состояний абстрактного автомата определяет при практической реализации число необходимых элементов памяти.

Кстати, первый рассмотренный автомат был (сознательно) построен избыточным. От состояния q_3 очень просто избавиться, передав его функции состоянию q_0 .

Существуют различные методы минимизации. К числу простейших относится *Метод Гилла*, позволяющий отыскивать классы эквивалентных состояний и заменять их отдельными состояниями.

Два автомата называются **эквивалентными**, если они имеют одинаковые входные и выходные алфавиты, и на одинаковые входные слова выдают одинаковые выходные слова (одинаковой длины).

Два состояния называются **1-эквивалентными**, если они не различимы с помощью одного входного сигнала (символа).

Состояния называются **k-эквивалентными**, если начиная с них неразличимы входные слова длиной в k .

Если состояния k -эквивалентны для любого k , то они называются **эквивалентными**.

Рассмотрим минимизацию методом Гилла на каком-то конкретном автомате Мили.

Т.В.	1	2	3	4	5	6
x_1	y_1	y_1	y_1	y_1	y_1	y_2
x_2	y_2	y_2	y_2	y_2	y_2	y_2

A B

Т.П.	1	2	3	4	5	6
x ₁	1/A	3/A	6/B	2/A	6/B	4/A
x ₂	2/A	1/A	3/A	2/A	5/A	5/A

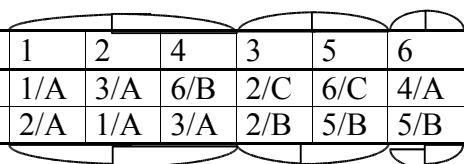
- I - эквивалентные



- II - эквивалентные

Т.П.	1	2	4	3	5	6
x ₁	1/A	3/A	6/B	2/C	6/C	4/A
x ₂	2/A	1/A	3/A	2/B	5/B	5/B

A B C



- III - эквивалентные

Т.П.	A	B	C
x ₁	A	C	A
x ₂	A	B	B

Т.В.	A	B	C
x ₁	y ₁	y ₁	y ₂
x ₂	y	y ₂	y ₂

Получен минимальный (с точностью до изоморфизма) автомат, в котором классы эквивалентных состояний заменены именами классов. Он эквивалентен (по поведению) исходному автомату, но имеет три состояния вместо шести.

Замечание. Классы, в процессе их выделения, могут дробиться, но не могут объединяться.

Ограниченность метода Гилла. В полученном автомате наглядно видно, что автомат не выйдет из начального состояния А – оно является **тупиковым**. Следовательно, автомат никогда не перейдет в состояния В и С, которые в данном случае будут **недостижимыми**.

Анализ тупиковых и недостижимых состояний может повлиять на конфигурацию минимального автомата, либо (что скорее всего) выявить ошибки в его построении.

Однако метод Гилла, как и большинство других методов минимизации, не учитывает влияния тупиковых и недостижимых состояний на результирующий автомат.

3.4. Особенности минимизации автомата Мура

Минимизация автомата Мура отличается первым шагом. Здесь в классы одно-эквивалентных попадают состояния, отмеченные одинаковыми выходными сигналами.

	A	B					
	y ₁	y ₂	y ₂	y ₂	y ₂	y ₂	y ₂
	1	2	3	4	5	6	7
x ₁	2/B	3/B	5/B	3/B	4/B	7/B	3/B
x ₂	1/A	1/A	1/A	1/A	1/A	1/A	1/A

	y_1	y_2
	A	B
x_1	B	B
x_2	A	A

3.5. Переход от автомата Мура к автомату Мили и наоборот

Автоматы Мура и Мили отличаются функцией выходов.

$y(t) = \varphi(q(t))$ – для автомата Мура

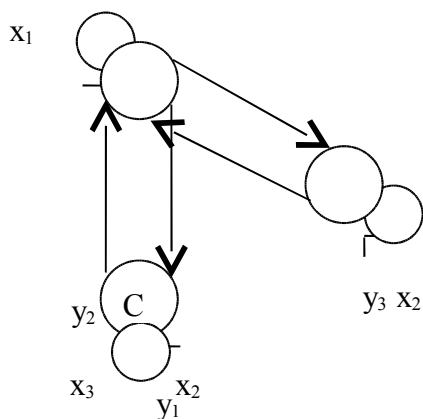
и

$y(t) = \varphi(q(t-1), x(t))$ – для автомата Мили

Переход от автомата Мура к автомату Мили заключается в построении таблицы переходов. Построение состоит в подстановке выходных сигналов, отмечающих состояния в расширенной таблице переходов, вместо состояний, в которые автомат переходит. Тем самым, если говорить в терминах графов, выходные сигналы от состояний сдвигаются на стрелки, которые в эти состояния заходят.

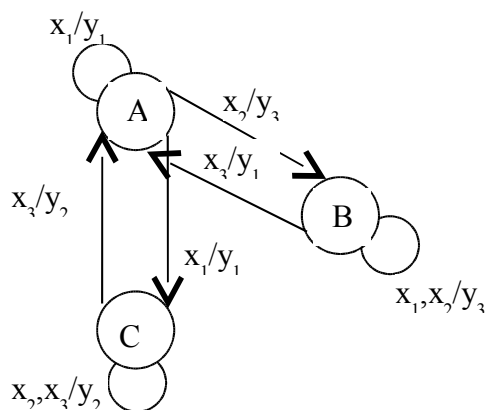
А таблица переходов автомата Мили получается из расширенной таблицы переходов автомата Мура отбрасыванием первой строки.

	y_1	y_3	y_2
x_2	A	B	C
x_1	A	B	A
x_2 x_3	B	B	C
x_3x_3	C	A	C



Т.П.	A	B	C
x_1	A	B	A
x_2	B	B	C
x_3	C	A	C

Т.В.	A	B	C
x_1	y_1	y_3	y_1
x_2	y_3	y_3	y_2
x_3	y_2	y_1	y_2



Переход от автомата Мили к автомату Мура. x_1

$q_i x_j \rightarrow$	q_1/b_0	q_2	q_3
x_1	q_1/b_{11}	q_3/b_{21}	q_2/b_{31}
x_2	q_2/b_{12}	q_1/b_{22}	q_3/b_{32}

Т.В.	q_1	q_2	q_3
x_1	y_3	y_1	y_2
x_2	y_4	y_5	y_6

		y_3	y_4	y_1	y_5	y_2	y_6
	b_0	b_{11}	b_{12}	b_{21}	b_{22}	b_{31}	b_{32}
x_1	b_{11}	b_{11}	b_{21}	b_{31}	b_{11}	b_{21}	b_{31}
x_2	b_{12}	b_{12}	b_{22}	b_{32}	b_{12}	b_{22}	b_{32}

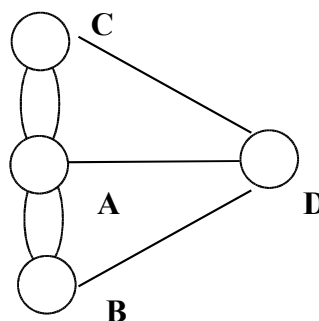
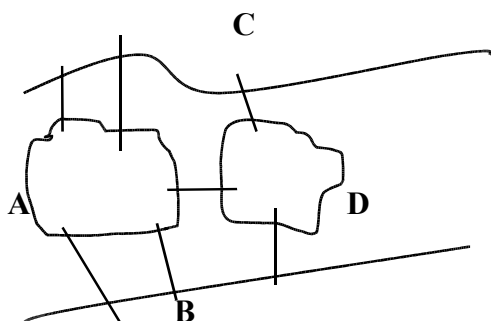
Теорема : (Глушкова)

Таким образом доказана конструктивная теорема, что для произвольного автомата Милли может быть построен эквивалентный ему автомат Мура имеющий не более $n * m + 1$ состояний, где n - число входных сигналов, m - число состояний исходного автомата Милли.

4.Теория графов

4.1. Понятие графа

Начало теории графов часто ведут от 1736 года и связывают с решением Эйлером знаменитой задачи о Кенигсбергских мостах.

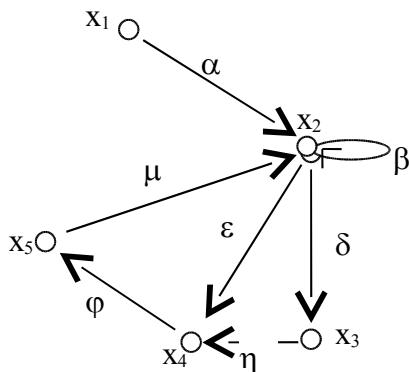


Жителям в те далекие времена, чтобы придать воскресному гулянию осмысленность, предлагалось выйдя из дома (на любом участке суши (A, B, C или D) пройти по всем мостам строго по одному разу и вернуться домой....

На втором рисунке этот корявый план нарисован в виде графа.

Следует отметить некоторые практические особенности теории графов. Слово *граф* однокоренное со словом *графика*. Поэтому не удивительно, что многие задачи теории графов представляются в виде специального рисунка – графа. Однако, это, как правило, возможно только для простейших вариантов задач. Рисовать графы для задач с сотнями вершин и тысячами дуг, если и возможно, то бессмысленно. Теряется главное преимущество рисунка – наглядность. Кроме того, сегодня при решении задач теории графов широко используется вычислительная техника, а для нее - решение задачи, заданной рисунком – одно из самых неудобных представлений, какие можно придумать. А наглядность компьютер понимает по-своему :-|

Граф G задается как совокупность двух сущностей: *множества вершин* X и множества соединений – *множества дуг или ребер*. $G = \langle \Gamma, X \rangle$,
Графически это может выглядеть следующим образом:



Традиционная «аналитическая» запись для этого рисунка будет:

$$\begin{array}{l|l} \Gamma_{x1} = \{x2\} & \Gamma_{x4} = \{x3, x3\} \\ \Gamma_{x2} = \{x2, x3, x4\} & \Gamma_{x5} = \{x2\} \\ \Gamma_{x3} = \emptyset & \end{array}$$

Другой способ задания графа - с помощью *матрицы инциденций*.

	α	β	δ	η	φ	ε	μ
x_1	-1						
x_2	+1	2	-1			-1	+1
x_3			+1	-1			
x_4				+1	-1	+1	
x_5					+1		-1

Самый популярный вид матрицы для графов – *матрица смежностей*

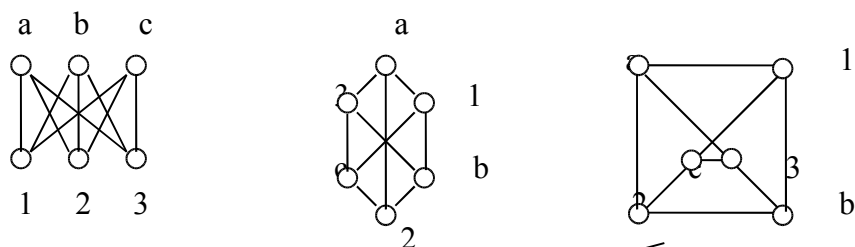
	x_1	x_2	x_3	x_4	x_5
x_1		1			
x_2		1	1	1	
x_3				1	
x_4					1
x_5		1			

Граф с ненаправленными соединениями (*ребрами*) - **неориентированный**.

Граф с направленными стрелками (*дугами*) – **ориентированный (орграф)**.

Мультиграф – граф, между вершинами которого может быть больше одной дуги.

В графах важно их топологическое свойство: то есть соединение определенных вершин. А само по себе взаиморасположение роли не играет, как и расстояния между объектами.



Один и тот же граф.

Дуга может выходить из вершины или заходить в нее. Она будет соответственно называться *исходящей* или *заходящей*.

Путем в орграфе называется последовательность дуг, такая, что каждое следующее ребро исходит из вершины, в которую заходит предыдущее.

Длина пути измеряется числом пройденных дуг.

Путь, начинающийся и заканчивающийся в одной и той же вершине - **контур**.

Контур длиной в единицу - **петля**.

Путь называется **простым**, если каждое из ребер встречается на этом пути один раз.

Путь называется **элементарным**, если любая вершина на этом пути один раз.

Дуга, исходящая (или заходящая) в вершину называется **инцидентной** данной вершине (и наоборот вершина **инцидентна** дуге).

Вершины инцидентные одной дуге называются **смежными**.

Полустепенью исхода вершины x - $\rho^-(x)$ называется число исходящих из нее дуг
 $\rho^-(x_3) = 3$.

Полустепенью захода вершины x - $\rho^+(x)$ называется число заходящих в нее дуг
 $\rho^+(x_3) = 2$.

$\rho = \rho^-(x) + \rho^+(x)$. - степень вершины x .

Теорема: В любом графе число вершин с нечетной степенью четно.

Доказательство исходит из того, что суммарная степень всех вершин – число четное (у каждой дуги 2 конца!). Если убрать степени всех четных вершин, то останется четное число суммарной степени нечетных вершин. А это возможно только если число вершин с нечетной степенью четно.

Теорема: В графе без петель, где вершин больше двух всегда найдется пара вершин с одинаковой степенью.

Доказательство заменим решением задачи «про шахматистов»:

Пусть среди n человек нет двух таких, кто сыграл бы одинаковое число партий в шахматном турнире. Тогда обязательно должно быть:

1-ый сыграл: 0 партий

2-ой сыграл: 1 партию

:
 n-ый сыграл n-1 партий
 т.е. из вершины n-го игрока исходит (n-1) стрелка, а в 0-ую не входит ни одна.
 Но этого не может быть.

Граф $G_A = \langle \Gamma_A, A \rangle$ - называется **подграфом** графа $G = \langle \Gamma, X \rangle$, если $A \subseteq X$, $\Gamma_A \subseteq \Gamma$.

Для неориентированных графов вместо *дуги* говорят *ребро*, вместо *пути* - *цепь*, вместо *контура* - *цикл*.

Граф называется **связным** (компонентом связности), если между любыми двумя вершинами есть цепь.

4.2. Теорема Эйлера

Цепь называется **эйлеровой**, если она является простой и проходит по всем ребрам графа.

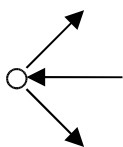
Эйлеровым циклом называется простой цикл, проходящий по всем ребрам.

Граф, имеющий эйлеровый цикл, называется **эйлеровым графом**.

Теорема: Для того чтобы связный граф был эйлеровым необходимо и достаточно, чтобы степени всех вершин его были четными.

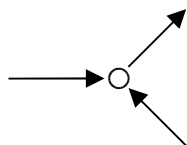
Доказательство: 1) Докажем необходимость.

Пусть есть вершина с нечетной степенью. Эта вершина не может быть первой



так как выходить и возвращаться в нее можно, используя четное число ребер. Нечетное ребро обусловит окончательный уход из этой вершины. Но эта вершина должна быть и последней, чтобы обеспечить цикл. Что не возможно.

Вершина с нечетной степенью не может быть и промежуточной, ибо в конечном итоге в этой вершине закончится цепь. Кроме ребра, по которому однажды в эту вершину зашли, останется четное число ребер, по которым будем уходить из вершины и обязательно в нее возвращаться.



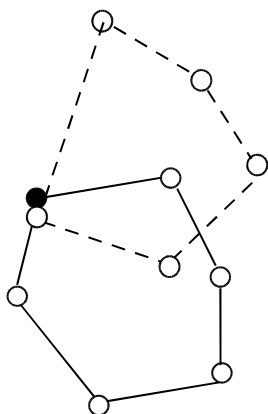
Таким образом доказана необходимость того, чтобы все вершины были четными.

2) Докажем достаточность требования четности вершин.

Возьмем любой граф, содержащий только вершины четной степени.

Строим из любой вершины простой цикл. Если пройдены все ребра, то теорема доказана иначе

Строим для любой вершины в контуре простой цикл из свободных ребер и вставляем новый цикл в предыдущий. (То есть при обходе прерываем в соответствующей вершине первый цикл и проходим второй, закончив его в вершине, из которой вышли. И заканчиваем первый цикл).

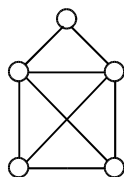


Если таким образом пройдены все ребра графа, то теорема доказана. Иначе выбирается новая вершина, инцидентная непройденным ребрам (их четное число) и строится новый цикл. И так до исчерпания непройденных ребер графа.

Таким образом, теорема доказана. А, следовательно, решена и задача о Кенигсбергских мостах. Слово «решена» здесь используется в расширенном понимании, принятом в обиходе у математиков, поскольку Эйлер на самом-то деле доказал, что задача не имеет решения.

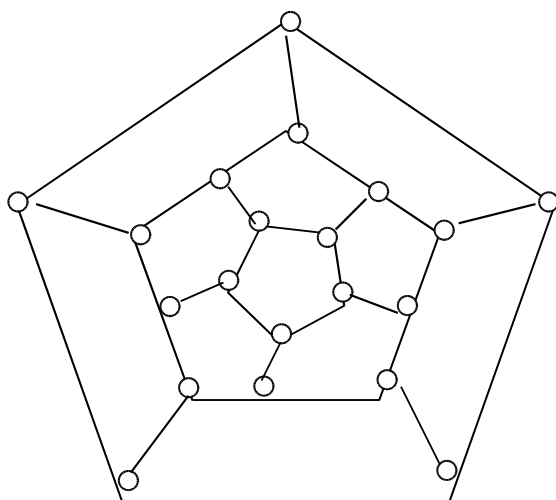
Следствие. Для того, чтобы в графе существовала Эйлерова цепь необходимо, чтобы в нем было ровно две вершины с нечетной степенью, причем эта цепь начинается в одной из этих вершин и заканчивается в другой.

Известная детская задача нарисовать, не отрывая карандаша, домик – лучшая иллюстрация к этому следствию.



Элементарный цикл, проходящий через все вершины, называется **гамильтоновым циклом**, а соответствующий граф – **гамильтоновым графом**.

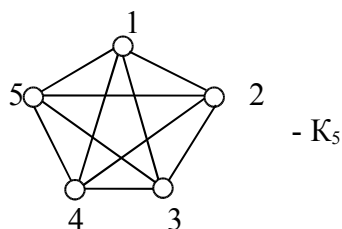
Пример гамильтонова графа



Однако, для гамильтоновых графов не удалось доказать красивой теоремы, наподобие теоремы Эйлера.

4.3. Полные графы и деревья

Граф называется **полным**, если любые две его вершины смежны, т.е. имеют общее ребро.



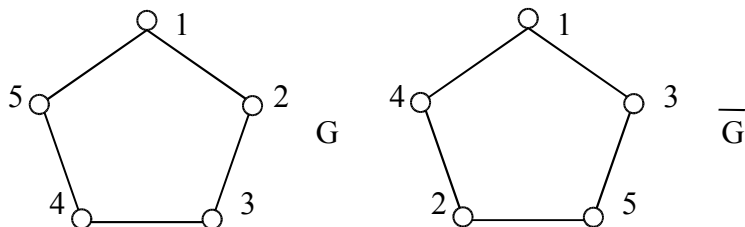
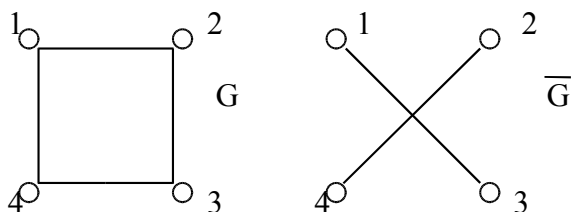
Теорема: В полном графе с n вершинами $\frac{n(n-1)}{2}$ ребер.

Доказательство. Каждая из n вершин полного графа связана с $n-1$ вершинами, то есть $n(n-1)$.

При таком подходе каждое из ребер учитывается дважды, поэтому надо разделить произведение на два.

В полном графе всегда существует гамильтонов цикл, и он определяется любой циклической подстановкой (см. *теорию групп*).

Граф \bar{G} называется **дополнением** графа G , если их объединение дает полный граф.



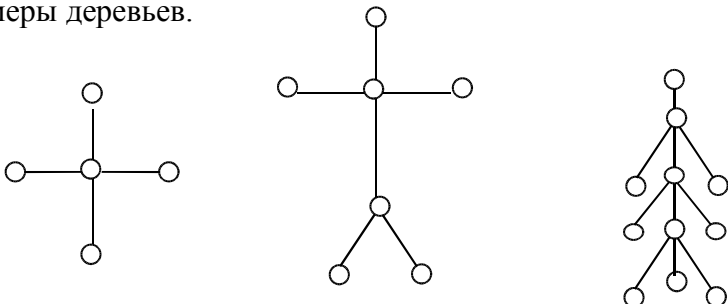
4.4. Деревья

Дерево - это связный граф без циклов. Можно дать другое определение дерева или вывести его из первого. Дерево – это граф, между любой парой вершин которого существует единственная цепь.

Теорема: В графе типа дерева с n вершинами $n-1$ ребер.

Доказательство. Для графа, состоящего лишь из одной вершины, это соотношение выполняется. Пусть оно выполняется и для графа с $n-1$ вершинами, тогда добавление новой дуги приводит к добавлению и одной вершины, что сохраняет соотношение.

Примеры деревьев.

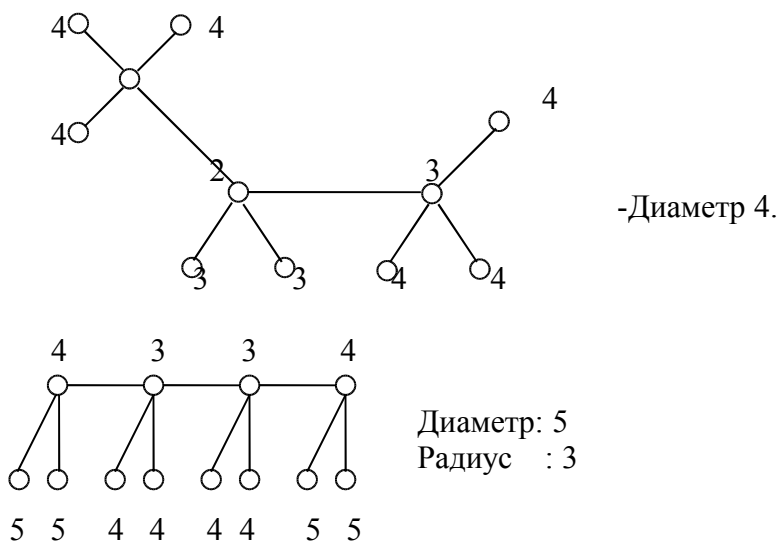


Лесом называется граф, состоящий из нескольких компонент связности, каждая из которых является деревом.

Диаметром для графов типа дерева является максимальное расстояние между его вершинами.

. Определим для каждой вершины ее расстояние от самого удаленного листа Минимальное число - **радиус**, эта вершина корневая (центральная).

В любом дереве существует одна или две (смежные) корневые вершины

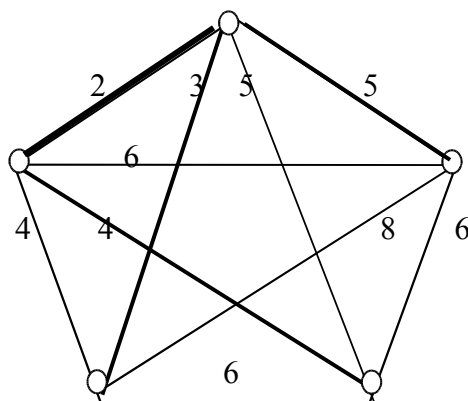


4.5. Алгоритм Краскала

Пусть дан полный граф. Ребрам приписаны штрафы. На основе этого графа строят дерево, имеющее минимальный суммарный штраф.

Для этого на каждом шаге выбирают ребро, имеющее минимальный штраф и не образующее цикл с уже выбранными ребрами.

Пример.

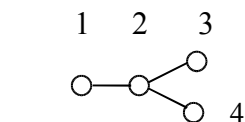
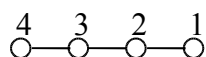
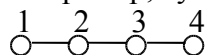


Жирными линиями выделено минимальное дерево

Теорема Кэли для раскрашенных деревьев.

Для n вершин существует n^{n-2} различных помеченных деревьев.

Например, существует 16 различных деревьев с четырьмя вершинами.



4 вершины $\Rightarrow 4^{4-2} = 16$ различных помеченных деревьев

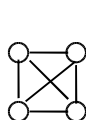
4.6.

Планарные графы

Граф - **плоский**, если он изображен на плоскости без пересечения ребер.

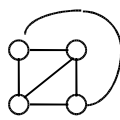
Граф - **планарный**, если он может быть изображен на плоскости без пересечения ребер.

Любой плоский граф может быть преобразован в граф с прямыми ребрами.



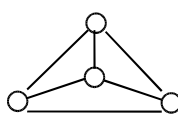
неплоский, но
планарный

\Rightarrow



плоский

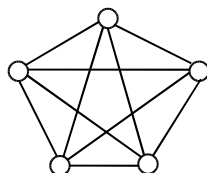
\Rightarrow



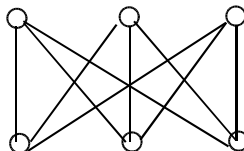
плоский с
прямыми ребрами

Граф, где все вершины соприкасаются с внешней гранью - **внешнепланарный**.

Два "замечательных" непланарных графа:



K_5



$K_{3,3}$

Приведем без доказательства две теоремы:

Любой граф, содержащий в качестве подграфа K_5 или $K_{3,3}$ - непланарен.

Два графа **гомеоморфны** если они тождественны с точностью до вершин со степенью $p=2$.

Любой граф, содержащий в качестве подграфа граф гомеоморфный K_5 или $K_{3,3}$ или - непланарен.

Теорема (Эйлера для планарных графов):

В любом планарном графе

$$B + \Gamma = P + 2.$$

где: B - число вершин

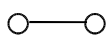
Γ - число граней

P - число ребер

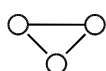
Доказательство:



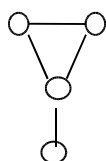
$$1 + 1 = 0 + 2$$



$$2 + 1 = 1 + 2$$



$$3 + 2 = 3 + 2$$



$$4 + 2 = 4 + 2$$

Пусть есть граф с n вершинами, для которого это соотношение верно.

Добавление ребра приводит к увеличению на единицу либо числа граней, либо вершин.

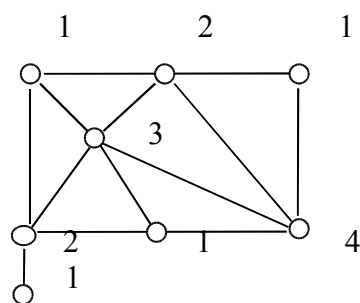
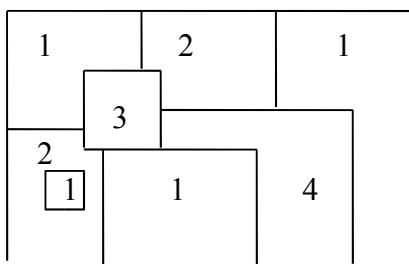
4.7. Задача о 4 красках

Это одна из самых знаменитых задач теории графов и математики вообще.

Достаточно ли четырех красок для раскраски любой политической карты мира так, чтобы два государства, имеющие общую границу, были раскрашены в разные цвета?

В качестве иллюстрации можно взять произвольную "карту". Для облегчения анализа представим государства в виде вершин графа. "Раскраску" отобразим цифрами.

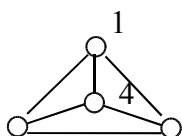
Дуги будут говорить о наличии общих границ. Не должно быть дуг, соединяющих вершины с одинаковыми цифрами.



Так что задачу можно переформулировать так:

Сколько необходимо красок в планарном графе, чтобы любые две смежные вершины были раскрашены различными цветами?

Теорема: Трех красок мало.



Пример доказывает, что 3-х красок мало

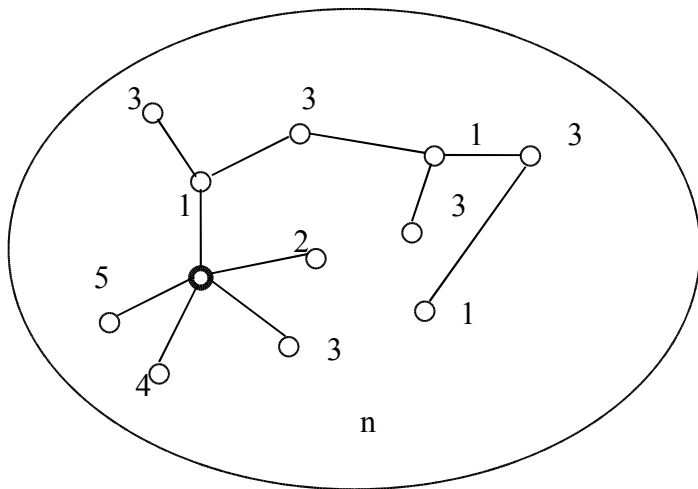
Теорема: Для раскраски любого планарного графа достаточно 5-ти красок

Доказательство:

1) Для любого планарного графа с $n \leq 5$ теорема справедлива.

2) Пусть любой планарный граф с n вершинами 5-раскрашиваемый.

Докажем справедливость этого и для графа с $n+1$ вершинами, опираясь на доказанный факт, что в любом *плоском графе* имеется хотя бы одна вершина степени не выше пяти. Объявим такую вершину $n+1$ -ой.



Если эта вершина имеет степень не больше четырех, то 5 красок хватит. Но если степень пять, то из 1-ой вершины строим все возможные цепи, где чередуются вершины 1 и 3:

Здесь возможны два случая:

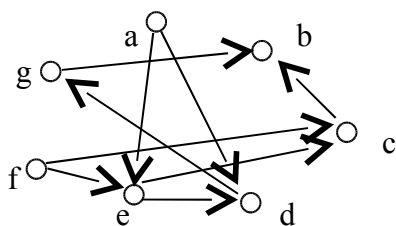
1). Ни одна из цепей не замкнулась на третью вершину. Тогда меняем цветами все 1-ые и 3-ие вершины и $n+1$ -ю вершину красим в 3-ий цвет;

2.) Одна из цепочек замкнулась на 3, тогда из вершины с номером 2 строим цепь 2-4. Ни одна из этих цепей не замкнется на 4-ю вершину (т.к. граф планарный!). Меняем цвета 2 и 4 в этой цепи. И красим $n+1$ -ю вершину в цвет 2.

Таким образом, то, что пяти красок достаточно, доказано.

Возвращаясь к четырем краскам следует сказать, что американскими математиками была доказана теорема, о том, что четырех красок достаточно. Однако, в этом доказательстве есть шаги, связанные с очень большими переборами вариантов, выполненные с использованием компьютера (пойди проверь). Так что с точки зрения «пуританской» математики можно считать, что теорема пока не доказана...

4.8. Определение путей в графе



Требуемые результаты получаются путем перемножения матриц смежности графа. M - матрица смежностей, показывает пути длиной в 1 в данном графе.

	a	b	c	d	e	f	g
a				1	1		
b		1					
c							
d							1
e			1	1			
f			1		1		
g		1					

M

×

	a	b	c	d	e	f	g
a				1	1		
b		1					
c							
d							1
e			1	1			
f			1		1		
g		1					

M

=

=

Матрица M^2 дает все пути длиной в 2

	a	b	c	d	e	f	g
a							1
b							
c							
d		1					
e							1
f		1		1			
g							

Матрица M^n - пути длиной в n.

Если M^i - нулевая матрица, то наибольший путь в графе имеет длину $i - 1$.

Для определения наличия путей между двумя вершинами можно использовать «транзитивное замыкание» матриц

$M^* = M^1 \cup M^2 \cup M^3 \dots$

Непустая клеточка ij будет говорить о наличии пути из i -ой вершины в j -ую.

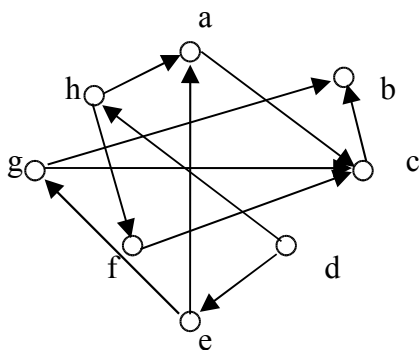
4.9. Приведение графа к ярусно-параллельной форме

Эти рассуждения имеют смысл для ориентированных графов без контуров.

Граф находится в **ярусно-параллельной форме** (ЯПФ),

если в нулевом ярусе размещаются вершины, имеющие нулевую полустепень захода, в i -том ярусе - вершины, в которые заходят дуги только из предыдущих ярусов, причем хотя бы одна дуга из $(i - 1)$ -го яруса.

Пусть дан произвольный граф без циклов.

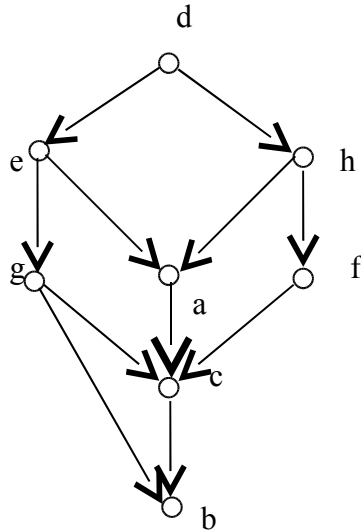


	a	b	c	d	e	f	g	h
a			1					
b								
c		1						
d					1			1
e	1						1	
f			1					
g		1	1					
h	1					1		

Алгоритм приведения к ЯПФ:

1. Матрица смежности графа просматривается, и в очередной ярус выбираются вершины с нулевой полустепенью захода, соответствующие нулевым столбцам.
2. Строки, соответствующие выбранным на предыдущем шаге вершинам, обнуляются.
3. Осуществляется возврат к шагу 1, до полного исчерпания матрицы.
4. Прорисовываются дуги.

В результате, вышеприведенный граф примет вид:

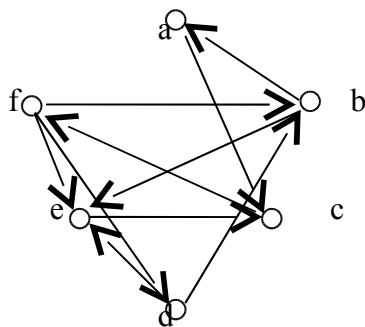


Ширина яруса определяется числом вершин в ярусе.

Ширина графа в ЯПФ определяется шириной самого большого яруса.

4.10. Внутренняя устойчивость графа

Множество внутренней устойчивости - множество несмежных вершин графа.



	a	b	c	d	e	f
a			1			
b	1				1	
c						1
d		1			1	
e			1			
f		1		1	1	

Поскольку одна любая вершина представляет внутренне устойчивое множество, то естественно, интерес представляют максимально возможные множества внутренней устойчивости.

Классический пример, задача о восьми ферзях: Как расставить на шахматной доске восемь ферзей, чтобы они не били друг друга. То есть построить граф с 64 вершинами (клеточками), где каждая клеточка соединена с клеточками, которые находятся под боем. Максимальные множества несмежных вершин и дают решение этой задачи.

Долго эта задача была классическим полигоном для систем искусственного интеллекта, как творческая задача, использующая нестрогие (эвристические) методы.

Последние годы ситуация изменилась.

Для нахождения таких множеств появился замечательный алгоритм Магу, который, Фактически дает аналитическое (!) решение этой задачи.

Алгоритм Магу.

1. По единицам матрицы строим парные дизъюнкты.

$$(a \vee b)(a \vee c)(b \vee e)(c \vee f)(d \vee b)(d \vee e)(e \vee c)(f \vee b)(f \vee d)(f \vee e)$$

2. Преобразуем в ДНФ, выполнив все возможные поглощения и операции идемпотентности.

$$\text{Получим: } bcde \vee bcef \vee adef \vee afeb \vee fbdc$$

3. Для каждой конъюнкции выписываем недостающие вершины, образующие множества внутренней устойчивости.

$$\{a, f\}, \{a, d\}, \{a, e\}, \{b, c\}, \{c, d\}$$

Максимальное из таких множеств дает **число внутренней устойчивости** (здесь оно равно 2).

4.11. Множество внешней устойчивости.

Ядро графа

Множество **внешней устойчивости** - такое множество вершин графа, что:

1) либо вершины принадлежат этому множеству.

2) либо они имеют дуги в этом множестве.

Это определение легче усвоить и запомнить, если отдавать себе отчет, что внешне устойчивое множество, прежде всего, определяется вершинами графа, которые в это множество *не входят* (пункт 2).

Множество всех вершин графа внешне устойчиво (подпадает под пункт 1). Поэтому интерес представляют минимально возможные множества внешней устойчивости.

Поиск внешне устойчивого множества происходит в другой классической задаче:

Как расставить минимальное число ферзей, чтобы все поля доски были под боем.

Для решения этой задачи также используется соответствующий алгоритм Магу.

Возьмем граф из предыдущего примера:

	a	b	c	d	e	f
a	1		1			
b	1	1			1	
c			1			1
d		1		1	1	
e			1		1	
f		1		1	1	1

Алгоритм Магу.

1. По главной диагонали проставляем 1.

2. Выписываем построчные дизъюнкции.

$$(a \vee c)(a \vee b \vee e)(c \vee f)(b \vee e)(c \vee e)(b \vee d \vee e \vee f)$$

3. Преобразуем в ДНФ, выполнив все возможные поглощения и операции идемпотентности.

Получим: $acd \vee aef \vee bc \vee ce$

Эти конъюнкции и дают множества внешней устойчивости.

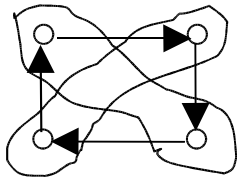
$\{a, c, d\}, \{a, e, f\}, \{b, c\}, \{c, e\}$

Минимальное из них дает **число внешней устойчивости** (здесь 2).

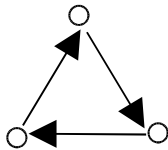
Множества, одновременно внутренне и внешне устойчивые называются **ядром** графа.

Для рассмотренного графа - $\{b, c\}$

В графе может быть несколько ядер (например - 2)

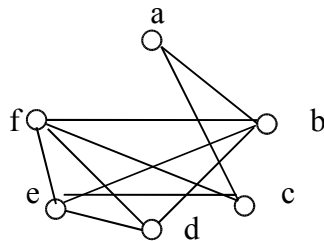


или не быть совсем.



4.12. Клика

Клика - максимально большой полный подграф данного графа.

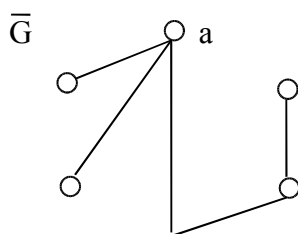


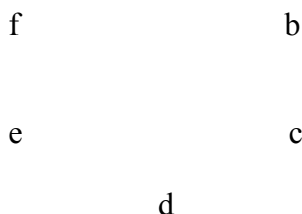
	a	b	c	d	e	f
a		1	1			
b				1	1	1
c					1	1
d					1	1
e						1
f						

	a	b	c	d	e	f
a				1	1	1
b			1			
c				1		
d						
e						
f						

Построение Клики.

1. Строим дополнительный граф исходного графа.





2. Найдем множество внутренней устойчивости для графа \bar{G} .

$$(a \vee d)(a \vee e)(a \vee f)(b \vee c)(c \vee d)$$

$$(a \vee de)(a \vee f)(c \vee bd)$$

$$(a \vee def)(c \vee bd)$$

$$ac \vee cdef \vee bdef \vee abd$$

$$\{b, d, e, f\}, \{c, e, f\}, \{a, b\}, \{a, c\}$$

3. Множества полученных вершин дают всевозможные полные подграфы исходного графа G . Причем, максимальный из подграфов дает клику.

5. Теория групп

Теория групп лежит в основе современной алгебры. Начала ее были созданы молодым гениальным математиком Э. Галуа (1811-1832) как инструмент для оценки возможности решения уравнений высших степеней в радикалах. Однако сфера применения и область интерпретации теории групп с тех пор многократно расширилась. Одна из самых значительных интерпретаций для групп – это различные типы симметрии.

5.1. Понятие группы

Группу можно задать как алгебру с одной операцией \otimes , удовлетворяющей следующим законам:

1. Существование операции.

$$\forall xy \exists z (x \otimes y = z)$$

2. Ассоциативность

$$\forall xyz (x \otimes (y \otimes z)) = ((x \otimes y) \otimes z)$$

3. Существование единицы (e)

$$\exists e \forall y (e \otimes y = y)$$

4. Существование обратного элемента.

$$\forall x \exists ! y (x \otimes y = e)$$

5. Коммутативность

$$\forall xy (x \otimes y = y \otimes x)$$

Выполнение лишь первого закона дает **группоид**. Если дополнительно выполняется второй – **полугруппа** (популярна при исследовании свойств формальных грамматик).

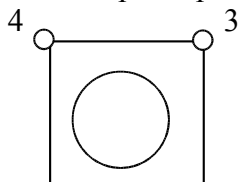
Выполнение первого, второго и третьего законов дает **моноид**.

Выполнение аксиом с первой по четвертую дает **группу**.

Если для группы выполняется также коммутативный закон, то группа называется **абелевой**.

5.2. Морфизмы групп

Рассмотрим вращения квадрата вокруг центра до совмещения вершин.



$$a_1 = 0^0$$

В качестве элементов – углы поворота.

$$\begin{array}{ccc}
 & a_2 = 90^0 & \text{В качестве операции - доворачивание.} \\
 & a_3 = 180^0 & \text{Выполняются все законы для группы.} \\
 & a_4 = 270^0 & \\
 \begin{array}{c} \Gamma \\ \circ \quad \circ \\ 1 \quad 2 \end{array} & &
 \end{array}$$

Например, $a_1 \circ a_2 = a_2$; $a_2 \circ a_2 = a_3$; $a_3 \circ a_3 = a_1$

Популярны со времен Галуа и так называемые *подстановки*. Можно записать подстановки, соответствующие каждому из четырех вращений предыдущего примера:

$$\begin{array}{cccc}
 \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 1 & 2 & 3 & 4 \end{pmatrix} &
 \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 4 & 1 & 2 & 3 \end{pmatrix} &
 \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 3 & 4 & 1 & 2 \end{pmatrix} &
 \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 2 & 3 & 4 & 1 \end{pmatrix}
 \end{array}$$

А в качестве операции взять композицию подстановок. Например,

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 2 & 3 & 4 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 2 & 3 & 4 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 3 & 4 & 1 & 2 \end{pmatrix}$$

В результате также получится группа.

Возьмем корни уравнения $x^4 - 1 = 0$
 $\{1, i, -1, -i\}$ - группа по операции умножения.

Таким образом, мы рассмотрели несколько конечных групп, содержащих по четыре элемента. Эти группы изоморфны между собой.

Например, можно отобразить друг в друга «единичные» элементы:

$$a_1 \leftrightarrow 0^0 \leftrightarrow \begin{pmatrix} 1 & 2 & 3 & 4 \\ | & & & | \\ 1 & 2 & 3 & 4 \end{pmatrix} \leftrightarrow 1$$

Так что речь может идти об *абстрактных группах*, то есть о таких группах, для которых конкретное множество и конкретная операция несущественны.

Пусть f - некоторое отображение элементов одной группы в другую или в ту же самую и

$$f(a \otimes b) = f(a) \oplus f(b) \quad a, b \in G; f(a), f(b) \in b_2.$$

то говорят, что f - **гомоморфизм**.

Если $f(a)=f(b)$, тогда и только тогда, когда $a = b$, то имеем **изоморфизм** (однозначный гомоморфизм).

Гомоморфизм группы в себя называется **эндоморфизмом**.

Инъективный гомоморфизм называется **мономорфизмом**.

Сюръективный гомоморфизм называется **эпиморфизмом**.

Изоморфизм в себя называется **автоморфизмом**.

Пример : $\{ \dots -3, -2, -1, 0, 1, 2, 3, \dots \}$

$$\begin{array}{c}
 \downarrow \downarrow \downarrow \downarrow \downarrow \\
 \{ \dots -6, -4, -2, 0, 2, 4, 6, \dots \}
 \end{array}$$
 - эндоморфизм, эпиморфизм, мономорфизм, изоморфизм, автоморфизм.

5.3. Инвариантные (нормальные) подгруппы

Группа H называется **подгруппой** группы G , если она состоит из элементов группы G и сама является группой.

Элемент $c = b^{-1}ab$ называется **трансформацией** элемента a с помощью элемента b . При этом элементы c и a называются **сопряженными**.

b^{-1} - **обратный элемент** для b .

Здесь a и b - элементы группы, а обычное (необозначаемое) умножение, фактически, групповая операция.

Если $b^{-1}ab = a$, то $ab = ba$ (т.к. данная группа абелева, следовательно, коммутативна).

Доказательство: умножим $b^{-1}ab = a$ слева и справа от знака равенства на b :
 $bb^{-1}ab = ba$

Теорема: Трансформация разбивает группу на классы сопряженных элементов.

Доказательство:

1. Рефлексивность : $a = 1^{-1}a1$

2. Симметричность : $c = b^{-1}ab \Rightarrow$

$$bcb^{-1} = bb^{-1}abb^{-1}$$

$$bcb^{-1} = a$$

$$(b^{-1})^{-1}cb^{-1} = a, \text{ пусть } B = b^{-1}$$

$$B^{-1}cB = a, \text{ т.е. если } a - \text{ трансформация } c, \text{ то } c - \text{ трансформация } a$$

3. Транзитивность: $c = b^{-1}ab, c = d^{-1}cd$

$$e = d^{-1}b^{-1}abd$$

$$e = (bd)^{-1}abd$$

$$e = D^{-1}aD$$

$$bdd^{-1}b^{-1} = 1, (bd)^{-1}(bd) = 1 \Leftrightarrow d^{-1}b^{-1} = (bd)^{-1}$$

Теорема: Трансформация подгруппы H элементом $b \in G$ есть подгруппа группы G , изоморфная H .

Доказательство:

1. $C_1 = b^{-1}x_1b$

$$C_2 = b^{-1}x_2b, \quad x_1, x_2 \in H$$

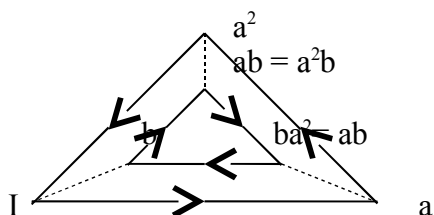
$$C_1C_2 = b^{-1}x_1bb^{-1}x_2b$$

2. $b^{-1}1b = 1$ (т.е. 1 исходной группы остается 1 полученной группы)

3. $a = b^{-1}xb$

$$a^{-1} = (b^{-1}xb)^{-1} = b^{-1}x^{-1}(b^{-1})^{-1} = b^{-1}x^{-1}b$$

Т.е. в результате (1- 3) мы получаем группу, причем эта процедура сохраняет функциональность, сюръективность, всюду определенность, инъективность, т.е. полученная группа изоморфна исходной.



Подгруппа K группы G называется **инвариантной (нормальной)**, если трансформация любого элемента подгруппы K с помощью любого элемента этой группы дает снова элемент подгруппы K .

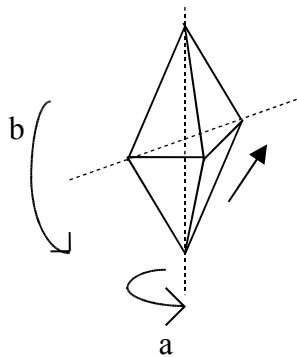
$K = \{ I, a, a^2 \}$ - подгруппа некоторой группы G

$$ab = ba^2 = ba^{-1} \quad (\text{или} \quad a^2 \cdot a = I \quad / \quad *a^{-1}, \quad a^2aa^{-1} = Ia^{-1}, \quad a^2 = a^{-1})$$

$$b^{-1}ab = b^{-1}ba^{-1}$$

$b^{-1}ab = a^{-1} \quad (= a^2)$ - трансформация элемента a с помощью элемента b и она есть элемент группы.

5.4. Группа Диэдра (D_3)



$$D_3 = \{I, a, a^2, b, ba, ba^2\}$$

Для этой группы будут следующие определяющие соотношения:

$$a^3 = b^2 = (ba)^2 = I$$

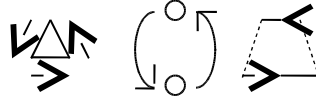


Таблица умножения данной группы:

	I	a	a^2	b	ba	ba^2
I	I	a	a^2	b	ba	ba^2
a	a	a^2	I	ba^2	b	ba
a^2	a^2	I	a	ba	ba^2	b
b	b	ba	ba^2	I	a	a^2
ba	ba	ba^2	b	a^2	I	a
ba^2	ba^2	b	ba	a	a^2	I

В каждой строке и каждом столбце элементы не повторяются.

а. $H = \{I, B\}$ пусть $f(I) = f(b) = I$ - некоторый гомоморфизм

$$a = Ia = (ba)^2a = baba = baba^2$$

$$f(a) = f(baba^2) = f(b) f(a) f(a) f(b^2) = f(a)f(a^2) = \text{(по предположению } f(b) = I \text{)}$$

$$= f(a^3) = f(I) = I$$

$$f(a^2) = f(a) f(a) = I I = I$$

$$f(ba) = f(b) f(a) = I I = I$$

$$f(ba^2) = f(b) f(a^2) = I I = I$$

Т.е. всю группу D_3 можно отобразить в единичный элемент.

$$\begin{array}{lcl} \text{а)} & f & f \\ H = \{I, b\} & D_3 \rightarrow G : D_3 \rightarrow I & \\ K = \{I, a, a^2\} & f & f \\ & D_3 \rightarrow G : D_3 \rightarrow \{I, f(b)\} & \end{array}$$

$$f(I) = f(a) = f(a^2) = I$$

I

$$f(ba) = f(b)f(a) = f(b)$$

$$f(ba^2) = f(b) = f(b)f(b) = f(b^2) = I$$

Группы, имеющие единственный (отличный от единицы) элемент такой, что какая-то степень этого элемента дает I , называется циклической группой n -ой степени.

Если для какой-то группы мы осуществляем гомоморфное отображение, причем какая-то ее подгруппа целиком отображается в единичный элемент группы, то такая подгруппа есть ядро гомоморфизма. Обозначается $f^{-1}(I)$.

5.5. Смежные классы

$$H = \{I, b\}$$

aH - смежный (левый) класс для H , если все элементы H слева умножены на a .

$$aH = \{a, ab\}$$

$$a^2H = \{a^2, a^2b\} = \{a, ba\}$$

$$K = \{I, a, a^2\}$$

$$bK = \{b, ba, ba^2\}$$

Все получаемые элементы различны между собой.

Теорема (Лагранжа): Порядок конечной группы кратен порядку любой его подгруппы.

Подгруппа K (группы G) есть инвариантная для G , если соответствующие смежные классы для нее совпадают.

Если группа коммутативная, то она инвариантная.

Ядро гомоморфизма является нормальной подгруппой.

5.6. Фактор-группы

Смежные классы группы G по ее нормальной подгруппе K образуют группу.

$$K = \{I, a, a^2\}$$

$$bK = \{b, ba, ba^2\}$$

$$1) K \bullet K$$

	I	a	b
I	I	a	a ²
a	a	a ²	I
b	a ²	I	a

$$2) K \bullet bK = bK$$

	b	ba	ba ²
I	b	ba	ba ²
a	ba ²	b	ba
b	ba	ba ²	b

$$3) bK \bullet K = bK$$

$$4) bK \bullet bK = K$$

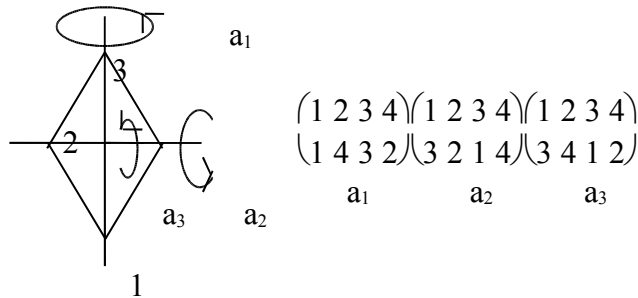
	K	bK
K	K	bK
bK	bK	K

- фактор - группа для группы G .

5.7. Группа Клейна четвертой степени

Это замечательная группа. Некоторые ученые, особенно предрасположенные к философии считают, что это одна из групп, лежащих в основе механизма мироздания. Не меньше и не больше. Но мы здесь философский аспект опускаем, а группу приводим.

	I	a ₁	a ₂	a ₃
I	I	a ₁	a ₂	a ₃
a ₁	a ₁	I	a ₃	a ₂
a ₂	a ₂	a ₃	I	a ₁
a ₃	a ₃	a ₂	a ₁	I



Свойства:

1. $a_i \cdot a_j = a_j \cdot a_i$
2. $a_i \cdot I = a_i$
3. $a_i \cdot a_i = I$
4. $a_i \cdot a_j = a_k$, где $a_i, a_j, a_k \neq I$
5. $a_i \cdot a_j \cdot a_k = I$, где $a_i, a_j, a_k \neq I$

6. Теория алгоритмов

6.1. Понятие алгоритма

Алгоритм - это строгое предписание по выполнению последовательности шагов, приводящее к решению задачи данного типа. Понятие алгоритма относится к понятиям фундаментальным и неопределяемым.

Свойства алгоритмов:

1. Массовость.
2. Пошаговость.
3. Элементарность отдельных шагов (что такое «элементарно» каждый Ватсон понимает по-своему).
4. Детерминированность (точно известно, что нужно делать после каждого шага).
5. Эффективность (алгоритм должен привести к решению за конечное число шагов).

Главное разочарование программистов относительно теории алгоритмов состоит в том, что *классическая теория алгоритмов не занимается «правилами построения алгоритмов»*. На законный вопрос, чем же она тогда занимается, можно достойно ответить: она занимается более важной проблемой – **проблемой алгоритмической разрешимости**. То есть занимается определением того, возможно ли вообще построить алгоритм для решения задач данного типа.

Другими словами, существуют алгоритмически не разрешимые задачи, на алгоритмизацию которых не стоит тратить время. Например, невозможно построить функцию

$$F(x) = \begin{cases} 1, & \text{если в числе } \pi \text{ есть последовательность из } x \text{ подряд цифр } 5 \\ 0, & \text{иначе.} \end{cases}$$

Любопытна в связи с этим теорема:

Теорема: Алгоритмически неразрешимых задач больше, чем алгоритмически разрешимых.
Доказательство: Мощность множества функций (если угодно – задач) даже заведомо ограничено:

$$f \\ N \rightarrow N$$

не менее, чем континуум- \aleph_1

Количество же вычислимых функций (если угодно – алгоритмов) счетно, т.е. \aleph_0 .

Действительно, всякий алгоритм конечен, следовательно, он может быть записан конечной строкой букв русского алфавита. А полученные конечные строки можно расположить в лексикографическом (алфавитном) порядке, то есть пересчитать.

Таким образом, $\aleph_1 - \aleph_0 = \aleph_1$

6.2. Конкретизация понятия алгоритма

Поскольку определить, что такое алгоритм, невозможно, то были предложены и успешно используются паллиативы, которые назовем *конкретизациями* понятия алгоритма. Наиболее известны следующие:

1. λ -нотация.
2. Рекурсивные функции.
3. Машины Тьюринга.
4. Нормальные алгорифмы Маркова.

В связи с этим задача переформулируется следующим образом:

Задача алгоритмически разрешима, если для нее можно построить λ -нотацию (рекурсивную функцию, машину Тьюринга, нормальный алгорифм Маркова). И наоборот, если невозможно построить λ -нотацию и т.п., то задача алгоритмически неразрешима. Важно, что все конкретизации алгоритма равнообъемны, то есть одновременно могут или не могут быть построены для исследуемых задач.

6.3. Сложность вычислений

В качестве небольшого, но важного отступления отметим в данном разделе, что алгоритмическая разрешимость еще не означает, что задача может быть реально решена. Дело в том, что для нас на практике задача, решение которой требует тысячу лет или ресурсы, превышающие все вычислительные ресурсы планеты, тоже неразрешима.

И здесь вступает в силу «математика практической осуществимости»...Такого рода проблемами занимается *теория сложности вычислений*.

Рассмотрим для иллюстрации таблицу, в которой четыре столбца, соответствующие «абстрактному» объему исходных данных (n)

Три строки соответствуют различным «функциям сложности вычислений». В таблице приведены времена вычислений в зависимости от объема данных и сложности вычислений (F).

$\begin{matrix} n \\ F \end{matrix}$	10	30	50	60
n	10^{-5} сек	$30 \cdot 10^{-6}$ сек.	$30 \cdot 10^{-6}$ сек.	$30 \cdot 10^{-6}$ сек.
n^5	0.1 сек.	24.3 сек.	5.2 мин.	13 мин.
2^n	10^{-3} сек.	17.9 мин.	35.7 лет	366 столетий

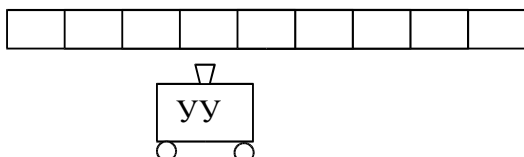
Бросается в глаза быстрый рост сложности вычислений в последней строке. Действительно, с точки зрения теории сложности вычислений, между последним и двумя первыми типами задач пропасть. Первые две задачи относятся к классу задач с полиномиальной сложностью. А последняя – к задачам с экспоненциальной сложностью. Такие задачи называются *трудноразрешимыми*. Класс этих задач формируется эмпирически и носит название класса *NP - полных задач*.

Есть какая-то аналогия между проблемами алгоритмической разрешимости и трудноразрешимости. Как из-за невозможности определить понятие алгоритма проблема алгоритмической разрешимости сводится к возможности построения конкретизации, так и трудноразрешимость устанавливается сведением исследуемой задачи к одной из «эталонных» NP - полных задач.

6.4. Машины Тьюринга

Известны случаи построения школьниками железных машин Тьюринга с колесами. Но машина Тьюринга – это все-таки прежде всего метод математического моделирования. Машина Тьюринга включает:

1. Потенциально бесконечную (вправо) ленту, разделенную на ячейки.
2. Считывающе-записывающую головку с устройством управления (УУ).
3. Алфавит внутренних состояний $\{q_0, q_1 \dots q_n\}$.
4. Входной-выходной алфавит.



Определяется начальная конфигурация. Головка смотрит на какую-то ячейку и устройство управления находится в начальном состоянии q_1 .

Устройство управления на основании считанного из ячейки символа и внутреннего состояния пишет в ячейку символ (возможно, тот же самый), совершает действие D и переходит в новое внутреннее состояние (возможно прежнее). Это и есть команда Машины Тьюринга, которую можно записать так:

$$a_i q_i \rightarrow a_j D_j q_j.$$

$D = \{Л, П, С\}$ - множество действий.

Л – влево, П – вправо, С – стоять.

Совокупность команд составляет **программу** машины Тьюринга, которая обычно оформляется в виде таблицы.

Машина заканчивает работу, когда переходит в состояние q_0 .

λ - пустой символ.

Пример: Построим машину Т, которая в сплошной последовательности 1 стирает первую и две последние. (λ - пустой символ).

	q_1	q_2	q_3	q_4
1	$\lambda П q_2$	$1 П q_2$	$\lambda Л q_4$	$\lambda С q_0$
λ	-	$\lambda Л q_3$	-	-

6.5. Нормальные алгоритмы Маркова

Автор - А.А. Марков, отдавал предпочтение транскрипции **алгорифм**. Нормальные алгорифмы Маркова представляются *нормальной схемой подстановок*, которая состоит из

совокупности подстановок, расположенных в определенном порядке. Подстановки имеют вид: $P \rightarrow (\cdot)Q$ ($P \rightarrow Q$ – (простая) подстановка, $P \rightarrow \cdot Q$ – заключительная подстановка).

Говорят, что строка R входит в строку L , если L имеет вид L_1RL_2 .

Говорят, что подстановка применима к слову, если строка, соответствующая левой части подстановки, входит в слово. Применение заключается в замене в преобразуемом слове левой строки подстановки правой.

Две особые подстановки:

$P \rightarrow$ - аннулирующая.

$\rightarrow Q$ - порождающая.

Механизм работы нормальных алгорифмов:

0) Дано (преобразуемое) слово – цепочка символов фиксированного алфавита и нормальная схема подстановок, содержащая фиксированную последовательность простых и заключительных подстановок.

1) Слово всегда просматривается слева направо.

Схема подстановок просматривается всегда начиная с первой подстановки и, если подстановку можно применить, то она применяется к самому левому вхождению этой строки в преобразуемое слово.

2) Работа алгоритма заканчивается тогда, когда ни одна из подстановок не применима, либо использована заключительная подстановка.

Примеры.

нормальная схема
подстановок

преобразуемое
слово

\downarrow	$xx \rightarrow y$ $xy \rightarrow x$ $yz \rightarrow x$ $zz \rightarrow \cdot z$ $yy \rightarrow x$	\longrightarrow <u>xxx</u> yyzzz y <u>xy</u> zzz y <u>xy</u> zzz y <u>xy</u> zzz yx <u>zzz</u> yxzzz
--------------	--	--

	МУХА
$X \rightarrow K$	МУКА
$M \rightarrow P$	РУКА
$KA \rightarrow ЛОН$	РУЛОН
$PY \rightarrow \cdot СЛ$	СЛОН

Примеры алгорифмов, использующие специальные символы, аннулирующие и порождающие подстановки:

Удвоение исходной строки

$\alpha x \rightarrow x\beta x\alpha$

$\alpha y \rightarrow y\beta y\alpha$

$\beta xx \rightarrow x\beta x$

$\beta xy \rightarrow y\beta x$

$\beta yx \rightarrow x\beta y$

$\beta yy \rightarrow y\beta y$

$\beta \rightarrow$

$\alpha \rightarrow \cdot$

$\rightarrow \alpha$

Обращение исходной строки

$\alpha\alpha \rightarrow \beta\alpha$

$\beta\alpha \rightarrow \beta$

$\beta x \rightarrow x\beta$

$\beta y \rightarrow y\beta$

$\beta \rightarrow .$

$\alpha x y \rightarrow y \alpha x$

$\alpha y x \rightarrow x \alpha y$

$\rightarrow \alpha$

6.6. Рекурсивные функции

Содержательная идея рекурсивных функций состоит в том, что все исходные данные (аргументы) задачи и ее решения (значения функций) *можно пересчитать*, даже если это далекая от математики задача, вроде решения для себя проблемы идти ли на дискотеку, в библиотеку или лучше на футбол...

Осуществив такого рода нумерацию аргументов и значений можно свести решение задач к нахождению функций ставящих в соответствие числовым аргументам числовые значения.

При этом как аргументы, так и функции находятся в области натуральных чисел - N .

Рекурсивные функции строятся на основе трех примитивных (заведомо однозначно понимаемых и реализуемых) функций.

1. **Нуль-функция:** $Z(x) = 0$.

Например, $Z(1) = 0$, $Z(5) = 0$, но $Z(-5)$ - не определено.

4. **Функция следования:** $S(x) = x + 1$.

Тонкость заключается в том, что операции сложения (+) мы здесь пока не определили и запись " $+ 1$ " понимается как взятие следующего элемента естественно упорядоченного множества N .

То есть в множестве N всегда можно найти следующий аргумент, например: $S(5) = 6$.

3. **Функция проектирования (выбора аргумента).** $I_i^{(n)}(x_1, x_2, \dots, x_i, \dots, x_n) = x_i$.

Или частный случай - **тождественная функция:** $I(x) = x$.

С примитивными функциями можно производить различные манипуляции, используя три оператора: **суперпозиции, примитивной рекурсии и наименьшего корня**.

1. **Оператор суперпозиции:** Позволяет из функции $f(y_1, \dots, y_m)$ и функций $h_1(x_1, \dots, x_n)$, $h_2(x_1, \dots, x_n)$, ..., $h_m(x_1, \dots, x_n)$ сконструировать функцию $f(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n))$.

Например, с помощью оператора суперпозиции можно получить любую константу

$S(S(\dots(Z(x))\dots)) = n$, где число вложенных функций следования равно n .

Или сдвига числа на константу n , также равную числу вложенных функций следования.

$S(S(\dots(S(x))\dots)) = x + n$,

2. **Оператор примитивной рекурсии.** Этот оператор позволяет получить $n + 1$ -местную функцию из n -местной и $n + 2$ -местной функций.

Оператор задается двумя равенствами:

$$\begin{cases} f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y) = h(x_1, \dots, x_n, y-1, f(x_1, \dots, x_n, y-1)) \end{cases}$$

Позволяет по $n+1$ -местной функции получить n -местную.

Частный случай - функция одного аргумента:

$$\left\{ \begin{array}{l} f(0) = \text{const} \\ f(y) = h(y - 1, f(y - 1)) \end{array} \right.$$

Функции, которые могут быть построены из примитивных с помощью операторов суперпозиции и примитивной рекурсии называются **примитивно-рекурсивными**.

Пример: *функция суммирования*.

$$f_z(x, 0) = g(x) = I(x) = x$$

$$f_z(x, 1) = h(x, 0, f_z(x, 0)) = h(x, 0, x) = h(I_{1,3}^{(3)}((x, 0, x))) = S(x) = x + 1$$

$$f_z(x, 2) = h(x, 1, f_z(x, 1)) = h(x, 1, x+1) = S(x+1) = x + 2$$

...

$$f_z(x, y) = h(x, 1, f_z(x, y - 1)) = S(f_z(x, y - 1)) = x + y$$

то есть в традиционной записи определение сложения, как примитивно-рекурсивной функции, будет:

$$x + y = x + (y - 1) + 1$$

Функция умножения.

$$f_p(x, 0) = y(x) = z(0) = 0$$

$$f_p(x, 1) = h(x, 0, f_p(x, 0)) = h(x, 0, 0) = h(I_{1,3}^{(3)}((x, 0, 0))) = f_z(x, 0) = x$$

$$f_p(x, 2) = h(x, f_p(x, 1)) = f_z(x, x) = 2x$$

$$f_p(x, y) = f_z(x, f_p(x, y - 1))$$

то есть в традиционной записи определение умножения, как примитивно-рекурсивной функции, будет

$$x * y = x * (y - 1) + x$$

3. μ -оператор.

$$\mu y[g(x_1, \dots, x_n, y) = 0]$$

где y - выделенная переменная.

Работу μ -оператора можно описать следующим образом.

Выделяется переменная (здесь – y). Затем фиксируется значение остальных переменных (x_1, \dots, x_n). Значение y последовательно увеличивается, начиная с нуля. Значением μ -оператора будет значение y , при котором функция впервые обратилась в ноль. Значение μ -оператора считается неопределенным, если функция вообще не принимает значения ноль, либо она принимает отрицательное значение до того как примет значение ноль.

Пример.

Пусть функция $g(x, y) = x - y + 3$. Зафиксируем $x = 1$

$$\mu y[g(1, y) = 0] = 4$$

так как $1 - 4 + 3 = 0$.

Класс (множество, которое может быть получено из примитивных функций с помощью операторов суперпозиции, примитивной рекурсии и μ -оператора, называется **множеством частично-рекурсивных функций**.

Множество частично-рекурсивных функций совпадает с множеством *вычислимых функций* - *алгоритмически разрешимых задач*.

6.7. λ -исчисление

λ -исчисление, основоположником которого считают Алонзо Черча, фактически, и стало основой теории алгоритмов и первой конкретизации алгоритма. λ -исчисление рассматривают также как основу таких важных разделов математики, как функциональное программирование и комбинаторная логика.

λ -исчисление (нотация, способ записи) формализует понятие функции не как множества или графика, а как *правила*.

В основе λ -исчисления лежит операция аппликации.

Аппликация - применение функции к аргументу (можно применить только известную функцию).

Язык состоит из:

1. Множества переменных ($x_1 \dots$).
2. Множества констант ($c_1 \dots$).
3. Символа аппликации \bullet .
4. Символа абстракции λ .
5. Символа равенства $=$.

λ -терм:

1. Переменная или константа - λ -терм.
2. Если x - переменная, и M - некоторый λ -терм, то $\lambda x.M$ тоже λ -терм.
3. Если M и N λ -термы, то MN тоже λ -терм.

Формула - любое выражение вида $M=N$, где M и N λ -термы.

Замечание. В литературе прикладного плана нередко используется несколько иная терминология и форма записи.

$f = \lambda x. x + x$

f - название, ранее не названной функции.

λ - оператор.

x - аргумент.

\bullet -комбинатор.

$x + x$ - выражение, задающее значение функции.

Аксиомы:

1. $M = M$.

2. $(\lambda x.M)N = M \{N/x\}$ β -редукция.

где $\{N/x\}$ – подстановка N вместо всех вхождений x в M .

[В альтернативном представлении часто используемая β -редукция записывается, например, так $(\lambda x.f(x))(a) = f(a)$]

3. $\lambda x.M = \lambda y.M$ при $\{y/x\}$ α -конверсия.

где $\{y/x\}$ – подстановка y вместо всех вхождений x в M .

Правила вывода:

1.
$$\frac{M = N}{N = M}.$$

2.
$$\frac{M = N, N = P}{M = P}.$$

3.
$$\frac{M = N}{PM = PN}.$$

4.
$$\frac{M = N}{MP = NP}.$$

5.
$$\frac{M = N}{\lambda x.M = \lambda x.N}.$$

Рассмотрим примеры β -редукции (в прикладной варианте записи)

$$(\lambda x.x + 2y)(a) = a + 2y$$

$$(\lambda y.x + 2y)(a) = x + 2a$$

$$(\lambda x.(\lambda y.x + 2y))(a)(b) = (\lambda y.a + 2y)(b) = a + 2b$$

$$(\lambda x.((\lambda y.xy)u))(\lambda v.v) = (\lambda y.(\lambda v.v)y)u = (\lambda v.v)u$$

$$(\lambda x.((\lambda y.xy)u))(\lambda v.v) = (\lambda x.xu)(\lambda v.v) = (\lambda v.v)u$$

$$(\lambda x.xx) (\lambda x.xx) = (\lambda x.xx) (\lambda x.xx) = (\lambda x.xx) (\lambda x.xx) = \dots$$

$$((\lambda x.x*3) (\lambda y.\text{if } y > 4 \text{ then } e = 2 \text{ else } x/2) (\lambda y.y>2)) (5) = 2*5 = 10$$

$$(\lambda n.(\lambda x.x-n))(2) = \lambda x.x-2$$

$$(\lambda f.2*f(8) - f(f(8)))(\text{half}) = 2*8/2 - (8/2)/2 = 6 \quad (\text{half} - \text{половина}).$$

7. Формальные грамматики

7.1. Понятие формальной грамматики

Формальная грамматика - это четверка $G = \langle V_N, V_T, P, S \rangle$, в которой V_N - *нетерминальный словарь* (множество нетерминальных символов); V_T - *терминальный словарь* (множество терминальных символов); P - *множество грамматических правил*; $S \in V_N$ - *начальный нетерминальный символ*.

Метаязык - язык, с помощью которого описывается язык:

$::=$ - есть по определению;

| - “исключающее или”;

$\langle \dots \rangle$ - внутри – один нетерминальный символ;

[] - необязательная часть;

, - запятая – разделитель при перечислении.

Пример: Построим грамматику G_1 :

$\langle \text{прог} \rangle ::= \langle \text{оп} \rangle \mid \langle \text{оп} \rangle; \langle \text{прог} \rangle$

$\langle \text{оп} \rangle ::= \langle \text{пер} \rangle := \langle \text{выр} \rangle$

$\langle \text{пер} \rangle ::= a \mid b \mid c$

$\langle \text{выр} \rangle ::= \langle \text{пер} \rangle \mid \langle \text{пер} \rangle \langle \text{зн} \rangle \langle \text{выр} \rangle$

$\langle \text{зн} \rangle ::= + \mid - \mid * \mid /$

$V = V_N \cup V_T$ - *обобщенный словарь*.

V^* - *цепочка символов (строка, слово) из обобщенного словаря*;

V_N^* - *цепочка символов (строка, слово) из нетерминального словаря*;

V_T^* - *цепочка символов (строка, слово) из терминального словаря*.

$\varepsilon \in V$ - *пустой символ*, входит в обобщенный словарь.

Строка α непосредственно порождает строку β и обозначается: $\alpha \Rightarrow \beta$,

если $\alpha = vxw$ $\beta = vyw$ и существует некоторое правило $p: x ::= y$,
где $v, w, \in V^*$, $x \in V_N$, $y = V^* \setminus \{\epsilon\}$

Строка α *порождает* строку β и обозначается $\alpha \Rightarrow^* \beta$, когда от строки α можно перейти к строке β с помощью последовательности непосредственных порождений.

Продолжая пример:

$\langle \text{прог} \rangle \Rightarrow \langle \text{оп} \rangle$; $\langle \text{прог} \rangle \Rightarrow \langle \text{оп} \rangle$; $\langle \text{оп} \rangle \Rightarrow \langle \text{пер} \rangle := \langle \text{выр} \rangle$; $\langle \text{оп} \rangle \Rightarrow^*$
 $a := b + c$; $c := a + b - c$;

Грамматика, использующая процедуры (непосредственного) порождения – порождающая грамматика.

Строка β *непосредственно свертывается* в строку α и обозначается: $\alpha \Leftarrow^* \beta$,

если $\alpha = vxw$ $\beta = vyw$ и существует некоторое правило $p: x ::= y$,
где $v, w, \in V^*$, $x \in V_N$, $y = V^* \setminus \{\epsilon\}$

Строка β *свертывается* в строку α и обозначается $\alpha \Leftarrow^* \beta$, когда от строки β можно перейти к строке α с помощью последовательности непосредственных свертываний.

Грамматика, использующая процедуры (непосредственного) свертывания – распознающая грамматика.

Строки символов из обобщенного словаря, получающиеся в процессе порождения или свертывания, называются **сентенциальными формами**.

Язык L , порождаемый данной грамматикой G – множество нетерминальных цепочек, порождаемых из начального нетерминального символа. Такие терминальные цепочки называются **предложениями** данного языка.

$$L(G) = \{ x \in V_N^* \mid S \Rightarrow^* x \}$$

Аналогично можно определить язык L через свертывание.

$$L(G) = \{ x \in V_N^* \mid S \Leftarrow^* x \}$$

Замечание. Другой вариант метаязыка

вместо $::=$ используется стрелка \rightarrow , терминальные символы записываются маленькими (строчными) буквами, а нетерминальные – большими (прописными) буквами.

Такой вариант мета языка чаще используется в математической литературе. Первый предпочитают использовать в литературе для программистов. Так что мы будем пользоваться и тем и другим вариантами...

7.2. Деревья вывода

Порождение и свертывание можно также представлять с помощью *деревьев вывода*.

Пусть дана грамматика.

$I \rightarrow T$

$I \rightarrow I + T$

$I \rightarrow I - T$

$T \rightarrow M$

$T \rightarrow T * M$

$T \rightarrow T / M$

$M \rightarrow (I)$

$M \rightarrow K$

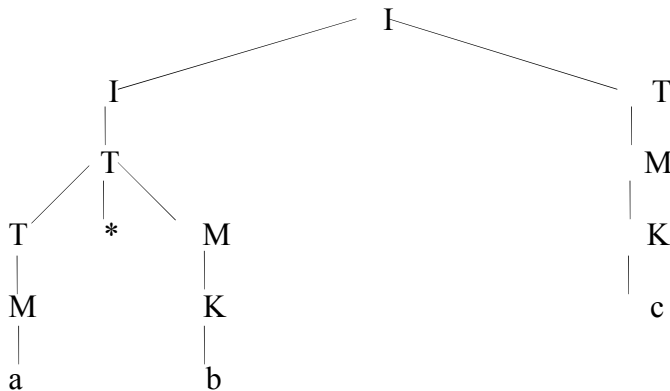
$K \rightarrow a$

$K \rightarrow b$

$K \rightarrow c$

Построим дерево вывода.

Для предложения $a * b + c$ дерево вывода будет:



Этот же результат можно получить и другим способом:

$I \rightarrow I + I$

$I \rightarrow I - I$

$I \rightarrow I * I$

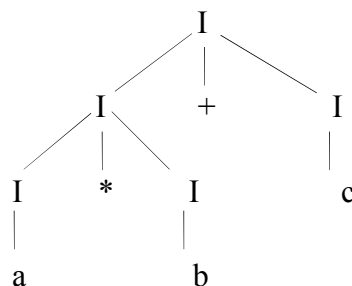
$I \rightarrow I / I$

$I \rightarrow (I)$

$I \rightarrow a$

$I \rightarrow b$

$I \rightarrow c$



7.3. Классификация языков по Хомскому

Н. Хомский предложил подразделять формальные грамматики, как и порождаемые ими языки на четыре типа.

Тип 0 - формальная грамматика, на правила которой не накладывается никаких ограничений. Для исследования они интересны не представляют – поскольку режим «делай, что хочешь» для математики и для практики редко представляют интерес.

Тип 1. К этому типу относятся грамматики, правила которых имеют вид:

$v\alpha w ::= v\beta w$, где

$v, w \in V^*$ - произвольные цепочки символов, возможно пустые;

$\alpha \in V_N$ - нетерминальный символ;

$\beta \in V^* \setminus \{\epsilon\}$ [иногда $\beta \in V^*$].

[$\beta \in V^*$].

Эти грамматики еще называют **контекстно-зависимыми** или **КЗ-грамматиками**.

Здесь строка α заменяется на строку β в рамках какого-то контекста. Часто используется на практике подмножество таких грамматик, называемое **грамматиками непосредственных составляющих**:

$vAw ::= v\alpha w$, где $v, w, \alpha \in V^*$, $A \in V_N$

При этом часто рассматриваются **неукорачивающиеся** грамматики (что обеспечивается непустой цепочкой β).

Тип 2. К этому типу относятся грамматики, правила которых имеют вид:

$\alpha ::= \beta$

$\alpha \in V_N$ - нетерминальный символ;

$\beta \in V^* \setminus \{\varepsilon\}$:

Здесь также представляют наибольший интерес грамматики непосредственных составляющих.

Такие грамматики называются **контекстно-свободными** граммами или **КС-граммами**.

Языки программирования большей частью описываются граммами этого типа.

Туп 3 . К этому типу относятся грамматики, правила которых имеют один из двух видов:

($A := Bc$)

($A := b$)

где $A, B, C \in V_N$; $b, c \in V_T$;

Это так называемый *леворекурсивный вариант*. В качестве альтернативы возможен и праворекурсивный вариант:

($A := cB$)

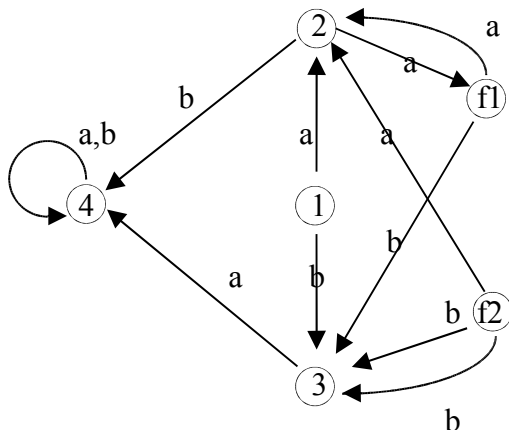
($A := b$)

Такие грамматики называют **регулярными** или **автоматными граммами**. Лексический анализ в компиляторе обычно наиболее целесообразно описывать с помощью этих грамматик.

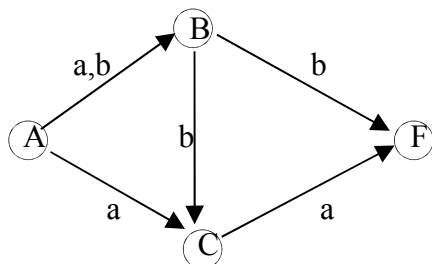
7.4. Распознающие автоматы

Распознающим называется автомат Мура с множеством выделенных состояний, называемых конечными. Говорят, что автомат распознает входное слово, если, начав свою работу в одном из начальных состояний, он заканчивает ее в одном из конечных.

Пример: Автомат, распознающий слова содержащие попарное вхождение букв a и b , вроде $aabbbbbaa$. $f1, f2$ - конечные состояния



Распознающий автомат – это, как правило, **недетерминированный частичный автомат**. То есть по одному и тому же сигналу можно перейти в различные состояния, а в некоторых состояниях нет перехода для ряда входных сигналов.



	0	0	0	1
	A	B	C	F
a	B,C		F	
b	B	C,F		

Кстати, строка, приписывающая состояниям выходные сигналы совсем не обязательна.

Представление этого автомата с помощью автоматной грамматики:

$A \rightarrow aB \mid bB \mid aC$

$B \rightarrow bC \mid b$

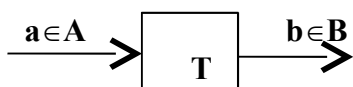
$C \rightarrow a$

Это праворекурсивная автоматная грамматика.

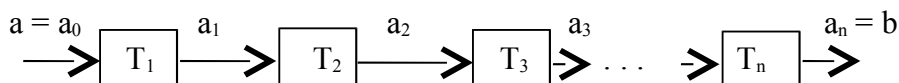
7.5. Понятие транслятора

Транслятор - программа или устройство, переводящее входную строку **a** языка **A** во выходную строку **b** языка **B** с сохранением смысла.

Это нестрогое определение, поскольку «сохранение смысла» можно понимать весьма различно.



Для того, чтобы облегчить переход от входного языка к выходному, а также с целью упростить оптимизацию, процесс трансляции часто разбивают на этапы, с трансляцией на промежуточные языки. Такие трансляторы называются *многопроходными*.



По типу трансляции трансляторы подразделяются на *компиляторы* и *интерпретаторы*. Компиляторы осуществляют перевод всего текста до начала выполнения (вычисления).

Интерпретатор транслирует исходный текст порциями. Он позволяет получать первые результаты уже на самых первых шагах обработки.

Интерпретатор обычно проще компилятора с аналогичного языка раз 10 – 100, но примерно во столько же раз дольше идет обработка и требуются большие машинные ресурсы на этапе выполнения.

Компилятор и интерпретатор дополняют друг друга и каждый хорош на своем месте.

Самыми широко известными примерами интерпретаторов, кроме интерпретаторов Бейсика, служат операционные системы. Особенно это наглядно и многообразно представлено в ОС UNIX.

По уровню транслируемого языка интерпретаторы подразделяются на собственно интерпретаторы и *ассемблеры*.

Ассемблеры – это машинно-зависимые языки (низкого уровня). Исходный текст ассемблера, а более строго – *макроассемблера* - состоит из команд и *макрокоманд*. Макрокомандам соответствуют настраиваемые заготовки на языке ассемблера - *макроописания*, которые после необходимых настроек вставляются в текст программы.

Главная особенность макроассемблеров – это преобразование программного текста (текстовая замена) до начала трансляции – *претрансляция*. Эту функцию выполняет *препроцессор*.

Ассемблеры позволяют использовать преимущества и особенности конкретной архитектуры. С другой стороны ассемблеры привязаны к архитектуре.

7.6. Основные функции компилятора.

Лексический анализ

1. *Лексический анализ* - приведение к некоторому стандартному виду ;
2. *Синтаксический анализ* - грамматический разбор ;
3. *Семантический анализ* - смысловой анализ;
4. *Генерация выходного текста*.

Лексический анализ выявляет *лексемы* - словарные единицы.

Основные функции лексического анализа:

1. выделение служебных слов языка (**begin, while, for, ...**);
2. обработка численных констант;
3. выделение идентификаторов;
4. выделение сложных символов (**:= <=>**);
5. внесение исправлений;
6. устранение различий устройств ввода;
7. устранение особенностей использования алфавитов, кодов.

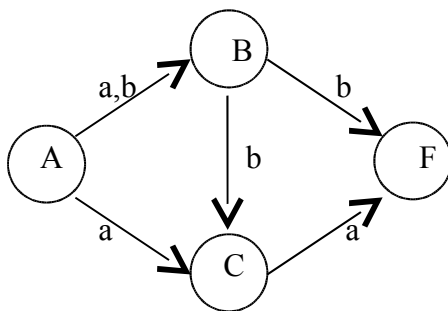
7.7. Переход от недетерминированного распознающего автомата к детерминированному

Состояния автомата и совокупности состояний, в который автомат переходит, объявляются множествами. Каждое из этих множеств становится состоянием нового детерминированного автомата. Переход из состояния, содержащего множество элементов, будет в состояние-множество, составленное из всех состояний, в которые в исходном

автомате осуществлялись переходы. Заметим, что пустые клеточки дают состояние - пустое множество.

	A	B	C	F	
a	B,C		F		$A \rightarrow aB \mid bB \mid aC$
b	B	C,F			$B \rightarrow bC \mid b$
					$C \rightarrow a$

	{A}	{B,C}	{B}	{F}	{CF}	{ }
a	{B,C}	{F}	{ }	{ }	{F}	{ }
b	{B}	{C,F}	{C,F}	{ }	{ }	{ }



7.8. Переход от праволинейной грамматики к автоматной

Праволинейная грамматика - грамматика с правилами вида:

$A \rightarrow \alpha$

$A \rightarrow \alpha B$

где $A, B \in V_N$, $\alpha \in V_T^*$

То есть это такая КС-грамматика, где вначале идет любое количество терминальных символов, а в конце возможен один нетерминальный символ

Пример:

Дана праволинейная грамматика:

1. $S \rightarrow aA$
2. $S \rightarrow bc$
3. $S \rightarrow A$
4. $A \rightarrow abbS$
5. $A \rightarrow cA$
6. $A \rightarrow \epsilon$

Правила 2, 3, 4 – нарушают требования к автоматным грамматикам.

Их можно последовательно заменить совокупностями автоматных правил.

$$\left\{ \begin{array}{ll} 4a & A \rightarrow a\langle bbS \rangle \\ 4b & \langle bbS \rangle \rightarrow b\langle bS \rangle \\ 4c & \langle bS \rangle \rightarrow bS \end{array} \right\} \text{ правило 4.}$$

$$\left\{ \begin{array}{l} 2a \ S \rightarrow b\langle cE \rangle \\ 2b \ \langle cE \rangle \rightarrow cE \\ 2c \ E \rightarrow \varepsilon \end{array} \right\} \quad \text{правило 2.}$$

$$\left\{ \begin{array}{l} 3a \ S \rightarrow a\langle bbS \rangle \\ 3b \ S \rightarrow cA \\ 3c \ S \rightarrow \varepsilon \end{array} \right\} \quad \text{правило 3}$$

7.9. LEX

lex и **уасс** - программы, содержащие средства для написания компилятора.

lex – программа (в терминах UNIX – команда) лексического анализа облегчает задачу выделения лексем.

уасс - программа синтаксического анализа.

Структура **lex** – программ:

```
%{ Вставка фрагмента программы на Си
}%
Раздел деклараций : имя_значение.
%%
Раздел правил : шаблон_действие.
%%
Пользовательский код.
Раздел деклараций:
    %token лексемы
Раздел правил:
нетерминал: | цепочка символов { код на Си }
                ;
%%
start : 'x'  lettera 'y' lettera '\n' { (printf("Ok\n")); }
                ;
lettera :      'z' letterb
                | 'Z'
                ;
letterb :      ',' 'z' letterb
                | ',' 'Z'
                ;
```

Пример 1:

```
%%
yyerror( str )
char *str;
{ printf( "error: %s",str); }
yylex()
{
int c=getchar();
return(c);
}
main()
{ yyparse();}
```

_____prog.y

```

%token X Y Z P
%%
text :      start
      | text start
start :    X lettera Y lettera ( printf("Ok"); )
;
lettera :  Z
      | Z P lettera
;
%%
yyerror( str )
char *str;
{ printf( "error: %s",str); }
_____prog.1
%{
#include "y.tab.h"
%}
%%
x      {return(X);}
y      {return(Y);}
z      {return(Z);}
[.]    {return(P);}
.      {return(yytext[0]);}
%%
main()
{ uuparse(); }

```

Для выполнения необходимы следующие действия :

```

$ yacc -d prog.y
# генерируется y.tab.c, содержащий основную программу
# по -d создается y.tab.h, в котором описываются макросы X Y Z P
$ lex prog.1
# создается lex.yy.c с функцией yylex - распознаватель лексем
# используется в функции uuparse
$ cc -o prog y.tab.c lex.yy.c
$ prog

```

Пример 2:

```

digit [0-9]
letter [a-z A-Z]
%%
{digit}+_ {printf("const\n");
return (const);}
{letter}({letter}|{digit})* {printf("var\n");return(var);}
"+ | "-" | "*" | "/" {printf("zn\n");
return(zn);}
"=" _ {printf("eq\n");
return(eq);}

```

Если ввести `a1=a1+c3-13` будет var

```

eq
var
zn
var
zn

```


7.10. Детерминированные автоматы с магазинной памятью (МП-автоматы)

Есть «промежуточная» математическая модель между автоматами и контекстно-свободными грамматиками – автомат с магазинной памятью. (МП-автомат).

Существует достаточно распространенная задача – задача определения парности скобок. Однако ее нельзя представить автоматной грамматикой.

Соответствующая грамматика может выглядеть следующим образом:

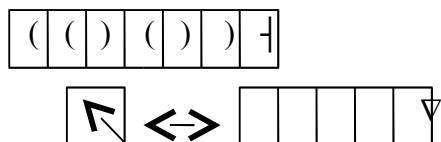
$S \rightarrow (S)$

$S \rightarrow SS$

$S \rightarrow \varepsilon$

МП-автомат состоит из входной ленты, в ячейках которой записывается анализируемая строка (\downarrow -конец строки), устройства управления и разбитого на ячейки магазина (стека). ∇ - символ пустого магазина. Устройство управления автомата может "помнить" состояние ($S1 \dots$).

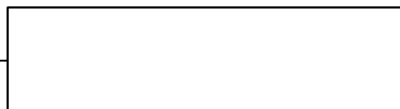
Требуется распознать: $((()())\downarrow)$



Работу автомата можно описать программой.

S1	X	∇
($\downarrow X$ \rightarrow	$\downarrow X$ \rightarrow
)	$\uparrow \rightarrow$	—
\downarrow	—	+

S1



Здесь и далее используются обозначения:

$\downarrow \alpha$ - поместить строку α в вершину магазина.

$\times \alpha$ - заменить верхний символ магазина на строку α .

$\uparrow \alpha$ - убрать символ из вершины магазина.

\rightarrow - сдвинуться на шаг вправо по входной строке.

$> - <$ - стоять на месте.

— - отвергнуть.

— - принять.

S - State - состояние МП-автомата (на каждое состояние своя таблица, здесь одно состояние S1).

$\nabla [S1] ((()())\downarrow)$
 \uparrow

$$\begin{array}{c}
 x\nabla [S1] (()) \downarrow \\
 \uparrow \\
 xx\nabla [S1] (()) \downarrow \\
 \uparrow \\
 x\nabla [S1] (()) \downarrow \\
 \uparrow \\
 xx\nabla [S1] (()) \downarrow \\
 \uparrow \\
 x\nabla [S1] (()) \downarrow \\
 \uparrow \\
 \nabla [S1] (()) \downarrow \\
 \uparrow
 \end{array}$$

Задача распознавания вложенных скобок "типа матрешка" сложнее и для ее распознавания требуется МП-автомат с двумя состояниями:

(())

S ₁	X	∇
(S ₁ ↓ X →	S ₁ ↓ X →
)	S ₂ ↑→	—
↓	—	+

S ₂	X	∇
(—	—
)	S ₂ ↑→	—
↓	—	+

При встрече первой закрывающей скобки МП автомат меняет состояние S₁ на состояние S₂.

7.11. Транслирующие грамматики

В этих грамматиках присутствует элемент аттракциона - транслирующая грамматика не только анализирует входное слово: но и транслирует его. В большинстве практических случаев эти процессы разделяют, поэтому-то такие грамматики можно рассматривать как некий казус.

- | | |
|---------------|------------------|
| 1. E → E + T. | 1. E → E + T{+}. |
| 2. E → T. | 2. E → T. |
| 3. T → T * P. | 3. T → T * P{*} |
| 4. T → P. | 4. T → P. |
| 5. P → (E). | 5. P → (E). |
| 6. P → a. | 6. P → a{a}. |
| 7. P → b. | 7. P → b{b}. |
| 8. P → c. | 8. P → c{c}. |

Проанализируем строку
(a + b) * c

1. E ⇒ T ⇒ T * P ⇒ P * P ⇒ (E) * P ⇒ (E + T) * P.
E ⇒ T ⇒ T * P{*} ⇒ P * P{*} ⇒

$$(E) * P\{*\} \Rightarrow (E + T\{+\}) * P\{*\} \Rightarrow (a\{a\} + b\{b\}) * c\{c\}\{*\}$$

Если выделить символы, заключенные в фигурные скобки, то получится исходное выражение, оттранслированное в постфиксную запись.

$$ab + c *$$

7.12. s и q - грамматики

s-грамматикой будем называть такую контекстно-свободную грамматику, правые части правил, которой начинаются с терминальных символов, причем для одного и того же левого символа правые части начинаются с разных символов.

Не s-грамматика :

$S \rightarrow aT$ - начинается с нетерминального. $T \rightarrow bT$.

$S \rightarrow TbS$

$T \rightarrow bT$

Аналогичная s-грамматика (распознает тоже)

:

$S \rightarrow abR$

$S \rightarrow bRbS$

$R \rightarrow a$

$R \rightarrow bR$

q-грамматика отличается от s-грамматики наличием аннулирующего правила (в правой части есть пустой символ) $\alpha \Rightarrow \varepsilon$.

1. $S \rightarrow aAS$

2. $S \rightarrow b$

3. $A \rightarrow cAS$

4. $A \rightarrow \varepsilon$

Из-за аннулирующих правил для q-грамматики вводится понятие следующего символа. N (A) - множество терминальных следующих (Next) за A символов.

В данном случае за A могут следовать a или b - {a,b}.

$$S \Rightarrow aAS \Rightarrow aAaAS \Rightarrow aAaAb$$

$E(1) = \{a\}$ - множество выбора для первого правила.

$E(2) = \{b\}$

$E(3) = \{c\}$

$E(4) = N(A) = \{a,b\}$

Данная грамматика может быть распознана МП-автоматом, в который добавлена операция замены α . В этом случае автомат начинает работать с непустым стеком.

	S	A	∇
a	1/ AS →	4/ ↑ >--<	—
b	2/ ↑→	4/ ↑ >--<	—
c	—	3/ AS →	—
⊥	—	—	+

7.13. LL(1) - грамматики. (left - leftmost)

LL(1) - грамматики относятся к нисходящим грамматикам (сверху - вниз).

Они отличаются от q-грамматик тем, что правые части могут начинаться с нетерминальных символов, но таких, которые после подстановок терминальных символов обеспечивают однозначность выбора грамматических правил.

В LL(1) - грамматиках разворачиваются самые левые нетерминальные символы сентенциальной формы и анализируется очередной самый левый терминальный входной строки. Возможен анализ k самых левых символов входной строки, Тогда грамматику называют LL(k) - грамматикой. Но, поскольку грамматики LL(k) и LL(1) эквивалентны в плане порождаемых языков, остановимся на рассмотрении только последней.

$F(\alpha)$ - множество терминальных символов, стоящих первыми (First)) в цепочках, выводимых из строки α .

$N(A)$ - множество терминальных символов, следующих (Next) в цепочках за данным нетерминальным символом A.

Множество выбора для каждого правила формируется с учетом множества первых и множества следующих символов.

LL(1) - это такая грамматика, у которой для правил с одинаковыми левыми частями множества выбора не пересекаются.

1. $S \rightarrow AbB$ $E(1) = F(AbB) = \{a, b, c, e\}$
2. $S \rightarrow d$ $E(2) = \{d\}$
3. $A \rightarrow CAb$ $E(3) = F(CAb) = \{a, e\}$
4. $A \rightarrow B$ $E(4) = F(B) \cup N(A) = \{c\} \cup \{b\} = \{c, b\}$
5. $B \rightarrow cSd$ $E(5) = F(cSd) = \{c\}$
6. $B \rightarrow \varepsilon$ $E(6) = F(\varepsilon) \cup N(B) = \{\emptyset\} \cup \{b, d, \vdash\}$
7. $C \rightarrow a$ $E(7) = \{a\}$
8. $C \rightarrow ed$ $E(8) = \{e\}$

	S	A	B	C	b	D	∇
a							
b							
c							
d							
e							

↓			6	>			+
			<				

7.14. Метод рекурсивного спуска

Метод рекурсивного спуска позволяет писать программы синтаксического анализа на языке, допускающем рекурсию, прямо по грамматическим правилам. Это на практике самый простой и самый любимый народом метод написания синтаксических анализаторов.

Пусть дана грамматика:

1. $S \rightarrow aAS$ $E(1) = \{a\}$
2. $S \rightarrow b$ $E(2) = \{b\}$
3. $A \rightarrow cASb$ $E(3) = \{c\}$
4. $A \rightarrow \varepsilon$ $E(4) = N(A) = \{a, b\}$

Программа на некотором паскале-подобном языке будет:

```
program descent;
var ch:char;
begin
  read(ch); {Встать на начало анализируемого текста}
  S;
  if ch<>'-' then - else +; {- и + здесь следует понимать как успешное или неуспешное
  завершение}
end.
```

```
procedure s;
begin
  case ch of
    'a':p1;
    'b':p2;
    'c', '↓': - ;
  end;
```

```
procedure p1;
begin
  read(ch);
  a;
  S;
end;
```

```
procedure p3;
begin
  read(ch);
  a;
  S;
  read(ch);
  if ch<>'b' then -;
end;
```

```
procedure a;
begin
  case ch of
    'a', 'b':p4;
    'c':p3;
    '↓': - ;
  end;
```

```
procedure p2;
begin
  read(ch);
end;
```

```
procedure p4;
begin
end;
```

7.15. LR - грамматики (left - rightmost)

Эти грамматики относятся к восходящим грамматикам (снизу - вверх).

В LR- грамматиках сворачиваются самые правые части правил для самых левых нетерминальных символов и анализируется очередной самый правый символ свертываемой части строки.

К числу LR- грамматик относятся *грамматики с предшествованием*.

Определим специальные отношения, которые могут возникать между символами стоящими рядом в сентенциальной форме. Здесь правые части грамматических правил будем называть *свертками*.

1. Если S_i и S_j - два рядом стоящие символа входят в одну свертку, то между ними существует отношение : $S_i = * S_j$ (назовем его *равно*);

$\underline{\quad \dots S_i S_j \dots \quad}$

Пример. В сентенциальной форме $AbC\underline{dE}fg$ при наличии правила $K \rightarrow CdE$, существуют отношения

$C = * d$, $d = * E$

2. Если S_i и S_j два рядом стоящие символа и с S_j начинается какая-то свертка, то между ними существует отношение: $S_i < * \cdot S_j$;

$S_i \underline{\quad S_j \dots \quad}$

Пример. В сентенциальной форме $AbC\underline{dE}fg$ при наличии правила $L \rightarrow dE$

Существует отношение

$C < * d$

3. а) Если S_i и S_j два рядом стоящие символа и S_i самый правый символ в свертке, то между ними существует отношение : $S_i * > S_j$;

$\underline{\quad \dots S_i S_j \quad}$

Пример. В сентенциальной форме $AbC\underline{dE}fg$ при наличии правила $L \rightarrow dE$

существует отношение

$E * > f$

б) Если S_i и S_j два рядом стоящие символа и S_i самый правый символ в одной свертке, а S_j - самый левый в другой, то между ними существует отношение : $S_i * > S_j$;

$\underline{\quad \dots S_i \quad S_j \dots \quad}$

Пример. . В сентенциальной форме $AbC\underline{dE}fg$ при наличии правил $L \rightarrow dE$ и $M \rightarrow fg$

существует отношение $E * > f$

Для удобства дальнейшей работы составим таблицу *левых и правых* символов, которые могут оказаться в подставленных вместо этих символов цепочках на месте данных нетерминальных символов. Таблица строится на основе анализа грамматических правил.

$A \rightarrow BC$

$B \rightarrow IC$

$B \rightarrow CA$

$C \rightarrow d$

	левые	правые
A	B l C d	C d
B	l C d	C A d
C	d	d

Выявим отношения:

$$B =^* C \quad l =^* C \quad C =^* A$$

$B <^* \cdot L(C)$ (множество левых для C)

$$B <^* \cdot L(C) = \{d\}$$

$$l <^* \cdot L(C)$$

$$C <^* \cdot L(C) = \{B, l, C, d\}$$

$$\{C, A, d\} = \Pi(B) \cdot ^* > C$$

$$0 = \Pi(l) \cdot ^* > C$$

$$\{d\} = \Pi(C) \cdot ^* > C \quad (3a)$$

$$\{C, A, d\} = \Pi(B) \cdot ^* > L(C) = \{d\} \quad (3b)$$

$$\{d\} = \Pi(C) \cdot ^* > L(A) = \{B, l, C, d\}$$

И сведем их в таблицу - матрицу предшествования.

	A	B	C	d	l
A			$\cdot ^* >$	$\cdot ^* >$	
B			$=^*$	$<^* \cdot$	
C	$=^* \cdot$	$<^*$	$^* > <^*$	$^* > <^*$	$<^*$
d	$^* >$	$^* >$	$^* >$	$\cdot ^* >$	$^* >$
l			$=^* \cdot$	$<^* \cdot$	

Грамматика называется **грамматикой с предшествованиями**, если между любыми двумя символами, стоящими рядом в сентенциальной форме, существует строго одно отношение предшествования.

Использование матриц с предшествованием.

$$S \rightarrow BC$$

$$B \rightarrow Axz$$

$$C \rightarrow xx$$

$$A \rightarrow xy$$

	A	B	C	x	y	z	\vdash
A				$=^* \cdot$			
B			$=^* \cdot$	$<^* \cdot$			
C							$\cdot ^* >$
x				$=^* \cdot$	$=^* \cdot$	$=^* \cdot$	$\cdot ^* >$
y				$\cdot ^* >$			
z				$\cdot ^* >$			
\vdash	$<^* \cdot$	$<^* \cdot$		$<^* \cdot$			

Считаем, что она построена.

Использование матрицы:

хухххх - проставляем все значки

$$\vdash <^* x =^* y \cdot ^* > x =^* z \cdot ^* > x =^* x \cdot ^* > \vdash$$

$$\begin{aligned} & \vdash \langle * A \quad \vdash x \vdash y \cdot * \rangle x \vdash x \cdot * \rangle \vdash \\ & \vdash \langle * B \langle * x = * x * \rangle \vdash \\ & \vdash \langle * B = * C * \rangle \vdash \\ & \vdash S \vdash \end{aligned}$$

Алгоритм распознавания:

1. Между символами строки вставляются отношения предшествования.
5. Строка просматривается слева направо до первого символа $\cdot * \rangle$, после этого просмотр идет в обратном направлении до первого встречаемого символа $* \langle \cdot$ - между этими символами и находится свертка, то есть правая часть правила, которая заменяется левой.
3. Восстанавливаются отношения предшествования.
4. Возвращение к первому пункту. Процесс продолжается до получения начального нетерминального символа. Если этот процесс не завершится успешно, строка не принадлежит данной грамматике.

У этого метода есть минусы:

1. Далеко не во всех случаях удастся построить грамматику с предшествованиями.
2. На практике символов может быть много сотен и в результате получается слабозаполненная матрица большой размерности.

7.16. Функции предшествования

Этот интересный метод придумал Р.Флойд – автор многих остроумных решений в программировании. Вместо матрицы строятся две специальные функции f и g , такие что:

1. Если $S_i \cdot * \rangle S_j \Rightarrow f(S_i) > g(S_j)$.
2. Если $S_i \langle * S_j \Rightarrow f(S_i) < g(S_j)$.
3. Если $S_i = * S_j \Rightarrow f(S_i) = g(S_j)$.

Тогда, вместо поиска с помощью матрицы отношения предшествования между символами, просто происходит сравнение числовых значений соответствующих функций на больше меньше равно.

Построение функций предшествования:

0. Строится матрица предшествования и начальные значения функций принимаются равными единице: $f(S_i) = g(S_j) = 1$.
1. Матрица просматривается по строкам в поисках отношений $\cdot * \rangle$ и, если $f(S_i) > g(S_j)$, то идем дальше, если же $S_i \cdot * \rangle S_j$, а $f(S_i) \leq g(S_j)$, то увеличиваем значение $f(S_i)$ - $f(S_i) = g(S_j) + 1$.
2. Матрица просматривается по столбцам в поисках отношений $\langle * \cdot$ и, если $f(S_i) < g(S_j)$, то идем дальше, если же $S_i \langle * S_j$, а $f(S_i) \geq g(S_j)$, то увеличиваем значение $g(S_j)$ - $g(S_j) = f(S_i) + 1$.
3. Матрица просматривается в поисках отношений $= *$ и, если $f(S_i) = g(S_j)$, то идем дальше, если $S_i = * S_j$, а $f(S_i) \neq g(S_j)$, то выравниваем значения функций путем увеличения меньшего из значений до большего - $f(S_i) = g(S_j) = \max[f(S_i), g(S_j)]$.
4. Возвращение к первому пункту.
Повторять до тех пор, пока рост функций не прекратится (или когда значение одной из функций не превысит 2^n , где n - размерность матрицы - в этом случае алгоритм не сходится).

Пример.

Уточняемые значения функций будем располагать левее строк и выше столбцов с соответствующими символами.

		5		4		3
	$g(S_j)$	4		4		3
$f(S_i) \diagdown$	2	2	3		2	
	1	1	1	1	1	1
		A	B	C	D	E
3 2 1	A		$\cdot >$	$< \cdot$	$\cdot >$	$\dot{=}$
1	B		$\dot{=}$			
4 3 1	C	$< \cdot$				
6 5 3 2 1	D	$\cdot >$				
2 1	E		\div		$\cdot >$	

	A	B	C	D	E
f	3	2	4	6	2
g	5	2	4	1	3

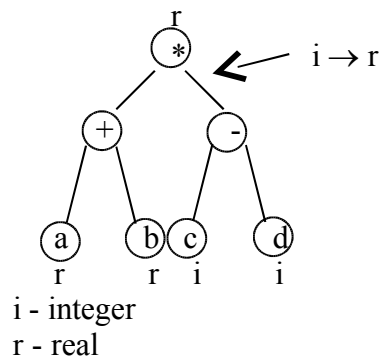
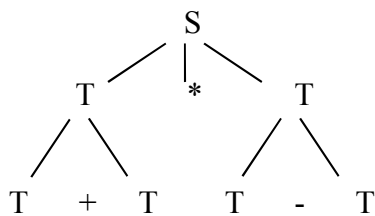
1. Алгоритм не всегда сходится (не всегда приводит к построению функций).
2. При переходе к функциям происходит «незаконное доопределение» матрицы. То есть как бы появляются отношения предшествования между парами символов, для которых в исходной матрице отношение отсутствовало.

Атрибутные грамматики, впервые предложенные Д. Кнудом, в идеале призваны контролировать *смысловую корректность* формальных грамматик. Смысл обычно задается операционно или декларативно.

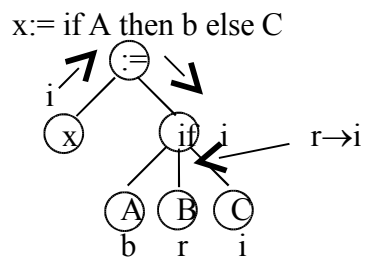
В примерах грамматики с *синтезируемыми атрибутами* (восходящие грамматики) и *наследуемыми атрибутами* (нисходящие грамматики).

$$\begin{aligned} S &\rightarrow T \mid T * T \mid (T) \\ T &\rightarrow T + T \mid T * T \mid a \mid b \mid c \mid d \end{aligned}$$

— 89 —



Пример наследуемых атрибутов приведен на следующем рисунке:



7.18. YACC

yacc - программа синтаксического анализа (**y**et **a**nother **c**ompile of **c**ompile**s**). Также как и **lex** – ‘она начально была написана, как команда для ОС UNIX. Часто эти команды используются совместно.

Пример.

```
% token CONST VAR ZN EQ
%%
pr: VAR EQ vyr ((выражение)) {printf("prog\n");}
vyr: CONST ZN vyr {printf("CONST ZN vyr\n");}
    VAR ZN vyr {printf("VAR ZN vyr\n");}
    CONST {printf("CONST\n");}
    VAR {printf("VAR\n");}
```

Работа над примером : a1 = a1 + c3 - 13

выдала :

CONST

VAR ZN vyr

VAR ZN vyr

prog

Для обработки ошибок есть стандартная функция ERROR.

flex - fast lexical analyzer generator

раздел деклараций

%%

раздел правил

%%

пользовательский код

Раздел деклараций :

имя значение

Раздел правил :

шаблон действие

Шаблоны :

x символ 'x'

. любой символ кроме '\n'

[abj-oZ] любой из 'a','b','Z' и диапазона от 'j' до 'o'

r* любое количество выражений "r"

r+ одно или более выражений "r"

r|s или "r", или "s"

(r|s)* любое количество выражений "r" или "s"

<<EOF>> конец файла

Пример :

```
int num_int=0;
digit [0-9]
%%
{digit}+"L" { num_int++;printf( "longconst\n"); }
%%
main()
{
  yylex();
  printf( "Number of long int is %d\n",num_int);
}
```

7.19. Область действия и передача параметров

Существует шесть основных способа передачи параметров :

1. **by value (значением)**. В вызываемой процедуре выделяется место (память) для параметров и туда помещаются их значения. Это самый аккуратный способ. Его еще можно назвать *самым математическим*. Изменения, которые претерпевает переданный параметр, ни как не повлияют на его значение в вызывающей программе. Это сродни тому, что от математики мы вправе ожидать естественного порядка вещей: вычисление функции не приводит к изменению значения аргумента.
2. **by result (результатом)**. Память для хранения значения параметра выделяется в вызывающей программе. Это достаточно экзотический способ. Он может иметь место, например, при вызове программы генерации случайных чисел.
3. **by value-result (значением-результатом)**. Сочетание первых двух способов.
4. **by reference (ссылкой)**. Память выделяется в вызывающей программе, а в качестве параметра передается ссылка (указатель) на эту память. Это наиболее часто используемый способ.
5. **by name (именем)** . При этом способе производится текстовая замена формального параметра фактически переданным. Этот способ прежде всего используется в различных претрансляторах и макроассемблерах.
6. **by stack (стеком)** Это "неклассический" способ, который получил распространение в связи с появлением "нестандартных" языков типа Форт.

Весьма условный (позаимствованный) пример. Их обычно приводят, чтобы поразить воображение.

Пример :

prog
B[1]:=1;

P(x); - фрагмент вызванной процедуры.
I:=1;

```

B[2]:=1;      x:=x+2;
I:=1;         B[I]:=10;
(*) P(B[I]);  I:=2;
  ^          x:=x+2;
  |
Фрагмент
вызывающей
программы.

```

Процедура вызывается в точке (*).

А поражает то, что при пяти основных способах передачи параметров получаются разные результаты вычислений:

Способ передачи	B[1]	B[2]
1	1	1
2	-	-
3	5	1
4	12	1
5	10	3

7.20. Генерация выходного текста.

Польская инверсная запись

При рассмотрении вопросов генерации выходного текста надо иметь в виду то, что реально выходной текст программы после трансляции - это, как правило, не выполняемый код, а некоторая промежуточная форма, поскольку программа в дальнейшем может быть загружена для выполнения в разные места памяти и т.д. С другой стороны, выполняемая программа (или программа в близком к такой форме виде) машинно-зависима, то есть использует конкретную систему команд и другие конкретные архитектурные особенности. Так что процесс генерации - трудоемкий процесс, требующий большой и весьма кропотливой работ. Это в значительной степени ремесло, а не наука, хотя есть и теоретические работы по этому вопросу. Интересно, что здесь заметный вклад внесли ученые женщины-программисты.

Поэтому рассмотрим лишь основную идею генерации и остановимся на использовании в вычислениях постфиксных представлений.

Генерация.

Здесь, лишь из соображений наглядной демонстрации идеи, в качестве выходной формы программы возьмем блок-схему.

Пусть транслируется условный оператор входного языка, имеющий вид:

IF (A - B) 10, 15, 20

Для данного входного языка определено, что

IF - служебное слово оператора условия.

A - B - арифметическое выражение.

10, 15, 20 - метки.

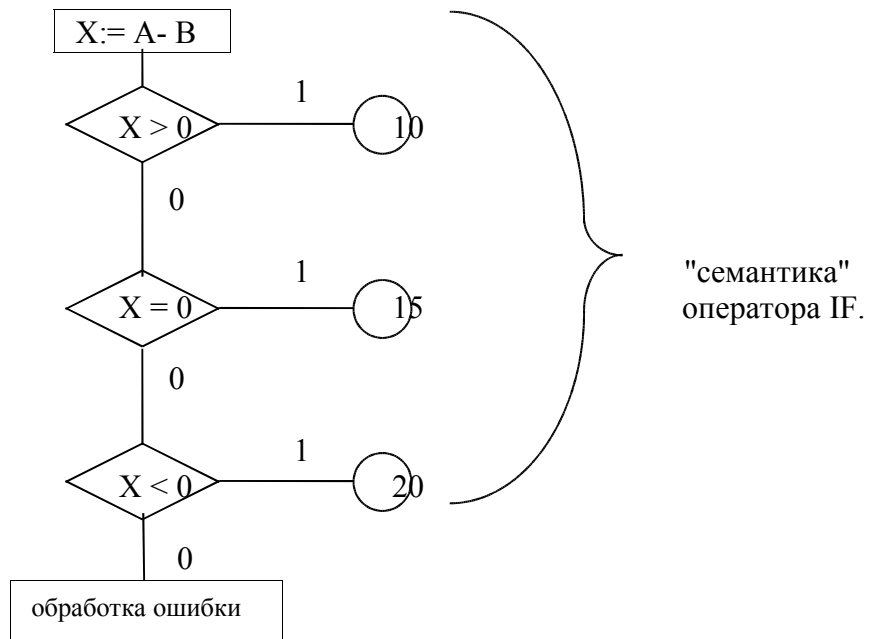
Работа оператора состоит в вычислении арифметического выражения

И сравнение полученного результата с нулем на больше, равно или меньше нуля.

В зависимости от трех возможных исходов сравнения осуществляется переход на соответствующую программную метку.

"Смыслу" оператора заранее поставлена в соответствие заготовка выходного кода - в данном случае - заготовка блок-схемы. При анализе содержания оператора извлекается

арифметическое выражение, требующее дальнейшей обработки и конкретные метки переходов.



Тут как раз хороший повод вернуться к вопросу трансляции арифметических выражений. Часто для них (и не только) выполняют преобразование в **польскую инверсную запись (ПОЛИЗ)**, что упрощает последующее выполнение-вычисление.

Вообще есть три способа записи операций:

1. Инфиксный: $a*b$ (например, $a + b$).
2. Префиксный: $*ab$ (например, $\Sigma(a,b)$).
3. Постфиксный: $ab*$ (например, a и b - просуммировать)

О третьем, постфиксном, способе и идет речь. Его еще называют

польской инверсной записью; в память о польском математике Яне Лукашевиче.

Преобразование выражений в ПОЛИЗ с легкой руки Э.Дейкстры (первой величины в области теоретического программирования) стали часто изображать в виде "железнодорожного разъезда".



Алгоритм преобразования арифметических выражений в ПОЛИЗ.

1. Поступающие на вход операнды сразу проходят на выход.
2. Поступающие на вход операторы сравниваются по приоритету. Если приоритет оператора на входе магазина больше, чем в вершущке магазина, то оператор со входа поступает в магазин (first in/last out). Если приоритет оператора на входе меньше или равен приоритету оператора в вершущке магазина, то оператор из вершущки магазина идет на

выход и сравнение повторяется. Эти процедуры выполняются до тех пор, пока не исчерпается строка.

Операции	Магазинный	Сравнительный
(∅	∞
:=	∅	∞
+ -	1	1
* /	2	2
(степень) ↑	3	3
)	-	∅

$y := a * (b + c) \uparrow e / (d - k)$

у сразу проходит на выход...

$yabc \xrightarrow{)} yabc + e \xrightarrow{/} yabc + e \uparrow * dk \xrightarrow{)} yabc + e \uparrow * dk - / :=$
1 23 2 32

При выполнении используется стек.

Операнды поступают в стек. Поступив на вход стека, оператор вытягивает из стека столько операндов, сколько ему нужно. Результат заталкивается в вершину стека.

$\xrightarrow{+} 2 \xrightarrow{\uparrow} 25 \xrightarrow{*} 3 \xrightarrow{-} 1 \xrightarrow{/} 25 \xrightarrow{:=}$
3 5 25 3 1 25
2 1 1 25 25 y
1 y y y y
y

7.21. Оптимизация программ

Если не принять специальных мер, в результате трансляции получаются программы, избыточные и по занимаемой памяти, и по вычислениям. Поэтому меры по оптимизации принимаются в практически используемых трансляторах.

Оптимизация на первых этапах (проходах) трансляции может быть эффективной потому, что программа в этот период компактна, хорошо обозрима, а главное – мобильна.

“Плюсы” оптимизации на последних этапах (проходах) очевидны – именно там аккумулируются результаты неоптимальных решений на всех предыдущих этапах трансляции.

Оптимизация на начальных этапах:

1. Предварительное вычисление выражений.

При данных $x := 2; y := 3;$

оператор $z := x + y + 10$

замена на $z := 15;$

2. Исключение невыполнимых ветвей. То есть тех ветвей, которые соответствуют невыполнимому сочетанию условий.

3. Выделение общих частей.

$a := (x + y) * z - 35;$ $b := ((x + y) * z) / a;$	заменяет на	$w := (x + y) * z;$ $a := w - 35;$ $b := w/a;$
---	-------------	--

4. *Вынесение за цикл.*

```
for i := 1 to 10 do begin
    x := x + i; P(x);
    k := b + c; /* выносится за цикл, т.к. не зависит
                  от параметра цикла */.
end;
```

Измененный вариант

```
k := b + c;
for i := 1 to 10 do begin
    x := x + i; P(x);
end;
```

5. *Вычисление логических выражений.*

$x \Rightarrow y \ \& \ (z = 5 \vee x \neq 5)$

Так, например, при ложном условии $x \Rightarrow y$ конъюнкция ложна вне зависимости от истинности условия в скобках. А само условие в скобках при истинности $z = 5$ истинно вне зависимости от выполнения второго скобочного условия.

6. *Изменение линейной последовательности команд* с целью оптимизации межрегистровых передач, обращений к памяти и т.п.

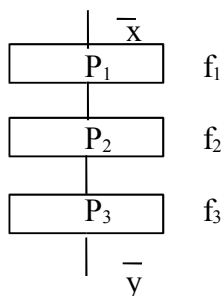
8. Функциональное программирование

Уместно упомянуть выдающуюся книгу П. Хендерсона “Функциональное программирование” [10].

Существует некоторая внешняя аналогия между процедурным и функциональным программированием.

Для решения задачи в *процедурном* программировании необходимо выполнить процедуру, которая в свою очередь обычно состоит из совокупности процедур.

Для решения задачи в *функциональном* программировании необходимо вычислить функцию, которая в свою очередь зависит от вычисления ряда входящих в нее функций.



$$\bar{y} = f_3(f_2(f_1(\bar{x})))$$

Процедурное программирование - это выполнение некоторых действий над памятью, в результате которых входные данные \bar{x} превращается в результирующие данные \bar{y} . В частном случае можно поставить в соответствие процедурам математические функции. Но,

главное, что в процедурном программировании термин **переменная** используется для обозначения *изменяемой константы*, не имеющей ничего общего с понятием переменной в традиционной математике.

Процедурное программирование - это "сплошные побочные эффекты".

Самой простой и фундаментальной иллюстрацией служит оператор присваивания:

$X := X + 1$, где правый X - это (относительный) адрес в памяти, где находится предыдущее значение X ... Или типичная процедура работы с массивом (матрицей) - в "вольном переводе" на математический язык, здесь "функция" меняет свой аргумент, который становится значением "функции".

Вместо циклов процедурного программирования (как раз отслеживающих значения изменяемых констант) в функциональном программировании используется *рекурсия*.

Процедурные языки можно считать машинно-зависимыми, поскольку они ориентированы на архитектуру машины Фон Неймана: *память - процессор*.

С точки зрения реализации, функциональному программированию присущи: представление программы в виде списковой структуры и вычисление в режиме интерпретации. (Поэтому тот же Э. Дейкстра назвал функциональное программирование самым извращенным использованием машины Фон-Неймана).

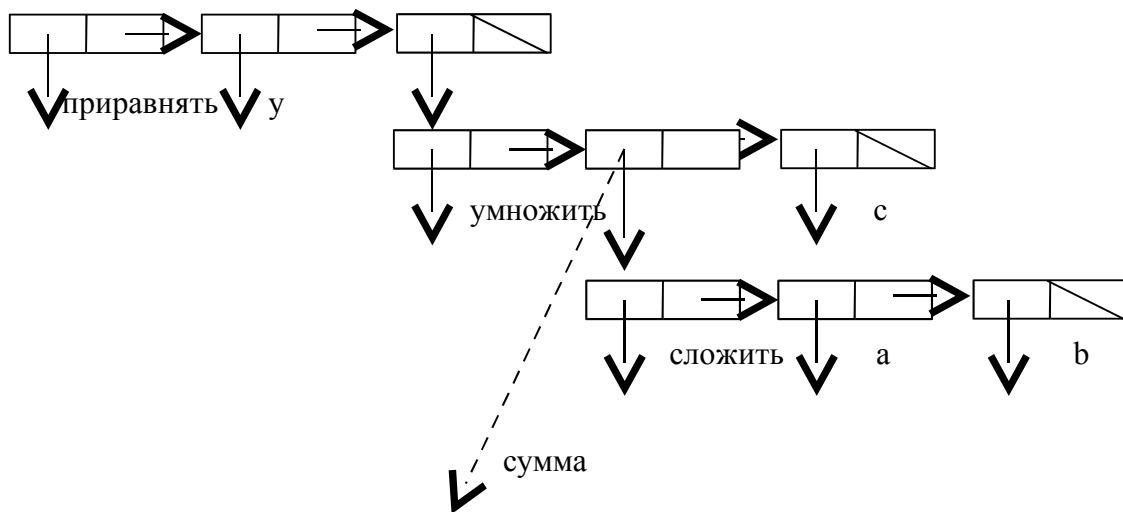
Пример: Пусть имеется оператор присваивания

$y := (a+b) * c$

и пусть ему соответствует функциональная запись

(приравнять y (умножить (сложить a b) c))

Списковая структура, представляющая эту функцию, будет:



Вычисление - это трансформация списка в режиме интерпретации. Вначале вычисляется функция сложения и ссылка на эту функцию заменяется ссылкой на полученную сумму. Затем вычисляется произведение и т.д.

Плюсы функционального программирования: большая гибкость, возможны и естественны параллельные вычисления. Нет "спрятанных состояний", а следовательно, нет побочных эффектов.

Минусы функционального программирования: Режим интерпретации в десятки раз снижает скорость вычисления; из-за необходимости хранить списковую структуры нерационально используется память.

Первым языком функционального программирования был язык LISP и многие базовые понятия этого языка стали классикой функционального программирования.

Базовые функции функционального программирования:

$\text{car}(x)$ - дает первый элемент списка x ;

$\text{cdr}(x)$ - хвост списка (список без первого элемента) ;

$\text{cons}(x,y)$ - добавляет элемент x к списку y ;

$\text{append}(x,y)$ - добавляет список y к списку x ;

Примеры.

Пусть $x = (1, 2, 3)$ $y = (4, 5)$

$\text{car}(x) = (1)$;

$\text{cdr}(x) = (2, 3)$;

$\text{cons}(x,y) = ((1, 2, 3), 4, 5)$;

$\text{append}(x,y) = (1, 2, 3, 4, 5)$.

Синтаксис LISP достаточно архаичный, поэтому воспользуемся способом записи функциональных программ с помощью специальной нотации, ставшей популярной с легкой руки Хендерсона.

Примеры.

Функция **дл** вычисления длины списка

$\text{дл}(x) \equiv \text{if}(x) = \text{nil} \text{ then } 0 \text{ else } \text{дл}(\text{cdr}(x)) + 1$

Вычислим функцию для конкретного списка.

$\text{дл}(A, B, C) = \text{дл}(B, C) + 1$

$\text{дл}(B, C) = \text{дл}(C) + 1$

$\text{дл}(C) = \text{дл}(\text{nil}) + 1$

$\text{дл}(\text{nil}) = 0$

"Пройдя" в обратном направлении можно получить числовое значение.

Обращение списка **обр**, то есть список A, B, C обратится в список C, B, A

$\text{обр}(x, y) \equiv \text{if } x = \text{nil} \text{ then } y \text{ else } \text{обр}(\text{cdr}(x), \text{cons}(\text{car}(x), y))$

Вычислим функцию для конкретного списка

$\text{обр}(A, B, C, \text{nil}) = \text{обр}((B, C), (A))$

$\text{обр}((B, C), (A)) = \text{обр}((C), (B, A))$

$\text{обр}((C), (B, A)) = \text{обр}((\text{nil}), C, B, A)$

Здесь использован популярный прием функционального программирования "сумка". Это позволяет получить результат сразу, без обратного прохода.

Функциональный язык Бэкуса.

LISP, созданный в начале 60-х годов, не единственный функциональный язык, хотя и самый распространенный.

Интересный функциональный язык РЕФАЛ был разработан в 70-х годах нашим соотечественником Турчиным и использовал математическую модель нормальных алгоритмов Маркова.

Но, пожалуй, наибольший резонанс у теоретиков программирования имел функциональный язык, предложенный Дж. Бэкусом 1979 году. Он был создан, одним из "отцов" фортрана не только как альтернатива Фортрану и всем прочим процедурным языкам, но в известной мере и как альтернатива LISP, синтаксис которого ориентировался на устаревшее представление об архитектуре компьютера, да и в области математической логики за тот период произошли заметные подвижки.

Следуя за Дж. Бэкусом сравним процедурный и функциональные стили программирования.

Пусть у нас есть фрагмент процедурной программы:

$c := 0$

for $i := 1$ to N do $c := c + a[i] * b[j]$

Беспристрастный, но строгий анализ показывает очевидные недостатки процедурного программирования:

1. Операторы работают с невидимыми значениями переменных.
2. Программа неиерархична (операторы одного уровня).
3. Программа динамична (чтобы её понять необходимо её выполнить).
4. Последовательно выполняются операции с отдельными элементами массива.
5. Часть данных находится в программе.
6. Программа называет свои операнды, предварительно их записав.
7. Отсутствует механизм сбора мусора - ненужные части программы продолжают находиться в памяти.

На языке, предложенном Бэкусом, та же самая задача решается проще:

$(/+)^{\circ}(\alpha^*)^{\circ}T :$

где

T – транспозиция (проектирование),

$^{\circ}$ - композиция,

$/$ - вставка,

α - применить ко всем (applay to all)

это *функциональные формы*, они оперируют функциями

$+$ и $*$ - традиционные функции.

Приведем пример выполнения этой программы:

$(/+)^{\circ}(\alpha^*)^{\circ}T : \langle \langle 1,2,3 \rangle, \langle 6,5,4 \rangle \rangle$

кортеж из двух кортежей

$(/+)^{\circ}(\alpha^*) : \langle \langle 1,6 \rangle, \langle 2,5 \rangle, \langle 3,4 \rangle \rangle$

$(/+): \langle 6,10,12 \rangle$

:28

Достоинства метода:

1. Нет невидимых данных.
2. Программа иерархична, т.к. есть не только функции, но и функциональные формы.
3. Статическое описание программы. Программа читается «за один проход».
4. Работаем сразу со всеми данными, а не с отдельными элементами.
5. В теле программы нет никаких данных.
6. В программе нет названий операндов.
7. Сбор мусора - программы и данные эволюционируют в процессе вычислений.

Принципиальное отличие функционального программирования от процедурного состоит в том, что функциональное программирование не требует от пользователя управления памятью.

9. Логическое программирование.

Язык Пролог

Логическое программирование идет дальше функционального. Здесь программист не только не занимается управлением памятью, но и не управляет вычислениями. (Для логической программы, например, нельзя нарисовать блок-схему).

Логическое программирование не является программированием в традиционном понимании этого слова, поскольку программист в данном случае пишет не программу-алгоритм, а логическую модель. Как правило, такая модель может быть использована для решения не одной, а ряда задач, определенного моделью круга.

Архитектура машины Фон-Неймана еще меньше приспособлена к специфике и потребностям логического программирования, чем функционального. То есть «эффективность вычислений» еще ниже.

Математическим фундаментом логического программирования служат аксиоматические теории. Например, как в случае Пролога – метод резолюции.

Язык Пролог (ПРОграммирование с помощью ЛОГики) создан А. Колмеройер 1970 году во Франции, распространен в Венгрии, Англии, Японии.

Программа на Прологе представляет из себя систему аксиом, представленных в виде не содержащих свободных переменных дизъюнктов. В Прологе используются только **хорновские дизъюнкты**, то есть дизъюнкты, в которых не больше одного положительного предиката. В Прологе их называют обычно *предложениями* или *клозами*.

То есть исходные аксиомы могут иметь в общем случае вид $A_1 \& A_2 \& \dots \& A_n \rightarrow B$

что при переводе в дизъюнкты будет $\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n \vee B$

Дизъюнкт, состоящий только из отрицательных предикатов - *вопрос*. А дизъюнкт, состоящий лишь из одного положительного предиката – *факт*.

Примеры программы на Прологе.

1. `append ([], L, L).`

2. `append ([x | L1], L2, [x | L3]) :- append (L1, L2, L3).`

Здесь `[]` выделяют список.

`|` - отделяет голову (первый элемент списка) от хвоста списка.

Добавим к предложениям, описывающим функцию `append`, вопрос: “Какой получится список при объединении списков `[a, b]` и `[c, d]`?”

3. `?-append ([a, b], [c, d], z).`

Выполнены:

2 – 3: `4: append ([a | b], [c, d], [a | z1]) :- append ([b], [c, d], z1).`

2 – 4: `5: append ([b | []], [c, d], [b | z2]) :- append ([], [c, d], z2).`

5 – 1: `6: append ([], [c, d], [c, d]).` □

В результате получим искомое `z`.

`z2 = [c, d]; z1 = [b | z2] = [b, c, d]`

`z = [a, z1] = [a, b, c, d].`

Даже на такой скромной модели можно решать не одну, а ряд задач. Например, вопрос 3`:

“Какой список надо добавить к `[a, b]`., чтобы получился список `[a, b, c, d]`.”?

3`. `?-append ([a, b], z, [a, b, c, d]).`

Обращение списка:

обращение (`[]`, `[]`).

обращение ([Y | T], L) :- обращение (T, Z), append (Z, [H], L).
?- обращение ([a, b, c], X).

Важной особенностью логического программирования то, что синтез конкретного алгоритма происходит

10. Объектно-ориентированное программирование

Появление объектно-ориентированного программирования (ООП) – самое значительное за последние лет двадцать продвижение в области методологии программирования.

Основная идея - прямое моделирование того мира, в котором предстоит решать задачи. Раньше программа воспринималась как процедура, состоящая в свою очередь из процедур (функция, имеющая аргументами функции). В ООП программа – это объект, состоящий в свою очередь из объектов.

Именно для удовлетворения потребностей прямого моделирования был создан язык **Smalltalk**, которому удалось, по сравнению с предшественниками, вроде *GPSS* и *Симулы*, совершить качественный скачок в этом направлении программирования. Сам по себе Smalltalk не получил большого распространения из-за весьма непривычного синтаксиса (и не менее экзотической терминологии), да и был он прежде всего все-таки экспериментальным языком.

Благодаря своим моделирующим возможностям (ООП) оказалось удобным для описания интерфейсов в интерактивных задачах. Во многих случаях ООП показало преимущества перед функциональным и логическим программированием. Это наводит на далеко идущие философские размышления... Вообще, первые же годы победного шествия (этот митинговый стиль достаточно точно отражает то, что реально имело место) методологии ООП дали большое количество прикладных методик и приемов, которые позволили более успешно решать многие задачи. Но, увы, методология ООП так же неадекватна Фон-Неймановской архитектуре современных компьютеров, как и функциональное, и логическое программирование.

Частичным решением этой проблемы (может быть самым экзотическим из реально на тот момент возможных) стало появление языка Си++, как расширение языка Си. С точки зрения теоретического программирования язык си – это Фортран 80-ых. (Против такого уничижительного определения не возражал и автор языка Си – Д. Ритчи). Этот язык, сочетающий в себе многие преимущества языка высокого уровня и ассемблера, дал программистам, по образному выражению некоторых, педаль газа, но заблокировал педаль тормоза. На Си компьютер может «мчаться» быстро, но рискованно. То есть Си, насаждая ссыльно-ассемблерное программирование, как бы имеет вектор в сторону, противоположную той, которая определяется теорией и методологией языков программирования.

Поэтому и Си++ - это довольно странное сочетание *некоторых* черт ООП и процедурного программирования.

Здесь мы обсудим лишь основные принципы ООП безотносительно конкретных языковых реализаций.

Основные принципы ООП.

1. Инкапсуляция данных (упрятывание данных, абстрактные типы данных).

То есть определение типов данных и возможных способов манипуляции ими. Классический пример – *стек*. Задается структура данных (например, одномерный массив) и процедуры помещений в стек, выталкивания из верхушки стека и проверка стека на пустоту. Другие способы доступа к информации стека запрещены. То есть, нельзя, например, обратиться напрямую к какому-то элементу соответствующего массива. Инкапсуляция, четко определяя границы объектов, что «укрупняет» систему и упрощает работу с ней. Впрочем, принципы

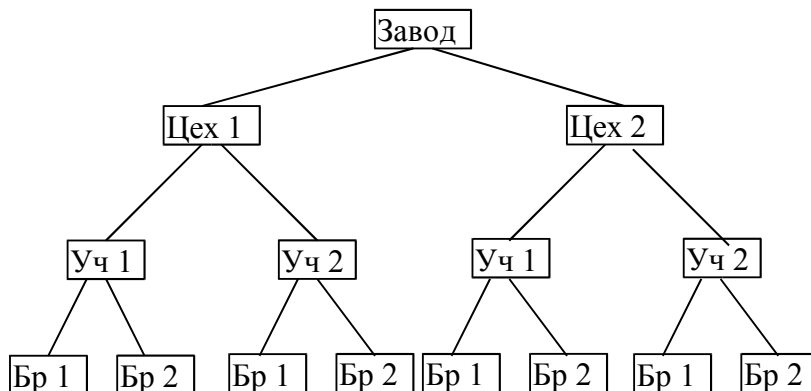
инкапсуляции сами по себе нашли наилучшее воплощение, среди практических языков, в процедурном языке Ада.

2. **Наследование.** Совместно с инкапсуляцией наследование составляет два основных принципа ООП. Именно их сочетание и дало качественно новый подход к программированию.

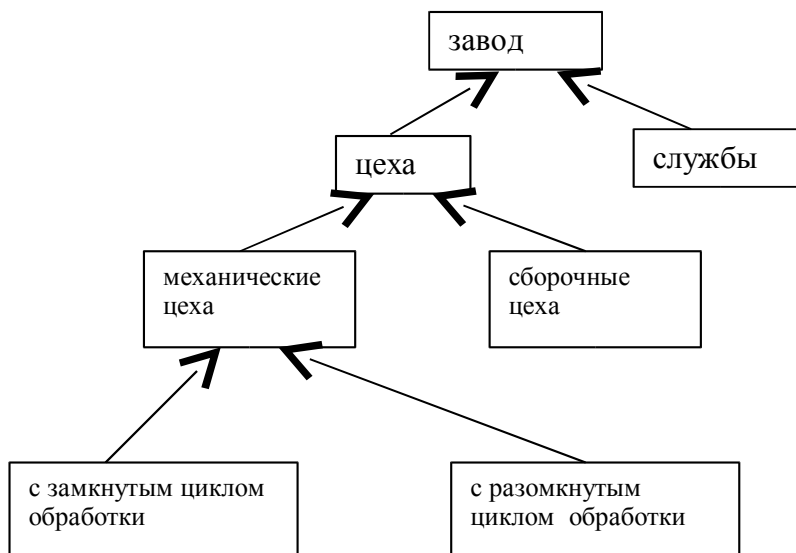
Различные виды отношений между объектами:

- 1). Генерация **is-a** “*есть некоторый*”.
- 2). Классификация **instance of** “*быть примером*”.
- 3). Агрегация **part of** “*быть частью*”.
- 4). Ассоциация **member of** “*быть элементом*”.

На практике традиционное программирование для борьбы со сложностью, занимаясь декомпозицией и классификацией, использует отношения «*быть частью*» - «*быть элементом*». В ООП прежде всего используются “*есть некоторый*” - “*быть примером*”. Традиционный подход к декомпозиции на примере завода можно представить так



При ООП используется «классификационный» подход:



Стрелками показано отношение «есть некоторый».

Такой подход дает возможность нижестоящим объектам наследовать свойства вышестоящих. В пересчете на программирование – использовать программы, «обслуживающие» вышестоящий объект.

3. Механизм обмена сообщениями. Объекты обмениваются сообщениями. Учитывая те же реалии прямого моделирования можно сказать, что механизм обмена сообщениями может существенно отличаться от механизмов передачи параметров и вызова процедур. Кстати, «передачи параметров» могут быть независимы по времени от «вызовов процедур».

4. Позднее связывание. Позднее связывание - это и методологический и технический принцип, который исходит из того, что решения следует принимать не раньше того момента, когда это необходимо, чтобы учесть сложившуюся на этот момент ситуацию. Мы можем закладывать в алгоритм, скажем, «долларовый эквивалент», а конкретный пересчет произойдет в момент расчета. Или, например, объект 5 может в разных контекстах восприниматься по-разному: как число, строка, символ и т.д., то есть менять тип.

Кстати, позднее связывание - это один из аргументов за режим интерпретации.

В настоящее время многие прикладные языки, прежде всего связанные с интернетом, строятся с оглядкой на ООП. Так что есть надежды.

Заключение

Вычислительная техника не может преодолеть жесткие рамки машины Фон-Неймана. Несмотря на интенсивнейшие работы и очень большое финансирование, прогресс в компьютерной сфере носит весьма локальный характер, поскольку неприкосновенной остается архитектура *процессор – память*.

Теоретическое программирование очертило многие принципиальные проблемы, которые предстоит решать. Движение в сторону «искусственного интеллекта» также расширяет круг задач, подлежащих формализации.

Все это говорит о том, что само по себе увеличение производительности компьютеров в тысячи раз, как то обещают, например, квантовые вычислители, не решает назревших проблем действительного прогресса в компьютеризации. Качественный скачек в производительности тем более потребует радикальных архитектурных изменений. Потребуется, естественно, и новых подходов к формализации, то есть новых подходов к математическому моделированию.

Литература

1. Новиков Ф.А. Дискретная математика для программистов. – СПб:Питер, 2000. – 304 с.
2. Кузнецов О.П., Адельсон-Вельский Г. М. Дискретная математика для инженера. 2-е изд. – М.: Энергоатомиздат, 1988.-480 с.
3. Кук Д., БейзГ. Компьютерная математика. –М.: Наука, 1990.- 384 с.
4. Кузин Л.Т. Основы кибернетики. том 2. –М.:Энергия, 1979.-584 с.
5. Мендельсон Э. Введение в математическую логику. –М.:Наука, 1971. –320 с.
6. Клини С. Математическая логика. –М: Мир,1973. –480 с.
7. Уилсон Р. Введение в теорию графов. –М.:Наука, 1977. –207 с.
8. Гроссман И., Магнус В. Группы и графы. –М.:Мир, 1971. –247 с.
9. Кофман А. Введение в прикладную комбинаторику. –М.:Наука, 1975. –479 с.
10. Хендерсон П. Функциональное программирование. Применение и реализация. –М.: Мир, 1983.-349 с.
11. Клоксин У., Меллиш К. Программирование на языке Пролог.- М.: Мир, 1987.- 336 с.
12. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е издание/Пер. с англ. - М.: "Издательство Бином", СПб: "Невский диалект", 1998 г. - 560 с.