Тук са всички теми за теоретичната част на държавния изпит за специалност информатика. Написани са от мен, Иван Димитров Георгиев (вече завършил) студент по информатика, електронната ми поща е ivandg@yahoo.com.
Материалът е съобразен изцяло с конспекта и съответните анотации, издадени и одобрени през май 2007 година. Лично аз съм се подготвял по тези материали.
За два въпроса (6 и 8) не съм сигурен, че изложението е пълно, т.е. ако ползвате този въпрос може да се допитате до съответния преподавател дали не трябва да се добави нещо. За въпрос 6 това е сигурно, тъй като ми се падна на изпита и на него имам 5 .
Пропуснато е изложението на въпрос номер 25, но за него има подходяща книга на доц. Черногорова, която се разпространява свободно в мрежата и може да бъде изтеглена от сайта на факултета.
Добре е да имате инсталиран MathType, за да нямате проблеми при разчитането. От този линк http://www.dessci.com/en/dl/MTW6.exe може да изтеглете trial версия за един месец.

1. Булеви функции. Теорема на Пост-Яблонски за пълнота.

Нека $\mathbf{J_2} = \{0, 1\}$. Всяка функция $f : \mathbf{J_2}^n \to \mathbf{J_2}$, $n \in \mathbb{N}$, $n \geq 1$ наричаме **двоична** (**булева**) **функция**.
Всяка функция $f : \mathbf{J_2}^n \to \mathbf{J_2}$ можем да разглеждаме като функция на n независими променливи $x_1, x_2, \ldots, x_n$.
С $\mathbf{F_2^n}$ ще означаваме множеството на всички двоични функции на n променливи. Очевидно е, че $|\mathbf{F_2^n}| = 2^{2^n}$.
Означаваме $\mathbf{F_2} = \bigcup_{n=1}^{\infty} \mathbf{F_2^n}$ - множеството на всички двоични функции.
Въвеждаме стандартна (лексикографска) наредба на $\mathbf{J_2^n}$:
$\alpha = a_1a_2\ldots a_n$ предшества $\beta = b_1b_2\ldots b_n$, ако съществува $i \in \{1, 2, .., n\}$, такова че $a_1 = b_1, a_2 = b_2, \ldots, a_{i-1} = b_{i-1}$, $a_i < b_i$ ($a_i = 0, b_i = 1$). При стандартно подредени вектори от $\mathbf{J_2^n}$ всяка булева функция се задава еднозначно с двоичен вектор-стълб с размерност 2^n. Това означава, че i-тата компонента на вектора-стълб е стойността на функцията за i-тия вектор от $\mathbf{J_2^n}$ в стандартната наредба.

Нека $f(x_1, x_2, \ldots, x_n) \in \mathbf{F_2^n}$, $g_i(y_1, y_2, \ldots, y_m) \in \mathbf{F_2^m}$, $i = 1, 2, \ldots, n$. Функцията $h(y_1, y_2, \ldots, y_m) = f(g_1(y_1, \ldots, y_m), g_2(y_1, \ldots, y_m), \ldots, g_n(y_1, \ldots, y_m))$ наричаме **суперпозиция** на $g_1, g_2, \ldots, g_n$ във f.

Казваме, че функцията $f : \mathbf{J_2}^n \to \mathbf{J_2}$ **не зависи съществено** от променливата x_i, ако $f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$. Още казваме, че x_i е **фиктивна променлива**.
Ясно е, че към всяка двоична функция на n променливи можем да добавим колкото искаме фиктивни променливи. Така можем да считаме, че суперпозицията на $g_1, g_2, \ldots, g_n$ във f е дефинирана дори когато $g_1, g_2, \ldots, g_n$ са функции на различен брой променливи.

Нека $F = \{f_0, f_1, \ldots\} \subseteq \mathbf{F_2}$. Нека $X = \{f, x, 0, 1, (,), <\text{запетая}>\}$.
По-нататък ще записваме думите $f_\alpha x_\beta$, където $\alpha, \beta \in \{0, 1\}^+$ като f_i, x_j, където α е двоичното представяне на числото i, β е двоичното представяне на числото j. Дефинираме индуктивно понятието **формула над множеството от функции** F:
База: За всяка функция $f_i \in F$ на n променливи думата $f_i(x_1, x_2, \ldots, x_n) \in X^*$ е формула над F.
Предположение: Нека $f_i \in F$ е функция на n променливи и $\alpha_1, \alpha_2, \ldots, \alpha_n \in X^*$ са формули над F или променливи, т.е. от вида x_k.
Стъпка: Тогава думата $f_i(\alpha_1, \alpha_2, \ldots, \alpha_n) \in X^*$ е формула над F.

Функция от вида $f(x_1, \ldots, x_n) = x_k$ наричаме **идентитет**.

Нека $F = \{ f_0, f_1, \ldots\} \subseteq \mathbf{F_2}$. На всяка формула Φ над F съпоставяме функция $f \in \mathbf{F_2}$ по следния начин:
База: Ако $\Phi = f_i (x_1, \ldots, x_n)$, то определяме $f = f_i \in F$.
Предположение: Нека $f_i \in F$ е функция на n променливи и $\Phi_1, \Phi_2, \ldots, \Phi_n$ са формули или променливи и съответните им функции са $g_1, g_2, \ldots, g_n \in \mathbf{F_2}$. Ако Φ_j е променливата x_k, то съответната функция g_j е идентитетът x_k.
Стъпка: Тогава на формулата $\Phi = f_i (\Phi_1, \ldots, \Phi_n)$ съпоставяме суперпозицията $f = f_i (g_1, \ldots, g_n)$.

С [F] ще означаваме множеството от всички двоични функции, съпоставени на формулите над F и ще го наричаме **затваряне** на F (относно суперпозицията).

Множеството от функции $F \subseteq \mathbf{F_2}$ е **пълно** в $\mathbf{F_2}$, ако $[F] = \mathbf{F_2}$.
Очевидно е, че $[\mathbf{F_2}] = \mathbf{F_2}$, но съществуването на пълни множества, различни от $\mathbf{F_2}$ не е очевидно.

Нека $F \subseteq \mathbf{F_2}$. Казваме, че F е **базис**, ако:
1. F е пълно, т.е. $[F] = \mathbf{F_2}$.
2. F е минимално по включване с това свойство, т.е. $G \subset F \Rightarrow [G] \neq \mathbf{F_2}$.

Дефинираме двоична функция $f(x, \sigma) = x^{\sigma}$ по следния начин:
$x^{\sigma} = x$, ако $\sigma = 1$ и $x^{\sigma} = \bar{x}$, ако $\sigma = 0$.

Лема: $0^{\sigma} = \bar{\sigma}$, $1^{\sigma} = \sigma$.

Формули от вида $x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \ldots x_{i_k}^{\sigma_k}$, където $x_{i_j} \neq x_{i_s}$ при $j \neq s$, $\sigma_j \in \{0, 1\}$, наричаме **елементарни конюнкции**.

Теорема (Бул): Множеството $\{x \cdot y, xy, \bar{x}\}$ е пълно.
Доказателство: Ако $f \neq 0$, можем да представим $f = x\bar{x}$ и тогава $f \in [\{x \cdot y, xy, \bar{x}\}]$.
Нека $f \neq 0$. Тогава $f(x_1, x_2, \ldots, x_n) = \bigvee_{\substack{\sigma_1\sigma_2\ldots\sigma_n \in \mathbf{J_2^n} \\ f(\sigma_1,\sigma_2,\ldots,\sigma_n)=1}} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n}$, което е формула над $\{x \cdot y, xy, \bar{x}\}$.

В означенията на доказателството, когато $f \neq 0$, формулата се нарича **съвършена дизюнктивна нормална форма** на f.

Теорема: Нека $F \subseteq \mathbf{F_2}$ е пълно множество, $G \subseteq \mathbf{F_2}$ и за всяка функция $f \in F$ имаме $f \in [G]$. Тогава G е пълно множество.

Следствие: $\{xy, \bar{x}\}$ е пълно, $\{x \oplus y, \bar{x}\}$ е пълно,
$\{0, \tilde{1}, xy, x \oplus y\}$ е пълно.
Доказателство: От теоремата на Бул и законите на Де Морган
$x \oplus y = \overline{\bar{x}\bar{y}}$, $xy = \overline{\bar{x} \oplus \bar{y}}$ и от $\bar{x} = x \oplus \tilde{1}$.

Да се спрем на последното пълно множество от следствието.
Нека $f \in \mathbf{F_2}$. Тогава f има формула над $\{0, \tilde{1}, xy, x \oplus y\}$.
Разкриваме скобите във формулата, като прилагаме дистрибутивния закон на конюнкцията спрямо събирането по модул 2, използваме идемпотентността на умножението ($xx = x$) и факта, че $f \oplus f = 0$. Накрая получаваме многократна сума по модул 2 от елементарни конюнкции без отрицания, като една елементарна конюнкция участва най-много веднъж. При това, от аналогията на конюнкцията с умножението по модул 2, получената формула може да разглеждаме като полином над полето GF (2) (полето с два елемента). Наричаме я **полином на Жегалкин**.

Теорема: Всяка булева функция има и то единствен полином на Жегалкин.
Доказателство: Всяка функция има полином, различните функции имат различни полиноми и броят на полиномите е $2^{2^n} = |\mathbf{F_2^n}|$.

Казваме, че множеството $F \subseteq \mathbf{F_2}$ е **затворено**, ако $[F] = F$.
Например $\mathbf{F_2}$ е затворено, тъй като $[\mathbf{F_2}] = \mathbf{F_2}$.
Множествата $\{0\}$, $\{\tilde{1}\}$, $\{x, \bar{x}\}$ също са затворени.

Теорема (критерий за затвореност): Нека $F \subseteq \mathbf{F_2}$ е такова, че
1. $f(x) = x \in F$;
2. за всяка функция $f(x_1, \ldots, x_n) \in F$ и $g_1, \ldots, g_n \in F$ имаме $h = f(g_1, \ldots, g_n) \in F$.

Тогава F е затворено множество.

Казваме, че функцията $f(x_1, \ldots, x_n) \in \mathbf{F_2}$ **запазва нулата**, ако $f(0, \ldots, 0) = 0$. Казваме, че функцията $f(x_1, \ldots, x_n) \in \mathbf{F_2}$ **запазва единицата**, ако $f(1, \ldots, 1) = 1$.

Означаваме с $\mathbf{T_0}$ множеството от всички булеви функции, които запазват нулата. Тези от тях които са на n променливи означаваме с $\mathbf{T_0^n}$. Означаваме с $\mathbf{T_1}$ множеството от всички булеви функции, които запазват единицата и с $\mathbf{T_1^n}$ тези от тях, които са на n променливи.
Например $xy \in \mathbf{T_0}$, $x \oplus y \in \mathbf{T_1}$, $x \oplus y \in \mathbf{T_1}$, $x \in \mathbf{T_0}$, $x \in \mathbf{T_1}$, $\bar{x} \in \mathbf{T_0}$, $\bar{x} \in \mathbf{T_1}$, $x \oplus y \in \mathbf{T_0}$, $x \oplus y \in \mathbf{T_1}$.
Така $\mathbf{T_0} \subset \mathbf{F_2}$ и $\mathbf{T_1} \subset \mathbf{F_2}$.

Очевидно, $|\mathbf{T_0^n}| = 2^{2^n-1} = |\mathbf{T_1^n}|$, тъй като всяка функция запазваща коя да е от константите се дефинира по произволен начин върху всички вектори освен един.

Теорема: Множествата $\mathbf{T_0}$ и $\mathbf{T_1}$ са затворени множества от булеви функции.
Доказателство: Директно от критерия за затвореност.

Нека $f(x_1, \ldots, x_n) \in \mathbf{F_2}$. Функцията $f^*(x_1, \ldots, x_n)$, определена по следния начин: за всеки вектор $a_1 \ldots a_n \in \mathbf{J_2^n}$,
$f^*(a_1, \ldots, a_n) = \overline{f(\overline{a_1}, \overline{a_2}, \ldots, \overline{a_n})}$ наричаме **двойнствена** на f.

Векторите $\alpha = a_1 a_2 \ldots a_n \in \mathbf{J_2^n}$ и $\bar{\alpha} = \overline{a_1}\,\overline{a_2} \ldots \overline{a_n} \in \mathbf{J_2^n}$ наричаме **противоположни вектори**.

Лема: В стандартната наредба на $\mathbf{J_2^n}$, противоположните вектори са симетрично разположени относно средата на таблицата.

От тази лема получаваме следният алгоритъм за намиране на двойнствена функция, зададена със своя вектор-стълб:
1. инвертираме всяка стойност в стълба;
2. завъртаме симетрично стълба около средата.

Като непосредствено следствие от алгоритъма получаваме:
за всяка функция $f \in \mathbf{F_2}$, $(f^*)^* = f$.

С прилагане на алгоритъма можем да установим следните двойнственост: $(0)^* = \tilde{1}$, $(x)^* = x$, $(\bar{x})^* = \bar{x}$, $(xy)^* = x \vee y$.

Лема (двойнственост на сложна функция):
Ако $h = f(g_1, \ldots, g_n)$, то $h^* = f^*(g_1^*, \ldots, g_n^*)$.

Функцията $f(x_1, \ldots, x_n) \in \mathbf{F_2}$ наричаме **самодвойнствена**, ако $f^* = f$. Със **S** означаваме множеството от всички самодвойнствени функции, със $\mathbf{S^n}$ тези от тях, които са на n променливи.

Например x, $\bar{x}$ са самодвойнствени, xy не е самодвойнствена.
Така $\mathbf{S} \subset \mathbf{F_2}$.

Лема: Функцията f на n променливи е самодвойнствена $\Leftrightarrow$ за всеки вектор $\alpha \in \mathbf{J_2^n}$ е в сила $f(\alpha) \neq f(\bar{\alpha})$.

Вече е ясно, че $|\mathbf{S^n}| = 2^{2^{n-1}}$, тъй като всяка самодвойнствена функция на n променливи се дефинира свободно точно върху един от всяка двойка противоположни вектори.

Теорема: Множеството **S** е затворено множество от булеви функции.
Доказателство: Прилагаме критерия за затвореност, като използваме лемата за двойнственост на сложна функция.

Въвеждаме нова релация r в $\mathbf{J_2^n}$:
ако $\alpha = a_1a_2...a_n$ и $\beta = b_1b_2...b_n$, то α r β $\Leftrightarrow$ $a_1 \le b_1, ..., a_n \le b_n$.
Релацията r очевидно е частична наредба, която не е линейна.
Въвеждаме релация ρ в $\mathbf{J_2^n}$:
ако $\alpha = a_1a_2...a_n$ и $\beta = b_1b_2...b_n$, $\alpha \rho \beta$ $\Leftrightarrow$ съществува $i \in \{ 1, 2, ..., n \}$, такова че $a_i < b_i$ ($a_i = 0, b_i = 1$) и $a_j = b_j$ при $j \ne i$.

Лема: Ако α r β и $\alpha \ne \beta$, то съществуват $\alpha_1, ..., \alpha_k \in \mathbf{J_2^n}$, такива че $\alpha \rho \alpha_1 \rho ... \rho \alpha_k \rho \beta$ (допускаме k = 0).

Казваме, че функцията $f(x_1, ..., x_n) \in \mathbf{F_2}$ е **монотонна**, ако за всеки $\alpha, \beta \in \mathbf{J_2^n}$, такива че α r β имаме $f(\alpha) \le f(\beta)$.
С **M** означаваме множеството от всички монотонни функции,
с $\mathbf{M_n}$ тези от тях, които са на n променливи.
Функциите x, xy, $x \vee y$, 0, 1 са монотонни, докато $\bar{x}$ не е монотонна, тъй като 0 r 1, но $1 = \bar{0} > \bar{1} = 0$.
Така $\mathbf{M} \ne \mathbf{F_2}$.

Задачата за намиране на явна формула за броя на монотонните функции на n променливи не е решена.

Лема: Нека $f \notin \mathbf{M}$. Тогава съществуват $\alpha, \beta \in \mathbf{J_2^n}$, такива че $\alpha \rho \beta$ и $f(\alpha) > f(\beta)$.
Доказателство: Тъй като $f \notin \mathbf{M}$, то съществуват α r β, $\alpha \ne \beta$, такива че $f(\alpha) > f(\beta)$, т.е. $f(\alpha) = 1$, $f(\beta) = 0$. От горната лема съществуват $\alpha_1, ..., \alpha_k \in \mathbf{J_2^n}$, такива че $\alpha \rho \alpha_1 \rho ... \rho \alpha_k \rho \beta$.
Да означим $\alpha_0 = \alpha, \alpha_{k+1} = \beta$. Да допуснем, че за всяко $i \in \{ 0, 1, ..., k \}$ имаме $f(\alpha_i) \le f(\alpha_{i+1})$.
Тогава $1 = f(\alpha_0) \le f(\alpha_1) \le ... \le f(\alpha_k) \le f(\alpha_{k+1}) = 0$ – противоречие.
Така съществува индекс i, такъв че $f(\alpha_i) > f(\alpha_{i+1})$ и $\alpha_i \rho \alpha_{i+1}$.

Теорема: Множеството **M** е затворено множество от булеви функции.
Доказателство: Прилагаме критерия за затвореност.

Всяка функция $f(x_1, ..., x_n) \in \mathbf{F}$ с полином на Жегалкин от вида $a_0 \oplus a_1x_1 \oplus ... \oplus a_nx_n$, наричаме **линейна функция**.
С **L** означаваме множеството от всички линейни функции,
с $\mathbf{L_n}$ тези от тях, които са на n променливи.
Например x, $\bar{x} = x \oplus \tilde{1} \in \mathbf{L}$, но $xy \notin \mathbf{L}$, $x \vee y = x \oplus y \oplus xy \notin \mathbf{L}$.
Така $\mathbf{L} \ne \mathbf{F_2}$.

Очевидно $|\mathbf{L_n}| = 2^{n+1}$, тъй като в общия вид линейните полиноми на Жегалкин имат n+1 свободни коефициенти.

Теорема: Множеството **L** е затворено множество от булеви функции.
Доказателство: Използваме критерия за затвореност.

Теорема (Пост-Яблонски): Нека $F \subseteq \mathbf{F_2}$. Тогава F е пълно $\Leftrightarrow$ F не е подмножество на нито едно от множествата $\mathbf{T_0}$, $\mathbf{T_1}$, **S**, **M**, **L**.
Доказателство:
Нека F е пълно и $K \in \{\mathbf{T_0}, \mathbf{T_1}, \mathbf{S}, \mathbf{M}, \mathbf{L}\}$. Да допуснем, че $F \subseteq K$.
Тогава $[F] \subseteq [K]$. От пълнотата на F имаме $[F] = \mathbf{F_2} \subseteq [K] = \mathbf{F_2}$.
От друга страна за множеството K вече показахме, че $[K] = K \neq \mathbf{F_2}$ – противоречие. Така F не е подмножество на нито едно от множествата $\mathbf{T_0}$, $\mathbf{T_1}$, **S**, **M**, **L**.
Нека F не е подмножество на нито едно от множествата $\mathbf{T_0}$, $\mathbf{T_1}$, **S**, **M**, **L**. Тогава съществуват функции $f_0, f_1, f_s, f_m, f_l \in F$, не непременно различни, такива че $f_0 \notin \mathbf{T_0}$, $f_1 \notin \mathbf{T_1}$, $f_s \notin \mathbf{S}$, $f_m \notin \mathbf{M}$, $f_l \notin \mathbf{L}$. Да означим $F' = \{f_0, f_1, f_s, f_m, f_l\}$, $F' \subseteq F$.
Ще покажем, че $\bar{x} \in [F']$.
Функцията $g_0(x) = f_0(x, x, \ldots, x) \in [F']$, тъй като във функцията f_0 сме заместили променливи. Имаме $g_0(0) = f_0(0, 0, \ldots, 0) = 1$, тъй като $f_0 \notin \mathbf{T_0}$. Функцията $g_1(x) = f_1(x, x, \ldots, x) \in [F']$, тъй като във функцията f_1 сме заместили променливи.
Имаме $g_1(1) = f_1(1, 1, \ldots, 1) = 0$, тъй като $f_1 \notin \mathbf{T_1}$.
За $g_0(1)$ и $g_1(0)$ имаме четири възможности:

1. $g_0(1) = 0$ и $g_1(0) = 0$ – тогава $g_0(x) = \bar{x}$, $g_1(x) = 0$ и така $0 \in [F']$ и $g_0(g_1(x)) = \tilde{1} \in [F']$;
2. $g_0(1) = 0$ и $g_1(0) = 1$ – тогава $g_0(x) = \bar{x}$ и $g_1(x) = \bar{x}$ и така $\bar{x} \in [F']$;
3. $g_0(1) = 1$ и $g_1(0) = 0$ – тогава $g_0(x) = \tilde{1}$ и $g_1(x) = 0$ и така $0 \in [F']$, $\tilde{1} \in [F']$;
4. $g_0(1) = 1$ и $g_1(0) = 1$ – тогава $g_0(x) = \tilde{1}$ и $g_1(x) = \bar{x}$ и така $g_1(g_0(x)) = 0 \in [F']$ и $\tilde{1} \in [F']$.

Така в три от случаите получихме двете константи.
Да разгледаме втория случай, при който получихме само отрицанието.
Функцията $f_s \in \mathbf{S}$ съществува $\mapsto = a_1a_2\ldots a_n \notin \mathbf{J_2^n}$, така че $f_s(\mapsto) = f_s(\bar{\mapsto})$. Функцията $g_s(x) = f_s(x^{a_1}, x^{a_2}, \ldots, x^{a_n}) \in [F']$:
ако $a_i = 1$, то $x^{a_i} = x$ и заместваме само променлива,
ако $a_i = 0$, то $x^{a_i} = \bar{x}$, но в този случай $\bar{x} \in [F']$ и заместваме променлива с формула над F', така че отново получаваме формула над F'.
Имаме $g_s(0) = f_s(0^{a_1}, 0^{a_2}, \ldots, 0^{a_n}) = f_s(\bar{a_1}, \bar{a_2}, \ldots, \bar{a_n}) = f_s(\bar{\mapsto})$,

$g_s(1) = f_s(1^{a_1}, 1^{a_2}, ..., 1^{a_n}) = f_s(a_1, a_2, ..., a_n) = f_s(\mapsto)$.
Така $g_s(0) = g_s(1)$.
Ако $g_s(0) = g_s(1) = 0$, то $g_s(x) = 0$ $\in$ [F$\square$] и $g_1(g_s(x)) = \tilde{1}$ $\in$ [F$\square$].
Ако $g_s(0) = g_s(1) = 1$, то $g_s(x) = \tilde{1}$ $\in$ [F$\square$] и $g_1(g_s(x)) = 0$ $\in$ [F$\square$].
Така с формули над $\{ f_0, f_1, f_s \}$ построихме константите 0 и $\tilde{1}$ във всички случаи.
Функцията $f_m \in$ **M** и тогава от лемата по-горе съществуват вектори $\mapsto, \updownarrow \in$ **J_2^n**, такива че $\mapsto < \updownarrow$ и $f_m(\mapsto) = 1$, $f_m(\updownarrow) = 0$.
Нека $\mapsto = a_1...a_{i-1}0a_{i+1}...a_n$, $\updownarrow = a_1...a_{i-1}1a_{i+1}...a_n$.
Функцията $g_m(x) = f_m(a_1, ..., a_{i-1}, x, a_{i+1}, ..., a_n) \in$ [F$\square$], тъй като заместваме променливи с вече получените формули 0, $\tilde{1}$ над F$\square$.
Имаме $g_m(0) = f_m(a_1, ..., a_{i-1}, 0, a_{i+1}, ..., a_n) = f_m(\mapsto) = 1$,
$g_m(1) = f_m(a_1, ..., a_{i-1}, 1, a_{i+1}, ..., a_n) = f_m(\updownarrow) = 0$.
Така $g_m(x) = \overline{x} \in$ [F$\square$].
Нека $xyz_1z_2...z_k$ ($k \neq 0$) е най-късият нелинеен член в полинома на Жегалкин за функцията f_l. Такъв има, тъй като $f_l \notin$ **L**. Във формулата за f_l заместваме променливите $z_1, ..., z_k$ с вече получената формула $\tilde{1}$ над F$\square$ и всички останали променливи без x и y с вече получената формула 0 над F$\square$. Така получаваме функция $g_l(x, y) \in$ [F$\square$]. Полиномът на Жегалкин за $g_l(x, y)$ има вида $g_l(x, y) = xy \oplus ax \oplus by \oplus c$, тъй като всички останали нелинейни членове съдържат поне една променлива, различна от $x, y, z_1, ..., z_k$ и се анулират. Да направим следната смяна на променливите: $x = u \oplus b$, $y = v \oplus a$. Тя е позволена:
ако $b = 0$, то $u \oplus b = u$ и заместваме само променлива,
ако $b = 1$, то $u \oplus b = \overline{u} \in$ [F$\square$] и заместваме променлива с формула над F$\square$, така че отново получаваме формула над F$\square$.
Така получаваме функция
$h_l(u, v) = (u \oplus b)(v \oplus a) \oplus a(u \oplus b) \oplus b(v \oplus a) \oplus c = uv \oplus d$,
където $d = ab \oplus c$.
Ако $d = 0$, то $h_l(u, v) = uv$,
ако $d = 1$, то $h_l(u, v) = \overline{uv}$ и $g_m(h_l(u, v)) = uv$.
Получихме, че конюнкцията е формула над F$\square$.
Така $\{ xy, \overline{x} \} \subseteq$ [F$\square$] $\subseteq$ [F] и $\{ xy, \overline{x} \}$ е пълно множество и от една теорема за пълни множества $\Rightarrow$ F е пълно множество.

Казваме, че функцията $f \in$ **F_2** е **Шеферова**, ако $[\{ f \}] =$ **F_2**.
Съгласно теоремата на Пост-Яблонски f е Шеферова $\Leftrightarrow$
$f \notin$ **T_0**, $f \notin$ **T_1**, $f \notin$ **S**, $f \notin$ **M**, $f \notin$ **L**. Ще покажем едно достатъчно условие за Шеферовост.

Теорема: Нека $f \in$ **F_2** и $f \notin$ **T_0**, $f \notin$ **T_1**, $f \notin$ **S**. Тогава f е Шеферова.
Доказателство:

Ще покажем, че f ∉ **M** и f ∉ **L** и тогава от теоремата на Пост-Яблонски функцията f е Шеферова.
Тъй като f ∉ $\mathbf{T_0}$, то f (0, 0, …, 0) = 1.
Също от f ∉ $\mathbf{T_1}$ ⇒ f (1, 1, …, 1) = 0.
Но (0, 0, …0) ≠ (1, 1, …, 1) и f (0, 0, …, 0) > f (1, 1, …, 1),
така че f ∉ **M**.
Да допуснем, че f ∉ **L**, т.е. $f = a_0 \oplus x_{i_1} \oplus x_{i_2} \oplus \ldots \oplus x_{i_k}$.
От f ∉ $\mathbf{T_0}$ ⇒ f (0, …, 0) = 1, т.е. $a_0 = 1$. От f ∉ $\mathbf{T_1}$ ⇒ f (1, 1, …, 1) = 0, т.е. $\underbrace{1 \oplus 1 \oplus \ldots \oplus 1}_{k+1 \text{ пъти}} = 0$ ⇒ k е нечетно. Сега

$$f^* = 1 \oplus (x_{i_1} \oplus 1) \oplus \ldots \oplus (x_{i_k} \oplus 1) \oplus 1 = \underbrace{1 \oplus 1 \oplus \ldots \oplus 1}_{k+2 \text{ пъти}} \oplus x_{i_1} \oplus \ldots \oplus x_{i_k} =$$

$= 1 \oplus x_{i_1} \oplus \ldots \oplus x_{i_k} = f$ ⇒ f ∉ **S** – противоречие. Така f ∉ **L**.

2. Крайни автомати. Регулярни изрази. Теорема на Клини.

Краен детерминиран автомат наричаме петорката
A = < **Q**, **X**, q_0, δ, **F** >, където
Q е крайно множество от вътрешни състояния,
X е крайна входна азбука,
q_0 ∈ **Q** е начално състояние,
δ : **Q** x **X** → **Q** е частична **функция на преходите**,
F ⊆ **Q** е множеството от заключителни състояния.

Представяме крайните детерминирани автомати с краен ориентиран мултиграф с върхове елементите от **Q**. В графа има ребро от q_i до q_j, надписано с x ∈ **X,** ако δ (q_i, x) = q_j. Обикновено началното състояние е посочено със стрелка, а заключителните състояния се изобразяват с двойни кръгчета.

Дефинираме $\hat{\delta}$: **Q** x **X*** → **Q** – **разширена функция на преходите** по следния начин: $\hat{\delta}$ (q, ε) = q за всяко q ∈ **Q**,
$\hat{\delta}$ (q, τx) = δ ($\hat{\delta}$ (q, τ), x), за всеки x ∈ **X**, q ∈ **Q** и τ ∈ **X***, за които δ ($\hat{\delta}$ (q, τ), x) е дефинирана, недефинирана, ако δ ($\hat{\delta}$ (q, τ), x) не е дефинирана.

Казваме, че автоматът **A** = < **Q**, **X**, q_0, δ, **F** > **разпознава думата** τ ∈ **X***, ако $\hat{\delta}$ (q_0, τ) ∈ **F**. Езикът L_A = { τ | τ ∈ **X***, $\hat{\delta}$ (q_0, τ) ∈ **F** } наричаме **език, разпознаван от автомата A**.

Ако функцията на преходите δ : **Q** x **X** → **Q** не е тотална, то можем да разширим автомата **A** до **A**′ следния начин:
A′ = < **Q** ∪ q*, **X**, q_0, δ*, **F** >, където q* ∉ **Q**,
δ* (q, x) = δ (q, x), ако δ (q, x) е дефинирана и

□* (q, x) = q*, ако □ (q, x) не е дефинирана.
Сега, ако **A** разпознава думата ↦ то очевидно **A**□ разпознава ↦
Обратното също е вярно, тъй като q*□ **F** □ $L_A = L_{A□}$. И така без да изменяме езика на автомата можем, ако е необходимо, да додефинираме функцията на преходите до тотална функция.

Теорема: За всеки краен детерминиран автомат
A = < **Q**, **X**, q_0, □, **F** >, съществува автоматна граматика Г, такава че $L_Г = L_A$.
Доказателство: Построяваме граматиката Г = < **Q**, **X**, q_0, **P** >,
където **P** = { q_i □ xq_j | ако □ (q_i, x) = q_j } □ { q_i □ x, ако □ (q_i, x) □ **F** }.
Изчистваме Г от аксиома в дясна част на правило и ако □ □ L_A (q_0 □ **F**), добавяме правилото q_0 □ □. Лесно се вижда, че получената граматика поражда езика L_A.

Краен недетерминиран автомат наричаме петорката
A = < **Q**, **X**, q_0, □, **F** >, където
Q е крайно множество от вътрешни състояния,
X е крайна входна азбука,
q_0 □ **Q** е начално състояние,
□ : **Q** x **X** □ **2^Q** е **функция на преходите**,
F □ **Q** е множеството от заключителни състояния.

Дефинираме λ̂ : **Q** x **X*** □ **2^Q** – **разширена функция на преходите** по следния начин: λ̂ (q, □) = { q } за всяко q □ **Q**,
λ̂ (q, x) = □ (q, x) за всяко x □ **X**, λ̂ (q, ↦x) = $\bigcup_{i=1}^{m}$ □ (q_{p_i}, x),
където λ̂ (q, ↦) = { q_{p_1}, q_{p_2}, …, q_{p_m} }.

Казваме, че автоматът **A разпознава думата** ↦□ **X***, ако λ̂ (q_0, ↦) □ **F** □ □. Езикът L_A = { ↦| ↦□ **X***, λ̂ (q_0, ↦) □ **F** □ □ } наричаме **език, разпознаван от автомата A**.

Теорема: За всяка автоматна граматика Г = < **N**, **T**, **S**, **P** > съществува краен недетерминиран автомат **A**, такъв че $L_A = L_Г$.
Доказателство: Без ограничение на общността можем да считаме, че в Г няма преименуващи правила. Нека E □ **N** □ **T**. Конструираме автомата **A** по следния начин:
A = < **N** □ { E }, **T**, **S**, □, **F** >, където
□ (A, x) = { B | A□ xB □ **P**} □ { E}, ако A□ x □ **P** и
F = { E }, ако □ □ $L_Г$ и **F** = { **S**, E }, ако □ □ $L_Г$.
Тривиално се проверява, че $L_A = L_Г$.

Теорема: За всеки краен недетерминиран автомат

A = < **Q**, **X**, q_0, □, **F** > съществува краен детерминиран автомат **A**□, такъв че $L_A = L_{A□}$.

Доказателство: Построяваме крайният детерминиран автомат **A**□ по следния начин: **A**□ = < **Q**□, **X**, { q_0}, □□, **F**□ >, където
Q□ □ **2^Q** е множеството от всички M □ **Q**, такива че съществува дума ↦□ **X***, такава че **λ** (q_0, ↦) = M. Функцията □□ : **Q**□ x **X** □ **Q**□ е дефинирана по следния начин: □□ ({ $q_{p_1}, q_{p_2}, \ldots, q_{p_m}$}, x) = $\bigcup_{i=1}^{m}$ □ (q_{p_i}, x)
и **F**□ = { M | M □ **Q**□, M □ **F** □ □ }.

С индукция по дължината на ↦ще покажем, че
λ (q_0, ↦) = **λ**□ ({ q_0 }, ↦).

База: Нека ↦= □, т.е. d (↦) = 0. Имаме **λ** (q_0, □) = { q_0 } и
λ□ ({ q_0}, □) = { q_0} □ **λ** (q_0, ↦) = **λ**□ ({ q_0 }, ↦).

Предположение: Нека твърдението е изпълнено за думата ↦

Стъпка: Ще покажем, че **λ** (q_0, ↦x) = **λ**□ ({ q_0}, ↦x).

Нека **λ**□ ({ q_0}, ↦) = { $q_{p_1}, q_{p_2}, \ldots, q_{p_m}$}. Тогава по дефиницията на □□ имаме **λ**□ ({ q_0}, ↦x) = □□ ({ $q_{p_1}, q_{p_2}, \ldots, q_{p_m}$}, x) = $\bigcup_{i=1}^{m}$ □ (q_{p_i}, x).

От индукционното предположение имаме
λ (q_0, ↦) = **λ**□ ({ q_0 }, ↦) = { $q_{p_1}, q_{p_2}, \ldots, q_{p_m}$} □
λ (q_0, ↦x) = $\bigcup_{i=1}^{m}$ □ (q_{p_i}, x) = **λ**□ ({ q_0}, ↦x).

Имаме **λ** (q_0, ↦) □ **F** □ □ □ **λ**□ ({ q_0}, ↦) □ **F** □ □ □ **λ**□ ({ q_0}, ↦) □ **F**□.
И така $L_A = L_{A□}$.

Като комбинираме трите теореми получаваме, че автоматните езици (тези които се разпознават от автоматни граматики), са точно тези езици, които се разпознават от крайни детерминирани (недетерминирани) автомати.

Оттук нататък считаме, че всички автомати са навсякъде дефинирани.

Крайните детерминирани автомати **A** и **A**□ наричаме еквивалентни, ако $L_A = L_{A□}$.

Крайният детерминиран автомат $\mathbf{A_0}$, който разпознава автоматния език L се нарича **минимален** за този език L, ако за всеки автомат **A**, който е еквивалентен на $\mathbf{A_0}$ имаме $|\mathbf{Q_0}| □ |\mathbf{Q}|$, където $\mathbf{Q_0}$ и **Q** са множествата от състоянията съответно на $\mathbf{A_0}$ и на **A**.

Нека X е произволна азбука. Релацията R □ X* x X* наричаме **дясно-инвариантна**, ако от (↦ ↕) □ R □ (↦→, ↕□) □ R
за всяка дума □ □ X*.

Нека **A** = < **Q**, **X**, q_0, □, **F** > е краен детерминиран автомат.
Определяме релацията R_A □ **X*** x **X*** по следния начин:
R_A = { (↦ ↥) | $\hat{\delta}$ (q_0, ↦) = $\hat{\delta}$ (q_0, ↥) }.

Лема: R_A е релация на еквивалентност и е дясно-инвариантна.
Доказателство: Тривиална проверка.

Нека X е произволна азбука и L □ X* е произволен език.
Дефинираме релацията R_L □ X* x X* по следния начин:
R_L = { (↦ ↥) | за всяка дума □ □ X*, ↦□ и ↥□
едновременно са в L или не са в L }.

Теорема: Релацията R_L е релация на еквивалентност и е дясно-инвариантна.
Доказателство: Тривиална проверка.

Нека **A** = < **Q**, **X**, q_0, □, **F** > е краен детерминиран автомат.
Състоянието q □ **Q** наричаме **достижимо**, ако съществува дума ↦□ **X***, такава че $\hat{\delta}$ (q_0, ↦) = q. Ако за всяка дума ↦□ **X*** имаме $\hat{\delta}$ (q_0, ↦) □ q, казваме че състоянието q е **недостижимо**.
Ясно е, че отстраняването на недостижимите състояния не променя езика, разпознаван от автомата.
Така можем да считаме, че автоматът **A** има само достижими състояния. Всеки клас на еквивалентност на R_A съдържа думите от **X***, които довеждат автомата до дадено негово състояние. Ще отбележим, че тъй като автоматът е навсякъде дефиниран всяка дума от **X*** довежда автомата до някое състояние. Така, ако автоматът **A** е навсякъде дефиниран и има само достижими състояния, индексът на релацията R_A е краен - точно |**Q**|.

Теорема: Нека L □ **X***. Релацията R_L □ **X*** x **X*** има краен индекс □ L е автоматен език.
Доказателство: Нека L е автоматен език. Тогава съществува краен детерминиран автомат **A** = < **Q**, **X**, q_0, □, **F** >, такъв че L_A = L.
Лесно се вижда, че ако (↦ ↥) □ R_A, то (↦ ↥) □ R_L. При това положение, всеки клас на еквивалентност на R_L се състои от класове на R_A и тъй като R_A има краен индекс, то R_L също има краен индекс.
Нека индексът на R_L е краен. Нека [↦] е класът на еквивалентност на релацията R_L с представител ↦□ **X***. Да означим с **Q** множеството от класовете на еквивалентност на R_L (то е крайно).
Построяваме краен детерминиран автомат
A = < **Q**, **X**, [□], □, **F** >. Лесно се вижда, че всеки клас на еквивалентност на релацията R_L съдържа думи само от L или думи не от L (от дефиницията на R_L при □ = □).
Така можем да дефинираме **F** = { [↦] | ↦□ L }.

Дефинираме δ ([$\vdash$], x) = [$\vdash$x] за всяко x $\in$ **X**. Тази дефиниция е коректна, т.е. не зависи от представителя на класа на еквивалентност, тъй като релацията R_L е дясно-инвариантна. Лесно се вижда, че L_A = L. Така L е автоматен език.

Следствие: Автоматът **A** = < **Q**, **X**, [δ], $\Box$, **F** > от теоремата е минимален за езика L.
Доказателство: В процеса на доказателството на теоремата показахме, че индексът на R_L е по-малък от индекса на R_A за всеки детерминиран автомат **A**, който разпознава L. От друга страна, построеният автомат има точно толкова състояния, колкото е индексът на R_L, така че той е минимален.

Нека **A** = < **Q**, **X**, q_0, δ, **F** > е краен детерминиран автомат и L_A = L. Считаме, че автоматът е навсякъде дефиниран и има само достижими състояния.
Състоянията q_i, q_j $\in$ **Q** наричаме **еквивалентни**, ако съответните им класове на еквивалентност на релацията R_A попадат в един и същ клас на еквивалентност на релацията R_L.
От тази дефиниция и от теоремата лесно се вижда, че минимизацията на автомата **A** се свежда до намиране на всички подмножества на **Q** от еквивалентни състояния.

Лема: Ако q_1, q_2 $\in$ **Q**, q_1 $\in$ **F** и q_2 $\notin$ **F**, то q_1 и q_2 не са еквивалентни.

Лема (тест на едната буква): Нека q_1, q_2 $\in$ **Q**. Ако съществува x $\in$ **X**, така че δ (q_1, x) не е еквивалентно на δ (q_2, x), то q_1 и q_2 не са еквивалентни.

Лема: Нека Q = { Q_1, Q_2, …, Q_s} е разбиване на **Q**, такова че Q_j $\subseteq$ **F** или Q_j $\subseteq$ **Q****F**, j = 1, 2, …, s. Нека за всяко Q_j, за всяка буква x $\in$ **X** и за всеки q_1, q_2 $\in$ Q_j имаме: δ (q_1, x) $\in$ Q_k, δ (q_2, x) $\in$ Q_k за някое k $\in$ { 1, 2, …, s}. Тогава всяко Q_j се състои само от еквивалентни състояния.

Трите леми водят до следния алгоритъм:

1. Образуваме разбиването S^0 = { Q_1^0, Q_2^0}, където Q_1^0 = **F** и Q_2^0 = **Q****F**, i = 0.
2. Нека сме построили разбиването S^i = { Q_1^i, Q_2^i, …, $Q_{l_i}^i$ }. Всяко Q_j^i разбиваме на { $Q_{j_1}^{i+1}$, $Q_{j_2}^{i+1}$, …, $Q_{j_m}^{i+1}$ }, такива че елементите на всяко от тях не се държат като нееквивалентни с теста на едната буква и обединяваме получените разбивания в S^{i+1}.
3. Ако S^{i+1} = S^i – край, иначе i++, премини към 2.

Нека S^r = S^{r+1} = … е последното разбиване.

Ясно е, че от последната лема подмножествата в S^r се състоят само от еквивалентни състояния. Освен това лесно се вижда, че не е възможно еквивалентни състояния да са в различни подмножества на разбиването S^r.
Сега вече лесно можем да построим минималният автомат $\mathbf{A_0} = < \mathbf{Q_0}, \mathbf{X}, t_0, \delta_0, \mathbf{F_0} >$, $\mathbf{Q_0} = S^r$. Избираме $t_0 = Q_p^r$, така че $q_0 \in Q_p^r$, тъй като началното състояние на автомата е класът на еквивалентност на R_L, който съдържа ε, а ε е в класът на еквивалентност на R_A, съответен на q_0. Множеството от заключителните състояния $\mathbf{F_0}$ определяме по следния начин:
$\mathbf{F_0} = \{ Q_p^r \mid Q_p^r \cap \mathbf{F} \}$, тъй като заключителните състояния са тези класове на еквивалентност на R_L, които съдържат само думи от L и тогава тези класове са образувани точно от класовете на R_A, съответни на заключителни състояния.
Функцията на преходите δ_0 дефинираме по следния начин:
$\delta_0 (Q_p^r, x) = Q_j^r$, ако за всяко $q \in Q_p^r$ имаме $\delta (q, x) \in Q_j^r$.
Дефиницията е коректна, тъй като във всички елементи на S^r, състоянията реагират еднакво на теста на едната буква.

Нека X е азбука. Въвеждаме операции в множеството от всички езици над азбуката X:

1. Нека $L_1 \subseteq X^*$, $L_2 \subseteq X^*$. Дефинираме **сума** на езиците L_1 и L_2: $L_1 + L_2 = L_1 \cup L_2$.
2. Нека $L_1 \subseteq X^*$, $L_2 \subseteq X^*$. Дефинираме **произведение** на езиците L_1 и L_2: $L_1.L_2 = \{ \alpha\beta \mid \alpha \in L_1, \beta \in L_2 \}$.
   Дефинираме $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^2 = LL$, …, $L^{n+1} = L^nL$.
3. Нека $L \subseteq X^*$. Дефинираме **итерация** на езика L: $L^* = \bigcup_{n=0}^{\infty} L^n$.

Нека $X = \{ x_1, x_2, \ldots, x_n \}$ е крайна азбука.
Разширяваме X до $X' = X \cup \{ \Lambda, \emptyset, *, +, ., (,) \}$.
Без ограничение на общността можем да считаме, че добавените символи не се срещат в X. Дефинираме индуктивно понятията **регулярен израз** и **език, съпоставен на този регулярен израз**:
(регулярният израз е дума над X', съпоставените езици са над X)
База: Думите Λ, $\emptyset$, x_i, $i = 1, 2, \ldots, n$ над X' са регулярни изрази и съответните им регулярни езици над X са
$\{\varepsilon\}$, $\emptyset$, $\{ x_i \}$, $i = 1, 2, \ldots, n$.
Предположение: Нека $\alpha, \beta \in X'^*$ са регулярни изрази и съответните им регулярни езици над X са L_α и L_β.
Стъпка: Тогава $(\alpha)+(\beta)$, $(\alpha).(\beta)$, $(\alpha)^* \in X'^*$ са регулярни изрази и съответните им регулярни езици над X са $L_\alpha + L_\beta$, $L_\alpha L_\beta$, L_α^*.

Въвеждаме приоритет на операциите в следния ред:
итерация, произведение и сума. По този начин можем съществено да ограничим употребата на скоби в регулярните изрази.

Теорема: Ако L_1 и L_2 са автоматни езици, то $L_1 + L_2$ е автоматен език.
Доказателство:
Нека езикът L_1 се поражда от граматиката $\Gamma_1 = < \mathbf{N_1}, \mathbf{T_1}, \mathbf{S_1}, \mathbf{P_1} >$.
Нека езикът L_2 се поражда от граматиката $\Gamma_2 = < \mathbf{N_2}, \mathbf{T_2}, \mathbf{S_2}, \mathbf{P_2} >$.
Без ограничение на общността можем да смятаме, че
$\mathbf{N_1} \cap \mathbf{N_2} = \emptyset$, $\mathbf{N_1} \cap \mathbf{T_2} = \emptyset$, $\mathbf{T_1} \cap \mathbf{N_2} = \emptyset$. Ако това не е изпълнено можем да преименуваме съответните нетерминали, което няма да измени езиците L_1 и L_2. Нека $\mathbf{S} \notin \mathbf{N_1} \cup \mathbf{N_2} \cup \mathbf{T_1} \cup \mathbf{T_2}$.
Построяваме граматиката $\Gamma = < \mathbf{N_1} \cup \mathbf{N_2} \cup \{\mathbf{S}\}, \mathbf{T_1} \cup \mathbf{T_2}, \mathbf{S}, \mathbf{P} >$,
$\mathbf{P} = (\mathbf{P_1} \cup \mathbf{P_2}) \setminus \{\mathbf{S_1} \to \varepsilon, \mathbf{S_2} \to \varepsilon\} \cup \{\mathbf{S} \to \mathbf{S_1}, \mathbf{S} \to \mathbf{S_2}\} \cup \{\mathbf{S} \to \varepsilon$, ако имаме $\mathbf{S_1} \to \varepsilon$ или $\mathbf{S_2} \to \varepsilon\}$. Получената граматика е автоматна и веднага се вижда, че $L_\Gamma = L_{\Gamma_1} + L_{\Gamma_2} = L_1 + L_2$.

Теорема: Всеки краен език е автоматен.
Доказателство: $\{\varepsilon\}$ е автоматен език, тъй като се поражда от автоматната граматика $\Gamma = < \{\mathbf{S}\}, \{a\}, \mathbf{S}, \{\mathbf{S} \to \varepsilon\} >$.
Граматиката $\Gamma = < \{\mathbf{S}\}, \{a\}, \mathbf{S}, \{\mathbf{S} \to a\mathbf{S}\} >$ е автоматна и поражда езика $\emptyset$. Нека $L = \{\vdash_1, \vdash_2, \ldots, \vdash_k\} = a_1a_2\ldots a_k$.
Тогава граматиката $\Gamma = < \{\mathbf{S}, A_1, A_2, \ldots, A_{k-1}\}, \{a_1, a_2, \ldots, a_k\}, \mathbf{S},$
$\{\mathbf{S} \to a_1A_1, A_1 \to a_2A_2, \ldots, A_{k-2} \to a_{k-1}A_{k-1}, A_{k-1} \to a_k\} >$ е автоматна и очевидно $L_\Gamma = L = \{\vdash_1, \vdash_2, \ldots, \vdash_k\}$.
Нека $L = \{\vdash_1, \ldots, \vdash_m\}$ е произволен краен език, $m \geq 1$.
Тогава $L = \{\vdash_1\} + \{\vdash_2\} + \ldots + \{\vdash_m\}$ и от предната теорема L е автоматен език, тъй като е крайна сума на автоматни езици.

Теорема: Ако L_1 и L_2 са автоматни езици, то $L_1.L_2$ е автоматен език.
Доказателство:
Нека езикът L_1 се поражда от граматиката $\Gamma_1 = < \mathbf{N_1}, \mathbf{T_1}, \mathbf{S_1}, \mathbf{P_1} >$.
Нека езикът L_2 се поражда от граматиката $\Gamma_2 = < \mathbf{N_2}, \mathbf{T_2}, \mathbf{S_2}, \mathbf{P_2} >$.
Отново без ограничение на общността можем да смятаме, че
$\mathbf{N_1} \cap \mathbf{N_2} = \emptyset$, $\mathbf{N_1} \cap \mathbf{T_2} = \emptyset$, $\mathbf{T_1} \cap \mathbf{N_2} = \emptyset$.
Нека $\varepsilon \notin L_1$, $\varepsilon \notin L_2$.
Построяваме граматиката $\Gamma = < \mathbf{N_1} \cup \mathbf{N_2}, \mathbf{T_1} \cup \mathbf{T_2}, \mathbf{S_1}, \mathbf{P} >$,
$\mathbf{P} = (\mathbf{P_1} \cup \mathbf{P_2}) \setminus \{A \to a \mid A \to a\mathbf{S_2} \in \mathbf{P_1}\} \cup \{A \to a\mathbf{S_2} \mid$ за всяко $A \to a \in \mathbf{P_1}\}$. Очевидно Γ е автоматна граматика и веднага се проверява, че $L_\Gamma = L_1.L_2$.
Нека $\varepsilon \notin L_1$, $\varepsilon \notin L_2$ (случаят $\varepsilon \notin L_1$, $\varepsilon \in L_2$ се разглежда аналогично).
Тогава $L_1 = L_1' + \{\varepsilon\}$, $\varepsilon \notin L_1'$, и L_1' също е автоматен език.
Сега $L_1.L_2 = (L_1' + \{\varepsilon\}).L_2 = L_1'.L_2 + L_2$. От токущо доказаното $L_1'.L_2$ е автоматен език, L_2 е автоматен език и от теоремата по-горе $L_1.L_2$ е автоматен език.
Нека $\varepsilon \in L_1$, $\varepsilon \in L_2$. Тогава $L_1 = L_1' + \{\varepsilon\}$ и $\varepsilon \notin L_1'$, $L_2 = L_2' + \{\varepsilon\}$ и $\varepsilon \notin L_2'$, L_1' и L_2' също са автоматни езици.

Имаме $L_1.L_2 = (L_1\square + \{\square\}).(L_2\square + \{\square\}) = L_1\square.L_2\square + L_1\square + L_2\square + \{\square\}$.
От доказаното по-горе $L_1\square.L_2\square$ е автоматен език, също $L_1\square$ и $L_2\square$ са автоматни езици и $\{\square\}$ е автоматен език, защото е краен $\square$
$L_1.L_2$ е автоматен език.

Теорема: Ако L е автоматен език, то L* е автоматен език.
Доказателство:
Нека езикът L се поражда от граматиката Г = < **N**, **T**, **S**, **P** >.
Без ограничение на общността можем да смятаме, че Г е без аксиома в дясна част на правило.
Построяваме граматика $\Gamma\square$ = < **N**, **T**, **S**, **P**$\square$ >, където
P$\square$ = **P**\{ **S** $\square$ $\square$}$\square$ { A $\square$ a | за всяко късо правило A $\square$ a $\square$ **P** }.
Очевидно $\Gamma\square$ е автоматна граматика, вижда се, че $L_{\Gamma\square} = L^*\backslash\{\square\}$.
И така $L^*\backslash\{\square\}$ е автоматен език $\square$ L* = $L^*\backslash\{\square\}\square\{\square\}$ е автоматен език.

Теорема (Клини): Множествата на регулярните езици и автоматните езици съвпадат.
Доказателство: С индукция по построението ще покажем, че всеки регулярен език е автоматен.
База: Езиците $\{\square\}$, $\square$, $\{x_i\}$ са автоматни езици, защото са крайни.
Предположение: Нека $L_{\mapsto}$ и $L_{\updownarrow}$ са регулярни езици, съответни на регулярните изрази $\mapsto$ и $\updownarrow$ и да допуснем, че $L_{\mapsto}$ и $L_{\updownarrow}$ са автоматни езици.
Стъпка: Тогава $L_{\mapsto} + L_{\updownarrow}$, $L_{\mapsto}L_{\updownarrow}$, $L_{\mapsto}^*$ са автоматни езици (теоремите по-горе).
Нека L е автоматен език. Ще докажем, че L е регулярен език.
Съществува краен детерминиран автомат
A = < **Q**, **X**, q_0, $\square$, **F** >, такъв че L_A = L. Да означим
Q = { q_0, q_1, …, q_n }, **F** = $\{q_{p_1}, q_{p_2}, \ldots, q_{p_r}\}$ и да си мислим, че автоматът **A** е представен с крайния ориентиран мултиграф G.
Означаваме с R_{ij}^k множеството от маршрутите в G от q_i до q_j, които не използват като вътрешни върховете q_k, q_{k+1}, …, q_n. Ясно е, че всеки маршрут от q_i до q_j определя дума $\mapsto\square$ **X***,
такава че $\lambda(q_i, \mapsto) = q_j$. Така можем да отъждествим всеки маршрут със съответната му дума от входната азбука и да считаме, че R_{ij}^k е език над **X**, т.е. $R_{ij}^k \square$ **X***.

Лема: За всеки $q_i, q_j \square$ **Q** е в сила: $R_{ij}^{k+1} \square R_{ij}^k \square R_{ik}^k.(R_{kk}^k)^*.R_{kj}^k$,
k = 0, 1, …, n.
Доказателство: Маршрутите от R_{ij}^{k+1} разбиваме на две подмножества – такива които не използват q_k като вътрешен връх и такива които използват q_k като вътрешен връх. Очевидно маршрутите от R_{ij}^{k+1}, които не използват q_k като вътрешен връх са точно маршрутите от R_{ij}^k.

Да разбием произволен маршрут от R_{ij}^{k+1}, който използва q_k като вътрешен връх на части по следния начин: q_i -- q_k -- q_k …q_k -- q_j. В междинните части не се среща q_k, така че маршрутът започва с маршрут от q_i до q_k, който не използва q_k като вътрешен връх, т.е. маршрут от R_{ik}^{k}, продължава с произволен брой цикли от q_k до q_k, които минават точно веднъж през q_k, т.е. маршрут от $(R_{kk}^{k})^*$ и завършва с маршут от q_k до q_j, който не използва q_k като вътрешен връх, т.е. маршрут от R_{kj}^{k}. Така маршрутите от R_{ij}^{k+1}, които използват q_k като вътрешен връх са точно $R_{ik}^{k}.(R_{kk}^{k})^*.R_{kj}^{k}$ и окончателно $R_{ij}^{k+1} = R_{ij}^{k} \cup R_{ik}^{k}.(R_{kk}^{k})^*.R_{kj}^{k}$.

Лема: Езикът R_{ij}^{k} е регулярен език за всеки i, j $\in$ { 0, 1, …, n }, k $\in$ { 0, 1, …, n+1 }.
Доказателство: Провеждаме индукция по k.
База: k = 0. Ако i = j, то R_{ii}^{0} са всички маршрути от q_i до q_i, които не минават през никой друг връх. Ако в графа G няма примки в q_i, тогава $R_{ii}^{0} = \{\varepsilon\}$ и R_{ii}^{0} е регулярен език. Ако в графа G има примки в q_i и на тях съответстват букви $x_{i_1}, x_{i_2}, \ldots, x_{i_r}$, то $R_{ii}^{0} = \{ x_{i_1}, x_{i_2}, \ldots, x_{i_r}, \varepsilon \}$ и R_{ii}^{0} е регулярен език, тъй като съответства на регулярния израз $x_{i_1} + x_{i_2} + \ldots + x_{i_r} + \varepsilon$.
Ако i $\neq$ j, то R_{ij}^{0} са ребрата от q_i до q_j. Ако в графа G няма ребро от q_i до q_j, то $R_{ij}^{0} = \emptyset$ и R_{ij}^{0} е регулярен език. Ако в графа G има ребра от q_i до q_j и на тях съответстват букви $x_{i_1}, x_{i_2}, \ldots, x_{i_r}$, то $R_{ij}^{0} = \{ x_{i_1}, x_{i_2}, \ldots, x_{i_r} \}$ и R_{ij}^{0} е регулярен език, тъй като съответства на регулярния израз $x_{i_1} + x_{i_2} + \ldots + x_{i_r}$.
Предположение: Нека за някое k езикът R_{ij}^{k} е регулярен език и нека $\vdash_{ij}^{k}$ е съответният регулярен израз.
Стъпка: Тогава езикът R_{ij}^{k+1} също е регулярен, тъй като съгласно горната лема този език съответства на регулярния израз $\vdash_{ij}^{k} \cup \vdash_{ik}^{k}.(\vdash_{kk}^{k})^*.\vdash_{kj}^{k}$.

Продължение на доказателството на теоремата на Клини:
Имаме, че R_{ij}^{n+1} са всички маршрути от q_i до q_j (без ограничение) и от горната лема R_{ij}^{n+1} е регулярен език. От дефиницията за езика L_A имаме, че $L_A = R_{0p_1}^{n+1} \cup R_{0p_2}^{n+1} \cup \ldots \cup R_{0p_r}^{n+1}$, тъй като думите, които се разпознават от автомата го довеждат от началното състояние

q_0 до някое от крайните $q_{p_1}, q_{p_2}, \ldots, q_{p_r}$. И така L_A е крайна сума на регулярни езици $\Rightarrow L = L_A$ е регулярен език. Така всеки автоматен език е регулярен.

3. Графи. Дървета. Обхождане на графи. Минимално покриващо дърво.

Нека V е крайно множество, $V = \{ v_1, v_2, \ldots, v_n \}$, елементите на V наричаме **върхове**. Нека E е крайно множество, $E = \{ e_1, e_2, \ldots, e_m \}$, елементите на E наричаме **ребра**.
Краен ориентиран мултиграф се нарича тройката $G (V, E, f_G)$ с функция $f_G : E \to V \times V$ (на всяко ребро се съпоставя наредена двойка върхове). f_G наричаме **свързваща функция**.

За по нагледно представяне на графите ще използваме диаграми – върховете означаваме като точки и всяко ребро означаваме със стрелка от един връх към друг.

Ребрата $e \in E$, такива че $f_G (e) = (v, v)$, $v \in V$ наричаме **примки**.

Нека $G (V, E, f_G)$ е краен ориентиран мултиграф и f_G е инекция. Тогава G се нарича **краен ориентиран граф**. Ясно е, че в такъв случай множеството E може да бъде определено като подмножество на V x V. И така при задаване на краен ориентиран граф G свързваща функция не е необходима – графът се означава с G (V, E) и се задава с множество от върхове V и множество от ребра $E \subseteq V \times V$.

Нека G (V, E) е краен ориентиран мултиграф и релацията $E \subseteq V \times V$ е рефлексивна и симетрична. В такъв случай G се нарича **краен неориентиран граф** или просто **граф**. При изобразяване на крайни неориентирани графи не рисуваме примките и стрелките между два върха заменяме с една отсечка.

Ако в краен неориентиран граф допуснем многократни ребра, то отново ще е необходима свързваща функция и в този случай графът наричаме **краен неориентиран мултиграф**.

Нека G (V, E) е граф. Върховете v_i, v_j наричаме **съседи**, ако $(v_i, v_j) \in E$. Ако графът G е ориентиран, казваме че v_i е **баща** на v_j или, че v_j е **син** на v_i. Още казваме, че v_i и v_j са **краища** на реброто (v_i, v_j).

Нека G (V, E) е краен неориентиран граф и $v \in V$. Дефинираме **степен** на върха v - d (v) = броят на ребрата, на които v е край.
Нека G (V, E) е краен ориентиран граф и $v \in V$. Дефинираме **полустепен на изхода** на върха v – d^- (v) = броят на ребрата,

излизащи от v и **полустепен на входа** на върха v – $d^+(v)$ = броят на ребрата, завършващи във v.

Нека G (V, E, f_G) е краен ориентиран мултиграф. Редицата $v_{i_0}, v_{i_1}, \ldots, v_{i_L}$, L ≥ 0, такава че за всяко j ∈ { 0, 1, …, L-1 } съществува e ∈ E, такова че $f_G(e) = (v_{i_j}, v_{i_{j+1}})$, наричаме **маршрут** от v_{i_0} до v_{i_L} в графа G с дължина L. При $v_{i_0} = v_{i_L}$, маршрутът наричаме **контур**.

Нека G (V, E) е краен неориентиран граф.
Редицата $v_{i_0}, v_{i_1}, \ldots, v_{i_L}$, L ≥ 0, такава че за всяко j ∈ { 0, 1, …, L-1 }, $(v_{i_j}, v_{i_{j+1}})$ ∈ E и за всяко j ∈ { 1, …, L-1}, $v_{i_{j-1}} \neq v_{i_{j+1}}$, наричаме **път** от v_{i_0} до v_{i_L} в графа G с дължина L. При $v_{i_0} = v_{i_L}$ и L ≥ 3, маршрутът наричаме **цикъл**. Ясно е, че по тази дефиниция считаме, че има **тривиален път** с дължина 0 от всеки връх v_i до v_i. Ще отбележим, че с дефинирането на тривиален път отчитаме примките в неориентирания граф, но им даваме тежест нула при образуване на дължината на пътищата.

Графът G (V, E) се нарича **свързан**, ако за всеки v_i, v_j ∈ V съществува път от v_i до v_j. Дефиницията е приложима и за ориентирания случай, както и за случая с мултиграф, със замяната на понятието път с понятието маршрут, но тъй като изискването е прекалено силно, ще въведем алтернативно понятие.
Ориентираният граф G (V, E) наричаме **слабо свързан**, ако за всеки v_i, v_j ∈ V съществува път от v_i до v_j или от v_j до v_i.

Казваме, че графът D (V, E) е **дърво**, ако D е свързан граф без цикли.

Ще направим индуктивна дефиниция на **кореново дърво** (с корен r):

1. База – графът D ({ r }, ∅) е кореново дърво с корен r и единствен **лист** r.
2. Предположение – нека D (V, E) е кореново дърво с корен r и листа $l_1, l_2, \ldots, l_k$.
3. Стъпка – нека u ∈ V, w ∉ V. Тогава D′ (V′, E′) = D′ (V ∪ { w }, E ∪ { (u, w)}) е също дърво с корен r. Ако u = l_i за i ∈ { 1, 2, …, k }, то неговите листа са $l_1, \ldots, l_{i-1}$, w, $l_{i+1}, \ldots, l_k$, в противен случай, те са $l_1, \ldots, l_k$, w.

Операцията в индукционната стъпка наричаме **присъединяване на връх**.

Теорема: Всяко кореново дърво е дърво.
Доказателство: Индукция по построението.

1. База – кореновото дърво D ({ r }, ∅) е свързан граф, тъй като съществува тривиален път от r до r и няма цикли, тъй като няма ребра, така че D ({ r}, ∅) е дърво.
2. Предположение – нека кореновото дърво D (V, E) е дърво.
3. Стъпка – нека u ∈ V, w ∉ V. Ще покажем, че кореновото дърво D′ (V′, E′) = D′ (V ∪ { w }, E ∪ { (u, w)}) е дърво.
   Нека v_i, v_j ∈ D′. Ако v_i, v_j ∈ D, то по индукционното предположение има път от v_i до v_j. Ако v_i = v_j = w, то съществува тривиален път от v_i до v_j. Ако v_j = w, v_i ≠ w, то от индукционното предположение съществува път от v_i до u и като добавим реброто (u, w) получаваме път от v_i до w – не е възможно w да съвпада с върха преди u, тъй като w ∉ V.
   И така D′ е свързан граф. Да допуснем, че в D′ има цикли. Тъй като по индукционното предположение в D няма цикли, то реброто (u, w) със сигурност участва в цикъл, което е противоречие, тъй като w е край единствено на реброто (u, w) и следователно, съгласно дефиницията на път, w не може да участва в цикъл. Така D′ е свързан граф без цикли, т.е. D′ е дърво.

Ще отбележим, че всяко дърво можем да предефинираме като кореново, ако изберем произволен негов връх за корен.

Теорема: Нека D (V, E) е дърво с корен r. Тогава |V| = |E| + 1.
Доказателство: Индукция по построението.
1. База – за кореновото дърво D ({ r }, ∅) имаме |V| = 1, |E| = 0 и тогава |V| = |E| + 1.
2. Предположение – нека за кореновото дърво D (V, E) имаме |V| = |E| + 1.
3. Стъпка – нека u ∈ V, w ∉ V. Тогава за кореновото дърво D′ (V′, E′) = D′ (V ∪ { w }, E ∪ { (u, w)}) имаме |V′| = |V| + 1, |E′| = |E| + 1 = |V| + 2 ⇒ |V′| = |E′| + 1.

Теорема: Нека D (V, E) е дърво и v_i, v_j ∈ V. Тогава съществува единствен път между v_i и v_j.
Доказателство: Поне един път между v_i и v_j има, тъй като D е свързан граф. Да допуснем, че съществуват два различни пътя $v_{i_0}, v_{i_1}, \dots, v_{i_L}$, $v_{i_0} = v_i$, $v_{i_L} = v_j$ и $v_{j_0}, v_{i_1}, \dots, v_{j_M}$, $v_{j_0} = v_i$, $v_{j_M} = v_j$.
Нека s е най-малкото число, такова че $v_{i_s} \neq v_{j_s}$ (такова s има, тъй като двата пътя са различни), s ≥ 1. Тогава $v_{i_L}, v_{i_{L-1}}, \dots, v_{i_s}, v_{i_{s-1}} \neq v_{j_{s-1}}, v_{j_s}, \dots, v_{j_M}$ е цикъл, което е противоречие.

Теорема: Нека D (V, E) е дърво и (v_i, v_j) ∉ E. Тогава в графа D′ (V, E′), E′ = E ∪ { (v_i, v_j) } има точно един цикъл.
Доказателство: Тъй като (v_i, v_j) ∉ E, то съществува единствен път и то с дължина поне 2 от v_i до v_j в D. Тогава при добавяне на реброто

(v_i, v_j) наистина се образува цикъл v_i, V_1, v_j, v_i (V_1 е непразна редица от върхове). Да допуснем, че при добавяне на реброто (v_i, v_j) се образува втори цикъл. Естествено, той съдържа реброто (v_i, v_j), тъй като D е дърво и няма цикли и има вида v_i, V_2, v_j, v_i ($V_1 \neq V_2$).
Без ограничение можем да смятаме, че реброто (v_i, v_j) се използва само накрая и в двата цикъла. Тогава v_i, V_1, v_j и v_i, V_2, v_j са два различни пътя между v_i и v_j в D, което е противоречие с предишната теорема.

Нека G (V, E) е граф, а D (V, E'), E' $\subseteq$ E е дърво. Тогава D се нарича **покриващо дърво** на графа G.

Теорема: Графът G (V, E) притежава покриващо дърво тогава и само тогава, когато G е свързан.
Доказателство: Ясно е, че ако G притежава покриващо дърво D, то G е свързан, тъй като в D, а тогава и в G (E' $\subseteq$ E) има път от всеки връх до всеки друг.
Нека G е свързан граф. Ще опишем алгоритъм, който построява покриващо дърво на G:

1. H = G.
2. докато в H има цикли
   H = H – ребро от цикъла.

Твърдим, че след изпълнение на алгоритъма H е покриващо дърво на G. Действително, алгоритъмът винаги завършва, тъй като в G има краен брой ребра, а съществуването на цикъл предполага наличието на поне три ребра. Ясно е, че H е граф с върхове V и ребра E' $\subseteq$ E. От самия алгоритъм се вижда, че H няма цикли.
Ще покажем, че на всяка стъпка H остава свързан граф.
Да допуснем, че на някоя стъпка премахването на реброто (v_i, v_j) води до несвързан граф. Нека v_k, v_l са върхове между които няма път. Тъй като на предната стъпка графът е бил свързан, то съществува път между v_k и v_l, който минава през реброто (v_i, v_j), но в такъв случай можем да използваме остатъка от цикъла (или подходяща негова част) и да получим път между v_k и v_l в новия граф, което е противоречие. И така H (V, E') е свързан граф без цикли, т.е. покриващо дърво на G.

Под **обхождане на граф** ще разбираме процедура, която по определени правила "посещава" всички върхове на графа.

Даден е краен неориентиран граф G (V, E). В резултат на **обхождането в ширина** V се разбива на **нива на обхождане** L_0, L_1, …, L_k по следния начин:

1. избираме начален връх на обхождането r $\in$ V. "Обхождаме" r. Нека L_0 = { r }, i = 0.
2. Образуваме нивото L_{i+1} = { v | v $\in$ V, v – необходен, съществува w $\in$ L_i, така че (w, v) $\in$ E }. "Обхождаме" всички върхове в L_{i+1}.

3. Ако $L_{i+1} \neq \emptyset$, тогава i++, премини към 2. Иначе край.

Ще отбележим, че обхождането ще посети всички върхове точно тогава, когато графът е свързан.

Ще приложим тази алгоритмична схема, за да построим покриващо дърво на свързания граф G (V, E):

1. Избираме начален връх на обхождането $r \in V$.
   Образуваме $D = (V_D, E_D)$, $V_D = \{ r \}$, $E_D = \emptyset$.
   Нека $L_0 = \{ r \}$, $i = 0$.
2. Образуваме нивото $L_{i+1} = \{ v \mid v \in V, v \notin V_D$, съществува $w \in L_i$, такъв че $(w, v) \in E \}$. $E_D = E_D \cup \{ (w, v) \mid w \in L_i, v \in L_{i+1}\}$, като за всяко $v \in L_{i+1}$ се добавя точно едно (w, v), такова че $w \in L_i$, $V_D = V_D \cup L_{i+1}$.
3. Ако $L_{i+1} \neq \emptyset$, тогава i++, премини към 2. Иначе край.

След края на изпълнението $V_D = V$, тъй като G е свързан и получаваме кореново дърво $D (V, E_D)$, $E_D \subseteq E$, което е покриващо дърво на графа G.

Даден е краен неориентиран граф G (V, E). При схемата **обхождане в дълбочина** основни понятия са **текущ връх** t и **баща** на върха t – p (t).

1. Избираме начален връх $r \in V$. “Обхождаме” r.
   Обявяваме t = r, p (t) е неопределен.
2. Търсим необходим съсед v на текущия връх t.
   Ако има такъв v, тогава p (v) = t, t = v, “обхождаме” v, премини към 2.
   Ако няма такъв v и $t \neq r$, t = p (t), премини към 2.
   Ако няма такъв v и t = r, край.

И в тази схема ще се обходят всички върхове точно тогава, когато графът е свързан. Ще приложим алгоритмичната схема за решаване на задачата за построяване на покриващо дърво на свързания граф G (V, E):

1. Избираме начален връх $r \in V$, образуваме $D (V_D, E_D)$, $V_D = \{ r \}$, $E_D = \emptyset$, t = r, p (t) – неопределен.
2. Търсим необходим съсед v на текущия връх t.
   Ако има такъв v, тогава p (v) = t, t = v, $V_D = V_D \cup \{ v\}$, $E_D = E_D \cup \{ (t, v)\}$, премини към 2.
   Ако няма такъв v и $t \neq r$, t = p (t), премини към 2.
   Ако няма такъв v и t = r, край.

След края на изпълнението $V_D = V$, тъй като G е свързан и получаваме кореново дърво $D (V, E_D)$, $E_D \subseteq E$, което е покриващо дърво на графа G.

И двата алгоритъма (в ширина и в дълбочина) строят коренови дървета и тогава те могат да бъдат използвани за превръщане на произволни дървета в коренови.

Нека G (V, E) е краен неориентиран свързан граф.
Казваме, че един път в G е **Ойлеров път**, ако той минава през всяко ребро точно по веднъж. Ако началото и края на Ойлеровия път съвпадат, казваме че той е **Ойлеров цикъл**.
Граф, в който има Ойлеров цикъл, наричаме **Ойлеров граф**.

Теорема (Ойлер): G (V, E) е Ойлеров граф □ за всеки връх v □ V, d (v) е четно число.

Следствие: В крайния неориентиран свързан граф G (V, E) има Ойлеров път □ в G има точно два върха с нечетна степен.

Ясно е, че теоремата и следствието са в сила и за крайни неориентирани мултиграфи.

Дефиницията за Ойлеров път и Ойлеров цикъл се пренася лесно за крайни ориентирани мултиграфи. В сила са следните

Теорема: Крайният свързан ориентиран мултиграф G (V, E, f_G) е Ойлеров □ за всеки връх v □ V, d^+ (v) = d^- (v).

Следствие: В крайния свързан ориентиран мултиграф G (V, E, f_G) има Ойлеров път □ съществуват w□, w□□ □ V, такива че
d^+ (w□) = d^- (w□)+1, d^+ (w□□)+1 = d^- (w□□) и
за всяко v □ V\{ w□, w□□}, d^+ (v) = d^- (v).

Даден е краен неориентиран свързан граф G (V, E).
Също така е дадена функция c : E □ $\mathbb{R}$. Стойността c (e), e □ E наричаме **цена (тегло)** на реброто e. Нека D (V, E□) е покриващо дърво на графа G. **Цена на дървото** наричаме сумата
$c(D) = \sum_{e \in E'} c(e)$. Покриващото дърво D (V, E□) наричаме **минимално покриващо дърво**, ако за всяко покриващо дърво D□ (V, E□□) имаме c (D) □c (D□).

Ясно е, че всеки граф притежава минимално покриващо дърво, тъй като графът притежава краен брой покриващи дървета, а всяко крайно множество от реални числа има най-малък елемент.

Теорема (МПД – свойство): Нека G (V, E) е свързан граф с ценова функция по ребрата c : E □ $\mathbb{R}$ и □ □ U □ V. Нека *e* = (v_i, v_j) □ E е такова, че v_i □ U, v_j □ V\U и *e* има минимално тегло от всички такива ребра. Тогава съществува минимално покриващо дърво D (V, E□) на G, такова че *e* □ E□.

Доказателство: Да допуснем противното, т.е. не съществува минимално покриващо дърво на G, което съдържа реброто *e*. Нека D (V, E□) е произволно минимално покриващо дърво на G. Образуваме графа G□ (V, E□ □ { *e* }). Тогава от една теорема по-горе в G□ има единствен цикъл, в който участва реброто *e*. Този цикъл съдържа друго ребро *e*□ (u1, u2), *e*□ □ *e*, за което u1 □ U, u2 □ V\U – ако допуснем противното, т.е. всички останали ребра в цикъла съдържат краища само от U или само от V\U, ще се окаже, че те не могат да образуват цикъл. Сега построяваме дървото D□ (V, E□ □ { *e* } \ { *e*□ }). Имаме c (D□) = c (D) + c (*e*) – c (*e*□) □c (D), тъй като c (*e*) □c (*e*□). Тъй като D е минимално покриващо дърво, то c (D□) = c (D) и тогава D□ също е минимално покриващо дърво, което съдържа *e* – противоречие.

Ще използваме полученото свойство за да опишем два алгоритъма, които строят минимално покриващо дърво.

Алгоритъмът на Прим строи кореново дърво със зададен корен r.

1. Избираме произволен връх r □ V, построяваме D (V_D, E_D), V_D = { r }, E_D = □ .
2. Ако V_D = V – край.
   Ако V_D □ V, търсим ребро *e* (v_i, v_j) с минимално тегло, такова че v_i □ V_D, v_j □ V\V_D. V_D = V_D □ { v_j}, E_D = E_D □ { *e* }.
   Премини към стъпка 2.

Алгоритъмът на Прим директно прилага МПД-свойството – на всяка стъпка избира най-доброто ребро и е сигурен, че в крайна сметка ще получи минимално покриващо дърво.

Алгоритъмът на Крускал построява дърво без корен.

1. Сортираме ребрата на G в нарастващ ред на теглата им e_1, e_2, …, e_m.
2. От всеки връх v □ V образуваме тривиално дърво D_v ({ v}, □).
3. За всяко ребро e_i = (u, v), i = 1, 2, …, m по реда на сортирането правим следното: ако u и v са в различни дървета D□ (V□, E□), D□□ (V□□, E□□), свързваме u и v с ребро и по този начин обединяваме двете дървета в едно.

Изборът на най-лекото възможно ребро на всяка стъпка се гарантира от предварителното сортиране на ребрата. Ако достигнем до ребро *e* = (u, v), такова че u и v са в едно и също дърво, то *e* не се включва в дървото, тъй като свързаността на u и v е постигната с по-леки ребра.

Съществуват различни начини за представяне на графи. Първият начин е чрез списък от ребрата, т.е. явно задаване на E. Вторият начин е чрез списъци на съседите (синовете за ориентирания случай) – за всеки връх задаваме множество (за

мултиграф задаваме списък, в който може да има повторения) от неговите съседи (синове).
За третия начин дефинираме **матрица на съседство** за краен ориентиран мултиграф G (V, E, f_G) – $M_G = \| a_{ij} \|_{|v|x|v|}$,
$a_{ij} = |\{ e \mid e \in E, f_G(e) = (v_i, v_j) \}|$. При неориентирани графи не отчитаме примките, т.е. $a_{ii} = 0$ за i = 1, 2, …, |V|. Ясно е, че при краен неориентиран мултиграф G матрицата M_G е симетрична и с нули по диагонала, при неориентирани графи тя е двоична, а при мултиграфи елементите са произволни естествени числа.

Тъй като всяко дърво е граф, то за представяне на дървета можем да използваме начините за представяне на графи. Съществуват, обаче, много по-ефективни представяния. Например, всяко кореново дърво D (V, E) с корен r може да бъде представено със **списък на бащите**: едномерен масив p с |V| елемента,
като p[v] е върхът, към който е присъединен върха v (бащата на v), по дефиниция p[r] = 0 – несъществуващ връх.

Нека D (V, E) е кореново дърво. За всеки връх $v \in V$ дефинираме $\rightarrow(v)$ – **разклоненост** на v като броят на синовете на v.
Максималната разклоненост на върховете в едно кореново дърво наричаме **разклоненост на дървото**. Кореново дърво с разклоненост 2 наричаме **двоично дърво**, с разклоненост 3 **троично дърво** и т.н. с разклоненост m **m-ично дърво**.
Във всяко двоично дърво може условно да се въведе наредба на синовете – **ляв** и **десен** син. По този начин двоичните дървета удобно се представят чрез наредба на синовете – за всеки връх се задава наредена двойка от синовете му – (ляв син, десен син), отсъствието на син задаваме например с 0.

Съвсем аналогично можем да въведем условна наредба на синовете и в m-ично дърво и да представяме m-ичните дървета с наредени m-торки от синове, но това е твърде неефективно, тъй като в таблицата ще има много празни полета. Затова ще разгледаме едно друго представяне.
Нека в коренovото дърво D е въведена наредба на синовете. Съвкупността от всички синове на даден връх ще наричаме **братство**. Първият син в братството наричаме **най-ляв син**, а съседът на всеки син, който стои непосредствено отдясно наричаме **десен брат**. Ясно е, че последният в братството няма десен брат. За коренови дървета с по-голяма разклоненост може да се използва представянето, наречето **“най-ляв син, десен брат”** – това е списък от наредени двойки, по една за всеки връх, на първо място стои най-левия син на дадения връх (0, ако няма синове), а на второ място стои десният брат на върха (0, ако той е последен в братството).

4. Семантика на рекурсивните програми с предаване на параметрите по стойност.

Ще считаме, че в нашия език за програмиране има само два типа данни: **Nat** и **Bool**. Nat са естествените числа, в Bool има две стойности **tt** - истина и **ff** - лъжа. Ще считаме, че разполагаме с определен набор от **основни операции** от следните видове:

- операции f : $Nat^n \to Nat$, n ≥ 1, например събиране (+), умножение (*) и т.н.;
- операции f : $Nat^n \to Bool$, n ≥ 1, например равенство (=), различните видове неравенства (<, ≤, >, ≥) и т.н.;
- операции f : $Bool^n \to Bool$, n ≥ 1, например конюнкция (&), дизюнкция (∨), отрицание (¬) и т.н.

Синтактичните елементи на езика са следните:

- константи от тип Nat и Bool за означаване на елементи на Nat и Bool;
- символи за основните операции;
- обектови променливи, които ще означаваме с главните букви X, Y, Z, …евентуално с индекси, тези променливи ще приемат стойности естествените числа от Nat;
- функционални променливи, които ще означаваме с F_k^n, където k е индекс, променливата F_k^n ще приема стойност частична функция над Nat, n указва броят на аргументите на функцията;
- точки, запетаи, скоби, чрез които ще постигаме еднозначен синтактичен анализ.

Дефинираме индуктивно понятието **терм от тип Nat**.
База:

1. Всяка константа от тип Nat е терм от тип Nat.
2. Всяка обектова променлива е терм от тип Nat.

Предположение: Нека $\tau_1, \tau_2, \ldots, \tau_n$ са термове от тип Nat.
Стъпка:

1. Ако f : $Nat^n \to Nat$ е основна операция, то f ($\tau_1, \tau_2, \ldots, \tau_n$) е терм от тип Nat.
2. Ако F_k^n е функционална променлива, то F_k^n ($\tau_1, \tau_2, \ldots, \tau_n$) е терм от тип Nat.

Дефинираме индуктивно понятието **терм от тип Bool**.
База:

1. Всяка константа от тип Bool е терм от тип Bool.
2. Ако f : $Nat^n \to Bool$ е основна операция и $\tau_1, \tau_2, \ldots, \tau_n$ са термове от тип Nat, то f ($\tau_1, \tau_2, \ldots, \tau_n$) е терм от тип Bool.

Предположение: Нека $\prec_1, \prec_2, \ldots, \prec_n$ са термове от тип Bool.
Стъпка: Ако f : $Bool^n \to Bool$ е основна операция, то f ($\prec_1, \prec_2, \ldots, \prec_n$) е терм от тип Bool.

Дефинираме понятието **условен терм**. Нека $\prec$ е терм от тип Bool, τ_1, τ_2 са термове от тип Nat. Тогава if $\prec$ then τ_1 else τ_2 е условен терм.

По-нататък, под терм ще разбираме кой да е от трите вида термове.

Ясно е, че ако τ е терм, то в τ участват краен брой обектови и функционални променливи. Ще използваме означението τ (X_1, …, X_n, $F_1^{n_1}$, $F_2^{n_2}$, …, $F_k^{n_k}$) за да укажем, че обектовите променливи на τ са сред X_1, …, X_n, а функционалните променливи на τ са сред $F_1^{n_1}$, $F_2^{n_2}$, …, $F_k^{n_k}$. Понякога ще използваме съкратен запис τ (**X**, **F**). Също, ще си позволяваме да изпускаме броя на аргументите за една функционална променлива, ако той не е съществен.

Нека τ_1, τ_2, …, τ_n са термове от тип Nat, τ (X_1, …, X_n, **F**) е терм. Означаваме с τ (X_1/τ_1, X_2/τ_2, …, X_n/τ_n, **F**) термът, който се получава от τ чрез едновременно заместване на всяка от обектовите променливи X_1, …, X_n с термовете τ_1, …, τ_n съответно. Понякога ще използваме съкратен запис τ (**X**/τ, **F**).

Термове, в които не участват обектови променливи ще наричаме **функционални термове**. Естествено, в горните означения, ако термовете τ_1, …, τ_n са функционални, то τ (**X**/τ, **F**) също е функционален терм.

Програма е синтактичен обект **R** от следния вид:
τ_0 (X_1, …, X_n, $F_1^{n_1}$, $F_2^{n_2}$, …, $F_k^{n_k}$), where
$F_1^{n_1}$ (X_1, …, X_{n_1}) = τ_1 (X_1, …, X_{n_1}, $F_1^{n_1}$, $F_2^{n_2}$, …, $F_k^{n_k}$)
$F_2^{n_2}$ (X_1, …, X_{n_2}) = τ_2 (X_1, …, X_{n_2}, $F_1^{n_1}$, $F_2^{n_2}$, …, $F_k^{n_k}$)
…
$F_k^{n_k}$ (X_1, …, X_{n_k}) = τ_k (X_1, …, X_{n_k}, $F_1^{n_1}$, $F_2^{n_2}$, …, $F_k^{n_k}$)
Тук τ_0 е терм от тип Nat, τ_1, …, τ_k са термове от тип Nat или условни термове.

Ще определим операционната семантика на една програма **R**.

Нека $\overline{+}$ е функционален терм. Израз от вида $\overline{+}$ c, където c е константа от типа на $\overline{+}$ наричаме **опростяване** на $\overline{+}$ в програмата **R**.
Ще дефинираме **правила**, по които ще се извършва опростяването.

0. c $\Rightarrow$ c за всяка константа c (от тип Nat или от тип Bool).

1. Нека f е n-местна основна операция, τ_1, …, τ_n са функционални термове, $\tau_1 \Rightarrow c_1$, $\tau_2 \Rightarrow c_2$, …, $\tau_n \Rightarrow c_n$ и $f(c_1, \ldots, c_n) = c$. Тогава $f(\tau_1, \tau_2, \ldots, \tau_n) \Rightarrow c$. Тук τ_1, …, τ_n имат типове, които съответстват на типа на операцията f.
2. Нека $\prec$ е функционален терм от тип Bool, τ_1, τ_2 са функционални термове от тип Nat.
   a. Нека $\prec \Rightarrow$ tt и $\tau_1 \Rightarrow c$. Тогава if $\prec$ then τ_1 else $\tau_2 \Rightarrow c$.
   b. Нека $\prec \Rightarrow$ ff и $\tau_2 \Rightarrow c$. Тогава if $\prec$ then τ_1 else $\tau_2 \Rightarrow c$.
3. Нека τ_1, τ_2, …, τ_{n_i} са функционални термове от тип Nat, $1 \le i \le k$.
   Нека $\tau_1 \Rightarrow c_1$, $\tau_2 \Rightarrow c_2$, …, $\tau_{n_i} \Rightarrow c_{n_i}$.
   Нека $\sigma_i(X_1/c_1, X_2/c_2, \ldots, X_{n_i}/c_{n_i}, F_1^{n_1}, F_2^{n_2}, \ldots, F_k^{n_k}) \Rightarrow c$.
   Тогава $F_i^{n_i}(\tau_1, \tau_2, \ldots, \tau_{n_i}) \Rightarrow c$.

От вида на правилото 3. следва, че преди да предадем параметрите на дадена подпрограма трябва задължително да намерим техните стойности.

Казваме, че опростяването $\tau \Rightarrow c$ е **изводимо по стойност** в програмата **R** и отбелязваме $\mathbf{R} \vdash_V \tau \Rightarrow c$, ако съществува крайна редица $S_0, S_1, \ldots, S_n$ от множества от опростявания, такава че:

1. $S_0 = \emptyset$, $\tau \Rightarrow c \in S_n$.
2. За всяко i = 0, 1, …, n-1 имаме, че $S_{i+1} = S_i \cup \{\tau' \Rightarrow c'\}$, където опростяването $\tau' \Rightarrow c'$ е извършено по правило 0. или по едно от правилата 1., 2., 3. с предпоставки елементи на S_i.

Числото n наричаме **дължина** на извода за опростяването $\tau \Rightarrow c$.

Лема (за извода по стойност): Нека $\mathbf{R} \vdash_V \tau \Rightarrow c$ с дължина на извода k.

1. Ако τ е константа, то $\tau = c$.
2. Ако $\tau = f(\tau_1, \tau_2, \ldots, \tau_n)$, то съществуват константи $c_1, c_2, \ldots, c_n$, такива че $\mathbf{R} \vdash_V \tau_i \Rightarrow c_i$, i = 1, 2, …, n, при това с дължина на изводите по-малка от k и $f(c_1, c_2, \ldots, c_n) = c$.
3. Ако τ = if $\prec$ then τ_1 else τ_2, то $\mathbf{R} \vdash_V \prec \Rightarrow$ tt и $\mathbf{R} \vdash_V \tau_1 \Rightarrow c$ с дължина на изводите по-малка от k или $\mathbf{R} \vdash_V \prec \Rightarrow$ ff и $\mathbf{R} \vdash_V \tau_2 \Rightarrow c$ с дължина на изводите по-малка от k.
4. Ако $\tau = F_i^{n_i}(\tau_1, \tau_2, \ldots, \tau_{n_i})$, то съществуват константи

$c_1, c_2, \ldots, c_{n_i}$, такива че $\mathbf{R} \vdash_V \mp_j \; c_j$, $j = 1, 2, \ldots, n_i$ с дължина на изводите по-малка от k и

$\mathbf{R} \vdash_V \leftarrow_i (X_1/c_1, X_2/c_2, \ldots, X_{n_i}/c_{n_i}, \mathbf{F}) \; c$ с с дължина на извода по-малка от k.

С помощта на лемата за извода, лесно може да се докаже следното твърдение:

Твърдение (еднозначност на опростяването по стойност):
Нека $\mp$ е функционален терм, c_1 и c_2 са константи от типа на $\mp$ и

$\mathbf{R} \vdash_V \mp \; c_1$ и $\mathbf{R} \vdash_V \mp \; c_2$. Тогава $c_1 = c_2$.

Дефинираме функция $\mathbf{O_V}$ (**R**) : $\mathbb{N}^{\mathbf{n}} \to \mathbb{N}$,

$\mathbf{O_V}$ (**R**) $(a_1, a_2, \ldots, a_n) = b \Leftrightarrow \mathbf{R} \vdash_V \leftarrow_0 (X_1/a_1, X_2/a_2, \ldots, X_n/a_n, \mathbf{F}) \; b$.

Функцията $\mathbf{O_V}$ (**R**) наричаме **операционна семантика** на програмата **R** с **предаване по стойност**.
Определението е коректно, т.е. $\mathbf{O_V}$ (**R**) действително е функция, тъй като е в сила еднозначност на опростяването по стойност.

Нека A е произволно множество, в което е въведена частична наредба $\sqsubseteq$. Нека съществува $\bot \in A$, който е най-малък относно $\sqsubseteq$, т.е. $\bot \sqsubseteq a$ за всяко $a \in A$. Нека, освен това, всяка монотонно растяща редица $a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_n \sqsubseteq \ldots$ има точна горна граница в A, т.е. съществува $a \in A$, такова че $a_n \sqsubseteq a$ за всяко $n \in \mathbb{N}$ и за всяко $b \in A$, такова че $a_n \sqsubseteq b$ за всяко $n \in \mathbb{N}$ имаме, че $a \sqsubseteq b$. Естествено, точната горна граница $a \in A$ е единствена, означаваме $a = \bigcup_n a_n$.
При тези предположения, наредената тройка $(A, \sqsubseteq, \bot)$ наричаме **област на Скот**.

Ще дадем пример за област на Скот.

С $\boldsymbol{F_n}$, $n \geq 1$ означаваме множеството на всички частични функции на n аргумента над множеството на естествените числа $\mathbb{N}$.
В множеството $\boldsymbol{F_n}$ въвеждаме частична наредба:
$(f, g \in \boldsymbol{F_n})$ $f \sqsubseteq g$ за всяко $\mathbf{x} \in \mathbb{N}^{\mathbf{n}}$, $y \in \mathbb{N}$ имаме $f(\mathbf{x}) = y \Rightarrow g(\mathbf{x}) = y$. Ако $f \sqsubseteq g$, казваме че f е **подфункция** на g или, че g е **продължение** на f. Наредбата притежава най-малък елемент – никъде дефинираната функция, която ще означаваме с $\bot$**!** – тя се продължава до всяка функция.

Твърдение: Нека $\{f_k\}$ е монотонно растяща редица от функции, $f_k \in \boldsymbol{F_n}$. Дефинираме функцията g по следния начин:
$(\mathbf{x} \in \mathbb{N}^{\mathbf{n}}, y \in \mathbb{N})$ $g(\mathbf{x}) = y \Leftrightarrow$ съществува $k \in \mathbb{N}$, такова че $f_k(\mathbf{x}) = y$.

Тогава g е добре дефинирана n-местна функция, която е точна горна граница на $\{ f_k \}$.
Доказателство: Тривиална проверка.

Като следствие тройката $\boldsymbol{F_n} = (\boldsymbol{F_n}, \sqsubseteq, \sqsubseteq!)$ е област на Скот.

Нека $A_1 = (A_1, \sqsubseteq_1, \sqsubseteq_1)$, $A_2 = (A_2, \sqsubseteq_2, \sqsubseteq_2)$, …, $A_n = (A_n, \sqsubseteq_n, \sqsubseteq_n)$, $n \geq 2$ са области на Скот. Означаваме $A = A_1 \times A_2 \times \ldots \times A_n$ – декартовото произведение на множествата $A_1, A_2, \ldots, A_n$.
В A въвеждаме релация $\sqsubseteq$: за всеки $a_i \in A_i$, $b_i \in A_i$, i = 1, 2, …, n, $(a_1, a_2, \ldots, a_n) \sqsubseteq (b_1, b_2, \ldots, b_n) \Leftrightarrow a_1 \sqsubseteq_1 b_1$ и $a_2 \sqsubseteq_2 b_2$ и …и $a_n \sqsubseteq_n b_n$.
Очевидно е, че $\sqsubseteq$ е частична наредба в A. При това $\bot = (\bot_1, \bot_2, \ldots, \bot_n)$ е най-малък елемент в A относно $\sqsubseteq$.

Твърдение: Нека $\{ (a_k^1, a_k^2, \ldots, a_k^n) \}$ е монотонно растяща редица от елементи на A. Нека $a_i \in A_i$ е точната горна граница на $\{ a_k^i \}$, i = 1, 2, …, n. Тогава $(a_1, a_2, \ldots, a_n) \in A$ е точна горна граница на редицата $\{ (a_k^1, a_k^2, \ldots, a_k^n) \}$.
Доказателство: Тривиална проверка.

И така показахме, че $A = (A, \sqsubseteq, \bot)$ е област на Скот, където $A = A_1 \times A_2 \times \ldots \times A_n$, $\sqsubseteq$ е дефинираната частична наредба, $\bot = (\bot_1, \bot_2, \ldots, \bot_n)$. Тази област на Скот наричаме **декартово произведение** на областите на Скот $A_1, A_2, \ldots, A_n$.

Нека $A_1 = (A_1, \sqsubseteq_1, \bot_1)$, $A_2 = (A_2, \sqsubseteq_2, \bot_2)$ са области на Скот.
Казваме, че изображението $f : A_1 \to A_2$ е **непрекъснато**, ако за всяка монотонно растяща редица $a_0 \sqsubseteq_1 a_1 \sqsubseteq_1 \ldots \sqsubseteq_1 a_n \sqsubseteq_1 \ldots$от елементи на A_1 имаме, че $f (\bigcup_n a_n)$ е точна горна граница на редицата $\{ f (a_n) \}$ в A_2.
Казваме, че изображението $f : A_1 \to A_2$ е **монотонно**, ако за всеки $a, b \in A_1$ имаме $a \sqsubseteq_1 b \Rightarrow f (a) \sqsubseteq_2 f (b)$.

Твърдение: Всяко непрекъснато изображение $f : A_1 \to A_2$ е монотонно.
Доказателство: За всеки $a, b \in A_1$ и $a \sqsubseteq_1 b$ разглеждаме монотонната редица a, b, b, …и използваме непрекъснатостта.

Така, ако $f : A_1 \to A_2$ е непрекъснато изображение и $a_0 \sqsubseteq_1 a_1 \sqsubseteq_1 \ldots \sqsubseteq_1 a_n \sqsubseteq_1 \ldots$е монотонно растяща редица в A_1, то $f (a_0) \sqsubseteq_2 f (a_1) \sqsubseteq_2 \ldots \sqsubseteq_2 f (a_n) \sqsubseteq_2 \ldots$е монотонно растяща редица в A_2 и $f (\bigcup_n a_n) = \bigcup_n f (a_n)$.
Нека $A = (A, \sqsubseteq, \bot)$, $A_i = (A_i, \sqsubseteq_i, \bot_i)$, i = 1, 2, …, n са области на Скот.

Твърдение: Нека $f_i : A \to A_i$ са непрекъснати изображения,

i = 1, 2, …, n. Дефинираме изображение
$f_1 \times f_2 \times \ldots f_n : A \to A_1 \times A_2 \times \ldots A_n$,
$(f_1 \times f_2 \times \ldots f_n)(a) = (f_1(a), f_2(a), \ldots, f_n(a))$ за всяко $a \in A$.
Твърдим, че $f_1 \times f_2 \times \ldots \times f_n$ е непрекъснато.
Доказателство: Тривиална проверка.

Нека $A = (A, \sqsubseteq, \bot)$ е област на Скот и $f : A \to A$ е тотално изображение.
Казваме, че $a \in A$ е **неподвижна точка** на f, ако $f(a) = a$.
Казваме, че $a \in A$ е **най-малка неподвижна точка** на f, ако
$f(a) = a$ и за всяко $b \in A$, такова че $f(b) = b$ имаме $a \sqsubseteq b$.
Естествено, ако f притежава най-малка неподвижна точка, тя е единствена.
Казваме, че $a \in A$ е **квазинеподвижна точка** на f, ако $f(a) \sqsubseteq a$.
Казваме, че $a \in A$ е **най-малка квазинеподвижна точка** на f,
ако $f(a) \sqsubseteq a$ и за всяко $b \in A$, такова че $f(b) \sqsubseteq b$ имаме $a \sqsubseteq b$.
Естествено, ако f притежава най-малка квазинеподвижна точка,
тя е единствена.

Твърдение: Нека $f : A \to A$ е монотонно изображение. Нека $a \in A$ е най-малка квазинеподвижна точка на f. Тогава a е най-малка неподвижна точка на f.
Доказателство: Достатъчно е да покажем, че a е неподвижна точка на f. Тъй като a е квазинеподвижна, то $f(a) \sqsubseteq a$. Тъй като f е монотонно, то $f(f(a)) \sqsubseteq f(a)$ $f(a)$ е квазинеподвижна точка на f.
Тъй като a е най-малка квазинеподвижна точка на f, то $a \sqsubseteq f(a)$.
Така $f(a) = a$.

Теорема (Кнастер-Тарски): Нека $f : A \to A$ е непрекъснато изображение. Тогава f притежава най-малка квазинеподвижна точка.
Доказателство: Конструираме редица $\{a_n\}$ от елементи на A с индукция по n.
База: При $n = 0$, $a_0 = \bot$.
Предположение: Нека е дефинирано a_n, $n \geq 0$.
Стъпка: Тогава $a_{n+1} = f(a_n)$.
Твърдим, че редицата $\{a_n\}$ е монотонна. Ще покажем това с индукция по n.
База: При $n = 0$, $a_0 = \bot \sqsubseteq a_1$.
Предположение: Нека $a_n \sqsubseteq a_{n+1}$, $n \geq 0$.
Стъпка: Тъй като f е монотонно, то $f(a_n) \sqsubseteq f(a_{n+1})$, т.е. $a_{n+1} \sqsubseteq a_{n+2}$.
Нека $a = \bigcup_n a_n$. Твърдим, че a е най-малка квазинеподвижна точка на f. Действително, $f(a) = f(\bigcup_n a_n) = \bigcup_n f(a_n) = \bigcup_n a_{n+1}$.

Имаме $\bigcup_n a_{n+1} \sqsubseteq \bigcup_n a_n$, тъй като $\bigcup_n a_n$ е горна граница на редицата $\{a_{n+1}\}$. Така $f(a) \sqsubseteq \bigcup_n a_n = a \sqsubseteq a$ е квазинеподвижна точка на f.

Нека $b \sqsubseteq A$ е квазинеподвижна точка на f, т.е. $f(b) \sqsubseteq b$.
С индукция по n ще покажем, че $a_n \sqsubseteq b$ за всяко $n \sqsubseteq \mathbb{N}$.
База: При $n = 0$, $a_0 = \bot \sqsubseteq b$.
Предположение: Нека $a_n \sqsubseteq b$, $n \geq 0$.
Стъпка: Тогава $a_{n+1} = f(a_n) \sqsubseteq f(b) \sqsubseteq b$. Използвали сме, че f е монотонно и че b е квазинеподвижна точка на f.
Така $a_n \sqsubseteq b$ за всяко $n \sqsubseteq \mathbb{N} \sqsubseteq a = \bigcup_n a_n \sqsubseteq b$.
Окончателно, $a \sqsubseteq A$ е най-малка квазинеподвижна точка на f.

От горното твърдение получаваме, че всяко непрекъснато изображение притежава най-малка неподвижна точка, която е същевременно най-малка квазинеподвижна точка.

Ще докажем едно обобщение на теоремата на Кнастер-Тарски.
Нека $A_i = (A_i, \sqsubseteq_i, \bot_i)$, i = 1, 2, …, n са области на Скот.
Да означим $A = A_1 \times A_2 \times \ldots \times A_n$, $A = (A, \sqsubseteq, (\bot_1, \bot_2, \ldots, \bot_n))$.
Нека $f_i : A \sqsubseteq A_i$, i = 1, 2, …, n са тотални изображения.
Съвкупността от формалните равенства
$f_1(x_1, x_2, \ldots, x_n) = x_1$
$f_2(x_1, x_2, \ldots, x_n) = x_2$
…
$f_n(x_1, x_2, \ldots, x_n) = x_n$
наричаме **система от уравнения** за изображенията $f_1, f_2, \ldots, f_n$.
Казваме, че $(a_1, a_2, \ldots, a_n) \sqsubseteq A$ е **решение** на системата, ако $f_i(a_1, a_2, \ldots, a_n) = a_i$ за всяко i = 1, 2, …, n.
Казваме, че $(a_1, a_2, \ldots, a_n) \sqsubseteq A$ е **най-малко решение** на системата, ако $(a_1, a_2, \ldots, a_n)$ е решение на системата и за всяко решение $(b_1, b_2, \ldots, b_n) \sqsubseteq A$ на системата имаме $(a_1, a_2, \ldots, a_n) \sqsubseteq (b_1, b_2, \ldots, b_n)$.
Естествено, ако системата има най-малко решение, то е единствено.
Казваме, че $(a_1, a_2, \ldots, a_n) \sqsubseteq A$ е **квазирешение** на системата, ако $f_i(a_1, a_2, \ldots, a_n) \sqsubseteq_i a_i$ за всяко i = 1, 2, …, n.
Казваме, че $(a_1, a_2, \ldots, a_n) \sqsubseteq A$ е **най-малко квазирешение** на системата, ако $(a_1, a_2, \ldots, a_n)$ е квазирешение на системата и за всяко квазирешение $(b_1, b_2, \ldots, b_n) \sqsubseteq A$ на системата имаме $(a_1, a_2, \ldots, a_n) \sqsubseteq (b_1, b_2, \ldots, b_n)$.
Естествено, ако системата има най-малко квазирешение, то е единствено.

Теорема: Нека $f_i : A \sqsubseteq A_i$ са непрекъснати изображения. Тогава системата от уравнения за $f_1, f_2, \ldots, f_n$ притежава най-малко решение, което е и най-малко квазирешение.

Доказателство: Да означим $f = f_1 \times f_2 \times \ldots \times f_n$. Тогава $f : A \to A$ и f е непрекъснато изображение, тъй като $f_1, f_2, \ldots, f_n$ са непрекъснати. От теоремата на Кнастер-Тарски f притежава най-малка неподвижна точка, която е и най-малка квазинеподвижна точка. Да я означим с $(a_1, a_2, \ldots, a_n) \in A$. Ще покажем, че $(a_1, a_2, \ldots, a_n)$ е най-малко решение на системата. Имаме
$f(a_1, a_2, \ldots, a_n) = (f_1(a_1, \ldots, a_n), f_2(a_1, \ldots, a_n), \ldots, f_n(a_1, a_2, \ldots, a_n)) =$
$= (a_1, a_2, \ldots, a_n)$, така че $f_i(a_1, \ldots, a_n) = a_i$ за всяко
$i = 1, 2, \ldots, n$. Така $(a_1, \ldots, a_n)$ е решение на системата.
Нека $(b_1, \ldots, b_n) \in A$ е решение на системата, т.е. $f_i(b_1, \ldots, b_n) = b_i$ за всяко $i = 1, 2, \ldots, n$.
Тогава $f(b_1, \ldots, b_n) = (b_1, \ldots, b_n)$, така че $(b_1, \ldots, b_n)$ е неподвижна точка на $f \sqsubseteq (a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n)$.
Аналогично, като се използва, че $(a_1, \ldots, a_n)$ е най-малка квазинеподвижна точка на f се показва, че $(a_1, \ldots, a_n)$ е най-малко квазирешение на системата за $f_1, \ldots, f_n$.

Нека $\tau(X_1, \ldots, X_n, F_1^{n_1}, F_2^{n_2}, \ldots, F_k^{n_k})$ е терм.
Нека $a_1, a_2, \ldots, a_n \in \mathbb{N}$, $\varphi_1 \in \boldsymbol{F}_{n_1}$, $\varphi_2 \in \boldsymbol{F}_{n_2}$, …, $\varphi_k \in \boldsymbol{F}_{n_k}$.
Дефинираме индуктивно понятието **стойност** на терма τ в $a_1, \ldots, a_n, \varphi_1, \ldots, \varphi_k$, която записваме $\tau(a_1, \ldots, a_n, \varphi_1, \ldots, \varphi_k)$ или съкратено $\tau(\mathbf{a}, \boldsymbol{\varphi})$:

1. Ако $\tau = c$, то $\tau(\mathbf{a}, \boldsymbol{\varphi}) = c$ (съответно булева константа или естествено число).
2. Ако $\tau = X_i$, то $\tau(\mathbf{a}, \boldsymbol{\varphi}) = a_i$.
3. Нека $\tau = f(\tau_1, \tau_2, \ldots, \tau_l)$.
   Тогава $\tau(\mathbf{a}, \boldsymbol{\varphi}) = f(\tau_1(\mathbf{a}, \boldsymbol{\varphi}), \tau_2(\mathbf{a}, \boldsymbol{\varphi}), \ldots, \tau_l(\mathbf{a}, \boldsymbol{\varphi}))$.
4. Нека $\tau = \text{if } \prec \text{ then } \tau_1 \text{ else } \tau_2$. Тогава
   $\tau(\mathbf{a}, \boldsymbol{\varphi}) = \tau_1(\mathbf{a}, \boldsymbol{\varphi})$, ако $\prec(\mathbf{a}, \boldsymbol{\varphi}) = tt$,
   $\tau(\mathbf{a}, \boldsymbol{\varphi}) = \tau_2(\mathbf{a}, \boldsymbol{\varphi})$, ако $\prec(\mathbf{a}, \boldsymbol{\varphi}) = ff$,
   $\varphi_i!\,\tau(\mathbf{a}, \boldsymbol{\varphi})$, ако $\varphi_i! \prec(\mathbf{a}, \boldsymbol{\varphi})$.
5. Нека $\tau = F_i^{n_i}(\tau_1, \tau_2, \ldots, \tau_{n_i})$.
   Тогава $\tau(\mathbf{a}, \boldsymbol{\varphi}) = \varphi_i(\tau_1(\mathbf{a}, \boldsymbol{\varphi}), \tau_2(\mathbf{a}, \boldsymbol{\varphi}), \ldots, \tau_{n_i}(\mathbf{a}, \boldsymbol{\varphi}))$.

Тъй като функциите φ_i са частични, стойността $\tau(\mathbf{a}, \boldsymbol{\varphi})$ може да не е навсякъде определена.

По-нататък за краткост ще означаваме $\boldsymbol{F} = \boldsymbol{F}_{n_1} \times \boldsymbol{F}_{n_2} \times \ldots \times \boldsymbol{F}_{n_k}$.

Лема (съгласуване на заместване и стойност):
Нека $\tau(X_1, \ldots, X_n, F_1^{n_1}, F_2^{n_2}, \ldots, F_k^{n_k})$ е терм. Нека $\boldsymbol{\varphi} \in \boldsymbol{F}$ и $c_1, c_2, \ldots, c_n$ са константи. Тогава $\tau(\mathbf{c}, \boldsymbol{\varphi}) = \tau(\mathbf{X}/\mathbf{c}, \mathbf{F})(\boldsymbol{\varphi})$.
Доказателство: Индукция по построението на τ.

Твърдение (за монотонност): Нека τ ($\mathbf{X}$, $\mathbf{F}$) е терм, σ, $\sigma' \in \boldsymbol{F}$, $\sigma \neq \sigma'$ и $\mathbf{a} \in \mathbb{N}^{\mathbf{n}}$. Тогава за всяка константа b от типа на τ имаме, че τ ($\mathbf{a}$, σ) $\neq$ b $\Rightarrow$ τ ($\mathbf{a}$, σ') $\neq$ b.
Доказателство: Индукция по построението на τ.

Твърдение (за компактност): Нека τ ($\mathbf{X}$, $\mathbf{F}$) е терм.
За всеки $\mathbf{a} \in \mathbb{N}^{\mathbf{n}}$, $\sigma \in \boldsymbol{F}$ и константа b от типа на τ е изпълнено:
τ ($\mathbf{a}$, σ) $\neq$ b $\Rightarrow$ съществуват крайни подфункции $\sigma' \subseteq \sigma$, такива че τ ($\mathbf{a}$, σ') $\neq$ b.
Доказателство: Индукция по построението на τ.
Ако τ е константа или $\tau = X_i$, $i \in \{ 1, 2, \ldots, n \}$, то можем да изберем $\sigma' = (\sigma^{!}_{n_1}, \sigma^{!}_{n_2}, \ldots, \sigma^{!}_{n_k})$, където индексът показва броя на аргументите.
Нека $\tau = f(\tau_1, \tau_2, \ldots, \tau_l)$ и твърдението е изпълнено за термовете $\tau_1, \tau_2, \ldots, \tau_l$. Нека τ ($\mathbf{a}$, σ) $\neq$ b. Тогава съществуват константи $c_1, c_2, \ldots, c_l$, такива че τ_1 ($\mathbf{a}$, σ) $\neq$ c_1, τ_2 ($\mathbf{a}$, σ) $\neq$ c_2, …, τ_l ($\mathbf{a}$, σ) $\neq$ c_l и $f(c_1, c_2, \ldots, c_l) = b$. От индукционното предположение съществуват крайни подфункции $\sigma_1 \subseteq \sigma$, $\sigma_2 \subseteq \sigma$, …, $\sigma_l \subseteq \sigma$, такива че τ_1 ($\mathbf{a}$, σ_1) $\neq$ c_1, τ_2 ($\mathbf{a}$, σ_2) $\neq$ c_2, …, τ_l ($\mathbf{a}$, σ_l) $\neq$ c_l.
Нека $\sigma_1 = (\sigma_{11}, \sigma_{12}, \ldots, \sigma_{1k})$, $\sigma_{1j} \subseteq \sigma_j$, $j = 1, 2, \ldots, k$.
Нека $\sigma_2 = (\sigma_{21}, \sigma_{22}, \ldots, \sigma_{2k})$, $\sigma_{2j} \subseteq \sigma_j$, $j = 1, 2, \ldots, k$.
…
Нека $\sigma_l = (\sigma_{l1}, \sigma_{l2}, \ldots, \sigma_{lk})$, $\sigma_{lj} \subseteq \sigma_j$, $j = 1, 2, \ldots, k$.
Нека $\sigma^1 = \sigma_{11} \cup \sigma_{21} \cup \ldots \cup \sigma_{l1}$. Тъй като $\sigma_{11}, \sigma_{21}, \ldots, \sigma_{l1}$ са крайни подфункции на една и съща функция σ_1, то σ^1 е добре дефинирана крайна подфункция на σ_1. Аналогично, нека $\sigma^2 = \sigma_{12} \cup \sigma_{22} \cup \ldots \cup \sigma_{l2}$, $\sigma^2 \subseteq \sigma_2$, …, $\sigma^k = \sigma_{1k} \cup \sigma_{2k} \cup \ldots \cup \sigma_{lk}$, $\sigma^k \subseteq \sigma_k$. Нека $\sigma' = (\sigma^1, \sigma^2, \ldots, \sigma^k)$.
Тогава $\sigma' \subseteq \sigma$ и σ' са крайни функции. Ясно е, че $\sigma_i \subseteq \sigma'$, $i = 1, 2, \ldots, l$.
От монотонността получаваме τ_1 ($\mathbf{a}$, σ') $\neq$ c_1, τ_2 ($\mathbf{a}$, σ') $\neq$ c_2, …, τ_l ($\mathbf{a}$, σ') $\neq$ c_l, също $f(c_1, c_2, \ldots, c_l) = b \Rightarrow \tau$ ($\mathbf{a}$, σ') $\neq$ b.
Случаят τ = if $\prec$ then τ_1 else τ_2 се разглежда аналогично на предния случай.
Нека $\tau = F_i^{n_i}(\tau_1, \tau_2, \ldots, \tau_{n_i})$ и твърдението е изпълнено за термовете $\tau_1, \tau_2, \ldots, \tau_{n_i}$. Нека τ ($\mathbf{a}$, σ) $\neq$ b.
Тогава съществуват константи $c_1, c_2, \ldots, c_{n_i}$, такива че τ_1 ($\mathbf{a}$, σ) $\neq$ c_1, τ_2 ($\mathbf{a}$, σ) $\neq$ c_2, …, τ_{n_i} ($\mathbf{a}$, σ) $\neq$ c_{n_i} и σ_i ($c_1, c_2, \ldots, c_{n_i}$) $\neq$ b. От индукционното предположение съществуват крайни подфункции $\sigma_1 \subseteq \sigma$, $\sigma_2 \subseteq \sigma$, …, $\sigma_{n_i} \subseteq \sigma$, такива че τ_1 ($\mathbf{a}$, σ_1) $\neq$ c_1, τ_2 ($\mathbf{a}$, σ_2) $\neq$ c_2, …, τ_{n_i} ($\mathbf{a}$, σ_{n_i}) $\neq$ c_{n_i}.
Нека $\sigma_1 = (\sigma_{11}, \sigma_{12}, \ldots, \sigma_{1k})$, $\sigma_{1j} \subseteq \sigma_j$, $j = 1, 2, \ldots, k$.
Нека $\sigma_2 = (\sigma_{21}, \sigma_{22}, \ldots, \sigma_{2k})$, $\sigma_{2j} \subseteq \sigma_j$, $j = 1, 2, \ldots, k$.
…
Нека $\sigma_{n_i} = (\sigma_{n_i 1}, \sigma_{n_i 2}, \ldots, \sigma_{n_i k})$, $\sigma_{n_i j} \subseteq \sigma_j$, $j = 1, 2, \ldots, k$.
Нека $\sigma^1 = \sigma_{11} \cup \sigma_{21} \cup \ldots \cup \sigma_{n_i 1}$, $\sigma^2 = \sigma_{12} \cup \sigma_{22} \cup \ldots \cup \sigma_{n_i 2}$, …,

$\Phi^k = \Phi_{1k} \Box \Phi_{2k} \Box \ldots \Box \Phi_{n_i k}$. Отново $\Phi^1, \Phi^2, \ldots, \Phi^k$ са крайни функции и $\Phi^1 \Box \Phi_1, \Phi^2 \Box \Phi_2, \ldots, \Phi^k \Box \Phi_1$. Нека $\Phi^{i*} = \Phi^i \Box \{ (c_1, c_2, \ldots, c_{n_i}, b) \}$.
Тогава Φ^{i*} е крайна подфункция на Φ_i, тъй като $\Phi_i (c_1, c_2, \ldots, c_{n_i}) \prime b$.
Нека $\Phi = (\Phi^1, \ldots, \Phi^{i-1}, \Phi^{i*}, \Phi^{i+1}, \ldots, \Phi^k)$. Тогава $\Phi \Box \Phi$ и Φ са крайни функции. Естествено, $\Phi_1 \Box \Phi, \Phi_2 \Box \Phi, \ldots, \Phi_{n_i} \Box \Phi$ и от монотонността получаваме, че $\rightarrow_1 (\mathbf{a}, \Phi) \prime c_1, \rightarrow_2 (\mathbf{a}, \Phi) \prime c_2, \ldots, \rightarrow_{n_i} (\mathbf{a}, \Phi) \prime c_{n_i}$.
Освен това, $\Phi^{i*} (c_1, c_2, \ldots, c_{n_i}) \prime b \Box \rightarrow (\mathbf{a}, \Phi) \prime b$.

Лема: Нека $\Phi \Box \boldsymbol{F_n}$ и Φ е крайна, $\{ f_k \}$ е монотонно растяща редица от елементи на $\boldsymbol{F_n}$. Нека $\Phi \Box \bigcup_k f_k$. Тогава съществува $m \Box \mathbb{N}$, такова че $\Phi \Box f_m$.
Доказателство: Тривиална проверка.

Твърдение (за непрекъснатост): Нека $\rightarrow$ (**X**, **F**) е терм.
Нека $\Phi_0 \Box \Phi_1 \Box \ldots \Box \Phi_n \Box \ldots$е монотонно растяща редица от елементи на $\boldsymbol{F}$. За всеки $\mathbf{a} \Box \mathbb{N}^n$ и константа b от типа на $\rightarrow$ имаме
$\rightarrow (\mathbf{a}, \bigcup_n \Phi_n) \prime b \Box$ съществува $n \Box \mathbb{N}$, такова че $\rightarrow (\mathbf{a}, \Phi_n) \prime b$.
Доказателство: Нека съществува $n \Box \mathbb{N}$, такова че $\rightarrow (\mathbf{a}, \Phi_n) \prime b$.
Имаме $\Phi_n \Box \bigcup_n \Phi_n$ и от монотонността получаваме $\rightarrow (\mathbf{a}, \bigcup_n \Phi_n) \prime b$.
Нека $\rightarrow (\mathbf{a}, \bigcup_n \Phi_n) \prime b$. От твърдението за компактност съществуват крайни подфункции $\Phi \Box \bigcup_n \Phi_n$, такива че $\rightarrow (\mathbf{a}, \Phi) \prime b$.
Нека $\Phi = (\Phi_1, \Phi_2, \ldots, \Phi_k)$.
Имаме $\bigcup_n \Phi_n = \bigcup_n (\Phi_{n1}, \Phi_{n2}, \ldots, \Phi_{nk}) \Box (\bigcup_n \Phi_{n1}, \bigcup_n \Phi_{n2}, \ldots, \bigcup_n \Phi_{nk})$.
Тъй като $\Phi \Box \bigcup_n \Phi_n$, то $\Phi_1 \Box \bigcup_n \Phi_{n1}, \Phi_2 \Box \bigcup_n \Phi_{n2}, \ldots, \Phi_k \Box \bigcup_n \Phi_{nk}$.
Имаме, че $\Phi_1, \Phi_2, \ldots, \Phi_k$ са крайни функции и от лемата получаваме, че съществуват $n_1, n_2, \ldots, n_k$, такива че $\Phi_1 \Box \Phi_{n_1 1}, \Phi_2 \Box \Phi_{n_2 2}, \ldots, \Phi_k \Box \Phi_{n_k k}$. Нека $n = \max (n_1, n_2, \ldots, n_k)$.
Тъй като редиците $\{ \Phi_{m1} \}, \{ \Phi_{m2} \}, \ldots, \{ \Phi_{mk} \}$ са монотонни, то $\Phi_1 \Box \Phi_{n1}, \Phi_2 \Box \Phi_{n2}, \ldots, \Phi_k \Box \Phi_{nk} \Box \Box \Box \Phi_n$. Накрая от твърдението за монотонност получаваме $\rightarrow (\mathbf{a}, \Phi_n) \prime b$.

Нека $\rightarrow (X_1, \ldots, X_n, F_1^{n_1}, F_2^{n_2}, \ldots, F_k^{n_k})$ е терм от тип Nat или условен терм.
Дефинираме изображение $\Gamma_{\rightarrow} : \boldsymbol{F} \Box \boldsymbol{F_n}$ по следния начин:

$\Gamma_{\tau}(\alpha_1, \alpha_2, \ldots, \alpha_k)(a_1, a_2, \ldots, a_n) \simeq \tau(a_1, \ldots, a_n, \alpha_1, \ldots, \alpha_k)$ за всяко $(a_1, a_2, \ldots, a_n) \in \mathbb{N}^n$, $(\alpha_1, \alpha_2, \ldots, \alpha_k) \in \boldsymbol{F}$.

Теорема: При въведените означения, Γ_{τ} е непрекъснато изображение на $\boldsymbol{F} = \boldsymbol{F}_{n_1} \times \boldsymbol{F}_{n_2} \times \ldots \times \boldsymbol{F}_{n_k}$ във $\boldsymbol{F}_n$.

Доказателство: Непосредствено следствие от твърденията за монотонност и непрекъснатост.

Нека **R** е програма от вида

$\tau_0(X_1, \ldots, X_n, F_1^{n_1}, F_2^{n_2}, \ldots, F_k^{n_k})$, where

$F_i^{n_i}(X_1, \ldots, X_{n_i}) = \tau_i(X_1, \ldots, X_{n_i}, F_1^{n_1}, F_2^{n_2}, \ldots, F_k^{n_k})$, i = 1, 2, …, k.

Нека $\Gamma_i : \boldsymbol{F} \to \boldsymbol{F}_{n_i}$, $\Gamma_i = \Gamma_{\tau_i}$, i = 1, 2, …, k.

Разглеждаме системата за изображенията $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$:

$\Gamma_1(\chi_1, \chi_2, \ldots, \chi_k) = \chi_1$

$\Gamma_2(\chi_1, \chi_2, \ldots, \chi_k) = \chi_2$

…

$\Gamma_k(\chi_1, \chi_2, \ldots, \chi_k) = \chi_k$

Тъй като $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ са непрекъснати изображения (предната теорема), то системата притежава най-малко решение $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$. При това α е и най-малко квазирешение.

Функцията $\mathbf{D_V}(\mathbf{R}) : \mathbb{N}^n \to \mathbb{N}$, дефинирана с $\mathbf{D_V}(\mathbf{R})(\mathbf{a}) \simeq \tau_0(\mathbf{a}, \alpha)$ за всяко $\mathbf{a} \in \mathbb{N}^n$, където α е точно това най-малко решение наричаме **денотационна семантика** на програмата **R** с **предаване по стойност**.

Сега ще покажем, че денотационната и операционната семантика съвпадат за всяка програма **R**.

По-долу с α ще означаваме най-малкото (квази)решение на системата за $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$.

Твърдение: Нека $\tau(\mathbf{F})$ е функционален терм, c е константа и $\mathbf{R} \vdash_{\mathrm{V}} \tau \downarrow c$. Тогава $\tau(\alpha) \simeq c$.

Доказателство: Провеждаме пълна индукция по дължината на извода $\mathbf{R} \vdash_{\mathrm{V}} \tau \downarrow c$. Нека $\mathbf{R} \vdash_{\mathrm{V}} \tau \downarrow c$ с дължина на извода k и твърдението е вярно за изводи $\mathbf{R} \vdash_{\mathrm{V}} \tau \downarrow c$ с дължини по-малки от k.

Разглеждаме случаи в зависимост от вида на τ.

Случаите, когато τ е константа, τ е основна операция или τ е условен терм следват непосредствено от индукционното предположение и дефиницията за стойност на терм.

Интересният случай е $\tau = F_i^{n_i}(\tau_1, \tau_2, \ldots, \tau_{n_i})$. Имаме $\mathbf{R} \vdash_{\mathrm{V}} \tau \downarrow c$ с дължина на извода k, така че от лемата за извода съществуват

константи $c_1, c_2, \ldots, c_{n_i}$, такива че $\mathbf{R} \models_V \tau_j[\beta] = c_j$, $j = 1, 2, \ldots, n_i$ с дължини на изводите по-малки от k и

$\mathbf{R} \models_V \alpha_i(X_1/c_1, X_2/c_2, \ldots, X_{n_i}/c_{n_i}, \mathbf{F})[\beta] = c$ с с дължина на извода по-малка от k. От индукционното предположение имаме $\tau_j[\beta] = c_j$, $j = 1, 2, \ldots, n_i$ и $\alpha_i(\mathbf{X}/\mathbf{c}, \mathbf{F})(\beta) = c$. От лемата за съгласуване на заместване и стойност получаваме
$\alpha_i(\mathbf{c}, \beta) = c$. Имаме $\tau[\beta] = \beta_i(\tau_1[\beta], \tau_2[\beta], \ldots, \tau_{n_i}[\beta]) = \beta_i(\mathbf{c}) =$
$= \Gamma_i(\beta)(\mathbf{c}) = \alpha_i(\mathbf{c}, \beta) = c$. Използвали сме, че β е решение на системата за $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$.

Теорема: $O_V(\mathbf{R}) \subseteq D_V(\mathbf{R})$.

Доказателство: Нека $O_V(\mathbf{R})(\mathbf{a}) = b$. Тогава $\mathbf{R} \models_V \alpha_0(\mathbf{X}/\mathbf{a}, \mathbf{F})[\beta] = b$.

От предното твърдение получаваме $\alpha_0(\mathbf{X}/\mathbf{a}, \mathbf{F})(\beta) = b$. Прилагаме лемата за съгласуване на заместване и стойност и получаваме $\alpha_0(\mathbf{a}, \beta) = b \Rightarrow D_V(\mathbf{R})(\mathbf{a}) = b$. Така $O_V(\mathbf{R}) \subseteq D_V(\mathbf{R})$.

За всяко $i \in \{1, 2, \ldots, k\}$ дефинираме функцията $\beta_i : N^{n_i} \to N$ по следния начин:

$\beta_i(a_1, a_2, \ldots, a_{n_i}) = b \Leftrightarrow \mathbf{R} \models_V \alpha_i(X_1/a_1, X_2/a_2, \ldots, X_{n_i}/a_{n_i}, \mathbf{F})[\beta] = b$.

Функцията β_i е добре дефинирана от еднозначността на опростяването по стойност.

Лема: Нека $\tau(\mathbf{F})$ е функционален терм и $\tau[\beta] = b$.

Тогава $\mathbf{R} \models_V \tau[\beta] = b$.

Доказателство: Индукция по построението на τ.
Случаите, когато τ е константа, τ е основна операция или τ е условен терм следват непосредствено от дефиницията за стойност на терм, от индукционното предположение и от правилата за опростяване. По-интересен е следният случай:
$\tau = F_i^{n_i}(\tau_1, \tau_2, \ldots, \tau_{n_i})$ и твърдението е изпълнено за термовете $\tau_1, \tau_2, \ldots, \tau_{n_i}$. Имаме $\tau[\beta] = b \Rightarrow$ съществуват $b_1, b_2, \ldots, b_{n_i}$, такива че $\tau_1[\beta] = b_1, \tau_2[\beta] = b_2, \ldots, \tau_{n_i}[\beta] = b_{n_i}$ и $\beta_i(b_1, b_2, \ldots, b_{n_i}) = b$. От индукционното предположение $\mathbf{R} \models_V \tau_j[\beta] = b_j$, $j = 1, 2, \ldots, n_i$. Освен това $\beta_i(b_1, b_2, \ldots, b_{n_i}) = b \Rightarrow \mathbf{R} \models_V \alpha_i(X_1/b_1, X_2/b_2, \ldots, X_{n_i}/b_{n_i}, \mathbf{F})[\beta] = b$. От правило 3. получаваме $\mathbf{R} \models_V \tau[\beta] = b$.

Следствие: β е квазирешение на системата за $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$.

Доказателство: Нека Γ_i ($\square$)(**a**) $\nearrow$ b. Тогава $\rightsquigarrow_i$ (**a**, $\square$) $\nearrow$ b. От лемата за съгласуване на заместване и стойност получаваме $\rightsquigarrow_i$ (**X**/**a**, **F**)($\square$) $\nearrow$ b и от предната лема **R** $\overset{\nabla}{\Vdash}$ $\rightsquigarrow_i$ (**X**/**a**, **F**) $\square$ b. Съгласно дефиницията на $\square_i$ получаваме $\square_i$ (**a**) $\nearrow$ b. Така Γ_i ($\square$) $\square$ $\square_i$, i = 1, 2, …, k $\square$ $\square$ е квазирешение на системата за Γ_1, Γ_2, …, Γ_k.

Следствие: $\square$ $\square$ $\square$.
Доказателство: Тъй като $\square$ е квазирешение на системата за Γ_1, Γ_2, …, Γ_k, а $\square$ е най-малкото квазирешение на тази система, то $\square$ $\square$ $\square$.

Теорема: $\mathbf{D_V}$ (R) $\square$ **$\mathbf{O_V}$ (R)**.
Доказателство: Нека **$\mathbf{D_V}$ (R)**(**a**) $\nearrow$ b. Тогава $\rightsquigarrow_0$ (**a**, $\square$) $\nearrow$ b, $\square$ $\square$ $\square$ и от твърдението за монотонност получаваме $\rightsquigarrow_0$ (**a**, $\square$) $\nearrow$ b. От лемата за съгласуване на заместване и стойност $\rightsquigarrow_0$ (**X**/**a**, **F**)($\square$) $\nearrow$ b и от лемата по-горе получаваме **R** $\overset{\nabla}{\Vdash}$ $\rightsquigarrow_0$ (**X**/**a**, **F**) $\square$ b, т.е. **$\mathbf{O_V}$ (R)**(**a**) $\nearrow$ b.
Така **$\mathbf{D_V}$ (R)** $\square$ **$\mathbf{O_V}$ (R)**.

Като комбинираме двете теореми получаваме **$\mathbf{O_V}$ (R)** = **$\mathbf{D_V}$ (R)**, с което показахме еквивалентността на операционната и денотационната семантика с предаване по стойност на рекурсивните програми.

Нека P е свойство на функциите от $\boldsymbol{F_n}$, т.е. P : $\boldsymbol{F_n}$ $\square$ { tt, ff }.
Казваме, че P е (**изброимо**) **непрекъснато**, ако за всяка монотонно растяща редица от функции { f_k}, f_k $\square$ $\boldsymbol{F_n}$, P (f_k) за всяко k $\square$ $\mathbb{N}$ $\square$ P ($\bigcup_k f_k$).

Теорема (правило на Скот): Нека P е непрекъснато свойство, **Γ** : $\boldsymbol{F_n}$ $\square$ $\boldsymbol{F_n}$ е непрекъснато изображение.
Нека P ($\square$**!**) и за всяка g $\square$ $\boldsymbol{F_n}$, P (g) $\square$ P (**Γ** (g)).
Тогава P (f), където f е най-малката неподвижна точка на **Γ**.
Доказателство: От доказателството на теоремата на Кнастер-Тарски имаме f = $\bigcup_k f_k$, f_0 = $\square$**!**, f_{k+1} = **Γ** (f_k) за всяко k $\square$ $\mathbb{N}$.
С индукция по k ще покажем, че P (f_k) за всяко k $\square$ $\mathbb{N}$.
База: При k = 0, f_0 = $\square$**!** и P (f_0) е вярно по условие.
Предположение: Нека е доказано P (f_k).
Стъпка: Имаме P (f_k) $\square$ P (**Γ** (f_k)), P (f_k) е вярно $\square$ P (**Γ** (f_k)), т.е. P (f_{k+1}).
Тъй като { f_k} е монотонно растяща редица и P е непрекъснато свойство имаме P ($\bigcup_k f_k$), т.е. P (f).

Ще дадем някои примери за непрекъснати свойства.

Свойство P от вида:
P (f) ⇔ за всяко **x** ∈ $\mathbb{N}^n$, (**!** f (**x**) & I (**x**)) ⇒ O (**x**, f (**x**)), където
I (**x**) и O (**x**, y) са произволни предикати над естените числа се нарича **условие за частична коректност**. Ясно е, че P (⊥**!**) е вярно при всеки избор на предикатите I, O. Лесно се вижда, че P е непрекъснато.

Свойство P от вида:
P (f) ⇔ за всяко **x** ∈ $\mathbf{N}^n$, I (**x**) ⇒ (**!** f (**x**) и O (**x**, f (**x**))), където I, O имат същия смисъл се нарича **условие за тотална коректност**. Ясно е, че P (⊥**!**) е вярно ⇔ I (**x**) ⇒ ff при всеки избор на O. Лесно се вижда, че P е непрекъснато. Правилото на Скот, обаче, не е приложимо за P, тъй като P (⊥**!**) не е вярно (освен в тривиалния случай когато I ⇒ ff).

Нека $\mathbf{\Gamma_1} : \boldsymbol{F_m} \to \boldsymbol{F_n}$ и $\mathbf{\Gamma_2} : \boldsymbol{F_m} \to \boldsymbol{F_n}$ са непрекъснати изображения. Тогава свойството P, дефинирано с P (g) ⇔ $\mathbf{\Gamma_1}$ (g) ⊑ $\mathbf{\Gamma_2}$ (g) е непрекъснато.

Твърдение: Нека P_1 и P_2 са непрекъснати свойства. Тогава свойството P (g), дефинирано с P (g) ⇔ P_1 (g) & P_2 (g) е непрекъснато.
Доказателство: Ако P е вярно за всеки член на някоя монотонно растяща редица, то P_1 и P_2 също са вярни за всеки член на редицата, откъдето лесно следва твърдението.

Следствие: Нека $\mathbf{\Gamma_1} : \boldsymbol{F_m} \to \boldsymbol{F_n}$ и $\mathbf{\Gamma_2} : \boldsymbol{F_m} \to \boldsymbol{F_n}$ са непрекъснати изображения. Тогава свойството P, дефинирано с P (g) ⇔ $\mathbf{\Gamma_1}$ (g) = $\mathbf{\Gamma_2}$ (g) е непрекъснато.
Доказателство: $\mathbf{\Gamma_1}$ (g) = $\mathbf{\Gamma_2}$ (g) ⇔ $\mathbf{\Gamma_1}$ (g) ⊑ $\mathbf{\Gamma_2}$ (g) & $\mathbf{\Gamma_2}$ (g) ⊑ $\mathbf{\Gamma_1}$ (g).

Твърдение: Нека P_1 и P_2 са непрекъснати свойства. Тогава свойството P (g), дефинирано с P (g) ⇔ P_1 (g) ⇒ P_2 (g) е непрекъснато.
Доказателство: Ако P е вярно за всеки член на някоя монотонно растяща редица, то поне едно от P_1 и P_2 е вярно за безброй много членове на редицата, откъдето лесно следва твърдението.

Ще отбележим, че лесно се проверява (чрез пример), че отрицание на непрекъснато свойство не винаги е непрекъснато.

5. Релационна алгебра. Релационно смятане. Нормални форми.

Домен е множество от стойности, подобно на тип данни (но не е тип, защото няма операции). Релационният модел изисква стойностите, които съдържат домените да са атомарни, т.е. да не

могат да бъдат разбити на по-малки компоненти. Например, домени са множеството на целите числа, множеството на символните низове (с или без фиксирана дължина).
Нека D_1, D_2, …, D_k са домени. **Релация** наричаме всяко подмножество R на декартовото произведение D_1 x D_2 x …x D_k. Числото k наричаме **размерност** (**степен**) на релацията.
Елементите на една релация R наричаме **кортежи**. Всеки кортеж на една k-мерна релация има k компонента. За краткост кортежите означаваме по следния начин: $v_1v_2…v_k$, където $v_i \in D_i$ е i-тият компонент на кортежа.
Тъй като става дума за бази данни, които физически се съхраняват върху някаква крайна памет, ще считаме че всички релации са крайни. Празното множество също е релация.
Полезно е релацията да се разглежда като таблица. Всеки ред в таблицата е кортеж и всяка колона съответства на един компонент от съответния домен. На колоните често им се дава имена, които се наричат **атрибути**. Обикновено атрибутите описват значението на стойностите в колоните на релацията.
Всяка релация си има име. Името на релацията, заедно с множеството от атрибутите на релацията образуват **схемата на релацията**. Ако името на релацията е REL и нейните атрибути са A_1, A_2, …, A_k, то релационната схема означаваме по следния начин: REL (A_1, A_2, …, A_k). Атрибутите в схемата на една релация са множество, а не списък, т.е. няма наредба на атрибутите. При разместване на атрибутите или кортежите, получаваме релация, която е еквивалентна на първоначалната. Въпреки това се предполага, че при визуализация атрибутите се записват в определен стандартен ред. Този стандартен ред винаги ще определяме от записа на схемата на релацията.
Схемата на една релационна база данни наричаме множеството от схемите на релациите, които участват в базата данни.
Релациите не са статични, те се променят в течение на времето – добавят се нови кортежи, променят се съществуващи кортежи, изтриват се съществуващи кортежи. По-рядко се променя схемата на една релация – добавяне или изтриване на атрибути. В една голяма релация това са твърде тежки операции.
Реализацията на една релационна база данни се характеризира с независимост на данните. С други думи, вътрешната структура на имплементация на базата данни се скрива от външната концептуална структура. Концептуално данните са организирани в таблици, физически имаме записи в някаква файлова структура, които представят кортежите. За да бъде подобрена ефективността на заявките се използват различни механизми за индексиране като хеширане и B^+ дървета.
Потребителите търсят информация в релационната база данни като образуват **заявки**, написани на някоя от формалните нотации за изразяване на операции върху релации. Тези нотации са два вида:

1. Алгебрическа нотация, наречена **релационна алгебра**, където заявките се изразяват чрез прилагане на специални операции към релациите.
2. Логическа нотация, наречена **релационно смятане**, където заявките се изразяват чрез писане на логически формули, които трябва да удоволетворяват кортежите в резултата.

Двете нотации (при някои ограничения) имат еквивалентна изразителна мощ, т.е. всяка от тях може да изрази произволна заявка, която другата може.

Релационната алгебра е нотация за описване на заявки към релации. Множеството от операциите, които осигурява релационната алгебра не е пълно по Тюринг, в смисъл че съществуват операции, които не могат да се опишат със средствата на релационната алгебра, но могат да се опишат например на C++ или на кой да е от обикновените езици за програмиране. Предимството е, че това дава възможност за по-ефективно оптимизиране на заявките.
Операциите от релационната алгебра формално се извършват върху релации, които се разглеждат като множества от кортежи.

Съществуват пет основни операции, които се дефинират в релационната алгебра.

Първите две основни операции са обикновени операции върху множества: **обединение** и **разлика**.
За да могат да се приложат тези операции върху две релации R и S, то трябва да са изпълнени следните условия:

- R и S трябва да имат едни и същи схеми с идентични множества от атрибути, както и идентични домени за тези атрибути;
- преди да се изпълни операцията колоните на R и S трябва да се подредят по един и същи начин.

Обединението на R и S бележим с R S и то представлява множеството от кортежи, които се срещат в R, в S или едновременно в R и S.
Разликата на R и S бележим с R - S и тя представлява множеството от кортежи, които се срещат в R, но не се срещат в S.

Третата основна операция служи за комбиниране на кортежите в две релации. Тя се нарича **декартово произведение**.
Декартовото произведение на релациите R и S бележим с R x S и то представлява множеството от всевъзможните кортежи, които са съединение на кортеж от R с кортеж от S. Схемата на R x S е обединение на схемите на R и S с тази особеност, че ако R и S имат атрибути с еднакви имена предварително трябва да се извърши преименуване на дублиращите се атрибути. Използваме точкова нотация - за да различаваме общия атрибут A на релациите R и S записваме R.A за атрибута от R и S.A за атрибута

от S. Ще считаме, че атрибутите на R предшестват атрибутите на S в схемата на R x S.

Последните две основни операции са унарни и чрез тях се премахва част от релация. Те се наричат **проекция** и **селекция**. Операцията проекция, приложена върху релация R се използва за образуване на нова релация, в която участват само някои от колоните на R. Бележим $\prec_{A_1,A_2,\ldots,A_n}(R)$, където A_1, A_2, …, A_n са част от атрибутите на R. Схемата на новата релация се състои само от атрибутите A_1, A_2, …, A_n. Кортежите на новата релация са проекции на кортежите на R върху атрибутите A_1, A_2, …, A_n, с други думи кортежът $v_1v_2\ldots v_n$ присъства в новата релация точно тогава, когато съществува кортеж на R, който има стойност v_i за компонентата, отговаряща на атрибута A_i за всяко i $\in$ { 1, 2, …, n }.
Резултатът от операцията селекция, приложена върху релация R е друга релация с множество кортежи, което е подмножество на кортежите на R. Кортежите, които се включват в резултата са точно онези кортежи от R, които удоволетворяват някакво условие C за стойностите на атрибутите.
Резултатът от селекцията бележим по следния начин: $\leftarrow_C (R)$.
Схемата на релацията $\leftarrow_C (R)$ съвпада със схемата на R.
C е условен израз, в който като операнди участват константи или имена на атрибути на R. В C могат да се използват различните операции за сравнение =, <, >, $\leq$, $\geq$, $\neq$, както и логическите съюзи $\wedge$, $\vee$, $\neg$. Условието C се прилага към всеки кортеж t на релацията R, като името на всеки атрибут в условието C се замества със съответната му стойност от кортежа t. Ако условието се оцени като истина, то кортежът t се включва в релацията $\leftarrow_C (R)$, в противен случай не се включва.

В релационната алгебра са дефинирани и някои допълнителни операции, които могат да бъдат изразени с помощта на основните пет операции.

Операцията **сечение** може да се прилага върху две релации R и S, за които са изпълнени условията, описани при операциите обединение и разлика. Сечението на R и S бележим с R $\cap$ S и то представлява множеството от кортежи, които се срещат едновременно в R и S. Сечението се изразява чрез разлика по следния начин: R $\cap$ S = R - (R - S).

Операцията **частно** може да се прилага върху две релации R и S, ако са изпълнени следните условия:

- ако схемата на R е R (A_1, A_2, …, A_n), то схемата на S трябва да има вида S (A_i, A_{i+1}, …, A_n), 1 < i $\leq$ n;
- релацията S е непразна.

Частното на R и S бележим с R $\div$ S и то представлява множеството от всички кортежи $v_1v_2\ldots v_{i-1}$, такива че за всеки кортеж $v_iv_{+1}\ldots v_n$

на S кортежът $v_1v_2...v_n$ е кортеж на R. С други думи, R ÷ S е най-голямата релация, такава че (R ÷ S) x S ⊆ R.
Частното се изразява чрез петте основни операции така:
$R \div S = \prec_{A_1,A_2,...,A_{i-1}}(R) - \prec_{A_1,A_2,...,A_{i-1}}(\prec_{A_1,A_2,...,A_{i-1}}(R) \times S - R)$.

Операцията **съединение** (**-съединение**) е операция за комбиниране на кортежите на две релации R и S.

Резултатът от операцията означаваме по следния начин: $R \bowtie_C S$.

Схемата на $R \bowtie_C S$ е обединение на схемите на R и S, при това при дублиране на атрибути отново трябва да се използва точковата нотация. С е условен израз, подобен на условния израз от селекцията, но в него могат да участват имена на атрибути и на R и на S. Кортежите на $R \bowtie_C S$ получаваме по следния начин:

- образуваме релацията R x S;
- избираме онези кортежи на R x S, които удоволетворяват условието С (след заместване на имената на атрибутите със съответните стойности).

Ясно е, че -съединението се изразява по следния начин чрез основните операции: $R \bowtie_C S = \sigma_C (R \times S)$.

Операцията **естествено съединение** също е операция за комбиниране на кортежите на две релации R и S. Резултатът от операцията означаваме по следния начин: $R \bowtie S$. Кортежите на $R \bowtie S$ са всевъзможни съединения на кортеж от R с кортеж от S, които се съгласуват по общите атрибути на релациите R и S. По-прецизно, нека $A_1, A_2, ..., A_n$ са общите атрибути в схемите на релациите R и S. Тогава схемата на $R \bowtie S$ е теоретико-множествено обединение на схемите на R и S, т.е. не се използва точковата нотация и атрибутите $A_1, A_2, ..., A_n$ участват само по веднъж в схемата на $R \bowtie S$. Два кортежа r на R и s на S се съединяват успешно, ако те се съгласуват по стойностите на $A_1, A_2, ..., A_n$. Тогава съответният кортеж на $R \bowtie S$ се съгласува по всички атрибути на R с кортежа r и по всички атрибути на S с кортежа s. Естественото съединение се изразява чрез основните операции така:
$R \bowtie S = \prec_L (\sigma_C (R \times S))$. Условието С има вида:
$R.A_1 = S.A_1 \wedge R.A_2 = S.A_2 \wedge ... \wedge R.A_n = S.A_n$.
Списъкът от атрибути L започва с атрибутите на R, последвани от тези атрибути на S, които не са атрибути на R.

Форма на логиката, наречена **релационно смятане** лежи в основата на повечето комерсиални езици за заявки, базирани на релационния модел. Разликата между релационната алгебра и релационното смятане е, че релационната алгебра е операционална – един израз на релационната алгебра дава стъпка

по стъпка процедурата, която трябва да бъде извършена, за да се изчисли резултата. От друга страна, релационното смятане е декларативно – това е език от предикати, които казват дали даден кортеж се съдържа или не в резултата.
Релационното смятане съществува в две разновидности – **релационно смятане с променливи върху кортежи** и **релационно смятане с променливи върху домени**.

Релационното смятане с променливи върху кортежи е вид релационно смятане, при което променливите означават кортежи, а не компоненти на кортежи. Ако s е променлива, A е атрибут, то ще означаваме със s[A] стойността на компонентата на s, отговаряща на атрибута A.
Под **атом** разбираме един от следните изрази:

- R (s), където R е име на релация, s е променлива, атомът R (s) се оценява с истина, ако s е кортеж на релацията с име R, в противен случай се оценява с лъжа;
- s[A] op t[B], където s и t са променливи, A и B са атрибути, op □ { =, <, >, □, □, □ } е операция за аритметично сравнение;
- s[A] op c, c op s[A], където s, A, op имат същия смисъл, а c е константа от домена на атрибута A.

Дефиницията за **формула** е индуктивна, успоредно дефинираме понятията **свързана** и **свободна променлива** на формула.
Всеки атом е формула, атомът няма свързани променливи и всички променливи, участващи в атома са свободни.
Ако F е формула, то □F е формула. Свободните променливи на □F са свободните променливи на F и свързаните променливи на □F са свързаните променливи на F. Формулата □F е истина тогава и само тогава когато F е лъжа.
Ако F_1 и F_2 са формули, то F_1 □ F_2 е формула. Свободните променливи на F_1 □ F_2 са свободните променливи на F_1 и F_2, свързаните променливи на F_1 □ F_2 са свързаните променливи на F_1 и F_2. Формулата F_1 □ F_2 е истина тогава и само тогава когато F_1 е истина и F_2 е истина.
Ако F_1 и F_2 са формули, то F_1 □ F_2 е формула. Свободните променливи на F_1 □ F_2 са свободните променливи на F_1 и F_2, свързаните променливи на F_1 □ F_2 са свързаните променливи на F_1 и F_2. Формулата F_1 □ F_2 е истина тогава и само тогава когато F_1 е истина или F_2 е истина.
Ако F е формула и s е свободна променлива на F, то (^ s)F е формула. Свободните променливи на (^ s)F са свободните променливи на F без s, а свързаните променливи на (^ s)F са свързаните променливи на F и s. Формулата (^ s)F е истина тогава и само тогава когато съществува кортеж с конкретни стойности за компонентите му, такъв че, ако заместим с него свободните срещания на s във F получаваме истинна формула.

Ако F е формула и s е свободна променлива на F, то (∃s)F е формула. Свободните променливи на (∃s)F са свободните променливи на F без s, а свързаните променливи на (∃s)F са свързаните променливи на F и s. Формулата (∃s)F е истина тогава и само тогава когато за всеки кортеж с конкретни стойности за компонентите му, ако заместим с него свободните срещания на s във F получаваме истинна формула.

Заявка на релационното смятане с променливи кортежи наричаме израз от вида { t | F }, където F е формула, която има единствена свободна променлива t. Отговорът на заявката е релацията, съставена от всички кортежи, чиито стойности, когато бъдат заместени в съответните компоненти на променливата t, правят F истинна.

Понякога заявките може да дават безкраен отговор. Затова се налага да се ограничим до разглеждането само на **безопасни** формули. Освен, че трябва да дефинират само крайни релации, тези формули трябва да отговарят и на някои други изисквания. За да определим безопасните формули ни трябват две допълнителни дефиниции.

Първо ще дефинираме какво означава компонентата s[A] на променливата s да бъде **ограничена** в една формула.
Във формулата R (s) всеки компонент на променливата s е ограничен.
Във формулата s[A] op t[B] няма ограничени компоненти.
Във формулите s[A] = c, c = s[A], компонентата s[A] на променливата s е ограничена.
Във формулата s[A] op c, c op s[A], където op е операция за аритметично сравнение, различна от = няма ограничени компоненти.
Във формулата ¬F няма ограничени компоненти.
Във формулата $F_1 \vee F_2$ една компонента на променлива е ограничена точно когато тази компонента е ограничена или във F_1 или във F_2.
Във формулата $F_1 \wedge F_2$ една компонента на променлива е ограничена точно когато тази компонента е ограничена във F_1 и във F_2.
Във формулата (^ s)F една компонента на променлива е ограничена точно когато тази компонента е ограничена във F и променливата е различна от s.
Във формулата (∃s)F една компонента на променлива е ограничена точно когато тази компонента е ограничена във F и променливата е различна от s.

След това дефинираме какво означава компонентите s[A] и t[B] на две променливи s и t да бъдат **изравнени** в една формула.

Във формулата R (s) няма изравнени компоненти на променливи.
Във формулата s[A] op t[B], където op не е = няма изравнени компоненти на променливи.
Във формулата s[A] = t[B], s[A] и t[B] са изравнени компоненти на променливи.
Във формулите s[A] op c, c op s[A], няма изравнени компоненти на променливи.
Във формулата □F няма изравнени компоненти на променливи.
Във формулата F_1 □ F_2 две компоненти на променливи са изравнени точно когато тези компоненти са изравнени или във F_1 или във F_2.
Във формулата F_1 □ F_2 две компоненти на променливи са изравнени точно когато тези компоненти са изравнени във F_1 и във F_2.
Във формулата (^ s)F две компоненти на променливи са изравнени точно когато двете променливи са различни от s и компонентите са изравнени във F.
Във формулата (□s)F две компоненти на променливи са изравнени точно когато двете променливи са различни от s и компонентите са изравнени във F.
Естествено, предполагаме рефлексивност, симетричност и транзитивност: s[A] е изравнена със s[A] във всяка формула, ако s[A] е изравнена с t[B] в една формула, то t[B] е изравнена със s[A] в същата формула и ако s[A] е изравнена с t[B] в една формула и t[B] е изравнена с u[C] в същата формула, то s[A] е изравнена с u[C] във формулата.

Казваме, че формулата F е безопасна, ако:

1. Всяка компонента на всяка свободна променлива на F е изравнена във F с някоя ограничена във F компонента на променлива.
2. За всяка подформула на F от вида (^ t)G имаме, че всяка компонента на t е изравнена в G с някоя ограничена в G компонента на променлива.
3. За всяка подформула на F от вида (□t)G имаме, че всяка компонента на t е изравнена в G с някоя ограничена в G компонента на променлива.

Казваме, че заявката { t | F} е безопасна, ако формулата F е безопасна.

Теорема: Всяка заявка, изразима в релационната алгебра е изразима в безопасното релационно смятане с променливи кортежи.

Релационното смятане с променливи върху домени е вид релационно смятане, при което променливите означават компоненти на кортежи.
Под **атом** разбираме един от следните изрази:

1. R (x_1, x_2, …, x_n), където R е име на релация, x_1, x_2, …, x_n са променливи или константи. Този атом се оценява с истина точно когато $x_1x_2…x_n$ е кортеж на релацията с име R.
2. x op y, където x, y са променливи или константи, op $\in$ { =, <, >, $\leq$, $\geq$, $\neq$ } е операция за аритметично сравнение.

Дефиницията за **формула** е индуктивна и е абсолютно аналогична на дефиницията за формула на релационното смятане с променливи кортежи, с тази разлика, че вместо променлива кортеж вече говорим за променлива върху домен. По-същия начин паралелно се дефинират понятията свободна и свързана променлива на формула.

Заявка на релационното смятане с променливи върху домени наричаме израз от вида { $x_1x_2…x_n$ | F }, където F е формула със свободни променливи x_1, x_2, …, x_n. Отговорът на заявката е релацията, съставена от всички кортежи $a_1a_2…a_n$, такива че, замествайки с a_i всяко свободно срещане на x_i във F за всяко i $\in$ { 1, 2, …, n } получаваме истинна формула.
Както при релационното смятане с променливи кортежи, отговорът на някои заявки може да бъде безкрайна релация. Затова отново се налага да разглеждаме само **безопасни** формули. Дефиницията е както при релационното смятане с променливи кортежи. Първо се дефинира какво означава променливата x да бъде **ограничена** в една формула. След това се дефинира какво означава две променливи x и y да бъдат **изравнени** в една формула. И след това, казваме, че формулата F е безопасна, ако:

1. Всяка свободна променлива на F е изравнена във F с някоя ограничена във F променлива.
2. За всяка подформула на F от вида ($\exists$ y)G имаме, че y е изравнена в G с някоя ограничена в G променлива.
3. За всяка подформула на F от вида ($\forall$y)G имаме, че y е изравнена в G с някоя ограничена в G променлива.

Казваме, че заявката { $x_1x_2…x_n$ | F } е безопасна, ако формулата F е безопасна.

Теорема: Всяка заявка, изразима в релационното смятане с променливи кортежи е изразима в релационното смятане с променливи върху домени.

Следствие: Всяка безопасна заявка, изразима в релационното смятане с променливи кортежи е изразима в безопасното релационно смятане с променливи върху домени.

Теорема: Всяка безопасна заявка, изразима в релационното смятане с променливи върху домени е изразима в релационната алгебра.

При проектиране на схема на една релация трябва да се избягват следните аномалии:

1. Излишество – информацията безсмислено да се дублира в кортежите.
2. Аномалии на обновяването – като следствие от излишеството при обновяване на данните да не се актуализират всички засегнати кортежи, т.е. да не се променят данните на всички места, където се срещат.
3. Аномалии при добавяне – при добавяне на нови данни, които засягат компоненти, отговарящи само на някои атрибути, компонентите, отговарящи на останалите атрибути задължително трябва да получат стойности, при това те трябва да се съгласуват със съответните им стойности в останалите кортежи.
4. Аномалии на изтриването – при изтриване на данни да се губи друга информация като страничен ефект.

Общоприетият начин за елиминиране на изброените аномалии е декомпозицията на релациите – една релация се разбива на две нови релации. По-формално, нека R (A_1, A_2, …, A_n) е релация. Тогава тя се **декомпозира** на две нови релации
S (B_1, B_2, …, B_m) и T (C_1, C_2, …, C_k), ако:

1. { B_1, B_2, …, B_m} □ { C_1, C_2, …, C_k} = { A_1, A_2, …, A_n}.
2. Кортежите в S са проекции на кортежите на R по B_1, B_2, …, B_m.
3. Кортежите в T са проекции на кортежите на R по C_1, C_2, …, C_k.

В много случаи известните факти за реалния свят, че не всяко крайно множество от кортежи може да е екземпляр на дадена релация, дори да имат правилна размерност и стойности от правилните домени. Можем да различим два вида ограничения:

1. **Ограничения**, определени от **семантиката** на елементите от домена – тези ограничения зависят от разбирането за това какъв е смисъла на компонентите на релацията. Например, ако компонентът означава височина на човек, тя не може да бъде 20 метра. Полезно е системата за управление на базите данни да проверява за такива неправдоподобни стойности.
2. **Ограничения**, определени от равенство или неравенство на стойности – тези ограничения не зависят от това каква стойност има даден кортеж в даден компонент, а само от това дали два кортежа са съгласувани по определени компоненти.

Най-важните ограничения от втория тип са **функционалните зависимости**. Функционална зависимост в една релация R наричаме твърдение, което е изпълнено за всички възможни екземпляри на R и има следния вид: ако два кортежа имат едни и същи компоненти, отговарящи на атрибутите A_1, A_2, …, A_n, то те имат едни и същи компоненти, отговарящи на атрибутите

$B_1, B_2, \ldots, B_k$. Бележим функционалната зависимост по следния начин: $A_1A_2\ldots A_n \rightarrow B_1B_2\ldots B_k$.
Казваме, че множеството $\{A_1, A_2, \ldots, A_n\}$ от един или повече атрибути на релацията R образуват **ключ** на R, ако за всеки атрубит B на R е в сила функционалната зависимост $A_1A_2\ldots A_n \rightarrow B$ и $\{A_1, A_2, \ldots, A_n\}$ е минимално по включване с това свойство, т.е. за всяко собствено подмножество $\{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\}$ на $\{A_1, A_2, \ldots, A_n\}$ съществува атрибут B на R, такъв че не е в сила функционалната зависимост $A_{i_1} A_{i_2} \ldots A_{i_k} \rightarrow B$.

Възможно е една релация да има повече от един ключ. Обикновено в такава ситуация се избира един ключ, който се обявява за **първичен ключ**. Първичният ключ се използва при съхранение на релацията. Понякога терминът **възможен ключ** се използва за означаване на произволен ключ на релацията, а терминът ключ се запазва за първичния ключ.

Казваме, че множеството $\{A_1, A_2, \ldots, A_n\}$ от атрибути на релацията R образуват **суперключ** на R, ако за всеки атрубит B на R е в сила функционалната зависимост $A_1A_2 \ldots A_n \rightarrow B$. С други думи, суперключ на една релация R се нарича множество от атрибути, което съдържа ключ.

Ще разгледаме някои правила, чрез които от даден набор функционални зависимости в релация можем да извличаме нови функционални зависимости в тази релация.

Функционалните зависимости за една релация често могат да бъдат представени по различен начин. По-формално, казваме че множествата S и T от функционални зависимости са **еквивалентни**, ако множеството от екземплярите на релацията, удоволетворяващи S съвпада с множеството от екземплярите на релацията, удоволетворяващи T.
По-общо, множеството функционални зависимости T **следва от** множеството от функционални зависимости S, ако всеки екземпляр на релацията, който удоволетворява S, удоволетворява и T.
Бележим $S \rightarrow T$. Естествено, $S \rightarrow T$ и $T \rightarrow S$ тогава и само тогава, когато S и T са еквивалентни.

Нека е дадена функционална зависимост $A_1A_2\ldots A_n \rightarrow B_1B_2\ldots B_m$. Тогава можем да заменим тази функционална зависимост със следните функционални зависимости: $A_1A_2\ldots A_n \rightarrow B_i$, $i = 1, 2, \ldots, m$. Това правило наричаме **правило за разделяне**. Обратно, функционалните зависимости $A_1A_2\ldots A_n \rightarrow B_i$, $i = 1, 2, \ldots, m$ можем да заменяме с една функционална зависимост $A_1A_2\ldots A_n \rightarrow B_1B_2\ldots B_m$. Това правило наричаме **правило за комбиниране**.

Очевидно е, че и в двата случая полученото множество функционални зависимости е еквивалентно на изходното.

Казваме, че функционалната зависимост $A_1A_2…A_n \rightarrow B_1B_2…B_m$ е **тривиална**, ако $\{B_1, B_2, …, B_m\} \subseteq \{A_1, A_2, …, A_n\}$.

Казваме, че функционалната зависимост $A_1A_2…A_n \rightarrow B_1B_2…B_m$ е **нетривиална**, ако съществува $i \in \{1, 2, …, m\}$, такова че $B_i \in \{A_1, A_2, …, A_n\}$.
Казваме, че функционалната зависимост $A_1A_2…A_n \rightarrow B_1B_2…B_m$ е **напълно нетривиална**, ако за всяко $i \in \{1, 2, …, m\}$, имаме $B_i \in \{A_1, A_2, …, A_n\}$.

Нека $A_1A_2…A_n \rightarrow B_1B_2…B_m$ е нетривиална функционална зависимост. Тогава можем да премахнем онези B_i, които съвпадат с някое A_j. Получаваме нова напълно нетривиална функционална зависимост $A_1A_2…A_n \rightarrow C_1C_2…C_k$, която очевидно е еквивалентна на изходната. Това правило наричаме **правило за отстраняване на тривиалните зависимости**.

Нека $A_1A_2…A_n \rightarrow B_1B_2…B_m$ и $B_1B_2…B_m \rightarrow C_1C_2…C_k$ са функционални зависимости. Тогава добавяме нова функционална зависимост $A_1A_2…A_n \rightarrow C_1C_2…C_k$. Това правило наричаме **транзитивно правило**.
Ясно е, че добавената функционална зависимост следва от изходните функционални зависимости, така че получаваме еквивалентно множество от функционални зависимости.
Аксиомите на Армстронг са правила, чрез които може да се извлече всяка функционална зависимост, която следва от дадено множество функционални зависимости. Те са следните:

1. **Рефлексивност** - ако $\{B_1, B_2, …, B_m\} \subseteq \{A_1, A_2, …, A_n\}$, то $A_1A_2…A_n \rightarrow B_1B_2…B_m$.
2. **Нарастване** – ако $A_1A_2…A_n \rightarrow B_1B_2…B_m$ и $C_1, C_2, …, C_k$ са произволни атрибути, то $A_1A_2…A_nC_1C_2…C_k \rightarrow B_1B_2…B_mC_1C_2…C_k$.
3. **Транзитивност** – ако $A_1A_2…A_n \rightarrow B_1B_2…B_m$ и $B_1B_2…B_m \rightarrow C_1C_2…C_k$, то $A_1A_2…A_n \rightarrow C_1C_2…C_k$.

Като цяло нормалните форми са условия, които ако са изпълнени в дадена релация, то в нея със сигурност няма аномалии от определен вид. Целта на декомпозицията е да разбие релациите на по-малки релации, за които е в сила условието за нормална форма.

Казваме, че една релационна схема R се намира в **първа нормална форма** (**1NF**), ако всичките и домени се състоят от атомарни (неделими) стойности.
Казваме, че една релационна схема R се намира във **втора нормална форма** (**2NF**), ако за всяка нетривиална функционална

зависимост $A_1A_2...A_n \rightarrow B$, която е изпълнена за R имаме, че B е елемент на ключ или $\{ A_1, A_2, ..., A_n \}$ не е собствено подмножество на ключ.

Казваме, че една релационна схема R се намира в **трета нормална форма** (**3NF**), ако за всяка нетривиална функционална зависимост $A_1A_2...A_n \rightarrow B$, която е изпълнена за R имаме, че B е елемент на ключ или $\{ A_1, A_2, ..., A_n \}$ е суперключ.

Казваме, че една релационна схема R се намира в **нормална форма на Boyce-Codd** (**BCNF**), ако за всяка нетривиална функционална зависимост $A_1A_2...A_n \rightarrow B$, която е изпълнена за R имаме, че $\{ A_1, A_2, ..., A_n \}$ е суперключ.

Естествено, BCNF $\subset$ 3NF $\subset$ 2NF.

Лесно може да се покаже, че всяка релационна схема с два атрибута е в BCNF.

Многозначна зависимост в една релация R наричаме твърдение, което е изпълнено за всички възможни екземпляри на R и има следния вид: ако компонентите на кортежите, отговарящи на атрибутите $A_1, A_2, ..., A_n$ съвпадат, то компонентите на кортежите, съответни на атрибутите $B_1, B_2, ..., B_m$ са независими от компонентите на кортежите, съответни на всички останали атрибути. Бележим многозначната зависимост по следния начин: $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$.
По-прецизно, казваме че многозначната зависимост $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ е в сила за една релация R, ако за всяка двойка кортежи t, u в екземпляр на R, които се съгласуват по $A_1, A_2, ..., A_n$ съществува кортеж v, такъв че:

- v се съгласува с t и u по $A_1, A_2, ..., A_n$;
- v се съгласува с t по $B_1, B_2, ..., B_m$;
- v се съгласува с u по всички останали атрибути на R.

С други думи при фиксирани стойности на $A_1, A_2, ..., A_n$ съответните стойности на $B_1, B_2, ..., B_m$ и стойностите на всички останали атрибути се комбинират по всевъзможни начини в различни кортежи на релацията R.
Ще разгледаме някои правила за многозначните зависимости, които наподобяват правилата за функционалните зависимости, но има някои разлики. Дефиницията за еквивалентност и следване на множества от функционални зависимости автоматично се пренася и върху многозначни зависимости.

Ако за една релация R е в сила многозначната зависимост $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$, то за нея е в сила многозначната зависимост $A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$, където $C_1, C_2, ..., C_k$ съдържат $B_1, B_2, ..., B_m$ и някои от $A_1, A_2, ..., A_n$. Също за R е в сила многозначната зависимост $A_1A_2...A_n \twoheadrightarrow D_1D_2...D_r$, където

$D_1, D_2, \ldots, D_r$ са тези от $B_1, B_2, \ldots, B_m$, които не са от $A_1, A_2, \ldots, A_n$. Тези две правила наричаме **правила за тривиалните зависимости**.

Ако за една релация R са в сила многозначните зависимости $A_1A_2\ldots A_n \twoheadrightarrow B_1B_2\ldots B_m$ и $B_1B_2\ldots B_m \twoheadrightarrow C_1C_2\ldots C_k$, то за R е в сила многозначната зависимост $A_1A_2\ldots A_n \twoheadrightarrow C_1C_2\ldots C_k$. Това правило наричаме **транзитивно правило**.

Може да се покаже с пример, че за многозначните зависимости не е в сила правилото за разделяне.

Ако функционалната зависимост $A_1A_2\ldots A_n \rightarrow B_1B_2\ldots B_m$ е в сила за една релация R, то за R е в сила многозначната зависимост $A_1A_2\ldots A_n \twoheadrightarrow B_1B_2\ldots B_m$. С други думи, всяка функционална зависимост е многозначна.

За многозначните зависимости е в сила следното **правило за допълнение**, което не е в сила за функционалните зависимости: ако за R е в сила многозначната зависимост $A_1A_2\ldots A_n \twoheadrightarrow B_1B_2\ldots B_m$, то за R е в сила многозначната зависимост $A_1A_2\ldots A_n \twoheadrightarrow C_1C_2\ldots C_k$, където $C_1, C_2, \ldots, C_k$ са всички атрибути на R без $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$.

Казваме, че многозначната зависимост $A_1A_2\ldots A_n \twoheadrightarrow B_1B_2\ldots B_m$ е **нетривиална**, ако:

1. $\{ B_1, B_2, \ldots, B_m \} \cap \{ A_1, A_2, \ldots, A_n \} = \emptyset$.
2. $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$ не изчерпват атрибутите на R.

Казваме, че една релационна схема R е в **четвърта нормална форма** (**4NF**), ако във всяка нетривиална многозначна зависимост $A_1A_2\ldots A_n \twoheadrightarrow B_1B_2\ldots B_m$, която е изпълнена за R имаме, че $\{ A_1, A_2, \ldots, A_n\}$ е суперключ.
Естествено, всяка релация, която се намира в четвърта нормална форма се намира и в трета нормална форма, т.е. 4NF $\subset$ BCNF. Това следва от факта, че всяка функционална зависимост е многозначна. Обратното не е вярно.

Всяка релация с два атрибута се намира в четвърта нормална форма, тъй като в нея не може да има нетривиални многозначни зависимости.

Нека R $(A_1, A_2, \ldots, A_n)$ е релация и R се декомпозира на две релации S $(B_1, B_2, \ldots, B_m)$ и T $(C_1, C_2, \ldots, C_k)$. Казваме, че декомпозицията е със **съединение без загуба**, ако $R = S \bowtie T$.

Теорема: Декомпозицията на релацията R $(A_1, A_2, \ldots, A_n)$ на две релации S $(B_1, B_2, \ldots, B_m)$ и T $(C_1, C_2, \ldots, C_k)$ е със съединение без

загуба тогава и само тогава, когато за R е изпълнена поне една от двете многозначни зависимости
$\{ B_1, B_2, \ldots, B_m\} \rightarrow \{ C_1, C_2, \ldots, C_k\}$ или $\{ B_1, B_2, \ldots, B_m\} - \{ C_1, C_2, \ldots, C_k\}$,
$\{ B_1, B_2, \ldots, B_m\} \rightarrow \{ C_1, C_2, \ldots, C_k\}$ или $\{ C_1, C_2, \ldots, C_k\} - \{ B_1, B_2, \ldots, B_m\}$.

Нека R е релация, която удоволетворява множеството от функционални и многозначни зависимости F. Да предположим, че S е нова релация, която се получава от R с премахване на атрибути. Казваме, че S е **проекция** на R. Тогава функционалните зависимости, които са в сила за S са точно онези, които следват от функционалните зависимости във F и в които участват само атрибути на S. Казваме, че тези функционални зависимости се **проектират** в S. Аналогично се дефинира проекция на многозначни зависимости. Възможно е, обаче, да съществуват многозначни зависимости, които са в сила за S, но не са в сила за R. Такива многозначни зависимости наричаме **вградени** и те могат да създадат проблеми при декомпозирането в 4NF.

Нека R ($A_1, A_2, \ldots, A_n$) е релация, за която е изпълнено множеството от функционални зависимости F и R се декомпозира на две релации S ($B_1, B_2, \ldots, B_m$) и T ($C_1, C_2, \ldots, C_k$). Казваме, че декомпозицията е със **съединение без загуба на функционалните зависимости**, ако $F_1 \cup F_2 \models F$, където F_1 и F_2 са проекциите на F съответно върху S и T.

Чрез подходящи декомпозиции, всяка схема на релация може да се декомпозира на няколко схеми, така че да са изпълнени следните условия:

1. Всички получени релации да са в BCNF (4NF).
2. Декомпозицията да е със съединение без загуба.

Стратегията за декомпозиция, която възприемаме е следната. Нека $A_1 A_2 \ldots A_n \rightarrow (\rightarrow) B_1 B_2 \ldots B_m$ е нетривиална функционална (многозначна) зависимост и $\{ A_1, A_2, \ldots, A_n\}$ не е суперключ. Тогава декомпозираме релацията R на следните две релации:

1. Първата релация има атрибути $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$.
2. Втората релация има атрибути $A_1, A_2, \ldots, A_n$ и всички останали атрибути на R, които не участват във функционалната (многозначната) зависимост.

Ако новополучените релации не са в BCNF (4NF), то към тях прилагаме същата процедура. При това, функционалните зависимости в новите релации се изчисляват чрез проектиране на функционалните зависимости от изходната релация. Многозначните зависимости в новите релации, обаче, както вече споменахме може да не се изчерпват с проектираните многозначни зависимости от изходната релация – възможно е съществуването на вградени многозначни зависимости. Процесът на декомпозиране ще е краен, тъй като винаги получаваме релации с по-малко атрибути, а всяка релация с два атрибута е в BCNF (4NF).

Ще отбележим, че в общия случай декомпозицията в BCNF (4NF) не е със съединение без загуба на функционалните зависимости. Съществува, обаче, алгоритъм за декомпозиция в 3NF, който е със съединение без загуба и запазва функционалните зависимости. Този алгоритъм в повечето случаи се справя с излишествата, породени от фукционални зависимости.

6. Компютърни архитектури – Формати на данните. Вътрешна структура на централен процесор – блокове и конвейерна обработка, инструкции. Структура и йерархия на паметта. Сегментна и странична преадресация. Система за прекъсване – приоритети и обслужване.

Под **персонален компютър** разбираме изчислителна машина, използвана от един човек, за решаване на специфични, лични алгоритмични задачи и проблеми.
Специфичните особености на персоналния компютър са, че той заема малък обем, има проста структура и система от команди, има ограничен обем на основната памет и опростен интерфейс, към който се свързват всички устройства в изчислителната система.
Въпреки голямото разнообразие на производители и размери, персоналните компютри се характеризират с еднакъв модел на вътрешна архитектура. Този модел е изграден от три основни компоненти:

- **централен процесор**;
- **основна памет**;
- **входно-изходни устройства**.

Взаимовръзката между основните компоненти определя логическата организация на съответната компютърна система. Обменът на сигнали, команди и данни се извършва по специални информационни линии. Съществуват два вида логическа организация на компютърните системи:

1. Индивидуални информационни канали.
2. Обща информационна шина.

При първата организация основните компоненти се свързват по схемата на пълен граф, т.е. по схемата “всеки-всеки”.
При втората организация основните компоненти си взаимодействат посредством обща **информационна шина**, която представлява специализирано устройство, отговарящо за пренасянето на информация между компонентите на компютърната система. Схематично организацията е следната:

Предимството на втората организация е, че тя има ниска цена, проста е и добре структурирана и е гъвкава, т.е. възможна е лесна надстройка или замяна на основните компоненти.
Основен недостатъкът на втората организация е неизбежното ограничение, че в даден момент могат да комуникират само два компонента и произтичащата от това по-ниска производителност.
По-нататък ще разглеждаме само втората организация (обща информационна шина).
Централният процесор е основен компонент в архитектурата на персоналния компютър. Всяка задача, която се изпълнява от компютъра, директно или индиректно се изпълнява и контролира от централния процесор. Централният процесор осъществява реализирането на машинните команди (инструкции). Някои негови съставни части са:

- управляващо устройство;
- аритметично-логическо устройство;
- регистри;
- блок за работа с числа с плаваща точка и др.

Регистрите представляват свръхбърза памет, която се използва за съхраняване на информация, върху която се извършват логическите и аритметични операции. Тази информация се съхранява докато процесора е под електрическо напрежение или докато някоя машинна команда не я измени. Според функционалността си регистрите се делят на три вида:

- регистри с общо предназначение – в тях се записват междинни резултати или се използват за адресация;
- регистри за работа с плаваща аритметика;
- служебни регистри – използват се за специални системни функции, към тях се включват флагов регистър (съдържа информация за препълване, пренос, посока на обработка на низове, маскиране на прекъсвания и др.), PC (програмен брояч, сочи към следващата инструкция за изпълнение), SP (стеков указател, сочи към върха на програмния стек), MSW (съдържа управляващи флагове, служещи за синхронизация на работата на централния процесор с входно-изходните устройства и с операционната система).

Централният процесор изпълнява една машинна команда чрез следната последователност от елементарни стъпки, която наричаме **основен цикъл на изпълнение на командата**:

1. Адресиране и извличане на предстоящата за изпълнение команда (адресирането става по съдържанието на PC).

2. Обновяване на PC (съдържанието на PC се увеличава с дължината на извлечената команда).
3. Дешифриране на командата (определят се вида на операцията, броя на операндите, типа на данните и размера на данните).

Тези три стъпки се изпълняват от управляващото устройство.

4. Адресиране на операндите (на базата на информацията от полето на командата по определени методи на адресация се определя местоположението на клетките от паметта, които съдържат операндите).
5. Извличане на операндите.
6. Изпълняване на операцията върху операндите с помощта на аритметично-логическото устройство.
7. Записване на резултата (определя се местоположението на резултата и той се занася в съответната клетка на паметта).
8. Проверява се има ли постъпил сигнал от някоя от сигналните линии за прекъсване. Ако да, управлението се предава на хардуерния механизъм за обработка на прекъсването. След обработване на прекъсването (ако е имало такова) се преминава към стъпка 1. (започва изпълнение на следващата команда).

Съществуват различни типове команди.
Първият тип команди са **аритметично-логическите команди**, които се изпълняват върху операнди, които са цели числа със или без знак. Част от тях изчисляват булевите функции конюнкция, дизюнкция, отрицание, сума по модул 2 и т.н. Друга част са операциите за сравнение – равно, различно, по-голямо, по-малко, по-голямо или равно, по-малко или равно и т.н. Последната част са аритметичните операции събиране, изваждане, умножение, целочислено деление и модул.
Вторият тип команди са **инструкциите за преход**. Те променят последователното изпълнение на програмата чрез промяна на стойността на програмния брояч PC. Включват инструкции за условен преход, инструкции за безусловен преход, инструкции за извикване на подпрограма и инструкции за връщане от подпрограма.
Третият тип команди са **управляващи**. Това са инструкции, които се изпълняват когато процесорът е в привилегирован режим – например чете или пише в служебните регистри. Процесорът преминава в привилегирован режим например при обработка на прекъсване и въобще при изпълнение на системна програма, която е част от операционната система.
Четвъртият тип команди са **транспортни** и те служат за прехвърляне на данни. Основните са команда за прехвърляне на данни от едно място на паметта в друго, команда за прехвърляне на данни от паметта в регистрите и команда за прехвърляне на данни от регистрите в паметта.

Има и други типове команди – например операции върху числа с плаваща точка (събиране, изваждане, умножение, деление, степенуване, коренуване и др.), операции върху низове (прехвърляне, сравнение, конкатенация), мултимедийни операции MMX (специални векторни операции, които имат голямо приложение при обработката на изображения).

Устройствата в персоналния компютър като централен процесор, основна памет, твърд диск, видеоконтролер и т.н. обменят помежду си данни и контролна информация по така наречените шини. **Шината** представлява магистрала, по която може да бъде обменяна информация между две или повече устройства. Шината е споделен ресурс, който всички устройства в конкурентен режим се мъчат да си разпределят. На централния процесор се дава пълен приоритет, когато трябва да използва шината. Налага се въвеждане на специално устройство, което се нарича контролер на шината или арбитър и то извършва разпределянето на шината, т.е. определя кое устройство за колко време да използва шината. Освен за обмен на данни, шината се използва и за предаване на управляваща и адресна информация. Затова условно шината се разделя на три подшини – **подшина за данни**, **подшина за адреси** и **управляваща подшина**. Подшината за данни прехвърля информация между определено място в паметта или входно-изходно устройство и централния процесор. Конкретното място в паметта или входно-изходното устройство се определя от адреса, който се предава по подшината за адреси. Управляващата подшина пренася електрически сигнали, които контролират комуникацията между централния процесор и останалите устройства. По нея се пренасят команди от централния процесор към устройствата и съответно се получават съобщения за статуса на устройствата. Например, управляващата подшина съдържа две линии за четене и за писане, които определят посоката в която се пренасят данните. Други линии са линията за синхронизация, линии по които се предават сигнали за прекъсване и др.
Шината има следните характеристики:

- ширина на подшината за данни – количествена мярка за броя битове, които могат да преминават едновременно по подшината за данни;
- ширина на подшината за адреси – определя максималният размер на основната памет, например чрез 16-битова адресна подшина могат да се адресират 64KB памет, с 32-битова шина могат да се адресират 4GB памет;
- скорост – определя колко пъти шината може да бъде използвана (заемана) в рамките на една секунда, измерва се в MHz.

Памет в персоналния компютър се нарича всеки ресурс, който има свойството да съхранява информация във времето. Паметта

представлява съвкупност от битове. Под един **бит** разбираме количество информация, за която отговаря един електронен елемент. С развитието на компютрите се оказва, че най-удобната адресуема единица е байта. Един **байт** е осем бита. Капацитетът на паметта се мери в байтове. Никоя памет концептуално не може да чете част от байт. Паметта се изгражда като едномерен линеен масив от байтове, които се номерират с последователни номера, наречени **адреси**. Номерацията започва от 0 и стига да последния наличен физически адрес. Паметта се изгражда като устройство, което може да извършва следните две операции:

- четене – подаване на адрес и извеждане съдържанието на съответния байт;
- писане – подаване на адрес и на байт, който се записва на този адрес.

Това е логическият модел на паметта. От архитектурна гледна точка паметта се реализира физически като тримерни матрици. Когато се подаде двоичен адрес има устройство, което го дешифрира – адресът насочва устройството към онези елементи, които съдържат съответните битове. Времето за дешифрация и времето за прочитане са крайни времена. Под време на достъп до паметта разбираме времето от подаването на адреса до извличането на данните. Времето на достъп нараства с обема на паметта.

В паметта данните се представят в двоичен вид.
С помощта на n-битово поле могат да се представят точно 2^n различни стойности.
Първият начин за представяне на целите числа е чрез **прав код**. При него най-старшият бит в n-битовото поле е знаков – 0 означава положително число, 1 означава отрицателно число. Останалите n-1 бита представят абсолютната стойност на числото в двоична бройна система. По този начин при прав код диапазонът от цели числа, който може да се представи с n-битово поле е $-2^{n-1}+1 \ldots 2^{n-1}-1$. При това нулата има две представяния – 000…0 и 100…0. Недостатъкът на правия код е, че събирането и изваждането на цели числа се реализират трудно апаратно.
Вторият начин за представяне на целите числа е чрез **обратен код**. Отново най-старшият бит е знаков. Разликата от правия код е в представянето на отрицателните числа – то се образува чрез инвертиране на двоичното представяне на абсолютната стойност на числото. Диапазонът при n-битово поле отново е $-2^{n-1}+1 \ldots 2^{n-1}-1$. Нулата отново има две представяния – 00…0 и 11…1. При обратния код събирането и изваждането се реализират по-ефективно – числата, независимо от знака си, се събират като числа в двоична бройна система, при това, ако има пренос от най-старшия бит навън, то към резултата се добавя 1.
Третият начин за представяне на целите числа е чрез **допълнителен код**. Отново най-старшият бит е знаков. Разликата от правия код е в представянето на отрицателните числа – то се

образува чрез инвертиране на двоичното представяне на абсолютната стойност на числото (както при обратния код) и след това добавяне на 1. Диапазонът при n-битово поле е -2^{n-1} ...$2^{n-1}-1$. Нулата вече има единствено представяне – 00...0. При допълнителния код събирането и изваждането се реализират най-ефективно – числата, независимо от знака си, се събират като числа в двоична бройна система, при това преносът от най-старшия бит навън се игнорира.
Има още един за представяне на числата – така наречените **двоично-десетични числа**. При тях всяка десетична цифра се представя с уникална двоична последователност. Предимството е, че се дава възможност за лесно извеждане на числа. Недостатъкът е, че се усложняват аритметичните операции. Една цифра обикновено се представя с четири бита, които в общия случай представят стойностите/цифрите/символите 0-9. Така 6 комбинации остават неизползвани. Тъй като компютрите съхраняват данни под формата на байтове има два начина за съхраняване на четири битовите цифри: всяка цифра се съхранява в отделен байт (непакетирана форма) или в един байт се съхраняват две цифри (пакетирана форма). Едно n-байтово пакетирано двоично-десетично число може да се състои от най-много 2.n-1 цифри, едната е резервирана за знак.
Реалните числа в паметта се представят чрез рационални апроксимации.
Първият подход е рационалните апроксимации да са с **фиксирана точка**. С други думи, в n-битово поле, първите m бита представят цялата част на числото, а останалите n-m бита представят дробната част на числото – десетичната точка е на фиксирана позиция. Проблемът на този подход е, че не винаги се използват всички битове.
Вторият подход е рационалните апроксимации да са с **плаваща точка**. Всяко число с плаваща точка има вида $m.2^p$, където m е цяло число, което се нарича мантиса, а p е цяло число, което се нарича порядък. Казваме, че едно число с плаваща точка е в нормализирана форма, ако най-старшият бит на мантисата е 1. Всяко число с плаваща точка може да бъде нормализирано чрез изместване на мантисата наляво и съответно отразяване в порядъка. Нормализираната форма на едно число с плаваща точка е единствена и обикновено тя се използва за представяне на числото.
За универсално представяне на символите в паметта се постъпва по следния начин : съставя се **кодова таблица**, в която всеки символ от дадена азбука се асоциира с битова поредица с определена дължина. Съществуват различни кодови таблици. Примери са EBCDIC – 8-битова кодова таблица на IBM, UNICODE – 16-битова универсална кодова таблица, ASCII – първоначално замислена като 7-битова таблица, но впоследствие разширена до 8-битова за нуждите на различни националности. В различните

ASCII таблици символите, кодирани с 0 до 127 са едни и същи, а символите, кодирани със 128 до 255 могат да бъдат различни.

Исторически, ограничаващият фактор в производителността на персоналния компютър е времето за достъп до паметта. Скоростта на паметта е с порядъци по-малка от скоростта, с която работи централният процесор. Теоретично адресното пространство трябва да е много голямо, за да може да побере цялата информация. Но в един малък интервал от време реално не се използват всичките тези адреси. Поради тази причина се изгражда йерархия на паметта. Целта е адресите, които в даден момент се използват от централния процесор да се докарат по-близо до него.
Схемата е следната: (в централния процесор) регистри □□ кеш на първо ниво ($L1 кеш) □□ кеш на второ ниво (L2 кеш) □□ (извън централния процесор) кеш на трето ниво (L3 кеш) □□ основна памет □□ виртуална памет (swap-файлове върху диск).
Естествено, в схемата се включва и входно-изходната система, която действа с данни извън адресното пространство и обменя информация с основната памет и със swap-файловете.

До регистрите има паралелен достъп от процесора. L1 кешът е разделен на две независими части – в едната се пазят само инструкции, в другата само данни. Знакът $ пред L1 означава, че тази структура работи невидимо за програмата. L2 кешът се намира в процесора и е част от общото адресно пространство, с което работят инструкциите. L3 кешът е извън процесора и той е по-голям от L2 кешът по обем памет. Основната памет не съдържа всички адреси от общото адресно пространство, реално всички адреси могат да се поместят само върху диска.

Йерархията на паметта се базира на общото положение, че за малък период от време са нужни малко адреси. Нейната задача е да се реализира бърза памет чрез двете концепции за локалност:

- **темпорална локалност** – най-вероятно, последно използваните данни ще бъдат използвани пак скоро, така всяко ниво помни последно използваните блокове от по-долното ниво;
- **пространствена локалност** – най-вероятно съседите в адресното пространство на последно използваните данни ще бъдат използвани скоро, така блоковете от по-ниските нива в йерархията са по-големи, тъй като освен че обхващат последно използваните блокове, те обхващат техните съседи.

Основната характеристика на кешовете е, че те са в горните нива на йерархията, непосредствено след регистрите. Кешовете изцяло се управляват от хардуера. Размерът на кешовете е значително по-малък от размера на основната памет и те оперират приблизително със скоростта на процесора.
Кешът представлява структура като множество от **рамки**. Всяка

рамка се състои от данни, етикет и състояние. Етикетът е характеристика на рамката, която е съществена при търсене на данни в кеша. Състоянието е два бита – бит, който показва дали данните в рамката са валидни и бит, който показва дали в рамката са били писани данни.
Основната логическа организация на кеша е рамките да се групират в **множества**. Кешът се реализира като матрица, всеки ред на която представлява едно множество от рамки.
Адресът за кеша се състои от три полета – етикет, индекс и отместване. Индексът определя множеството, в което се намира търсената рамка. При търсене след като се определи това множество, данните и етикетите на всички рамки от него влизат в асоциативна схема с търсене по съдържание. На другия вход на тази схема влиза входящия етикет. В схемата се прави паралелно търсене на входящия етикет и ако той съвпадне с някой етикет на рамка от множеството, на изхода на схемата излизат данните в намерената рамка. Отместването показва мястото на търсения байт в блока данни на търсената рамка.
Всяко множество се състои от един и същ брой рамки. Ако този брой е k, казваме че кешът е **k-асоциативен**. Най-простият случай е когато всяко множество от рамки се състои само от една рамка. В този случай казваме, че кешът е директно изобразим. При директно изобразими кешове, индексът показва определена рамка от кеша, така че етикет не е необходим. Кешове при които има само едно множество от рамки се наричат пълно асоциативни. Естествено, при пълно асоциативните кешове няма нужда от индекс. Недостатъкът на пълно асоциативните кешове е голямата сложност на асоциативното търсене.
Съществените характеристики на кешовете са следните:

- асоциативност;
- размер на блока – ако размерът на блока е малък се губи принципа за пространствена локалност, ако е много голям може да имаме излишно прехвърляне на байтове;
- капацитет – по-големите кешове работят по-бавно, докато при по-малките се губи темпоралната локалност;
- механизъм за изхвърляне на блокове от данни – ако търсеният блок от данни не бъде намерен, коя рамка да бъде изхвърлена, за да се освободи място за търсената рамка, която ще се изтегли от по-долно ниво в йерархията, стратегиите са следните – **LRU** (изхвърляне на рамката, която не е използвана най-отдавна), **LFU** (изхвърляне на рамката, която е използвана най-рядко), **random** (изхвърляне на произволна рамка), **NMRU** (изхвърляне на произволна рамка, която е различна от най-често използваната);
- начин на записване на блок от данни в кеша – когато се записват данни в кеша стои въпросът дали тези данни да се записват и в основната памет, двете стратегии са **write-through** (при нея всички записи към кеша се насочват и към основната памет, недостатъкът е, че се намалява

производителността в работата на кеша при много операции за запис) и **write-back** (при нея данните се записват само в налична рамка в кеша, при опит за извърхляне на тази рамка, тя трябва да се запише в паметта преди да се изхвърли, предимството на тези кешове е, че се намалява трафикът към основната памет, недостатъкът е, че паметта губи консистентност – данни, които са записани в кеша в един момент може да не присъстват в основната памет).

В началото основната памет е много скъпа, докато магнитните дискове, макар по-бавни, са много по-евтини за запомняне на един бит. От друга страна програмите започвали да нарастват и да се нуждаят от по-големи адресни пространства. Въвежда се принципът на **виртуалната памет** – при нея в компютрите се работи с виртуален адрес, който по размер е по-голям от физическия адрес на основната памет. Така виртуалното адресно пространство може да е много по-голямо по размер от реалното адресно пространство. Виртуалното адресно пространство е разделено на блокове, наречени **страници**. Виртуалният адрес се състои от номер на виртуалната страница и отместване в рамките на страницата. Процесът на преобразуване на виртуален адрес към реален физически адрес наричаме **транслация**. При транслацията номерът на виртуалната страница се преобразува в номер на физическа страница в реалната памет, отместването във физическата страница е същото като във виртуалната. Апаратно или от операционната система се поддържа вътрешна таблица на страниците, която показва в даден момент от времето какво е изображението на виртуалната памет върху физическата. Съществено е да се отбележи, че отделните индивидуални процеси не извършват транслацията - тя се извършва от системен процес, който е част от ядрото на операционната система.

В началото виртуалната памет решава нуждата от по-голямо адресно пространство. В наши дни физическата памет нараства и поевтинява. Въпреки това, виртуалната памет се запазва, тъй като с нейното развитие се проявяват някои други полезни качества. Основното е, че всеки процес използва виртуални адреси в едно виртуално адресно пространство. В резултат от адресната транслация еднакви виртуални адреси от различни процеси се изобразяват в различни физически адреси. Така едно и също реално адресно пространство може да се използва от различни процеси без припокриване на адресите.

Виртуалната памет има следните предимства:

- защита - процесите са изцяло отделени един от друг;
- програмирането се опростява, тъй като всеки процес се развива в хомогенно виртуално адресно пространство с начален адрес 0.

Основен недостатък на виртуалната памет – какво става при ненамиране на физическата страница в паметта. Характерното за виртуалната памет е, че на магнитен диск се отделя файл

(**swap-file**), в който се помества образ на пълното виртуално адресно пространство. В реалната памет се поместват само някои активни страници. При ненамиране липсващата страница трябва да се прехвърли от диска в реалната памет. Това прехвърляне е твърде бавно и поради тази причина, реалната памет винаги е write-back, т.е. никога от процесора не се записват данни в диска, така се поддържа връзка само между процесора и реалната памет. Транслацията се извършва по следния начин: при заявка за достъп до дадена страница, първо се проверява дали тя се намира във физическата памет – ако да, връща адреса на страницата, ако не зарежда търсената страница от диска и след това връща нейния адрес, при това ако няма място в паметта за новата страница, някоя от наличните страници трябва да бъде подменена по някой от алгоритмите, които споменахме при кешове, при това, тъй като основната памет е write-back, ако подменяната страница е била променяна, тя трябва да се запише върху диска.

Освен на страници, виртуалната памет може да бъде организирана и на **сегменти**, те за разлика от страниците са с променлива дължина и могат да се припокриват.
В машините PENTIUM при сегментацията адресът се получава от 16-битов сегментен регистър и 32-битово отместване.
Всеки сегментен регистър съдържа индекс, който показва отместване в глобална или локална дескрипторна таблица. Дескрипторните таблици съдържат сегментни дескриптори, като всеки сегментен дескриптор описва един сегмент чрез адреса на началото на сегмента, размера на сегмента и различни битове за състояние. Глобалната дескрипторна таблица е единствена и се използва най-вече за дескриптори на системни сегменти. Локалната дескрипторна таблица не е единствена и се използва най-вече за дескриптори на приложни сегменти. В даден момент може да се използва точно една локална дескрипторна таблица.
Получаването на адреса при сегментацията става по следния начин: (зададени са сегментен регистър и отместване)

- първо се определя в глобалната или в локална дескрипторна таблица да се търси сегментния дескриптор;
- след като е определен сегментният дескриптор се взима адресът на началото на сегмента и към него се прибавя отместването.

Полученият адрес след сегментацията се нарича линеен адрес и той е част от линейното адресно пространство.
Възможни са два случая:

- линейното адресно пространство директно се изобразява върху физическото адресно пространство;
- линейното адресно пространство е виртуално – извършва се странична преадресация.

В случай на странична преадресация, линейният адрес се изпраща към блок, който го преобразува към физически адрес.

Системата за прекъсване цели във всеки момент да се даде възможност за регистрация на случващо се събитие. Механизмът на прекъсванията е ефективен начин за обмяна на информация с бавните входни-изходни устройства и за сигнализиция за особени състояния в работата на централния процесор. Чрез нея се избягва необходимостта от периодични проверки на флагове за дадени събития. На всяко прекъсване се съпоставя точно определен номер. За всеки вид прекъсване има специална програма, която се изпълнява при възникването на този вид прекъсване. В основната памет има специална фиксирана област, наречена таблица на **векторите на прекъсванията**. Всеки вектор на прекъсване съдържа указател към подпрограма за обработка на прекъсването и състояние.
Когато възникне прекъсване, централният процесор прекъсва изпълнението на текущата програма и съдържанието на програмния брояч PC и регистъра на състоянието MSW се запазват в стек за да може да има вложени прекъсвания. Новото състояние на процесора при влизане в прекъсване се взима от съответния вектор на прекъсване, който се намира по номера на прекъсването. Всяка програма, която обслужва прекъсване завършва с инструкция за връщане от прекъсване, която от върха на стека прочита предишното състояние на PC и MSW и ги зарежда в процесора и по този начин изпълнението се връща към прекъснатата програма.

За да се отчете важността на събитията, които може да настъпят се въвежда **приоритет** на прекъсванията. По време на изпълнението си програмата за обработка на едно прекъсване може да бъде прекъсната само от прекъсване с по-висок приоритет. Съществуват различни видове прекъсвания:

- прекъсвания по **машинна грешка** – грешка в апаратурата, те са най-високо приоритетни;
- **входно-изходни** прекъсвания – те се активират в резултат на изпълнение на входно-изходна операция;
- **външни** прекъсвания – свързани са с някакъв външен елемент (например бутон RESET) и са аналогични на входно-изходните прекъсвания;
- **програмни** прекъсвания – те са синхронни с програмата, която се изпълнява и подтискат изпълнението на някаква инструкция, например деление на 0;
- **програмно-активирани** (**SVC**) прекъсвания – при тях текущата програма издава специална команда за провеждане на прекъсване.

Друго разделяне на прекъсванията е на **маскируеми** и **немаскируеми**. Маскируемите прекъсвания могат да се игнорират, т.е. да не се обработват, докато немаскируемите прекъсвания задължително се обработват. Така немаскируемите прекъсвания винаги имат приоритет пред маскируемите.

7. Файлова система. Логическа организация и физическо представяне. Основни системни примитиви.

Файлът е основната единица, чрез която операционните системи осигуряват съхраняването на постоянни обекти данни.

Името на файл е символен низ с определена максимална дължина. Допустимите символи за името на файла, както и максималната дължина варират в различните операционни системи.
В UNIX/LINUX системите се допускат всички възможни символи и не се прави разлика между малки и главни латински букви.
В MSDOS са допустими само цифрите, буквите и символът за подчертаване (_) и не се прави разлика между малки и главни латински букви.
В някои операционни системи името на файла има следната форма: *име_на_файл[.разширение]*. Разширението се използва за да се означи типа или формата на файла и то не е задължително. Пример за такава операционна система е MSDOS.
Основният момент е доколко ядрото на операционната система прави разлика между името и разширението.
В UNIX/LINUX системите за ядрото няма име и разширение – името на файла е едно цяло. Разширенията могат да се поставят за по-удобна връзка между потребителя и обслужващите програми в обвивката на ядрото, но това не засяга самото ядро.
За разлика от UNIX/LINUX системите, в MSDOS разделянето е заложено в ядрото – например, файловете с изпълним код задължително имат разширение *exe* или *com*.

В по-старите операционни системи има единствен **тип** файлове.
В по-новите операционни системи такива файлове се наричат **обикновени файлове** - служат за съхраняване на информация върху външен носител.
В съвременните операционни системи има и други типове файлове. Те са файлове, свързани с входно-изходни устройства или файлове, свързани с услуги, предоставени от ядрото.
В UNIX/LINUX файловете за входно-изходни устройства се наричат **специални файлове**. Те са два типа – символен специален файл и блоков специален файл. Ползата от тях е, че програмите стават независими от устройствата – системните примитиви за четене и писане са едни и същи за всички устройства.
В UNIX/LINUX файловете, свързани с услуги, предоставяни от ядрото са няколко вида. **Програмните канали** са файлове, които осигуряват механизъм за комуникация между паралелно работещи процеси. Такъв тип файлове са и така наречените **socket**, но те се използват за процеси в мрежова среда. В по-новите UNIX системи има **символни връзки**. Символните връзки са тип файлове, чрез които се дава възможност един файл да е достъпен с различни

имена. Същата възможност предоставят и така наречените **твърди връзки**, но за разлика от символните връзки те не се разглеждат като тип файлове.

Файловата система на всяка операционна система е структурирана, като информацията за самата структура е запазена върху диска. Файловете се организират в **каталози**. Въпросът е дали каталозите са реализирани като тип файлове.
В по-старите системи като OS/360 файловете върху един диск са организирани в един каталог, който не е тип файл – поместен е отделно от файловата система върху диска.
В съвременните операционни системи каталозите са реализирани като тип системни файлове. Всеки каталог съдържа информация за файловете в него – за всеки файл има поне един запис, който съдържа името на файла и допълнителна информация.

Като **вътрешна структура**, най-общо файлът е последователност от данни. Един възможен подход е файлът да е последователност от записи с определен формат и дължина. В OS/360 има два формата на файловете – записи с фиксирана или с променлива дължина. При такава структура системните примитиви за четене и писане трябва да са организирани по записи. Това предизвиква проблеми, защото при различните типове файлове записите трябва да са с различна дължина и формат, което изисква различен подход при операциите четене и писане.
В съвременните операционни системи файлът се разглежда като последователност от байтове. Това опростява структурата на файла и съответно опростява системните примитиви за работа с файлове. Най-съществено е, че такава структура може да се приложи към всички изброени по-горе типове файлове.

Атрибутите са допълнителна информация за файловете, която операционната система съхранява. Основните атрибути са:

- размер на файла в зависимост от структурата му;
- атрибути за дата/време – дата/време на създаване на файла, дата/време на последен достъп, дата/време на последна модификация, дата/време на последно изменение на атрибутите;
- атрибути за сигурност – собственик на файла, права на достъп до файла за различните потребители, пароли за достъп до файла (най-вече за достъп в мрежова среда);
- атрибути-флагове – тези атрибути се реализират като един бит, примери за такива флагове в MSDOS са: readonly (ако е 1, файлът е само за четене), system (ако е 1, файлът е системен, т.е. съдържа част от операционната система), hidden (ако е 1, файлът е скрит), archive (ако е 1, файлът ще бъде включен в следващия back-up на системата), флагове има и във версиите на LINUX – например, флагът security deletion (ако е 1 за някой файл, то при неговото

унищожаване се форматират всички блокове, които е заемал файла) и флагът undelete (ако е 1 при изтриване на файла се съхранява информация, подмогаща евентуалното му възстановяване).

Най-простата организация на файловата система е за всеки диск да се изгради един каталог, който съдържа информация (име, атрибути, дискови адреси) за всички файлове, разположени върху диска. Такава организация има при по-старите операционни системи, например OS/360. Такава организация създава редица неудобства – всички файлове върху един диск трябва да са с различни имена, което е проблем при многопотребителски системи, освен това се затруднява търсенето на файлове. Поради тези причини в по-новите операционни системи се изгражда **йерархична организация** на файловата система. В началото тя е била с фиксиран брой нива – пример е операционната система CTSS на MIT, в която файловата система е на три нива. По-новите операционни системи поддържат йерархична организация с произволен брой нива. Най-горното ниво е главен каталог. Всеки каталог може да съдържа както информация за файлове, така и информация за други каталози. По този начин се получава дървовидна организация – в листата на дървото са файловете, а във вътрешните върхове на дървото са каталозите. Имената на файловете трябва да са уникални само в рамките на един каталог. В такава организация трябва да има начин за унифициране на името на всеки файл. За целта се въвеждат **пълни имена** на файлове. **Абсолютното** пълно име на файл се образува от имената на каталозите по единствения път от корена на дървото до листото с файла. **Относителното** пълно име на файл е свързано с понятието **текущ каталог**. Това име се образува от имената на каталозите по единствения път от текущия каталог до листото с файла. В UNIX/LINUX системите има единна йерархия за всички файлове и за всички устройства. Това се постига с техниката на монтиране. Поддържа се текущ каталог за всеки отделен процес. В MSDOS системите специалните файлове не са част от йерархичната структура. Освен това за всяко дисково устройство се изгражда собствена йерархична структура със собствен корен. Въвеждат се имена на различните устройства и се поддържа текущо устройство. Текущ каталог се поддържа само за отделните устройства, но не и за всички процеси.

Стратегиите за управление на дисковото пространство се обуславят в зависимост от комбинирането на решението на три основни въпроса.
Първият основен въпрос е кога се разпределя дискова памет за файловете. Съществуват два основни подхода:

- **статично разпределение** – при него памет за един файл се разпределя еднократно, при неговото създаване;

- **динамично разпределение** – при него памет за един файл се разпределя многократно с нарастването на файла.

Статичното разпределение затруднява потребителя – той трябва предварително да знае колко голям ще бъде файла, за да може да задели достатъчно памет. Динамичното разпределение е по-гъвкава стратегия, но е по-трудна за реализация.
Вторият основен въпрос е колко непрекъснати дискови области заема един файл. Под **непрекъсната дискова област** разбираме последователни сектори върху една писта, последователни писти върху един цилиндър, т.е. физически съседни области, отчитайки физическата организация на диска. Тук отново има два подхода:

- всеки файл заема една непрекъсната област върху диска, достатъчно голяма по размер;
- всеки файл заема много непрекъснати области върху диска, които не са съседни.

Третият основен въпрос е каква е единицата за разпределение върху дисковото пространство:

- с фиксиран размер в рамките на всички файлове;
- с променлив размер, дори в рамките на един файл.

В по-старите операционни системи се използва **статично и непрекъснато разпределение**. Предимства на такава организация са лесната реализация и ефективен достъп при последователна обработка. Недостатък на статичното непрекъснато разпределение е силната **фрагментация** на свободното дисково пространство – върху диска има много области, които са свободни, но файловата система не може да разпредели дискова памет за файл, защото няма достатъчно голяма непрекъсната област, въпреки че сумарно дисковото пространство е достатъчно.

При съвременните операционни системи се използва **динамично и поблоково разпределение**. Всеки файл се разделя на **блокове**, като всички блокове на всички файлове имат еднакъв размер. Всеки блок на един файл се разполага в една непрекъсната област. Логически съседните блокове на файла, обаче, не са физически съседни върху диска. Дискова памет за блок се разпределя когато файлът нараства, а не при неговото създаване. Такава организация решава повечето проблеми на статичното непрекъснато разпределение. Основният въпрос е какъв да бъде размера на блока. Ако блокът е прекалено голям, то ще се появят проблеми от фрагментацията, въпреки че последователният достъп ще е по-ефективен. Ако блокът е прекалено малък, то обратно - ще се постигне ефективно използване на дисковата памет, но последователният достъп ще е неефективен. При по-малък блок се появява и друг проблем – файловата система трябва да помни всички адреси на блоковете на един голям файл, т.е. обемът на системната информация на файловата система нараства.

Оттук нататък ще се спрем по-подробно на динамичното поблоково разпределение. При такава организация на файловата система дискът представлява последователност от блокове, номерирани от 0 нататък.
Файловата система трябва да съхранява структури върху диска, които да дават информация за текущото разпределение на дисковата памет. Тези структури се наричат **системни структури** и те трябва да съдържат информация за свободните блокове, за разпределените блокове за всеки файл и за други параметри на файловата система.

Когато операционната система трябва да задели памет, тя се нуждае от информация кои от блоковете са свободни – тази информация се съхранява в **системна структура за свободните блокове**. Ще разгледаме трите най-популярни реализации.

Първата реализация на тази структура е **свързан списък на свободните блокове**. При нея всеки свободен блок съдържа адреса на следващия свободен блок. Файловата система трябва да разполага с адреса на първия свободен блок – този адрес е параметър на файловата система. При заделяне на памет се заделя първия свободен блок, а адреса на следващия свободен блок се записва като адрес на първи свободен блок. Предимство на подхода е лесната реализация. Основният недостатък е ненадеждност – структурата е пръсната по всички свободни блокове.

Втората реализация е **свързан списък от свободни блокове с номера на свободните блокове**. При тази структура първият свободен блок съдържа не само адреса на втория свободен блок, но и адресите на следващите няколко свободни блока, доколкото позволява размера. Следващият свободен блок, който се използва от структурата е последния блок, чийто адрес е записан в първия свободен блок и т.н. По този начин структурата става по-компактна. Освен това, тя дава възможност да се реализират по-ефективни алгоритми за разпределяне и освобождаване на блокове. Файловата система отново трябва да разполага с адреса на първия свободен блок.

Третата реализация е така наречената **карта** или **таблица** на свободните блокове. Структурата е масив от елементи, като всеки елемент позиционно съответства на блок върху диска, т.е. съседни елементи на масива съответстват на физически съседни блокове. Всеки елемент описва състоянието на съответния блок. При най-простата реализация е достатъчен един бит за един елемент – 1 означава, че блокът е свободен, 0 означава, че блокът е зает. В този случай картата се нарича **битова карта** (bitmap). Предимство

на битовата карта е компактността на структурата – тя е с фиксиран размер в зависимост от размера на диска.
Основно предимство, обаче, е възможността за разработка на ефективни алгоритми за разпределение на блокове, които отчитат физическото съседство на блоковете – може да се постигне по-голяма непрекъснатост на файловете, въпреки че разпределението е поблоково.

За всеки файл операционната система трябва да има информация за това кои блокове в каква последователност са разпределени за този файл – тази информация се съхранява в **системни структури за разпределените блокове за файлове**. Тук отново има три популярни реализации.

Първата реализация на структурата е **свързан списък на блоковете на файла**. Всеки блок, разпределен за файла съдържа данни и адрес на логически следващия блок на файла. Последният блок на файла трябва да съдържа маркер за край на файла.
При такава организация за всеки един файл трябва да се пази адрес на първия блок, разпределен за файла. Обикновено, този адрес е записан в каталога заедно с името и атрибутите на файла.
Недостатък на такава реализация е ненадежността – смесват се данни и адреси, адресите са пръснати по всички блокове. Друг недостатък е неефективност при обработка на файловете с произволен достъп.

Втората реализация на структурата е **карта** или **таблица**, подобно на системната структура за свободните блокове. Размерът на елементите на битовата карта, обаче, трябва да се увеличи – състоянието на един блок отново е свободен или зает, но ако блокът е зает той съдържа адреса на логически следващия блок във файла или маркера за край на файла, ако той е последният блок във файла. Отново за всеки един файл трябва да се пази адрес на първия разпределен за него блок и обикновено тази информация се пази в каталога. Предимствата на такава структура са, че цялата нужна информация се съдържа в структура с фиксиран размер, разположена на фиксирано място в диска. По отношение на надеждността – възможно е създаване на няколко копия на таблицата. Основен недостатък на таблицата е неразширяемост, поради фиксирания размер.

Третата реализация е структура, в която за всеки файл се съхранява логическата последователност от блоковете, разпределени за този файл. Такава структура се нарича **индекс**. Физическата организация на индекса трябва да е такава, че да не ограничава размера на файловете и да осигурява бърз произволен достъп. В различните операционни системи се използват свързани списъци, дървета, B+ дървета и др.

Освен системните структури за свободните блокове и за блоковете, разпределени за файлове, операционната система трябва да разполага и със системни структури за общите параметри на файловата система – те съхраняват важна информация за файловата система на глобално ниво, например, размер на блока, размер на цялата файлова система, адреси на началото и размери на различните области, общ брой свободни блокове и др.

Ще разгледаме някои конкретни примери за физическа организация на файловата система.

При файловата система **UNIX System 5**, дисковото пространство е последователност от блокове, номерирани от 0 до N, където N зависи от големината на дисковото пространство.
Размерът на блока най-често е 1K, 2K или 4K.
Дисковото пространство при UNIX System 5 се разпределя на четири области: boot block, super block, индексна област и област за данни.
Блокът с номер 0 се нарича **boot block** и подобен блок има във всяка файлова система. Той съдържа програмата, с която се зарежда операционната система.
Блокът с номер 1 се нарича **super block** и той съдържа общи параметри на файловата система.
Индексната област е системната структура от данни, която съхранява информация за разпределените блокове. Тя представлява последователност от записи, наречени **индексни описатели** (inode). За всеки файл в индексната област има собствен inode. В индексната област има независимо адресиране на inode от 1 до S, където S е размера на индексната област. Максималният брой файлове е ограничен от броя на индексните описатели, тъй като всеки файл трябва да има индексен описател.
В **областта за данни** са блоковете данни на файловете. Освен тях в тази област има блокове за системна информация – косвени блокове и блокове, които реализират списъка на свободните блокове.
Ще разгледаме структурата на индексния описател (inode).
За всеки файл има точно един inode, който съдържа атрибутите на файла и адресна информация. Индексните описатели имат фиксиран размер 64B в UNIX System 5. Първите 24B в inode е частта за атрибути на файла, а останалите 40B служат за адресна информация. Първият атрибут е **mode**. Той заема 2B и съдържа следните полета:

- тип на файла - четири бита, които задават дали файлът е обикновен, специален, каталог, символна връзка и т.н., ако типът на файла в даден inode е 0000, то този inode е свободен, т.е. не описва никакъв файл;
- битове SUID и SGID, които се използват когато файлът е изпълним и управляват правата по време на изпълнение;

- бит sticky bit, който се използва най-вече за защита на каталози, ако файлът е каталог;
- три групи по три бита read, write, execute, които съответстват на правата на собственика, правата на групата на собственика и правата на останалите потребители.

Вторият атрибут е **nlink**. Той заема 2B и съдържа броя на твърдите връзки на файла. Третият атрибут е **uid**. Той е 2B и съдържа числов идентификатор на собственика на файла. Четвъртият атрибут е **gid**. Той е 2B и съдържа числов идентификатор на групата на собственика на файла. Следващият атрибут е **size**. Той е 4B и съдържа размера на файла в брой байтове. Последните три атрибута са по 4B и са атрибути за време. Атрибутът **atime** показва дата/време на последен достъп до файла. Атрибутът **mtime** показва дата/време на последна промяна на файла. Атрибутът **ctime** показва дата/време на последната промяна на индексния описател на файла.
Адресната част се разпределя на 13 адресни полета по 3B и един свободен байт за подравняване. Тези адресни полета съдържат адреси на блокове в областта за данни. Първите десет адресни полета съдържат адресите на първите десет блокове данни на файла. Единадесетото адресно поле съдържа адрес на косвен блок в областта за данни, който съдържа адреси на следващите блокове данни на файла. По този начин се реализира **косвена адресация**. Дванадесетото адресно поле съдържа адрес на косвен блок, който съдържа адреси на косвени блокове, които съдържат адреси на следващите блокове данни на файла. По този начин се реализира **двойна косвена адресация**. Чрез тринадесетото адресно поле се реализира **тройна косвена адресация**. По този начин блоковете от данни на файла са организирани в дърво, което има дълбочина най-много три. Това означава ефективен произволен достъп до файла – всеки блок от данни на файла е достъпен най-много с четири дискови операции.

Тъй като адресните полета са по 3B, то максималният брой блокове в областта за данни е 2^{24} = 16 милиона блокове.
От дървовидната структура се пресмята, че максималният размер на един файл при размер на блока 1K е 16GB.

Суперблокът съдържа общи параметри на файловата система:
- размер на цялата файлова система;
- размер на индексната област;
- общ брой свободни блокове данни;
- общ брой свободни индексни описатели;
- масив **s_free[B]**, който съдържа номера на свободни блокове и брояч **s_nfree**, който съдържа броя на тези номера;
- масив **s_inode[I]**, който съдържа номера на свободни inode и брояч **s_ninode**, който съдържа броя на тези номера.

Последните два компонента на суперблока са свързани с механизмите за разпределяне и освобождаване на блокове в областта за данни и inode в индексната област.

В UNIX System 5 системната структура, която съдържа информация за свободните блокове е списък от блокове, които съдържат номера на свободни блокове. Към тази структура се включва и масива s_free. Всеки от блоковете в структурата в началото си съдържа адрес към следващия блок от структурата и още B-1 адреса на свободни блокове. За начало на структурата се счита масива s_free. Така в елемента s_free[0] се съдържа адреса на първия блок от областта за данни, който съдържа номера на свободни блокове. При зареждането на операционната система целият суперблок се копира в оперативната памет и по този начин чрез масива s_free системата има бърз достъп до номера на свободни блокове. Масивът s_inode се използва по аналогичен начин на s_free, само че при разпределянето и освобождаването на inode. Номера на свободни inode се съдържат само в масива s_inode, но не и в областта за данни както при свободните блокове. Естествено, s_inode не може да съдържа номерата на всички свободни inode. Системата, обаче, може да разпознава свободните inode – те имат тип на файла 0000. Така, ако номерата на свободни inode в s_inode се изчерпят системата може да организира търсене за свободни inode.

Каталогът е тип файл в UNIX System 5. Както всички други файлове, всеки каталог притежава индексен описател, в който полето тип на файла е каталог. Всеки каталог притежава блокове данни в областта за данни. Тези блокове съдържат записи, като всеки запис описва един файл в каталога. Структурата на записа е следната:

- 2B за номера на индексния описател на файла;
- 14B за собственото име на файла.

Така в UNIX System 5 има ограничение за дължината на името на файла – то е до 14 символа. Получаваме ограничение и за размера на индексната област. Тя не може да съдържа повече от $2^{16} = 65536$ индексни описатели. Хубавото е, че всички записи са с фиксиран размер – постига се простота на структурата.
Записите в каталога не са сортирани - те са в хронологичен ред, т.е. в реда на създаване на файловете. При изтриване на файл, записът му в каталога не се изтрива, а се записва 0 като номер на индексен описател. Да напомним, че индексните описатели се адресират от 1, така че номер 0 може да се използва като признак за свободен запис. При създаване на файл, в каталога се търси свободен запис (с номер 0), в който да се запише информация за създадения файл или се добавя нов запис, ако няма свободни записи. Това води до неприятен ефект – каталозите могат само да нарастват, но е невъзможно да се свиват. Като цяло за обработка на каталозите не се използват ефективни алгоритми.

Всеки каталог съдържа два стандартни записа в началото си, дори ако в каталога няма никакви файлове. В първия запис името на файл е **..** и е записан номера на индексния описател на родителския каталог. Във втория запис името на файл е **.** и е записан номера на собствения индексен описател на каталога.

Твърдите връзки позволяват един файл, който се съхранява на едно място в областта от данни и има единствен inode да бъде достъпен с няколко имена, дори в един и същи каталог.
При създаване на твърда връзка на файл трябва да се увеличи броячът nlink в индексния описател на файла.
Всички твърди връзки на един файл са напълно равноправни – няма значение кой е оригиналния файл. Поради тази причина броячът nlink помага на системата да установи кога е нужно един файл да се изтрие физически, а не само от каталога - при всяко изтриване на един файл броячът nlink намалява с 1 и ако е станал 0, то файлът трябва да се изтрие физически. Важно е, че твърди връзки могат да се създават само към обикновени файлове.

Символните връзки дават същата възможност, както твърдите връзки но имат по-различна реализация. Символните връзки са тип файл – всяка символна връзка има собствен inode, в който тип на файла е символна връзка, запис в каталога и област за данни. В първия адресен блок в индексния описател за символна връзка е записан адрес на блок от областта за данни, в който е записано пълното име на файла, към който се осъществява връзката.
Предимствата на символните връзки пред твърдите са следните:

- символни връзки могат да се създават към файлове от произволен тип;
- възможно е символни връзки да се създават между файлове в различни файлови системи, което е невъзможно за твърдите връзки.

Естествено, за да се реализират символните връзки са необходими значително повече ресурси. Също, достъпът до файла през тях е косвен, докато през твърдите връзки е директен.

Дисковото пространство при файловата система **LINUX EXT2** е последователност от блокове, които се адресират от 0 до N, където N зависи от големината на дисковото пространство. Блоковете са с фиксирана дължина – 1K, 2K или 4K. Дисковото пространство при LINUX EXT2FS се разпределя така: първият блок е boot block, а останалата част от блокове се разделя на еднакви по размер групи.
Boot block заема един блок и той съдържа програмата, с която се зарежда операционната система.
Всички **групи** са с фиксиран размер и съдържат част от файловата система. Всички групи имат еднаква структура, която се състои от следните полета:

- суперблок;
- описатели на групи;

- битова карта на блоковете;
- битова карта на inode;
- индексна област;
- област за данни.

Суперблокът, както в UNIX System 5, заема един блок и съдържа общи параметри на файловата система. Разликата от UNIX System 5 е, че той има много копия – по едно за всяка група. Частта **описатели на групи** съдържа последователност от описатели на всички групи на файловата система. Суперблокът и описателите на групи са глобални за файловата система – те са едни и същи за всички групи и имат копие във всяка група. Останалите четири области от една група характеризират самата група.
Битовата карта на блоковете се помества в един блок и се интерпретира като масив от битове, които позиционно съответстват на блоковете в областта за данни в групата. Ако един бит е 1, то съответният блок е свободен, ако е 0, то съответният блок е зает.
Битовата карта на inode заема един блок и има аналогична роля на битовата карта на блоковете, но съдържа информация за индексните описатели в групата.
Индексната област и **областта за данни** имат аналогична роля на UNIX System 5 – индексната област е разделена на записи, наречени индексни описатели (inode), като на всеки файл съответства точно един inode, областта за данни съдържа блоковете с потребителските данни. Естествено, те се отнасят за конкретната група.
Размерът на една група се определя така, че битовата карта на свободните блокове да заема точно един блок. Например, ако блокът е 1K, то групата е малко повече от 8MB.

Един описател на група е 32B и съдържа информация за групата:
- адрес на блока с битовата карта на блоковете;
- адрес на блока с битовата карта на inode;
- адрес на първия индексен описател в индексната област;
- брой свободни блокове в областта за данни на групата;
- брой свободни inode в индексната област на групата;
- брой файлове с тип каталог в групата.

Всички описатели на групи се записват след суперблока в рамките на всяка група в областта описатели на групи.

Адресацията на индексните описатели е обща, въпреки че индексната област е разпръсната между различните групи. Размерът на всеки inode е 128B. В сравнение с UNIX System 5 са добавени нови атрибути:
- брой блокове, заети от файла;
- нов атрибут за време **dtime** – дата/време на последното изтриване на файла;

- атрибути-флагове – реализират се чрез 1 бит и са поместени в поле от 4B; някои от тях са: флаг **secure deletion** (при изтриване на файла се форматират всички блокове), флаг **undelete** (когато файлът се унищожава, за него се съхранява информация, подмогаща евентуалното му възстановяване), флаг **immutable** (не могат да се променят нито атрибутите, нито данните на файла), флаг **compress** (данните на файла се компресират при запис и декомпресират при четене), флаг **synchronous write** (операцията за запис е синхронна с физическото записване на данните върху диска).

Размерът на адресната област в inode е 60B и тя се разделя на 15 адресни полета по 4B. Първите 12 полета съдържат адресите на първите 12 блока данни на файла. Тринадесетото, четиринадесетото и петнадесетото поле реализират съответно косвена, двойно косвена и тройно косвена адресация.
Адресът е увеличен на 4B, за разлика от UNIX System 5, където той е 3B. По този начин могат да се адресират по-голям брой блокове – до 4 милиарда, което при размер на блока 1K е напълно достатъчно за съвременното състояние на технологиите.
Каталогът, за разлика от UNIX System 5, има различна структура на записите:

- номер на inode на файла (4B);
- дължина на записа (2B);
- дължина на името на файла (2B);
- име на файла (до 255B).

Записите са с променлива дължина. При това те се подравняват по 4B и затова в записа се пазят дължините и на името и на записа. Така името на файла при LINUX EXT2 може да е до 255 символа. По отношение на обработката на каталозите няма съществени изменения от UNIX System 5 – отново е неефективна.
От размера на полето за номер на inode се вижда, че броят на inode може да достигне 4 милиарда.
LINUX EXT2 поддържа твърди и символни връзки по същия начин, по който те се поддържат от UNIX System 5.

Дисковото пространство при файловата система на **MSDOS** е последователност от сектори. Минималната единица за разпределение на дисково пространство се нарича **клъстер** и представя един или няколко (степен на двойката) физически съседни сектора. Размерът на клъстера е фиксиран.
Дисковото пространство се разпределя на следните части:

- boot sector;
- FAT;
- копие на FAT;
- коренен каталог;
- област за данни.

В **boot sector** се съхранява програмата за зареждане на операционната система плюс някои общи параметри на файловата система.
FAT (**file allocation table**) е таблица на файловата система, която съдържа информация за разпределените клъстери и за свободните клъстери. Съхраняват се няколко копия на FAT (обикновено 2) с цел надеждност на файловата система.
Коренният каталог е изнесен извън областта за данни.
Областта за данни съдържа клъстерите и останалите каталози.

Някои от общите параметри, които се пазят в boot sector са следните:

- размера на клъстера;
- размера на таблицата FAT;
- броя на копията на таблицата FAT, които се съхраняват;
- размер на коренния каталог;
- размер на цялата файлова система.

FAT е масив от елементи с определен размер. Всеки елемент позиционно съответства на един клъстер в областта за данни и описва неговото състояние. Първите два елемента от масива съдържат служебна информация и те нямат отношение към клъстерите. Елементите с номера от 2 нататък съответстват на клъстерите, затова адресацията на клъстерите в областта за данни започва от 2. Всеки клъстер може да е в три състояния:

- **свободен** – тогава съответният му елемент във FAT съдържа код 0;
- **разпределен за файл** – тогава съответният му елемент във FAT съдържа адресът на логически следващия клъстер, който е разпределен за файла или маркер за край на файла (само единици), ако това е логически последният клъстер, който е разпределен за файла;
- **повреден** – тогава съответният му елемент във FAT съдържа специален код, който като стойност е по-голям от максималния адрес на клъстер.

В по-старите версии на файловата системи на MSDOS размерът на елементите в масива е 12 бита, клъстерът е 1K (два съседни сектора). Така могат да се адресират до 2^{12} = 4096 клъстера, т.е. до 4MB дисково пространство, което е крайно недостатъчно. Поради това таблицата се разширява. Появява се **FAT16**, където размерът на елементите на масива е 16 бита и могат да се адресират до 2^{16} = 65536 клъстера. Освен това се увеличава и максималният размер на клъстера – във FAT16 един клъстер може да достигне 64KB. Така максималният размер на дисковото пространство достига 4GB. В WINDOWS/95 таблицата се разширява до **FAT32** – 32-битови елементи на масива. Файловата система на MSDOS е пример за файлова система, която е неефективно разширяема.

При всяко стартиране на операционната система цялата таблица FAT се зарежда в оперативната памет за да има бърз достъп до информацията за разпределените и свободните клъстери. Увеличаване на размера на таблицата води до неудобство цялата таблица да се държи в паметта. Освен това увеличаването на размера на клъстера води до неефективно използване на паметта – за всеки файл задължително са разпределени цяло число клъстери.
В MSDOS **каталозите** са тип файлове. Всеки каталог съдържа записи с фиксиран размер (32B), като всеки запис описва един файл в каталога.
Разпределението на тези 32B е следното

- име на файла (8B);
- разширение на файла (3B);
- атрибути-флагове на файла (1B);
- резервирано пространство (10B) за разширяване;
- дата на последна модификация (2B);
- време на последна модификация (2B);
- номер на първи клъстер (2B);
- размер на файла в байтове (4B).

Използват се следните атрибути-флагове:

- **read-only** – ако е 1, файлът е само за четене;
- **system** – ако е 1, файлът е системен, т.е. съдържа част от операционната система;
- **hidden** – ако е 1, файлът е скрит за някои команди на операционната система;
- **catalog** – ако е 1, файлът е каталог;
- **archive** – ако е 1, файлът се включва следващия back-up на файловата система.

Коренният каталог е изнесен извън областта за данни и има фиксиран размер. Това води до ограничение на броя на записите в коренния каталог. Такова ограничение липсва при останалите каталози. Във всеки каталог, с изключение на коренния, се поддържат два стандартни записа с име **.** и **..**, без разширение, които описват съответно самия каталог и родителския каталог. По този начин се реализира йерархичната структура на файловата система.
С развитието на файловата система на MSDOS започват да се поддържат файлове с произволно дълги имена. Също така, в по-новите версии за сметка на резервираното пространство във всеки запис се добавя допълнителни атрибути дата/време на създаване на файла и на последен достъп до файла.

Файловата система **NTFS** се използва в по-новите версии на WINDOWS – WINDOWS 2000, WINDOWS XP.
В термините на NTFS дисковото пространство се нарича **том** (volume), като един том може да заема няколко дискови дяла.

Минималната единица за разпределение на дисково пространство е **клъстер** – последователност от няколко (1, 2, 4 или 8) физически

съседни сектори. Размерът на клъстера е фиксиран в рамките на една файлова система. Всеки клъстер има два адреса:

- **LCN** (logical cluster number) – логически адрес на клъстера в рамките на тома;
- **VCN** (virtual cluster number) – логически адрес на клъстера в рамките на файла, ако клъстера е разпределен за някакъв файл.

Адресирането и в рамките на тома и в рамките на всеки файл започва от 0 до максималния наличен адрес.
Основният принцип на файловата система NTFS е, че цялата информация за файловата система се съхранява във файлове, т.е. върху диска няма системни области. Така в NTFS има три типа файлове – обикновени, специални и системни файлове. В системните файлове се съхраняват метаданните.
Предимството на този принцип е, че по този начин системните файлове могат да нарастват динамично, подобно на обикновените файлове.
Дисковото пространство се разпределя на следните области:

- boot файл;
- MFT файл;
- MFT зона;
- първа половина на областта за данни;
- други системни файлове;
- втора половина на областта за данни;
- копие на boot файла.

Boot файла съдържа програмата за зареждане на операционната система. Поддържа се скрито копие на boot файла в края на дисковото пространство с цел надеждност.
MFT файла е сърцето на файловата система – той съдържа описанията на всички файлове. Стремежът е MFT файла да е непрекъснат за по-голяма ефективност. Поради тази причина непосредствено след MFT файла е предвидена **MFT зона**, която се поддържа празна, докато е възможно. В нея се разпределят клъстери за MFT файла когато той нараства.
Клъстери за системните файлове се разпределят някъде по средата на тома.
MFT файла съдържа записи с фиксиран размер от 1K, които се адресират от 0 нататък. Всеки файл в тома е описан с поне един основен запис в MFT файла. Адресът на този запис се използва като идентификатор на файла в рамките на файловата система.
Първите няколко записа на MFT файла са резервирани за описание на системните файлове (например коренния каталог, битова карта на свободните клъстери, системен файл за общи параметри на системата, системен файл с информация за повредените клъстери и др.).
Другият основен принцип на файловата система NTFS е, че файлът се състои само от атрибути. Данните на файла се разглеждат като един от типовете атрибути на файла. Всички атрибути се съхраняват по един и същи начин. Един файл може да

има няколко атрибута от един и същи тип, в частност няколко атрибута данни. Някои от типовете атрибути са следните:

- атрибут **$FILE_NAME** – съдържа собственото име на файла;
- атрибут **$STANDART_INFORMATION** – съдържа информация за файла – размер на файла, дата/време на създаване, дата/време на последен достъп, дата/време на последна модификация и др.;
- атрибут **$DATA** – съдържа данни на файла;
- атрибути **$INDEX_ROOT**, **$INDEX_ALLOCATION**, **$BITMAP**, които се използват при представянето на каталозите.

Атрибутите са два типа – **резидентни** и **нерезидентни**. Един атрибут е резидентен, ако той изцяло се съхранява в MFT-записа на съответния файл. Ако един атрибут се съхранява в клъстери извън MFT-файла, той е нерезидентен.
Например, атрибутите **$**FILE_NAME, **$**STANDART_INFORMATION и **$**INDEX_ROOT винаги са резидентни.
Обикновено нерезидентни са онези атрибути, които динамично нарастват много бързо.
Всеки MFT–запис има заглавие, последвано от един или повече атрибути. В заглавието се съдържат общи параметри за записа. Всеки атрибут има уникален числов код и в рамките на един запис атрибутите се съхраняват в последователност, отговаряща на нарастването на числовите им кодове. Всеки атрибут има заглавие, в което се описва типа на атрибута и друга информация. Ако атрибутът е резидентен, съдържанието му е поместено в MFT-записа. Ако атрибутът е нерезидентен, то в MFT-записа е поместено само заглавието на атрибута. В този случай, заглавието съдържа информация за клъстерите, разпределени за съдържанието на атрибута в областта за данни. Тази информация се пази под формата на екстенти от физически последователни клъстери. Всеки екстент се описва с три компонента:

- VCN на началния клъстер на екстента;
- LCN на началния клъстер на екстента;
- размер на екстента в брой клъстери.

Самите екстенти са организирани в списък.
Ако заглавието на нерезидентния атрибут не може да побере всичката адресна информация, то се разпределя нов атрибут в MFT-записа от същия тип.
Каталогът в NTFS е файл, който съдържа записи с променлива дължина. Всеки файл в каталога се описва с един запис и този запис съдържа:

- собствено име на файла;
- адресът на основния запис на файла в рамките на MFT-файла;
- копие на атрибута **$**STANDART_INFORMATION на файла.

Копието на стандартната информация в записите се поддържа с цел ускоряване на справките за каталога. От друга страна промяната на стандартната информация за един файл трябва да

се извършва на две места – в атрибута **$**STANDART_INFORMATION и в записа за файла в неговия каталог.
Записите във всеки каталог са съдържание на атрибута **$**INDEX_ROOT и са сортирани в азбучен ред по името на файла. Ако записите не се побират в атрибута **$**INDEX_ROOT, тогава за каталога се разпределят допълнителни екстенти, наречени **index buffers** с дължина 4K. Адресите на разпределените index buffers са съдържание на атрибута **$**INDEX_ALLOCATION. Атрибутът **$**BITMAP съдържа информация за свободните клъстери във всички индексни буфери. Записите от **$**INDEX_ROOT заедно със допълнителните записи в index buffers са организирани в B-дърво. Целта на тази структура е ускоряване на търсенето на файл по име.

Като цяло **системните примитиви** реализират услуги, които се предоставят на потребителските програми от ядрото.
Файлов дескриптор е цяло число от 0 до N-1, където N е максималният възможен брой отворени файлове. Той представлява идентификатор на отворен файл, който се използва от системните примитиви при манипулиране с файла. Свързването на файл с дескриптор се осъществява при отварянето на файла и тази връзка се разкъсва при затваряне на файла. Файловите дескриптори имат локално значение за процесите – всеки процес разполага с N файлови дескриптора. Файловите дескриптори 0, 1, 2 са свързани съответно със стандартния вход, стандартния изход и стандартния изход за грешки.
При всяко отваряне на файл с него се свързва указател на **текуща позиция** за това отваряне. Тя управлява позицията в която се пише или от която се чете във файла. Текущата позиция е цяло число и се измерва с отместването относно началото на файла в брой байтове. При всяко отваряне по подразбиране текущата позиция има стойност 0.
При всяко отваряне на файл с него се свързва и **режим на отваряне**. Той определя какви операции могат да се изпълняват върху отворения файл – например само четене, само писане, четене и писане.
Системният примитив **open** има следния прототип:
int open (const char *filename, int flag[, mode_t mode]).
Той може да се използва както за отваряне на съществуващ файл, така и за създаване на нов файл. С open могат да се отварят и специални файлове, но могат да се създават само обикновени файлове. Аргументът filename задава името на файла – абсолютно, собствено или относително спрямо текущия каталог в текущия процес. Аргументът flag е цяло число, което се интепретира като флагове, задаващи режима на отваряне. Аргументът mode се използва само при създаване на файл и той задава код на защита. При изпълнението на open, с отворения файл се свързва първият свободен файлов дескпритор на текущия процес. При успешно

завършване open връща файловият дескриптор на отворения файл. При неуспех open връща -1.

В по-ранните UNIX системи чрез системния примитив open не може да се създава файл. За тази цел се използва системният примитив **creat**, който е запазен и в по-новите UNIX системи. Подобно на open, чрез creat могат да се създават само обикновени файлове. Системният примитив creat има следния прототип:
int creat (char *filename, mode_t mode).
Той създава файл с име filename и код на защита mode и след това го отваря в режим само за писане. Ако файлът е съществувал, информацията в него се изтрива. При успех creat връща файловият дескриптор на отворения файл, при неуспех creat връща -1.

Системният примитив **close** се използва за затваряне на вече отворени файлове, т.е. за прекъсване на връзката между процеса и файл. По-прецизно, чрез close се освобождава файлов дескпритор. Той има следния прототип: int close (int fd). Аргументът fd задава файлов дескриптор на файла, който ще се затваря. При успешно завършване close връща 0, иначе -1.

Read е системен примитив за четене от файл. Той има следния прототип: ssize_t read (int fd, void *buf, size_t count).
Аргументът fd задава файлов дескриптор на отворен файл, от който ще се чете. Аргументът buf задава адрес в паметта, където ще се записват прочетените данни. Аргументът count задава брой байтове, които ще се прочетат. Четенето от файла започва от текущата позиция. При успех read връща броя на действително прочетените байтове, който може да е по-малък от count. При неуспех read връща -1. След изпълнението на read текущата позиция се увеличава с броя на действително прочетените байтове. При опит да се чете от текуща позиция след края на файла, read връща 0.

Write е системен примитив за писане във файл. Той има следния прототип: ssize_t write (int fd, void *buf, size_t count).
Аргументът fd задава файлов дескриптор на отворен файл, в който ще се пише. Аргументът buf задава адрес в паметта, откъдето ще се взимат данните за писане. Аргументът count задава брой байтове, които ще се записват във файла. Писането във файла започва от текущата позиция. При успех write връща броя на действително записаните байтове. При неуспех write връща -1. След изпълнението на write текущата позиция се увеличава с броя на действително записаните байтове. При опит да се пише от текуща позиция след края на файла, то се увеличава размера на файла и за него се разпределят нови блокове, ако се наложи.

Lseek е системен примитив за позициониране във файл. Той има следния прототип: off_t lseek (int fd, off_t offset, int flag).
Аргументът fd е файлов дескриптор на отворен файл. Lseek променя указателя на текущата позиция за отварянето на файла, съответно на файловия дескриптор fd. Аргументът offset задава отместването, с което ще се промени текущата позиция в брой байтове. Аргументър flag определя откъде ще се отчита отместването (от началото, от края или от текущата позиция). Lseek връща 0 при успех и -1 при неуспех.

Системният примитив **dup** има следния прототип: int dup (int fd). Той се използва за дублиране на файлов дескриптор.
Аргументът fd е файлов дескритор на отворен файл. Dup търси първият свободен файлов дескриптор в таблицата на файловите дескриптори за съответния процес и копира в него съдържанието на файловия дескриптор fd. При успех dup връща новия файлов дескритор, при неуспех връща -1. Dup има смислено приложение при пренасочване на входа и изхода.

Системният примитив **mkdir** се използва за създаване на каталог. Той има следния прототип:
int mkdir (const char *dirname, mode_t mode).
Аргументът dirname задава името на каталога (абсолютно или относително). Аргументът mode задава код на защита за създавания каталог. При изпълнението на mkdir за новосъздавания каталог се разпределя нов индексен описател и в родителския каталог се създава нов запис. Разликата от създаването на обикновени файлове е, че в новосъздадения каталог се инициализират двата стандартни записа с имена **.** и **..**. Също, за разлика от open и creat новосъздадения каталог не се отваря. При успех mkdir връща 0, при неуспех връща -1.

Системният примитив **rmdir** се използва за изтриване на каталози.Той има следния прототип:
int rmdir (const char *dirname).
Аргументът dirname задава името на каталога. За да може да се изтрие този каталог, той трябва да е празен, т.е. да съдържа само двата стандартни записа **.** и **..** При изпълнението на rmdir се освобождават индексния описател на каталога и съответния запис в родителския каталог. При успех rmdir връща 0, при неуспех връща -1.

Системният примитив **chdir** се използва за смяна на текущия каталог на текущия процес. Той има следния прототип:
int chdir (const char *dirname).
Аргументът dirname задава името на новия текущ каталог.
При успех chdir връща 0, при неуспех връща -1.

Системният примитив **link** се използва за създаване на твърда връзка към файл. Той има следния прототип:
int link (const char *oldname, const char *newname).
Аргументът oldname задава име на съществуващ файл. Този файл трябва да е обикновен. Аргументът newname задава името на новата твърда връзка към файла. При изпълнението на link в родителския каталог на newname се добавя нов запис с име newname и номер на inode, съвпадащ с inode на oldname. При това в този inode броячът на твърдите връзки се увеличава с 1. Link връща 0 при успех и -1 при неуспех.

Системният примитив **symlink** се използва за създаване на символна връзка към файл. Той има следния прототип:
int symlink (const char *toname, const char *fromname).
Аргументът toname е име на файл, който може да не съществува. Аргументът fromname задава името на новата символна връзка. По време на изпълнението на symlink се създава нов файл с име fromname – разпределя се нов inode, който се инициализира с тип символна връзка и се разпределя блок от данни, в който се записва toname, точно както е зададено в обръщението, след което в родителския каталог на fromname се добавя нов запис за новосъздадения файл. При успех symlink връща 0, иначе -1.

Системният примитив **unlink** се използва за унищожаване на файлове. Той има следния прототип:
int unlink (const char *filename).
Аргументът filename задава име на файл, който ще се унищожава. Файлът може да е от произволен тип, без каталог. При изпълнението на unlink в родителския каталог на файла се освобождава записа за файла и в индексния описател на файла броячът на твърдите връзки се намалява с 1. Ако новата стойност на брояча е 0, то индексният описател на файла, заедно с блоковете, разпределени за файла в областта за данни се освобождават. При успех unlink връща 0, при неуспех връща -1.

В многопотребителските системи трябва да е ясно какви са правилата за достъп на отделните потребители до файловете.
Ще разгледаме как е реализирана системата за защита в операционните системи UNIX и LINUX. Различните типове достъп до файл са достъп за четене (r), достъп за писане (w) и достъп за изпълнение (x).
Всеки потребител в системата принадлежи на една или повече групи.
С всеки файл при създаването му се свързват три атрибута, които се съхраняват в индексния описател на файла. Това са uid, gid и mode.
Атрибутът uid се инициализира с числовия идентификатор на собственика на процеса, който е създал файла.

Атрибутът gid се инициализира с числовия идентификатор на групата на собственика на процеса, който е създал файла. Атрибутът mode се инициализира чрез аргумент в системния примитив, който е създал файла.
Спрямо всеки отделен файл потребителите се разбиват в следните четири категории:

- **администратор** (root);
- **собственик** – потребител, който е собственик на файла;
- **група** – потребители, които не са собственик на файла, но попадат в групата на собственика;
- **други** – потребители, които не попадат в предните класове.

Администраторът има неограничен достъп до цялата файлова система.
За останалите три категории типовете разрешен достъп се определят от кода на защита mode, в който се съхраняват три групи по три бита (rwx), които определят правата на достъп до файла за другите потребители, за групата на собственика и за самия собственик. Ако някой от тези битове е 1, то съответната категория има съответното право върху файла, в противен случай тя няма това право.

Системният примитив **chmod** се използва за промяна на кода на защита на файл. Той има следния синтаксис:
int chmod (const char *name, mode_t mode).
Аргументът name задава името на файла, чийто код на защита ще се променя. Няма ограничение за типа на файла. Аргументът mode задава новия код на защита. При изпълнението на chmod в индексния описател на файла се променя неговия код на защита. За да завърши успешно chmod, процесът, който го изпълнява трябва да принадлежи на администратора или на собственика на файла. При успех chmod връща 0, при неуспех -1.

Системният примитив **chown** се използва за промяна на собственика или групата на файл. Той има следния синтаксис:
int chown (const char *name, uid_t owner, gid_t group).
Аргументът name задава име на файл. Аргументът owner задава числов индентификатор на новия собственик на файла, аргументът group задава числов идентификатор на новата група на файла. Възможно е да се зададе -1 за owner (group) – в такъв случай собственика (групата) на файла не се променя. При изпълнението на chown в индексния описател на файла се променят атрибутите uid и gid. За да се промени собственика на файла, процесът който изпълнява chown трябва да принадлежи на администратора. За да се промени групата на файла, процесът който изпълнява chown трябва да принадлежи на администратора или на собственика на файла. При това, собственикът на файла може да смени групата на файла само с група, на която той е член. При успех chown връща 0, при неуспех -1.

8. Компютърни мрежи и протоколи – OSI модел. Канално ниво. Маршрутизация. IP, TCP, HTTP.

Ще разгледаме моделът **OSI** (open system interconnection), създаден от международната организация **ISO** (international standard organization) за връзка между отворени системи. Всъщност OSI-моделът е абстрактен модел на мрежова архитектура, който описва предназначението на слоевете, но не се обвързва с конкретен набор от протоколи.
Всеки слой от модела на една машина взаимодейства с едноименния слой на друга машина. Правилата по които се осъществява това взаимодействие се определят от **протокол**, който отговаря на съответния слой. На практика при комуникацията между съответните слоеве на двете машини не се предават данни. Всеки слой предава данни и контролна информация на непосредствено по-долния слой, докато се достигне най-долния слой. Под него е физическата среда за предаване, където се осъществява реалната комуникация между машините. В приемника получените данни се разпространяват в обратна посока - от най-долния слой нагоре, като всеки слой премахва контролната информация, която се отнася до него. Всеки слой предоставя **интерфейс** на непосредствено по-горния слой, който определя какви функции и услуги му се предоставят. В OSI-модела има седем слоя – физически, канален, мрежов, транспортен, сесиен, представителен, приложен – изброени са в последователност от най-долния към най-горния слой.
Физическият слой има за задача да реализира предаването на битове през физическата среда. Основна функция на физическия слой е да управлява кодирането и декодирането на сигналите, представящи двоичните цифри 0 и 1. Този слой не се интересува от предназначението на битовете. Физическият слой трябва да осигурява възможност на по-горния канален слой да активизира, поддържа и прекратява физическите съединения.
Основна функция на **каналният слой** е откриването и евентуалното коригиране на грешки при предаването на данните. Данните на канално ниво се обменят на порции, наречени **кадри** (обикновено с дължина от няколко стотин до няколко хиляди байта). При надеждна комуникация приемникът трябва да уведомява изпращача за всеки успешно получен кадър като му изпраща обратно потвърждаващ кадър. Функциите на каналния слой обикновено се реализират смесено - апаратно и програмно.
Мрежовият слой отговаря за функционирането на комуникационната подмрежа. Приложните програми, които се изпълняват в двете крайни системи взаимодействат помежду си посредством **пакети** от данни. Основна задача на мрежовия слой е маршрутизирането на тези пакети. Пакетите са с фиксирана големина в рамките на една мрежа. За системите, реализиращи възлите на комуникационната подмрежа този слой е последен.

Функциите на мрежовия слой, както и на по-горните слоеве се реализират програмно.
Транспортният слой осигурява транспортирането на произволно дълги съобщения от източника до получателя. Той е най-ниският слой, който реализира връзка от тип “край-край” между комуникиращите системи. В транспортния слой на изпращача съобщенията се разбиват на пакети и се подават на мрежовия слой, а в транспортния слой на получателя подадените от мрежовия слой пакети се реасемблират. Транспортният слой освобождава по-горния сесиен слой от грижата за надеждното и ефективно транспортиране на данните между крайните системи.
Сесийният слой е отговорен за диалога между две комуникиращи програми. Съобщения се обменят след като двата крайни абоната установят **сесия**. Сесийният слой осигурява различни режими на диалог – двупосочен едновременен диалог, двупосочен алтернативен диалог, еднопосочен диалог. Освен това той предоставя възможност за прекъсване на диалога и последващо възстановяване от мястото на прекъсването. При липсата на сесиен слой всяко съобщение се предава независимо от другите съобщения.
Представителният слой е най-ниският слой, който разглежда значението на предаваната информация.
Първата функция на този слой е да определи общ синтаксис за предаване на съобщенията. Втората функция на слоя е да унифицира вътрешната структура на представените данни в съобщенията. По този начин за по-горния приложен слой няма значение дали двете крайни системи използват различни представяния на данните.
Приложният слой е най-горният слой, към който се свързват потребителските процеси в двата крайни абоната. Някои потребителски процеси са интерактивни - взаимодействат си в голям период от време с кратки съобщения от тип заявка-отговор. Други потребителски процеси взаимодействат с малко на брой големи по обем порции от данни. За двата вида процеси се предвиждат различни протоколи на приложния слой.

Каналното ниво има три основни функции - да осигури подходящ интерфейс на по-горното мрежово ниво, да открива грешки по време на предаването и да управлява информационният обмен. Данните за каналното ниво представляват последователност от **кадри** (frame). Каналното ниво взима пакетите, които му се подават от мрежовото ниво и ги затваря в кадри. Всеки кадър се състои от заглавна част (header), поле за данни, което съдържа пакета и опашка (trailer). Дължината на кадъра обикновено е ограничена отгоре. Физическото ниво възприема информацията от каналното ниво като поток от битове, без да се интересува от нейната структура. Получателят идентифицира в потока от битове кадрите и въз основа на служебната информация в тях ги контролира за грешки. За целта опашката на кадъра съдържа

контролна сума (обикновено 2 байта), която се изчислява върху останалата част от кадъра преди той да бъде предаден. Когато кадърът пристигне в получателя, контролната сума се преизчислява и ако тя е различна от предадената контролна сума, то получателят отхвърля кадъра и евентуално изпраща съобщение за грешка към източника. Ако контролните суми съвпаднат, то получателят приема кадъра и изпраща потвъждаващ кадър до източника. След това се премахва служебната информация на кадъра и информационният поток се предава на мрежовото ниво вече под формата на пакети.

Разглеждаме ситуация при която машината *A* изпраща кадри към машината *B* по канал с шум. Кадрите могат да се изкривят по време на предаването или изцяло да се изгубят.
Когато в *B* постъпи нов кадър от *A*, *B* изчислява контролната сума на кадъра. Ако тази контролна съвпадне с изпратената контролна сума, *B* изпраща данните на неговото мрежовото ниво, формира потвърждаващ кадър и го изпраща към *A*. Ако контролната сума не съвпадне, то кадърът е сгрешен и се отхвърля от *B* и *B* не изпраща потвърждение или изпраща негативно потвърждение обратно към *A*.
Възможно е *A* да изпрати кадър към *B*, но този кадър да се изгуби. Тогава *B* не може да реагира, тъй като не е регистрирал грешка. За да се избегне тази ситуация, *A* стартира брояч на време (**time-out**) с изпращането на всеки кадър. Времето, което отчита брояча трябва да е по-голямо от времето за предаване на кадъра, обработката му в приемника и получаване на потвърждение. Ако кадърът не се потвърди в рамките на това време, то *A* предава кадърът отново. Тук се обхваща случая в който *A* получава от *B* негативно потвърждение (това е равносилно с изтичане на времето за потвърждение).
Възможно е *A* да изпрати кадър към *B*, този кадър да се получи в *B*, но потвърждението да се изгуби. В този случай *A* не знае дали изпратеният кадър въобще е пристигнал до *B*.
При всички положения *A* изпраща наново кадъра и ако не се вземат мерки, *B* ще получи същия кадър и ще го изпрати към мрежовото ниво, което ще доведе до недопустимо дублиране на данните. За целта с всеки кадър се свързва пореден номер.
В случая е достатъчно номерът да е един бит (0 или 1). Във всеки един момент *B* очаква кадър с определен номер. Ако *B* получи кадър с друг номер, този кадър е дубликат и се отхвърля. Ако *B* получи кадър с очаквания номер, кадърът се приема и очакваният номер на кадър се инвертира (ако е бил 0 става 1, ако е бил 1 става 0). От своя страна *A* номерира алтернативно кадрите, които изпраща към *B*. Естествено, ако даден кадър бъде изпратен отново неговият номер не се променя.

Едно подобрение на разгледания подход е следния – тъй като кадрите текат и в двете посоки, потвържденията може да се

закачат за кадрите с данни (за целта се използва поле в заглавието на кадъра). Проблемът е, че данните от мрежовото ниво, към които трябва да се прикрепи потвърждението може да се забавят прекалено дълго и броячът на време да изтече, което ще доведе до повтаряне на кадъра. Обикновено решението е следното - изчаква се фиксиран брой милисекунди и ако дотогава не пристигне пакет от мрежовото ниво се изпраща самостоятелен потвърждаващ кадър.
Ще разгледаме някои **протоколи с прозорци**. Те са по-ефективни от горния протокол спри и чакай, тъй като позволяват изпращане на повече от един кадъра преди да се чака за потвърждение.
При тези протоколи всеки кадър се номерира с число от 0 до някакъв максимум, обикновено от вида
2^n - 1, така че номерът да се вмества точно в n бита.
Във всеки един момент предавателят поддържа множество от поредни номера на кадри, които попадат в **прозореца на предавателя**. От друга страна, получателят поддържа **прозорец на получателя**.
Поредните номера в рамките на прозореца на предавателя съответстват на кадри, които вече са били изпратени и чакат потвърждение. Когато от мрежовото ниво на предавателя пристигне нов пакет, той разширява прозореца откъм горната му граница. Когато в предавателя пристигне потвърждение, долната граница на прозореца се придвижва напред. Предимството е, че с едно потвърждение могат да се потвърдят повече от един последователно номерирани кадри. Тъй като кадрите в прозореца на предавателя могат да се изкривят или изгубят, те трябва да се съхраняват за евентуалното им повторно изпращане. Така предавателят трябва да разполага с достатъчен брой буфери.
Ще отбележим една особеност – прозорецът на предавателя никога не трябва да е изцяло запълнен (т.е. да съдържа всички номера), тъй като потвърждаването на изцяло запълнен прозорец не може да се интерпретира еднозначно – то може да означава както, че всички кадри са били приети, така и че всички кадри са били отхвърлени.
Номерата на кадрите в прозореца на получателя съответстват на кадри, които могат да бъдат получени. Когато в получателя пристигне кадър, чийто номер съвпада с долната граница на неговия прозорец, данните от този кадър се предават към мрежовия слой на получателя и прозорецът се завърта напред, т.е. придвижват се и горната и долната му граница. Ако номерът на пристигналия кадър попада в прозореца, но не съвпада с долната му граница, този кадър не се отхвърля, а се буферира.
За разлика от прозореца на предавателя, прозорецът на получателя има фиксиран размер. За да поддържа прозорец на предавателя трябват буфери и битова карта, която показва кои буфери са запълнени.
При наличие на сгрешен или изгубен кадър, предавателят ще продължи да предава кадри, преди да разбере че има проблем.

Въпросът е какво да прави получателят с успешно получените кадри след сгрешен или изгубен кадър.
Едната стратегия (**go back n**) е тези кадри да се отхвърлят. Тя съответства на прозорец на получателя с размер 1. С други думи, получателят приема единствено следващия поред кадър, който трябва да се предаде към мрежовия слой. В даден момент броячът на време на предавателя ще изтече и той ще изпрати наново всички кадри, започвайки от сгрешения (изгубения).
Другата стратегия (**selective repeat**) е получателят да буферира успешно получените кадри след сгрешен или изгубен кадър. Когато броячът на време в предавателя изтече, той изпраща наново само най-стария сгрешен (изгубен) кадър. Ако повторното изпращане е успешно, получателят може последователно да изпрати към своя мрежов слой кадрите, които е буферирал. Обикновено при тази стратегия получателят изпраща служебен кадър, който известява на предавателя за сгрешен или изгубен кадър - това води до по-бързо повторно предаване на съответния кадър. Стратегията съответства на размер на прозореца на получателя по-голям от 1. Всеки успешно получен кадър, чийто номер попада в прозореца на получателя се буферира и се изпраща към мрежовия слой чак след като са изпратени предшестващите го в прозореца кадри.

Мрежите с общодостъпно предаване се характеризират с общ комуникационен канал, който се споделя от всички машини, включени в мрежата. Всеки изпратен кадър минава през общия канал и достига до всички машини в мрежата. Адресно поле в кадъра посочва за кой е предназначен този кадър. Когато една машина получи кадър, тя проверява дали той е предназначен за нея. Ако това е така, кадърът се приема и обработва, в противен случай се отхвърля. Общодостъпни многоточкови канали се използват най-вече при локалните мрежи.
Най-разпространената локална мрежа е **Ethernet**. Един персонален компютър се свързва в Ethernet мрежа с помощта на NIC (Network Interface Card) - това е каналната станция, която осъществява обмена по Ethernet канала.
Преди да изпрати кадър, каналната станция проверява състоянието на канала. Ако той е свободен, тя веднага започва предаване. Ако каналът не е свободен (т.е. предава друга станция), то станцията изчаква неговото освобождаване. След като започне предаването, каналната станция продължава да подслушва канала. Ако се открие изкривяване на предавания сигнал, това означава, че по същото време е започнала да предава друга станция и е настъпила **колизия**. В този случай двете станции спират предаването и всяка от тях изчаква случаен интервал от време преди да предава отново.
Кадрите в Ethernet имат максимална дължина 1500 байта. Когато една предаваща станция разбере за конфликт, тя веднага спира предаването като орязва настоящия кадър. За да може да се прави разлика между валидни и орязани кадри, дължината на

кадъра трябва да е поне толкова голяма, че да може предаването да не е завършило, преди станцията да разбере за конфликта. Затова кадрите имат минимална дължина 64 байта.
Адресите са по шест байта. Адрес на получател, състоящ се само от 1 е предназначен за всички станции.
В началото в Ethernet се използва коаксиален кабел и скоростта на предаването е достигала 10 Mb/s. По-нататък се въвежда използването на **хъбове** (hub) и скоростта на предаване скача десетократно – до 100 Mb/s. Каналните станции се свързват към хъба чрез две медни усукани двойки. По една от усуканите двойки се предава към хъба, а по другата се приема от него. Ако хъбът получи кадър по някоя линия, той изпраща този кадър по всички останали линии. Важно е да се отбележи, че хъбът работи на физическо ниво и не знае адресите на каналните станции.
Алтернатива на хъбовете са по-интелигентни устройства, които работят вече на канално ниво – **мостовете** и **превключвателите**, които не предават кадрите по всички възможни линии. Те имат информация (във формата на таблици), чрез която въз основа на адреса на кадъра определят по коя изходна линия да се изпрати този кадър.

Основната функция на мрежовото ниво е да маршрутизира пакетите от източника към получателя. В повечето мрежи пакетите ще изминат това разстояние за няколко хопа.
Маршрутен алгоритъм е част от софтуера на мрежовото ниво, която определя по коя от изходните линии да се изпрати пристигнал пакет. За целта всеки маршрутизатор притежава **маршрутна таблица**.

Маршрутизиращите алгоритми са два вида - **неадаптивни** и **адаптивни**. При неадаптивните алгоритми маршрутизацията не се извършва на базата на текущата топология на мрежата. Маршрутите между всеки два възела в мрежата се изчисляват предварително и маршрутните таблици се попълват ръчно от мрежовите администратори. При промяна на топологията на мрежата (например при отпадане на възел или на връзка), администраторите ръчно трябва да променят маршрутните таблици, така че всеки два възела да останат свързани.
Това прави неадаптивните алгоритми приложими само в малки мрежи, при които рядко настъпват промени.
Неадаптивните алгоритми се наричат още **статични**.
Една маршрутна таблица в един възел съдържа по един ред за всяко възможно местоназначение. При статична маршрутизация, един ред съдържа толкова полета, колкото са непосредствените съседи на конкретния възел. За всеки ред, с всеки съсед се свързва едно тегло между 0 и 1, така че сумата от теглата на всеки ред да е точно 1. Когато във възела пристигне пакет, първо се определя кое е местоназначението, след това се генерира едно

случайно число и в зависимост от теглата в реда на съответното местоназначение се определя към кой съсед да се изпрати пакета. Така колкото е по-голямо теглото на един съсед, толкова повече пакети ще се изпращат към него.

При адаптивните алгоритми маршрутните таблици се променят динамично за да отразяват промени в топологията и натовареността на трафика. Важна характеристика на един адаптивен алгоритъм е неговата **скорост на сходимост** - тя се определя от времето, което е необходимо да се преизчислят маршрутните таблици на всички маршрутизатори в мрежата при промяна в топологията или трафика.

При **централизираните** адаптивни алгоритми в мрежата се създава един маршрутен управляващ център. Той изчислява маршрутните таблици на всички възли и им ги изпраща. За да се адаптират маршрутните таблици към текущата топология и текущия трафик, всички възли трябва да изпращат информация към маршрутния център. На базата на получените сведения, маршрутният център изчислява теглата на ребрата и след това пресмята оптималният маршрут между всеки два възела. Добре е да се поддържат алтернативни пътища между възлите.
Информацията от по-близките до маршрутния център възли ще пристигне по-бързо отколкото от по-далечните. Поради тази причина, периодът на обновяване на маршрутните таблици трябва да е поне два пъти по-голям от времето за преминаване на пакет от маршрутния център до най-отдалечения от него възел.
Преизчислена маршрутна таблица, получена в един възел не трябва да се използва веднага, тъй като маршрутните таблици пристигат по различно време в различните възли.
Ако по някаква причина маршрутният център отпадне, мрежата остава без управление. За целта може да се дублира маршрутният център, но тогава служебният трафик би се увеличил твърде много.
При маршрутизацията с **вектор на разстоянието** всеки маршрутизатор изгражда и поддържа маршрутна таблица, в която всеки ред съдържа адрес на дадено местоназначение, адрес на следващата стъпка към това местоназначение по най-добрия известен до момента път и дължината на този път.
Маршрутизаторите разменят на фиксиран брой милисекунди съдържанието на маршрутните си таблици само с директно свързаните към тях съседни маршрутизатори.
Предполага се, че всеки маршрутизатор знае метриката на връзките до своите съседи. Да предположим, че за метрика е избрано време-закъснението на пакетите.
Нека даден маршрутизатор J получи маршрутната таблица на съседа си X, като X_i е обявеното от X закъснение до маршрутизаторът i. Ако закъснението от J до X е m, то от J до всеки маршрутизатор i има път през X със закъснение $X_i + m$.

Възможни са четири случая:

- ако в маршрутната таблица на J няма ред за направлението i, то J добавя такъв ред и записва в него следваща стъпка X и закъснение $X_i + m$;
- ако в маршрутната таблица на J има ред за направлението i и в него е записана следваща стъпка X, то стойността на закъснението се актуализира с $X_i + m$ независимо дали тя е по-голяма или по-малка от предходната стойност;
- ако в маршрутната таблица на J има ред за направлението i, в него е записана следваща стъпка различна от X и закъснение по-голямо от $X_i + m$, то редът се актуализира като за следваща стъпка се записва X, а за закъснение $X_i + m$;
- ако в маршрутната таблица на J има ред за направлението i, в него е записана следваща стъпка различна от X и закъснение по-малко или равно на $X_i + m$, то редът не се променя.

Сериозен недостатък на маршрутизиращите алгоритми с вектор на разстоянието е ниската им скорост на сходимост. Добрите новини се разпространяват бързо в мрежата, но лошите новини обикновено изискват твърде голям брой периодични съобщения за да достигнат до всички маршрутизатори. Този проблем се нарича **броене до безкрайност**. За него съществуват някои частични, но не изчерпателни решения (като например разделяне на хоризонта).

При маршрутизирането със **следене състоянието на връзката**, всеки маршрутизатор трябва да извършва следните пет основни действия:

1. Откриване на съседните маршрутизатори и техните мрежови адреси.
2. Измерване на цените на връзките до съседните маршрутизатори.
3. Конструиране на пакети с информация за състоянието на връзките.
4. Изпращане на тези пакети до всички останали маршрутизатори.
5. Изчисляване на най-късия път до всеки маршрутизатор в мрежата.

След включването на един маршрутизатор неговата първа задача е да научи кои са съседите му. Това се постига чрез изпращане на “ехо” пакет по всяка от изходящите линии на маршрутизатора. От своя страна, всеки от съседите отговаря като съобщава името си. Това име трябва да бъде уникално в мрежата.

Всеки маршрутизатор трябва да може да определи време-закъснението до своите съседи. Най-простият начин е маршрутизаторът да изпрати "ехо" пакет към всеки свой съсед на който трябва директно да се отговори. Времето от изпращането на

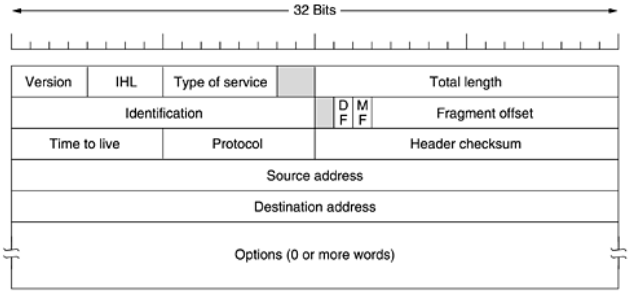
"ехо" пакета до получаване на отговора се дели на две и по този начин се получава времето-закъснение до съответния съсед.
Друг въпрос е дали при измерването да се взима предвид натовареността на възлите. Разликата се постига в зависимост от това кога маршрутизаторът стартира измерването - когато пакетът постъпва в съответната изходяща опашка или когато пакетът се придвижи в началото на опашката.

След като събере необходимата информация за състоянието на връзките си, следващата задача на маршрутизатора е да конструира пакет, който съдържа тази информация.
Пакетът трябва да съдържа уникалното име на подателя, пореден номер, срок на годност и списък със съседите на подателя, като за всеки съсед е указана цената на връзката до него.
Определянето на момента, в който трябва да бъдат подготвени и изпратени пакетите е важна задача.
Един възможен начин е това да става през определени равни интервали от време. Друга по-добра възможност е пакетите да се подготвят и изпращат само при промяна в топологията на мрежата - след отпадане или поява на нов съсед или промяна в цената на някоя връзка.

Най-съществената част на алгоритъма е надеждното доставяне на пакетите с информацията за състоянието на връзката до всички маршрутизатори.
За разпространението на пакетите се използва методът на наводняването (flooding). При него всеки пакет се изпраща по всички линии, освен линията по която е пристигнал.
Обработката на всеки пристигнал пакет започва с проверка дали пакетът има по-голям пореден номер в сравнение с най-големия пореден номер, който е пристигнал до този момент от този източник. Ако номерът е по-голям, информацията от пакета се записва в таблицата с информация за състояние на връзките и пакетът се предава по останалите линии. Ако номерът е по-малък или равен, пакетът се отхвърля.
Този алгоритъм има някои проблеми. Ако поредният номер не е достатъчно голям, той може да се превърти. Затова се използват 32-битови поредни номера. В полето за срок на годност маршрутизаторът-подател указва продължителността на интервала от време в секунди, през който пренасяната от него информация трябва да се счита за валидна. Всеки маршрутизатор, който получи даден пакет намалява с единица стойността на това поле преди да го предаде към своите съседи. Освен това, след като маршрутизаторът запише данните от пакета в своята таблица, той продължава да намалява срока на годност на тези данни на всяка следваща секунда. Ако срока на годност стане 0, данните се изтриват.

След като един маршрутизатор получи пълна информация за състоянието на връзките на всички останали маршрутизатори, той може да приложи алгоритъма на Дейкстра.
Изчислените маршрути се записват в маршрутните таблици.
Големите по размер мрежи изискват използване на маршрутизатори с голям обем памет.

От гледна точка на мрежовото ниво Internet е съвкупност от **автономни системи**. Всяка автономна система се състои от една или повече мрежи. В рамките на една автономна система има фиксирани правила за предаване и фиксиран размер на пакета. Internet всъщност изгражда правила за връзка между отделните автономни системи. За целта се използва протоколът **IP**. Между автономните системи данните се придвижват под формата на **дейтаграми**. Задачата на IP е да извърши успешно предаване на дейтаграмите от източника до получателя без значение дали те са в една и съща мрежа или в различни мрежи. Всяка дейтаграма се изпраща самостоятелно, като по пътя тя може да се фрагментира на по-малки единици. Когато фрагментите достигнат до получателя той ги реасемблира за да получи оригиналната дейтаграма.
Ще разгледаме формата на IP-дейтаграмата във версия 4 (4-байтови адреси). IP-дейтаграмата се състои от заглавна част и част за данни. Заглавната част е 20B+опции с променлива дължина и има следния формат:



Полето Version указва версията на протокола, към който принадлежи дейтаграмата.
Полето IHL указва дължината на заглавната част в 32-битови думи. То е необходимо, тъй като полето Options има променлива дължина.
Полето Type of service показва какво обслужване очаква дейтаграмата. Различните видове данни, например видеоизображение, глас, файлове предполагат различно обслужване. Практически сегашните маршрутизатори не обръщат внимание на това поле.
Полето Total length съдържа общата дължина на дейтаграмата (заглавна част + данни).
Полето Identification съдържа номер на дейтаграмата. Всички фрагменти на една и съща дейтаграма имат еднакъв номер и по този начин получателя разбира кой фрагмент към коя дейтаграма принадлежи.

Флагът DF (don't fragment) указва на маршрутизаторите да не фрагментират дейтаграмата. Всички автономни системи трябва да могат да приемат фрагменти от поне 576B. Ако размерът на фрагментите е по-голям и флагът DF е 1, то дейтаграмата може да пропусне някоя автономна система с по-малка дължина на пакета, дори тя да се намира на оптималния маршрут.
Флагът MF (more fragments) за всички фрагменти на дейтаграмата, освен последния е 1, а за последния е 0, т.е. той показва дали получен фрагмент е последен в дейтаграмата или не.
Полето Fragment offset указва къде се намира фрагмента в оригиналната дейтаграма.
Полето Time to live е брояч, който ограничава продължителността на живота на дейтаграмата. Това поле се намаля с единица на всеки hop, а освен това се намаля с единица и за всяка секунда престой в маршрутизатор. Когато полето стане 0, дейтаграмата се премахва и в обратна посока се изпраща предупредителен пакет.
Полето Protocol указва протокола на транспортно ниво, към който трябва да се предаде дейтаграмата. Той може да бъде TCP, UDP или някой друг.
Полето Header checksum е контролна сума само на заглавната част. Тя трябва да се преизчислява на всеки hop, тъй като поне едно поле се променя - Time to live.
Полетата Source Address и Destination address съдържат съответно адрес на източника и адрес на получателя.
Възможни са различни опции в полето Options. Най-често се използват опции, които налагат ограничения върху пътя, който трябва да измине дейтаграмата и опции, в които се записва пътя по който е минала дейтаграмата – те се използват с цел по-лесно да се проследяват грешки при маршрутизирането.
Всеки хост и маршрутизатор в мрежата има IP-адрес.
Всички IP-адреси са 32-битови. Всеки IP-адрес се дели на две части – номер на мрежа и номер на хост. Номерът на мрежата е непрекъсната порция от битове в лявата част на адреса, а номерът на хоста е останалата непрекъсната порция от битове в дясната част на адреса.
В зависимост от структурата си IP-адресите се делят на пет класа. Класовете се различават по първите няколко бита на адреса, които се наричат **сигнални битове**.
За клас A сигналният бит е 0, номерът на мрежата е 7 бита, така че в него са възможни приблизително 120 мрежи, номерът на хоста е 24 бита, така че всяка мрежа има приблизително 16000000 хоста.
За клас B сигналните битове са 10, номерът на мрежата е 14 бита, така че в него са възможни приблизително 16000 мрежи, номерът на хоста е 16 бита, така че всяка мрежа е с приблизително 65000 хоста.
За клас C сигналните битове са 110, номерът на мрежата е 21 бита, така че са възможни приблизително 2000000 мрежи,

номерът на хоста е 8 бита, така че всяка мрежа е с приблизително 250 хоста.
Клас D, със сигнални битове 1110 е предназначен за работа с групови адреси, а клас E със сигнални битове 1111 е резервиран за бъдеща употреба.

Големият недостатък на IP-адресацията е, че половината адреси са от клас A и се разпределят само между малко повече от 120 автономни системи, въпреки че всяка от тях може да съдържа милиони хостове.
Всяка мрежа трябва да има уникален номер и всички хостове в дадена мрежа трябва да имат един и същ номер на мрежата. Това води до проблеми при нарастване на броя на мрежите. Решението на проблема е да се разреши разделянето на една мрежа на **подмрежи**, но за външния свят тя да изглежда като една мрежа. За целта полето за мрежов номер се разширява надясно, като се отнемат битове от номера на хост.
За имплементация на подмрежите маршрутизаторите се нуждаят от **мрежова маска**, която определя границата между номера на мрежата + номера на подмрежата и номера на хоста.

За адресация в Internet се използват 32-битови IP-адреси. Хостовете, свързани към локална мрежа Ethernet, притежават уникални 48-битови физически адреси от тази мрежа.
За установяване на съответствието между IP адреса и Ethernet адреса на хостовете в локалната мрежа се използва протокол за право преобразуване на адресите **ARP**. Когато даден хост трябва да изпрати дейтаграма към машина от локалната мрежа, чийто IP адрес е известен, но не е известен Ethernet адреса, мрежовият слой разпространява в локалната мрежа ARP пакет-заявка. Този пакет-заявка е от тип broadcast, т.е. предава се до всички машини. В полето “Ethernet адрес на подателя” е записан съответният адрес на хоста, който изпраща ARP заявката.
В полето “Данни” е записано ARP съобщение от вида “who is X.X.X.X tell Y.Y.Y.Y”, където X.X.X.X и Y.Y.Y.Y са IP адреси съответно на получателя и на подателя. Всички машини от локалната мрежа игнорират заявката с изключение на хоста, чийто адрес съвпада с X.X.X.X. Този хост изпраща ARP пакет-отговор само на подателя, тъй като вече знае неговия Ethernet адрес от получената заявка. В полето “Данни” на пакета-отговор е записано ARP съобщение от вида “X.X.X.X is hh:hh:hh:hh:hh:hh”, където hh:hh:hh:hh:hh:hh е Ethernet адреса на хоста, изпращащ пакета-отговор. Обикновено хоста, който изпраща ARP заявката запомня (кешира) получените 48-битови Ethernet адреси, за да могат да се използват при следващо предаване. При определяне на Ethernet адреса на получателя на дадена дейтаграма първо се проверява дали този адрес вече е кеширан и ако не е, се изпраща ARP заявка. Освен това всеки хост при първоначалното си стартиране изпраща broadcast съобщение, което съдържа неговият

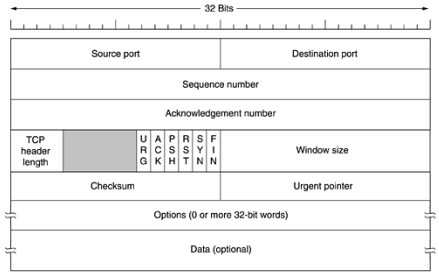
IP адрес и Ethernet адрес, което се получава от всички останали хостове в локалната мрежа и те записват тази информация в своите кешове.
Протоколът **RARP** е за намиране на IP адреси по Ethernet адреси. За функционирането му е необходимо в мрежата да е включен хост, който функционира като RARP сървър. Този сървър съхранява съответствието между Ethernet и IP адреси на станциите в мрежата. Обикновено RARP се използва от машини без твърди дискове, които чрез него научават своя IP адрес.

Големият проблем на IP протокола е недостига на адреси. По принцип съществуват над 2 милиарда адреси, но организацията им по класове е неефективна и заради нея се губят милиони адреси. Друг проблем е големината на маршрутните таблици. Съхраняването на таблици с милиони редове да кажем е възможно, въпреки че повечето маршрутизатори съхраняват таблиците си в оперативната памет, но друг сериозен проблем е сложността на алгоритмите за маршрутизация. Освен това, маршрутизаторите периодично трябва да изпращат маршрутните си таблици – колкото са по-големи, толкова по-голяма е вероятността части от тях да се загубят при предаването.
Решение на описаните проблеми е използване на безкласова маршрутизация **CIDR**. Основната идея е, че незаетите IP адреси се разпределят по блокове с различна големина (степен на 2), без да се взимат под внимание класовете.
Премахването на класовете усложнява маршрутизацията. Всеки ред от маршрутната таблица се разширява с 32-битова маска. По този начин маршрутната таблица е масив от тройки (IP адрес, маска, изходяща линия). Когато пристигне пакет, се извлича IP адресът на получателя. После маршрутната таблица се сканира ред по ред, като за всеки ред съответната маска се прилага към IP адреса и се сравнява получения номер с номера в реда. Възможни са няколко съвпадения – в такъв случай се избира реда с най-дълга маска.

Най-важните протоколи, обслужващи транспортния слой, са **TCP** (transmission control protocol) и **UDP** (user datagram protocol).

Предназначението на TCP е да осигурява надеждно предаване на данните между предавателя и приемника чрез установяване на връзка. Обменът на информация, който осъществява TCP се извършва посредством **сегменти**. При предаване TCP получава данни от по-горния слой, разделя ги на части, опакова ги в сегменти и ги изпраща на IP протокола. Той от своя страна опакова сегментите в дейтаграми и извършва маршрутизирането на всяка дейтаграма. При приемане IP протоколът разопакова пристигналите дейтаграми, след което предава получените сегменти на TCP протокола, който сглобява и подрежда данните от

сегментите в съобщения към по-горните слоеве така, както те са били изпратени.
Всеки край на TCP връзката се идентифицира с IP адреса на съответния хост и с 16-битово число, наречено **номер на порт**, което определя съответната приложна програма, използваща тази връзка. Комбинацията от адреса на хоста и номера на порта се нарича **socket**. Всеки TCP сегмент съдържа номерата на портовете на източника и на получателя и това позволява на TCP протокола да определи за коя приложна програма е предназначен съответния сегмент. Комбинацията от socket-а на източника и socket-а на получателя е уникална и тя идентифицира TCP връзката. Първите 1024 номера на портове са така наречените **well-known** портове, които са резервирани за най-често използваните стандартни приложни програми.
TCP сегментът се състои от заглавна част и част за данни и има следния формат:



Заглавната част включва задължителни полета с фиксиран размер 20 байта, към които може да бъде добавено поле Options. След опциите (ако има такива) следва полето на обменяните данни - Data, което също не е задължително.
Полетата Source port и Destination port са двубайтови и представляват номер на порта на източника и на получателя съответно, които заедно с IP адресите на източника и на получателя образуват номера на socket-и, идентифициращи уникално връзката.
Полето Sequence number е поредния номер на първия байт (в рамките на последователността от байтове, предавани от източника), който е записан в полето Data на сегмента.
Полето Acknowledgement number е номерът на първия байт данни, който се очаква да се получи със следващия сегмент, изпратен от другия край на TCP връзката.
Полето TCP header length е 4-битово и определя дължината на заглавната част на TCP сегмента в 32-битови думи. То е задължително, тъй като полето за опции е с променлива дължина.
Заглавната част на TCP сегмента съдържа и 6 еднобитови флага. Те имат следното предназначение:

- URG – указва, че е валиден е указателят за спешни данни;
- ACK – валиден е номерът на потвърждение, записан в полето Acknowledgement number на заглавната част;
- PSH – при активирането на този флаг, програмните модули управляващи транспортния слой на източника и на

приемника трябва да изпратят незабавно наличните данни колкото е възможно по-бързо към техния получател;
- RST – сегмент, в който е установен този флаг, служи за прекратяване на TCP връзката;
- SYN – сегмент с установен флаг SYN се използва при установяване на TCP връзка и за изпращане на началния номер, от който ще бъдат номерирани байтовете на изходящия информационен поток;
- FIN – сегмент, в който е установен този флаг, означава, че изпращачът прекратява предаването на данни.

Полето Window size определя темпа на информационния обмен от гледна точка на получателя на информационния поток. Стойността на прозореца указва на отсрещната страна колко байта могат да бъдат изпратени и съответно приети без препълване на входия буфер след последния потвърден номер на байт. При получаване на данни, размерът на прозореца намалява. Ако той стане равен на 0, изпращачът трябва да престане да предава данни. След като данните се обработят, получателят увеличава размера на своя прозорец, което означава, че е готов да получава нови данни.
Полето Urgent pointer се използва да укаже позицията на първия байт на спешните данни спрямо началото на полето данни.
Полето Checksum се изчислява върху целия TCP сегмент. При неговото изчисляване участват и някои полета от заглавната част на IP дейтаграмата, в която е опакован сегмента.
Полето Options на заглавната част на TCP сегмента е предназначено да предостави допълнителни възможности за управление на обмена. Най-важната възможност е указване на максимална дължина на сегмента. Всеки хост указва своята максимална дължина на сегмента и за осъществяване на обмена се приема по-малката от двете. Ако максималната дължина на сегмента не се договори се приема по подразбиране, че нейната стойност е 556 байта, което е допустимо за всички интернет хостове.
При първоначално отваряне на връзката между два хоста е необходимо всеки от тях да изпрати на другия началния номер на байтовата последователност, която ще изпраща, и съответно да получи насрещното потвърждение за получаването на този номер. Процедурата за установяване на връзка се нарича **трикратно договаряне** и в нормалния случай е следната:

1. Хостът (клиентът), който отваря връзката, изпраща SYN сегмент. В същия сегмент клиентът указва номера на порта на сървъра, с който трябва да се установи връзка, и началният номер x на потока байтове, който клиентът ще предаде към сървъра.
2. Сървърът отговаря със собствен SYN сегмент, включващ началния номер y на неговия поток от байтове. В сегмента се съдържа потвърждение за SYN сегмента с номер на

потвърждението, равен на x+1, тъй като за самия SYN сегмент е необходим един пореден номер.
3. Клиентът трябва да потвърди получаването на SYN сегмента от сървъра, като изпрати сегмент с потвърждаващ номер y+1.

Затварянето на връзката също се извършва чрез трикратно договаряне. Тъй като TCP връзката е пълен дуплекс, тя се затваря, когато всеки от двата хоста прекрати своя изходящ информационен поток. Такъв тип затваряне на връзката се нарича още симетрично. Вариантът, при който даден хост прекратява информационния обмен и в двете посоки се нарича асиметрично прекратяване на връзката.
За симетричното затваряне на една връзка е необходим обмен на 4 сегмента – по два за всяка посока. Даден хост може да инициира затваряне на своята част на връзката, когато изпрати сегмент с установен флаг FIN, след като приключи с предаването на данни. Хостът, получил този сегмент, може да продължи да изпраща данни при положение, че не е затворил връзката. Всеки хост, който получи FIN сегмент, изпраща обратно потвърждение с номер, равен на получения пореден номер + 1, тъй като FIN сегментът изисква един пореден номер.
За асиметрично затваряне на връзката се изпраща сегмент с вдигнат флаг RST.

UDP е прост транспортен протокол за предаване на дейтаграми в мрежите с комутация на пакети. За разлика от TCP, той не осъществява надежден транспорт. Дейтаграмите се изпращат от източника без да се контролира дали са достигнали до получателя. Затова форматът на заглавната част на UDP дейтаграмата е много по-прост: тя съдържа само номерата на портовете на източника и на получателя, дължината на дейтаграмата и контролна сума. UDP има смисъл да се използва при мрежи с висока надеждност.
Най-мощното приложение на Internet е **WWW**.
Основното, на което се базира web-технологията е хипертекста. Това е текст, който съдържа в себе си информация как да бъде изобразен на екрана. Изобразяването става чрез специална програма, наречена хипертекстов browser.
Хипертекстът се оформя като съвкупност от страници. Всяка страница си има уникално URL – уникален адрес, който еднозначно указва местоположението на страницата в целия Internet. Другото нещо е хиперлинкът – под част от текста, който се изобразява отдолу стои URL на друга страница. С други думи хипертекстовите страници съдържат препратки към други хипертекстови страници.
Browser е клиентът, който изтегля и изобразява страниците.
Web server е сървърът, който съхранява страниците. За комуникация между browser-ите и сървърите е създаден протокола **HTTP**.

Едно URL съдържа протокол, име на домейн и пътя на страницата върху диска на сървъра. При осъществяване на връзка между browser-а и сървъра по URL-то първо по името на сървъра, а после по пътя върху диска на сървъра страницата физически се изтегля от сървъра, предава се на browser-а и той я изобразява. Една страница се прехвърля в рамките на една HTTP-сесия. Ако човек кликне върху линк на изтеглената страница, browser-ът установява нова сесия, подава се новото URL, изтегля се страницата от сървъра и отнове се изобразява от browser-а.
Протоколът HTTP се базира на TCP. HTTP клиентът отваря сесия на произволен порт с номер по-голям от 1024. HTTP сървърът слуша за заявки на порт 80.
При HTTP протокола имаме подготвителна фаза – прави се заявка за HTTP сесия към сървъра, след това се прави HELLO към сървъра, потвърждава се от сървъра и се изпращат методи на HTTP. Методите са GET, HEAD, POST, PUT, TRACE, CONNECT. Основният метод е GET. Като аргумент му се подава пълното име на страницата върху диска на сървъра. Страниците могат да се кешират върху proxy-сървъри, затова в метода GET има if-условие. Всяка хипертекстова страница има заглавие, което съдържа описание на възрастта на страницата, датата на последната модификация и др. Методът HEAD взима само заглавието на страницата. Той позволява на browser-а бързо да провери дали я има физически страницата и кога е модифицирана последно (спестява се тегленето на тялото на страницата).
Методът PUT служи за прехвърляне на страница върху сървъра. Той е свързан с обмен на два етапа – първо се дава адресът на страницата, след това се прехърля самата страница. Методът POST е аналогичен на метода PUT с тази разлика, че той добавя новите данни към съществуващ адрес.
Методът TRACE връща от сървъра получените данни по заявката. Той се използва за тестване – да видим дали сървърът е получил това, което сме изпратили.
При първите версии на HTTP протокола за изпълнението на всеки метод се прави отделна HTTP сесия – тя отваря TCP съединение, праща нещо, след това затваря последователно съединението и сесията. С други думи имаме 1:1 – една сесия, едно съединение. След това започва развитие – стремежът е да не се затваря съединението, т.е. да има няколко сесии върху едно съединение. Ако за всяко изпълнение на GET се затваря съединението, а предстои четене на серия от страници това е много неефективно. При версията 1.0 на HTTP на едно съединение отговаря една сесия.
При версията 1.1 на HTTP на едно TCP съединение отговарят няколко сесии (няколко команди). Тези команди касаят различни хипертекстови страници, но достъпът до тях се прави с едно TCP съединение. За целта се създава съобщителен канал. По него в пълен дуплекс текат заявки, а в обратна посока – отговори. Така не се работи по метода спри и чакай за всяка заявка.

Друга особеност е, че зад един IP адрес може да има няколко имена на сървъри (така наречените виртуални сървъри). За клиента те са различни сървъри, но реално зад тях стои един и същ IP адрес. По-късно те могат да мигрират към други компютри, но вътре в тях URL-то ще се запази и по този начин не е необходима промяна на хиперлинковете към страници върху виртуалните сървъри.

9. Растеризиране на отсечка, окръжност и елипса.

Компютърната графика е наука за методите за съхранение, създаване и обработване на модели и техните визуални образи с помощта на компютър. Интерактивността в компютърната графика изисква визуализацията да става чрез устройства, които много бързо могат да променят образа. Най-широко използвани са графичните дисплеи с електронно-лъчева тръба. От тесния край на тръбата се излъчва сноп от електрони и той се насочва чрез специални магнитни системи. В другия на тръбата е екрана, който се състои от точки, които са достатъчно гъсто разположени, че за човешкото око да се създаде илюзия за непрекъснат образ. Тези точки се наричат **пиксели**. Един пиксел се формира от три зърна специално вещество, наречено **луминофор**, което има свойството да свети, когато снопът от електрони се удари в него. Трите зърна светят в червено (R), зелено (G) и синьо (B). Останалите цветове се получават чрез подходяща комбинация на тези три основни цвята. Тъй като светлинният импулс на луминофора намалява много бързо с времето, образът се нуждае от непрекъснато прерисуване, дори когато е статичен. За да се постигне илюзията на постоянен образ, честотата на опресняване трябва да е най-малко 60 Hz.
В зависимост от начина на регенерация на образа дисплеите се делят на два вида – **векторни** и **растерни**. Ние ще разглеждаме цветни растерни дисплеи. При тях във всеки един момент изображението представлява една правоъгълна матрица от пиксели. С всеки пиксел се свързват (обикновено) 24 бита – по 3 бита за всяка от основните цветови компоненти на пиксела. По този начин се осигуряват над 16000000 цветове. Основни характеристики на тези дисплеи са броят на пикселите (размерите на матрицата, например 800/600, 1024/768) и гъстотата на пикселите (броят на пикселите, които се събират в един инч, варира от 72 до 130).
Така при растерните дисплеи възниква необходимостта от създаването на ефективни алгоритми за растеризация на отсечки и криви линии, т.е. за преобразуване на криви и отсечки в растерен формат. Когато е дадена една отсечка (например, чрез двата си края) или крива (например, чрез уравнението си), за да се изчертае тя, трябва да се определи кои пиксели на дисплея да се оцветят, така че да се да се получи достатъчно близка апроксимация на отсечката или кривата.

Обикновено на всеки пиксел се съпоставя двойка координати – номера на стълба x и номера на реда y на пиксела. За начало на координатната система се счита горният ляв ъгъл, така че като се придвижваме надясно x нараства, а като се придвижваме надолу y нараства.

При всички алгоритми за растеризиране на отсечка, считаме, че отсечката е зададена с двата си края (X1, Y1) и (X2, Y2), където координатите са с цели числа. Целта на тези алгоритми е да се оцветят подходящи пиксели, така че да се създаде илюзията, че на екрана е начертана отсечката с начало в пиксела с координати X1 и Y1 и край в пиксела с координати X2 и Y2.

Първо разглеждаме **алгоритъма на Брезенхем** за растеризиране на отсечка. Първо изчисляваме наклонът на отсечката.
Той е m = dY/dX, където dY = Y2 – Y1, dX = X2 – X1 и уравнението на правата, определена от отсечката е y = m.x + b, където b = Y2 – m.X2. Ще считаме, че за наклона m е изпълнено -1 ≤ m ≤ 1. Ако това не е така, извършваме симетрия относно правата y = x и след това при изчертаване просто променяме ролите на двете координати.
Също така, ще считаме, че dX > 0. Ако това не е така, просто разменяме краищата на отсечката (при тази размяна наклонът m не се променя). Ще отбележим, че dX ≠ 0 – това е осигурено от първото съображение.
Така след тези две предварителни съображения, (X1, Y1) е левият край и (X2, Y2) е десният край на отсечката. Също, тъй като |m| ≤ 1, във всеки стълб между X1 и X2 (за всяка фиксирана първа координата между X1 и X2) трябва да се оцвети точно един пиксел. Първият оцветен пиксел е $(x_0, y_0) = (X1, Y1)$.
Нека (x_i, y_i) е i-тият оцветен пиксел. Отсечката пресича стълба $x_i + 1$ при $y = m.(x_i + 1) + b$. При това, i-тият оцветен пиксел ще е така избран, че $y_i \leq m.(x_i + 1) + b \leq y_i + 1$, ако dY е неотрицателно и $y_i - 1 \leq m.(x_i + 1) + b \leq y_i$, ако dY е отрицателно. Сега трябва да изберем (i+1)-ият пиксел за оцветяване. Ако dY е неотрицателно, той ще е един от двата пиксела $(x_i + 1, y_i)$ и $(x_1 + 1, y_i + 1)$. Ако dY е отрицателно, той ще е един от двата пиксела $(x_i + 1, y_i)$ и $(x_i + 1, y_i - 1)$. И в двата случая, кой точно пиксел да се избере се определя от това кой е по-близо до пресечната точка на отсечката със стълба $x_i + 1$.
При dY ≥ 0, разстоянието от $(x_i + 1, y_i)$ до $(x_i + 1, m.(x_i + 1) + b)$ е $t = m.(x_i + 1) + b - y_i$, а разстоянието от $(x_i + 1, y_i + 1)$ до $(x_i + 1, m.(x_i + 1) + b)$ е $s = 1 - t = y_i + 1 - m.(x_i + 1) - b$.
Така $t - s = m.(x_i + 1) + b - y_i - (y_i + 1 - m.(x_i + 1) - b) =$
$= 2.m.(x_i + 1) + 2.b - 2.y_i - 1$. Умножаваме полученото равенство с dX и получаваме $dX.(t - s) = 2.dY.(x_i + 1) + 2.dX.b - 2.dX.y_i - dX$.
Така знакът на t – s съвпада със знака на числото
$d_i = 2.dY.(x_i + 1) + 2.dX.b - 2.dX.y_i - dX$.

Ако $d_i > 0$, тогава избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i + 1)$.
Ако $d_i \leq 0$, тогава избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$.
При $dY < 0$, разсъжденията са аналогични: тогава
$d_i = - 2.dY.(x_i + 1) – 2.dX.b + 2.dX.y_i – dX$ и
ако $d_i > 0$, избираме $(x_{i+1}, y_{i+1}) = (x_{i+1}, y_{i+1}) = (x_i + 1, y_i – 1)$,
ако $d_i \leq 0$, тогава избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$.
За по-лесно изчисляване да съобразим следното:
$d_0 = 2.dY.(X1 + 1) + 2.dX.b – 2.dX.Y1 – dX = 2.dY – dX$, ако $dY \geq 0$,
$d_0 = - 2.dY – dX$, ако $dY < 0$. Двата случая можем да обединим така:
$d_0 = 2.abs\ (dY) – dX$.
След това, при $dY \geq 0$, $d_{i+1} = 2.dY.(x_{i+1} + 1) + 2.dX.b – 2.dX.y_{i+1} – dX$ ⇒
⇒ $d_{i+1} – d_i = 2.dY.(x_{i+1} – x_i) – 2.dX.(y_{i+1} – y_i)$.
При $dY < 0$, $d_{i+1} = - 2.dY.(x_{i+1} + 1) – 2.dX.b + 2.dX.y_{i+1} – dX$ ⇒
⇒ $d_{i+1} – d_i = - 2.dY.(x_{i+1} – x_i) + 2.dX.(y_{i+1} – y_i)$.
Двата случая можем да обединим така:
ако $d_i > 0$, $d_{i+1} = d_i + 2.abs\ (dY) – 2.dX$,
ако $d_i \leq 0$, $d_{i+1} = d_i + 2.abs\ (dY)$.

Така достигаме до следния алгоритъм:
Вход: (X1, Y1), (X2, Y2).
1. Ако abs (X2 – X1) < abs (Y2 – Y1), тогава
разменяме X1, Y1 и X2, Y2 и rev = 1, иначе rev = 0; към 2.
2. Ако X2 < X1, тогава разменяме X1, X2 и Y1, Y2; към 3.
3. dY = Y2 – Y1, dX = X2 – X1; към 4.
4. $d_0 = 2.abs\ (dY) – dX$, $x_0 = X1$, $y_0 = Y1$, i = 0; към 5.
5. Ако $dY \geq 0$, incY = 1, иначе incY = -1; към 6.
6. Ако rev = 0, чертаем (x_i, y_i), иначе чертаем (y_i, x_i); към 7.
7. Ако $x_i < X2$, към 8., иначе към 10.
8. Ако $d_i > 0$, тогава $d_{i+1} = d_i + 2.abs\ (dY) – 2.dX$, $x_{i+1} = x_i + 1$,
$y_{i+1} = y_i + incY$, иначе $d_{i+1} = d_i + 2.abs\ (dY)$, $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$; към 9.
9. i = i+1; към 6.
10. Край.

Сега ще разгледаме **алгоритъма на средната точка** за растеризиране на отсечка. Отново трябва да се направят двете съображения, както при алгоритъма на Брезенхем.
Така можем да считаме, че за наклона m имаме $– 1 \leq m \leq 1$ и освен това, че $dX > 0$. Правата, която е определена от точките (X1, Y1) и (X2, Y2) има уравнение $F (x, y) = dY.x – dX.y + C = 0$, където $C = dX.Y1 – dY.X1$. Тази права определя две полуравнини: за всички точки (x, y) над правата е изпълнено $F (x, y) < 0$ и за всички точки (x, y) под правата е изпълнено $F (x, y) > 0$.
Първият оцветен пиксел е $(x_0, y_0) = (X1, Y1)$.
Нека (x_i, y_i) е i-тият оцветен пиксел. Трябва да изберем (i+1)-ият пиксел за оцветяване. Както при Брезенхем, ако dY е неотрицателно, той ще е един от двата пиксела $(x_i + 1, y_i)$ и

$(x_1 + 1, y_i + 1)$. Ако dY е отрицателно, той ще е един от двата пиксела $(x_i + 1, y_i)$ и $(x_i + 1, y_i – 1)$. Тук обаче, изборът кой точно пиксел да се избере се определя по друг начин.
Ако dY 0, нека M_i е средата на отсечката $(x_i + 1, y_i)$ $(x_i + 1, y_i + 1)$, ако dY < 0, нека M_i е средата на отсечката $(x_i + 1, y_i)$ $(x_i + 1, y_i – 1)$. Изчисляваме $d_i = F(M_i)$ и определяме в коя полуравнина се намира точката M_i относно правата и по този начин определяме кой от двата пиксела да изберем (естествено, този който лежи в противоположната полуравнина).

При dY 0, $d_i = F(M_i) = dY.(x_i + 1) – dX.(y_i + \frac{1}{2}) + C$. Ако $d_i > 0$, избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i + 1)$, ако d_i 0 избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$.

При dY < 0, $d_i = F(M_i) = dY.(x_i + 1) – dX.(y_i - \frac{1}{2}) + C$. Ако d_i 0, избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$, ако $d_i < 0$ избираме $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i – 1)$.
Отново можем да съобразим някои неща за да изчисляваме по-лесно.

Ако dY 0, $d_0 = F(M_0) = dY.(X1 + 1) – dX.(Y1 + \frac{1}{2}) + C = dY - \frac{dX}{2}$,

$d_{i+1} = F(M_{i+1}) = dY.(x_{i+1} + 1) – dX.(y_{i+1} + \frac{1}{2}) + C =$

$= d_i + dY.(x_{i+1} – x_i) – dX.(y_{i+1} – y_i)$ при $d_i > 0$, $d_{i+1} = d_i + dY – dX$, при d_i 0, $d_{i+1} = d_i + dY$.

Ако dY < 0, $d_0 = F(M_0) = dY.(X1 + 1) – dX.(Y1 - \frac{1}{2}) + C = dY + \frac{dX}{2}$,

$d_{i+1} = F(M_{i+1}) = dY.(x_{i+1} + 1) – dX.(y_{i+1} - \frac{1}{2}) + C =$

$= d_i + dY.(x_{i+1} – x_i) – dX.(y_{i+1} – y_i)$ при d_i 0, $d_{i+1} = d_i + dY$, при $d_i < 0$, $d_{i+1} = d_i + dY + dX$.
Тук привидно имаме недостатък – резултатът от делението на 2 може да не е целочислено, но ние лесно можем да го поправим като разглеждаме уравнението $2.F(x, y) = 2.dY.x – 2.dX.y + 2.C = 0$ за правата.
Така достигаме до следния алгоритъм:
Вход: (X1, Y1), (X2, Y2).
1. Ако abs (X2 – X1) < abs (Y2 – Y1), тогава разменяме X1, Y1 и X2, Y2 и rev = 1, иначе rev = 0; към 2.
2. Ако X2 < X1, тогава разменяме X1, X2 и Y1, Y2; към 3.
3. dY = 2.(Y2 – Y1), dX = 2.(X2 – X1); към 4.
4. Ако dY 0, $d_0 = dY – dX/2$, иначе $d_0 = dY + dX/2$; към 5.
5. $x_0 = X1$, $y_0 = Y1$, i = 0; към 6.
6. Ако rev = 0, чертаем (x_i, y_i), иначе чертаем (y_i, x_i); към 7.
7. Ако $x_i < X2$, към 8., иначе към 12.
8. Ако dY 0, към 9., иначе към 10.
9. Ако $d_i > 0$, тогава $d_{i+1} = d_i + dY – dX$, $x_{i+1} = x_i + 1$,

$y_{i+1} = y_i + 1$, иначе $d_{i+1} = d_i + dY$, $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$; към 11.
10. Ако $d_i \geq 0$, тогава $d_{i+1} = d_i + dY$, $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$, иначе $d_{i+1} = d_i + dY + dX$, $x_{i+1} = x_i + 1$, $y_{i+1} = y_i - 1$; към 11.
11. i = i+1; към 6.
12. Край.

Третият метод, който ще разгледаме е **алгоритъмът на порциите** за растеризиране на отсечка. Отново искаме да са в сила предположенията, че $-1 \leq m \leq 1$ и $dX > 0$. Тук, обаче, добавяме и допълнително предположение (X1, Y1) = (0, 0). Това не е никакво ограничение, тъй като просто можем при изчертаването да правим подходяща транслация. Полагаме H = X2, H > 0, V = Y2. Тогава $m = \frac{V}{H}$. Във всеки фиксиран ред (при фиксирана втора координата) при растеризиране на отсечката трябва да се изчертава известна порция последователни пиксели. При направените предположения, първата порция трябва да е в реда 0 и започва от стълба 0, втората порция в реда 1 (-1, при V < 0) започва точно с един стълб по надясно от последния стълб в първата порция и т.н. Порциите се определят по следния начин: в реда y порцията е от всички x, такива че $y - \frac{1}{2} \leq \frac{V}{H}.x \leq y + \frac{1}{2}$. Оттук нататък предполагаме, че V > 0 – случаят V = 0 ще се разглежда отделно в алгоритъма, а случаят V < 0 лесно може да се сведе към V > 0, тъй като порциите и в двата случая са едни и същи.

Умножаваме последното неравенство с $\frac{H}{V}$ и получаваме:

$\frac{2.H}{2.V}.y - \frac{H}{2.V} \leq x \leq \frac{2.H}{2.V}.y + \frac{H}{2.V}$.

Сега нека $\frac{H}{2.V} = c_0 + \frac{r_0}{2.V}$, където $c_0, r_0 \in \mathbb{Z}$ и $0 \leq r_0 < 2.V$ и

$\frac{2.H}{2.V} = c_1 + \frac{r_1}{2.V}$, където $c_1, r_1 \in \mathbb{Z}$ и $0 \leq r_1 < 2.V$.

Тогава, $(c_1 + \frac{r_1}{2.V}).y - c_0 - \frac{r_0}{2.V} \leq x \leq (c_1 + \frac{r_1}{2.V}).y + c_0 + \frac{r_0}{2.V}$

$c_1.y - c_0 + \frac{r_1.y - r_0}{2.V} \leq x \leq c_1.y + c_0 + \frac{r_1}{2.V}.y + \frac{r_0}{2.V}$. Така първата порция при y = 0 е 0, 1, …, c_0, тъй като $\frac{r_0}{2.V} < 1$. Да положим $mod_0 = r_0$.

Очевидно е, че $0 = \frac{r_0 + r_1.0}{2.V} - \frac{mod_0}{2.V}$. Втората порция при y = 1 започва точно с един стълб по-надясно от края на първата порция, т.е. в $c_0 + 1$ и продължава до $c_0 + c_1 + \left\lfloor \frac{r_0 + r_1}{2.V} \right\rfloor$. При това,

$\left\lfloor \frac{r_0 + r_1}{2.V} \right\rfloor$ = 0 или 1, тъй като 0 $\leq r_0 + r_1 < 4.V$. Полагаме

$mod_1 = mod_0 + r_1$, ако $mod_0 + r_1 < 2.V$ (т.е. $\left\lfloor \frac{r_0 + r_1}{2.V} \right\rfloor = 0$) или

$mod_1 = mod_0 + r_1 - 2.V$, ако $mod_0 + r_1 \geq 2.V$ (т.е. $\left\lfloor \frac{r_0 + r_1}{2.V} \right\rfloor = 1$).

Тогава е ясно, че $\left\lfloor \frac{r_0 + r_1}{2.V} \right\rfloor = \left\lfloor \frac{mod_0 + r_1}{2.V} \right\rfloor = \frac{r_0 + r_1.1}{2.V} - \frac{mod_1}{2.V}$.

По-общо, нека порцията в реда y завършва в c. Имаме

$c = c_1.y + c_0 + \left\lfloor \frac{r_0 + r_1.y}{2.V} \right\rfloor = c_1.y + c_0 + \frac{r_0 + r_1.y}{2.V} - \frac{mod_y}{2.V}$.

Тогава порцията в реда y+1 започва в c+1 и завършва в

$\left\lfloor c_1.(y+1) + c_0 + \frac{r_1}{2.V}.(y+1) + \frac{r_0}{2.V} \right\rfloor =$

$\left\lfloor c_1.y + c_0 + c_1 + \frac{r_1}{2.V}.y + \frac{r_1}{2.V} + \frac{r_0}{2.V} \right\rfloor - \left\lfloor c_1.y + c_0 + \frac{r_1}{2.V}.y + \frac{r_1}{2.V} + \frac{r_0}{2.V} \right\rfloor + c_1 =$

$\left\lfloor \frac{mod_y}{2.V} \right\rfloor - \left\lfloor \frac{r_1}{2.V} \right\rfloor + c + c_1$.

Първото събираемо е равно или на 0 или на 1, тъй като
$mod_y + r_1 < 4.V$. Ако $mod_y + r_1 < 2.V$, тогава порцията в реда y + 1 завършва в c + c_1 и тогава полагаме $mod_{y+1} = mod_y + r_1$.
Ако $mod_y + r_1 \geq 2.V$, тогава порцията в реда y+1 завършва в 1 + c + c_1 и тогава полагаме $mod_{y+1} = mod_y + r_1 - 2.V$.
Естествено, полагането е такова, че

$\left\lfloor \frac{mod_y + r_1.y}{2.V} \right\rfloor = \frac{mod_y + r_1.y}{2.V} - \frac{mod_{y+1}}{2.V} \quad \left\lfloor \frac{r_0 + r_1.y}{2.V} \right\rfloor - \left\lfloor \frac{r_0 + r_1.y}{2.V} \right\rfloor - \left\lfloor \frac{r_1.y}{2.V} \right\rfloor =$

$= \frac{r_0 + r_1.y}{2.V} - \left\lfloor \frac{r_0 + r_1.y}{2.V} \right\rfloor + \frac{r_1.y}{2.V} - \frac{mod_{y+1}}{2.V} \quad$

$\left\lfloor \frac{r_0 + r_1.(y+1)}{2.V} \right\rfloor = \frac{r_0 + r_1.(y+1)}{2.V} - \frac{mod_{y+1}}{2.V}$.

Този процес продължава естествено до момента, в който края на някоя от порциите достигне (или надмине) H.
Така достигаме до следния алгоритъм:
Вход: (X1, Y1), (X2, Y2).
1. Ако abs (X2 – X1) < abs (Y2 – Y1), тогава
разменяме X1, Y1 и X2, Y2 и rev = 1, иначе rev = 0; към 2.
2. Ако X2 < X1, тогава разменяме X1, X2 и Y1, Y2; към 3.
3. H = X2 – X1, V = abs (Y2 – Y1); към 4.
4. Ако Y2 > Y1, incY = 1, иначе incY = -1; към 5.
5. y = 0, x = 0, ако V = 0 към 6., иначе към 7.
6. c = H; към 8.
7. r_1 = (H + H) % (V + V), c_1 = (H + H) / (V + V), mod_0 = H % (V + V), c = H / (V + V); към 8.
8. Ако c < H, към 9., иначе към 10.

9. Ако rev = 1 изчертаване на пикселите (Y1 + y, X1 + x), (Y1 + y, X1 + x +1), …, (Y1 + y, X1 + c), иначе изчертаване на пикселите (X1 + x, Y1 + y), (X1 + x + 1, Y1 + y), …, (X1 + c, Y1 + y); към 11.
10. Ако rev = 1 изчертаване на пикселите (Y1 + y, X1 + x), (Y1 + y, X1 + x +1), …, (Y1 + y, X1 + H), иначе изчертаване на пикселите (X1 + x, Y1 + y), (X1 + x + 1, Y1 + y), …, (X1 + H, Y1 + y); към 14.
11. x = c + 1; към 12.
12. Ако $mod_y + r_1 \square V + V$, $mod_{y+1} = (mod_y + r_1) – (V + V)$, $c = c + c_1 + 1$, иначе $mod_{y+1} = mod_y + r_1$, $c = c + c_1$; към 13.
13. y = y + incY; към 8.
14. Край.

По-нататък ще разгледаме алгоритми за растеризиране на окръжност. Навсякъде ще предполагаме, че окръжността има център (0, 0) и радиус цяло неотрицателно число R. Естествено, чрез подходяща транслация могат да се растеризират окръжности с произволен център. Целта на алгоритмите е да се оцветят по такъв начин пикселите, че да се създаде илюзията, че на екрана окръжността с център в пиксела (0, 0) и радиус R.
Естествено, при тези предположения, окръжността има уравнение $F(x, y) = x^2 + y^2 – R^2 = 0$. Тя разделя равнината на три части – вътрешността на кръга (тези с $F(x, y) < 0$), самата окръжност (контура на кръга, тези с $F(x, y) = 0$) и точките, извън кръга (тези с $F(x, y) > 0$).

Първо разглеждаме **алгоритъмът на Брезенхем** за растеризиране на окръжност. При алгоритъма се растеризира само дъгата от окръжността, която се намира в първи квадрант и се използва, че окръжността е симетрична спрямо координатните оси и спрямо центъра. Първият пиксел, който оцветяваме е $(x_0, y_0) = (0, R)$.
Нека е изчертан i-тият пиксел (x_i, y_i).
Да означим $H_i = (x_i + 1, y_i)$, $D_i = (x_i + 1, y_i – 1)$, $V_i = (x_i, y_i – 1)$.
Следващият пиксел за чертане ще е един от H_i, D_i, V_i. За да определим точно кой изчисляваме
$d_i = F(D_i) = F(x_i + 1, y_i – 1) = (x_i + 1)^2 + (y_i – 1)^2 – R^2$.
Ако $d_i \square 0$ окръжността минава под D_i, но не и под V_i и затова ще избираме един от двата пиксела D_i и V_i. За целта изчисляваме $\rightarrow_i = F(D_i) + F(V_i)$. Тъй като $F(D_i) \square 0$, то $F(D_i)$ е най-близкото разстояние от D_i до окръжността. Тъй като $F(V_i) \square 0$, то $F(V_i)$ е най-близкото разстояние от V_i до окръжността, но взето с обратен знак. Така, ако $\rightarrow_i \square 0$, трябва да изберем V_i, а когато $\rightarrow_i < 0$ трябва да изберем D_i. Имаме $\rightarrow_i = F(D_i) + F(V_i) = d_i + x_i^2 + (y_i – 1)^2 – R^2 =$
$= 2.d_i + x_i^2 – (x_i + 1)^2 = 2.d_i – 2.x_i – 1$. Така $\rightarrow_i \square 0 \square 2.d_i – 2.x_i – 1 \square 0 \square$
$d_i \square x_i + \frac{1}{2} \square d_i > x_i$, тъй като x_i и d_i са цели числа. Също,

$\Delta_i < 0 \Leftrightarrow d_i < x_i + \frac{1}{2} \Leftrightarrow d_i \leq x_i$, тъй като x_i и d_i са цели числа.
Така в случая $0 \leq d_i \leq x_i$ избираме D_i, а в случая $d_i > x_i$ избираме V_i.
Вторият случай е $d_i < 0$ и тогава окръжността минава над D_i, но не и над H_i и затова ще избираме един от двата пиксела D_i и H_i. За целта изчисляваме $\Delta_i = F(D_i) + F(H_i)$. Тъй като $F(D_i) < 0$, то $F(D_i)$ е най-близкото разстояние от D_i до окръжността, взето с обратен знак. Тъй като $F(H_i) \geq 0$, то $F(H_i)$ е най-близкото разстояние от H_i до окръжността. Така, ако $\Delta_i \geq 0$, трябва да изберем D_i, а когато $\Delta_i < 0$ трябва да изберем H_i. Имаме $\Delta_i = F(D_i) + F(H_i) =$
$= d_i + (x_i + 1)^2 + y_i^2 - R^2 = 2.d_i + y_i^2 - (y_i - 1)^2 = 2.d_i + 2.y_i - 1.$
Така $\Delta_i \geq 0 \Leftrightarrow 2.d_i + 2.y_i - 1 \geq 0 \Leftrightarrow d_i \geq - y_i + \frac{1}{2} \Leftrightarrow d_i > - y_i$, тъй като y_i и d_i са цели числа. Също, $\Delta_i < 0 \Leftrightarrow d_i < - y_i + \frac{1}{2} \Leftrightarrow d_i \leq - y_i$, тъй като y_i и d_i са цели числа. Така в случая $- y_i < d_i < 0$ избираме D_i, а в случая $d_i \leq - y_i$ избираме H_i.
Като комбинираме двата случая получаваме:
ако $d_i \leq - y_i$ избираме $(x_{i+1}, y_{i+1}) = H_i$, ако $- y_i < d_i \leq x_i$ избираме $(x_{i+1}, y_{i+1}) = D_i$, ако $x_i < d_i$ избираме $(x_{i+1}, y_{i+1}) = V_i$.

За по-ефективно изчисление да съобразим, че
$d_0 = F(d_0) = F(x_0 + 1, y_0 - 1) = F(1, R - 1) = 1^2 + (R - 1)^2 - R^2 =$
$= 2 - 2.R.$
Освен това, $d_{i+1} = F(D_{i+1}) = F(x_{i+1} + 1, y_{i+1} - 1) =$
$= (x_{i+1} + 1)^2 + (y_{i+1} - 1)^2 - R^2 = d_i - (x_i + 1)^2 - (y_i - 1)^2 + (x_{i+1} + 1)^2 +$
$+ (y_{i+1} - 1)^2 = d_i + (x_{i+1} - x_i).(x_{i+1} + x_i + 2) + (y_{i+1} - y_i).(y_{i+1} + y_i - 2).$
Така, при $(x_{i+1}, y_{i+1}) = H_i = (x_i + 1, y_i)$ имаме
$d_{i+1} = d_i + 2.x_i + 3$, при $(x_{i+1}, y_{i+1}) = D_i = (x_i + 1, y_i - 1)$ имаме
$d_{i+1} = d_i + 2.x_i + 3 - (y_i - 1 + y_i - 2) = d_i + 2.x_i - 2.y_i + 6,$
при $(x_{i+1}, y_{i+1}) = V_i = (x_i, y_i - 1)$ имаме
$d_{i+1} = d_i - (y_i - 1 + y_i - 2) = d_i - 2.y_i + 3.$

Така достигаме до следния алгоритъм:
Вход: R.
1. $x_0 = 0$, $y_0 = R$, $d_0 = 2 - 2.R$, $i = 0$; към 2.
2. Изчертаване на пиксела (x_i, y_i) и трите му симетрични точки относно координатните оси и относно центъра; към 3.
3. Ако $y_i = 0$ към 7., иначе към 4.
4. Ако $d_i > - y_i$, $d_{i+1} = d_i - 2.y_i + 3$, $y_{i+1} = y_i - 1$, иначе $y_{i+1} = y_i$, $d_{i+1} = d_i$; към 5.
5. Ако $d_i \leq x_i$, $d_{i+1} = d_{i+1} + 2.x_i + 3$, $x_{i+1} = x_i + 1$, иначе $x_{i+1} = x_i$; към 6.
6. $i = i + 1$, към 2.
7. Край.

Сега ще разгледаме **алгоритъма на Михенер** за растеризиране на окръжност. При този алгоритъм се растеризира само горния октант в първи квадрант и се използва, че окръжността е

симетрична относно координатните оси, центъра и ъглополовящите на квадрантите.
Първият пиксел, който оцветяваме е $(x_0, y_0) = (0, R)$.
Нека сме оцветили i-тият пиксел (x_i, y_i).
Отново да означим $H_i = (x_i + 1, y_i)$, $D_i = (x_i + 1, y_i - 1)$.
Тогава (i+1)-ият пиксел ще е един от H_i, D_i. С други думи, в означенията от алгоритъма на Брезенхем, при изчертаване на въпросния октант никога няма да се прави избор на V_i.
За да определим кой от двата пиксела да оцветим, изчисляваме $d_i = F(H_i) + F(D_i)$. Тъй като се намираме във въпросния октант, винаги имаме $F(H_i) \geq 0$ и $F(D_i) < 0$. Така $F(H_i)$ е най-близкото разстояние от H_i до окръжността и $F(D_i)$ е най-близкото разстояние от D_i до окръжността, но взето с обратен знак.
И следователно при $d_i \geq 0$ трябва да вземем $(x_{i+1}, y_{i+1}) = D_i$, а при $d_i < 0$ трябва да вземем $(x_{i+1}, y_{i+1}) = H_i$.
За по-ефективно изчисление да съобразим, че
$d_0 = F(H_0) + F(D_0) = F(1, R) + F(1, R - 1) = 1^2 + R^2 - R^2 + 1^2 +$
$+ (R - 1)^2 - R^2 = 3 - 2.R.$
Освен това, $d_{i+1} = F(H_{i+1}) + F(D_{i+1}) = (x_{i+1} + 1)^2 + y_{i+1}^2 - R^2 +$
$+ (x_{i+1} + 1)^2 + (y_{i+1} - 1)^2 - R^2 = d_i + (x_{i+1} + 1)^2 + y_{i+1}^2 - (x_i + 1)^2 - y_i^2 +$
$+ (x_{i+1} + 1)^2 + (y_{i+1} - 1)^2 - (x_i + 1)^2 - (y_i - 1)^2 = d_i +$
$+ 2.(x_{i+1} - x_i).(x_{i+1} + x_i + 2) + (y_{i+1} - y_i).(y_{i+1} + y_i) + (y_{i+1} - y_i).(y_{i+1} + y_i - 2) =$
$= d_i + 2.(x_{i+1} - x_i).(x_{i+1} + x_i + 2) + 2.(y_{i+1} - y_i).(y_{i+1} + y_i - 1).$
Така, при $(x_{i+1}, y_{i+1}) = H_i = (x_i + 1, y_i)$, $d_{i+1} = d_i + 4.x_i + 6$, а при $(x_{i+1}, y_{i+1}) = D_i = (x_i + 1, y_i - 1)$, $d_{i+1} = d_i + 4.x_i + 6 - 2.(2.y_i - 2) =$
$= d_i + 4.x_i - 4.y_i + 10.$

Така достигаме следния алгоритъм:
Вход: R.
1. $x_0 = 0$, $y_0 = R$, $d_0 = 3 - 2.R$, $i = 0$; към 2.
2. Изчертаване на пикселите (x_i, y_i) и (y_i, x_i) и шестте им симетрични точки относно двете координатни оси и центъра на координатната система; към 3.
3. Ако $x_i \geq y_i$ към 6., иначе към 4.
4. Ако $d_i \geq 0$, $d_{i+1} = d_i + 4.x_i - 4.y_i + 10$, $y_{i+1} = y_i - 1$, иначе $d_{i+1} = d_i + 4.x_i + 6$, $y_{i+1} = y_i$; към 5.
5. $x_{i+1} = x_i + 1$, $i = i + 1$, към 2.
6. Край.

Следващият алгоритъм за растеризиране на окръжност е **алгоритъма на средната точка**. При този алгоритъм също се растеризира само горния октант в първи квадрант.
Първият пиксел, който оцветяваме е $(x_0, y_0) = (0, R)$.
Нека сме оцветили i-тият пиксел (x_i, y_i).
Отново да означим $H_i = (x_i + 1, y_i)$, $D_i = (x_i + 1, y_i - 1)$.
Както при алгоритъма на Михенер, тъй като изчертаваме въпросният октант, (i+1)-ият пиксел ще е един от H_i, D_i (няма да се прави избор на V_i). За да определим кой от двата пиксела ще се

оцвети, въвеждаме $M_i = (x_i + 1, y_i - \frac{1}{2})$ – средата на H_iD_i и

изчисляваме $d_i = F(M_i) = F(x_i + 1, y_i - \frac{1}{2}) = (x_i + 1)^2 - (y_i - \frac{1}{2})^2 - R^2$.

Ако $d_i \geq 0$, т.е. окръжността минава под M_i, тогава е ясно, че трябва да изберем D_i, ако $d_i < 0$, т.е. окръжността минава над M_i, то е ясно, че трябва да изберем H_i.

За да улесним изчисленията забелязваме следното:

$d_{i+1} = F(M_{i+1}) = (x_{i+1} + 1)^2 + (y_{i+1} - \frac{1}{2})^2 - R^2 = d_i + (x_{i+1} + 1)^2 - (x_i + 1)^2 +$

$+ (y_{i+1} - \frac{1}{2})^2 - (y_i - \frac{1}{2})^2 = d_i + (x_{i+1} - x_i).(x_{i+1} + x_i + 2) +$

$+ (y_{i+1} - y_i).(y_{i+1} + y_i - 1)$. Така, ако $(x_{i+1}, y_{i+1}) = D_i = (x_i + 1, y_i - 1)$,
$d_{i+1} = d_i + 2.x_i + 3 - (2.y_i - 2) = d_i + 2.x_i - 2.y_i + 5$, а ако
$(x_{i+1}, y_{i+1}) = H_i = (x_i + 1, y_i)$, $d_{i+1} = d_i + 2.x_i + 3$. Освен това,

$d_0 = F(M_0) = F(1, R - \frac{1}{2}) = 1^2 + (R - \frac{1}{2})^2 - R^2 = \frac{5}{4} - R$.

Тук достигаме до недостатък, тъй като $\frac{5}{4}$ не е цяло число.

Това, обаче, лесно може да се преодолее по следния начин:

полагаме $e_i = d_i - \frac{1}{4}$ за всяко i. Естествено, горните зависимости

между d_{i+1} и d_i автоматично се пренасят и за e_{i+1} и e_i, също
$e_0 = 1 - R$. Оттук се вижда, че за всяко i, e_i е цяло число $\Leftrightarrow$
$e_i \geq 0 \Leftrightarrow d_i \geq 0$, $e_i < 0 \Leftrightarrow d_i < 0$. Така при определянето на това кой пиксел да се избере можем да използваме знака на e_i вместо знака на d_i.

Така достигаме до следния алгоритъм:
Вход : R.
1. $x_0 = 0$, $y_0 = R$, $e_0 = 1 - R$, $i = 0$; към 2.
2. Изчертаване на пикселите (x_i, y_i) и (y_i, x_i) и симетрични им точки относно двете координатни оси и центъра на координатната система; към 3.
3. Ако $x_i \geq y_i$ към 6., иначе към 4.
4. Ако $e_i \geq 0$, $e_{i+1} = e_i + 2.x_i - 2.y_i + 5$, $y_{i+1} = y_i - 1$, иначе
$e_{i+1} = e_i + 2.x_i + 3$, $y_{i+1} = y_i$; към 5.
5. $x_{i+1} = x_i + 1$, $i = i + 1$, към 2.
6. Край.

Последният алгоритъм за растеризиране на окръжност, който ще разгледаме е **алгоритъма на вторите крайни разлики**. Отново окръжността растеризираме само в горния октант на първи квадрант и използваме нейната богата симетрия.

Първият оцветен пиксел е $(x_0, y_0) = (0, R)$. Нека сме оцветили i-тият пиксел (x_i, y_i). Тогава, както в алгоритъма на средната точка, за (i+1)-ви пиксел избираме D_i или H_i, в зависимост от знака на $d_i = F(M_i) = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - R^2$. Хитрото тук е начинът на изчисляване на d_i. Дефинираме d_i^H като изменението на d_i, ако на i-тата стъпка е избран пиксела H_i, т.е. $d_i^H = d_{i+1} - d_i$, ако $(x_{i+1}, y_{i+1}) = H_i$. Дефинираме d_i^D като изменението на d_i, ако на i-тата стъпка е избран пиксела D_i, т.е. $d_i^D = d_{i+1} - d_i$, ако $(x_{i+1}, y_{i+1}) = D_i$.
Ако на i-тата стъпка сме избрали H_i, то от по-горе $d_{i+1} = d_i + 2.x_i + 3 \Rightarrow d_i^H = 2.x_i + 3$. Ако на i-тата стъпка сме избрали D_i, то от по-горе $d_{i+1} = d_i + 2.x_i - 2.y_i + 5 \Rightarrow d_i^D = 2.x_i - 2.y_i + 5$.
Нека на i-тата стъпка е избрана H_i, т.е. $(x_{i+1}, y_{i+1}) = H_i$.
Тогава $d_{i+1}^H = d_{i+2} - d_{i+1}$, ако на (i+1)-та стъпка е избрана H_{i+1}.
Така $d_{i+1}^H = F(M_{i+2}) - F(M_{i+1}) = (x_i + 3)^2 + (y_i - \frac{1}{2})^2 - R^2 - (x_i + 2)^2 - (y_i - \frac{1}{2})^2 + R^2 = 2.x_i + 5 = d_i^H + 2$.
Също, $d_{i+1}^D = d_{i+2} - d_{i+1}$, ако на (i+1)-та стъпка е избрана D_{i+1}.
Така $d_{i+1}^D = F(M_{i+2}) - F(M_{i+1}) = (x_i + 3)^2 + (y_i - \frac{3}{2})^2 - R^2 - (x_i + 2)^2 - (y_i - \frac{1}{2})^2 + R^2 = 2.x_i + 5 - (2.y_i - 2) = 2.x_i - 2.y_i + 7 = d_i^D + 2$.
Нека на i-тата стъпка е избрана D_i, т.е. $(x_{i+1}, y_{+1}) = D_i$.
Тогава $d_{i+1}^H = d_{i+2} - d_{i+1}$, ако на (i+1)-та стъпка е избрана H_{i+1}.
Така $d_{i+1}^H = F(M_{i+2}) - F(M_{i+1}) = (x_i + 3)^2 + (y_i - \frac{3}{2})^2 - R^2 - (x_i + 2)^2 - (y_i - \frac{3}{2})^2 + R^2 = 2.x_i + 5 = d_i^H + 2$.
Също, $d_{i+1}^D = d_{i+2} - d_{i+1}$, ако на (i+1)-та стъпка е избрана D_{i+1}.
Така $d_{i+1}^D = F(M_{i+2}) - F(M_{i+1}) = (x_i + 3)^2 + (y_i - \frac{5}{2})^2 - R^2 - (x_i + 2)^2 - (y_i - \frac{3}{2})^2 + R^2 = 2.x_i + 5 - (2.y_i - 4) = 2.x_i - 2.y_i + 9 = d_i^D + 4$.
Освен това, $d_0^H = d_1 - d_0$, ако на 0-та стъпка е избрано H_i, т.е.
$d_0^H = F(M_1) - F(M_0) = 2^2 + (R - \frac{1}{2})^2 - R^2 - 1^2 - (R - \frac{1}{2})^2 + R^2 = 3$.
Също, $d_0^D = d_1 - d_0$, ако на 0-та стъпка е избрано D_i, т.е.
$d_0^D = F(M_1) - F(M_0) = 2^2 + (R - \frac{3}{2})^2 - R^2 - 1^2 - (R - \frac{1}{2})^2 + R^2 =$
$= 3 - (2.R - 2) = 5 - 2.R$.
Имаме $d_0 = \frac{5}{4} - R$ и затова отново въвеждаме $e_i = d_i - \frac{1}{4}$ за всяко i.
Тогава $e_0 = 1 - R$ и освен това в процеса на изчисление e_i се изменят по същия начин, както d_i – ако заменим навсякъде по-горе d_i с e_i ще получим коректни резултати. Но e_i са цели □

$e_i \geq 0 \Rightarrow d_i \geq 0$, $e_i < 0 \Rightarrow d_i < 0$ и следователно можем да използваме знакът на e_i за да определяме кой пиксел да изберем на i-тата стъпка. Сега е ясно следното: ако на i-тата стъпка сме избрали H_i, т.е. $(x_{i+1}, y_{i+1}) = H_i \Rightarrow e_{i+1} = e_i + d_i^H$, а ако на i-тата стъпка сме избрали D_i, т.е. $(x_{i+1}, y_{i+1}) = D_i \Rightarrow e_{i+1} = e_i + d_i^D$. Така като знаем e_0 и d_i^H, d_i^D за всяко i ние можем да изчисляваме e_i за всяко i.

Така стигаме до следния алгоритъм:
Вход: R.
1. $e_0 = 1 - R$, $d_0^H = 3$, $d_0^D = 5 - 2.R$, $x_0 = 0$, $y_0 = R$, $i = 0$; към 2.
2. Изчертаване на пикселите (x_i, y_i) и (y_i, x_i) и шестте им симетрични точки относно двете координатни оси и центъра на координатната система; към 3.
3. Ако $x_i \geq y_i$ към 6., иначе към 4.
4. Ако $e_i \geq 0$, $e_{i+1} = e_i + d_i^D$, $d_{i+1}^H = d_i^H + 2$, $d_{i+1}^D = d_i^D + 4$, $y_{i+1} = y_i - 1$, иначе $e_{i+1} = e_i + d_i^H$, $d_{i+1}^H = d_i^H + 2$, $d_{i+1}^D = d_i^D + 2$, $y_{i+1} = y_i$; към 5.
5. $x_{i+1} = x_i + 1$, $i = i + 1$, към 2.
6. Край.

Всички разгледани алгоритми могат да се използват за растеризиране на дъга от окръжност по следната схема:
да предположим, че трябва да се изчертае дъгата на окръжността $F(x, y) = x^2 + y^2 - R^2 = 0$, от ъгъл $\mapsto$до ъгъл $\updownarrow$(спрямо абсцисата).
Тогава трябва да действаме по следната схема: първо изчисляваме $StartX = R.\cos\mapsto$, $StartY = R.\sin\mapsto$, $EndX = R.\cos\updownarrow$, $EndY = R.\sin\updownarrow$
и след това чертаем цялата окръжност по някой от алгоритмите, като оцветяваме само тези пиксели, чийто номер на стълб е между StartX и EndX и чийто номер на ред е между StartY и EndY.
Естествено, тук прибягваме до изчисляване на тригонометричните функции, но няма как.

Накрая ще разгледаме едно обобщение на алгоритъма на средната точка, което ни позволява да разстеризираме елипса.
Предполагаме, че елипсата има главни направления, успоредни на координатните оси и, че нейният център е в началото. Това не е голямо ограничение, тъй като с подходяща ротация и транслация всяка елипса може да се изобрази в този вид. Така елипсата има уравнение $F(x, y) = \frac{x^2}{a^2} + \frac{y^2}{b^2} - 1 = 0$, $a > 0$, $b > 0$. Еквивалентен запис е $F(x, y) = b^2.x^2 + a^2.y^2 - a^2.b^2 = 0$. Елипсата ще растеризираме в първи квадрант и ще използваме, че тя е симетрична относно координатните оси. Първият оцветен пиксел е $(x_0, y_0) = (0, b)$.
Нека сме оцветили i-тият пиксел (x_i, y_i). Тогава (i+1)-ят пиксел се избира между пикселите $H_i = (x_i + 1, y_i)$, $D_i = (x_i + 1, y_i - 1)$ и $V_i = (x_i, y_i - 1)$. При това растеризацията се разделя на два етапа: през първия етап градиента на елипсата в изчертаваната точка сключва ъгъл по-голям от 45 градуса с абсцисната ос, а през втория етап градиента на елипсата в изчертаваната точка

сключва ъгъл по-малък от 45 градуса с абсцисната ос. Ясно е, че през първия етап няма да има избор на V_i, а през втория етап няма да има избор на H_i. Границата между двата етапа е първият момента, в който ъгълът между градиента на елипсата в изчертаваната точка и абсцисната ос стане по-малък или равен на 45 градуса (в началото, естествено, този ъгъл е 90 градуса).

За градиента имаме $\text{grad } F(x_i, y_i) = (\frac{\partial F}{\partial x}(x_i, y_i), \frac{\partial F}{\partial y}(x_i, y_i)) =$

$= (2.b^2.x_i, 2.a^2.y_i)$. Тогава първият етап се определя от неравенството $b^2.x_i < a^2.y_i$. В първия момент, в който $b^2.x_i \geq a^2.y_i$ трябва да започне вторият етап.

Нека сме в първия етап. За да определим кой пиксел ще се оцветява – H_i или D_i, полагаме $M_i = (x_i + 1, y_i - \frac{1}{2})$ и изчисляваме

$d_i = F(M_i) = b^2.(x_i + 1)^2 + a^2.(y_i - \frac{1}{2})^2 - a^2.b^2$. Ако $d_i \geq 0$, т.е. M_i не попада във вътрешността на елипсата, то естествено трябва да изберем $(x_{i+1}, y_{i+1}) = D_i$, ако $d_i < 0$, т.е. M_i е във вътрешността на елипсата, трябва да изберем H_i. Ако сме избрали D_i, тогава

$d_{i+1} = F(M_{i+1}) = b^2.(x_i + 2)^2 + a^2.(y_i - \frac{3}{2}) - a^2.b^2 = d_i + b^2.(x_i + 2)^2 +$

$+ a^2.(y_i - \frac{3}{2}) - b^2.(x_i + 1)^2 - a^2.(y_i - \frac{1}{2}) = d_i + b^2.(2.x_i + 3) - a^2.(2.y_i - 2) =$

$= d_i + 2.b^2.x_i - 2.a^2.y_i + 3.b^2 + 2.a^2$. Ако сме избрали H_i, тогава

$d_{i+1} = F(M_{i+1}) = b^2.(x_i + 2)^2 + a^2.(y_i - \frac{1}{2}) - a^2.b^2 = d_i + b^2.(x_i + 2)^2 +$

$+ a^2.(y_i - \frac{1}{2}) - b^2.(x_i + 1)^2 - a^2.(y_i - \frac{1}{2}) = d_i + b^2.(2.x_i + 3) =$

$= d_i + 2.b^2.x_i + 3.b^2$.

Нека сме във втория етап. За да определим кой пиксел ще се оцветява – V_i или D_i, полагаме $M_i = (x_i + \frac{1}{2}, y_i - 1)$ и изчисляваме

$d_i = F(M_i) = b^2.(x_i + \frac{1}{2})^2 + a^2.(y_i - 1)^2 - a^2.b^2$. Ако $d_i \geq 0$, т.е. M_i не попада във вътрешността на елипсата, то естествено трябва да изберем $(x_{i+1}, y_{i+1}) = V_i$, ако $d_i < 0$, т.е. M_i е във вътрешността на елипсата, трябва да изберем D_i. Ако сме избрали V_i, тогава

$d_{i+1} = F(M_{i+1}) = b^2.(x_i + \frac{1}{2})^2 + a^2.(y_i - 2) - a^2.b^2 = d_i + b^2.(x_i + \frac{1}{2})^2 +$

$+ a^2.(y_i - 2) - b^2.(x_i + \frac{1}{2})^2 - a^2.(y_i - 1) = d_i - a^2.(2.y_i - 3) =$

$= d_i - 2.a^2.y_i + 3.a^2$. Ако сме избрали D_i, тогава

$d_{i+1} = F(M_{i+1}) = b^2.(x_i + \frac{3}{2})^2 + a^2.(y_i - 2) - a^2.b^2 = d_i + b^2.(x_i + \frac{3}{2})^2 +$

$+ a^2.(y_i - 2) - b^2.(x_i + \frac{1}{2})^2 - a^2.(y_i - 1) = d_i + b^2.(2.x_i + 2) - a^2.(2.y_i - 3) =$

$= d_i + 2.b^2.x_i - 2.a^2.y_i + 2.b^2 + 3.a^2$.

Освен това, $d_0 = F(M_0) = b^2.1^2 + a^2.(b - \frac{1}{2})^2 - a^2.b^2 = b^2 + a^2.b^2 - b.a^2 + \frac{1}{4}.a^2 - a^2.b^2 = b^2 - b.a^2 + \frac{1}{4}.a^2$. Също, при прехода между етапите, естествено, трябва в началото да изчислим d_i директно.

Тук привидно има недостатък, тъй като $\frac{1}{4}$ не е цяло число. Това лесно може да се избегне, ако вместо уравнението $F(x, y) = 0$ за елипсата използваме еквивалентното уравнение $4.F(x, y) = 0$.

Така достигаме до следния алгоритъм:
Вход: a, b.
1. $x_0 = 0$, $y_0 = b$, $d_0 = 4.b^2 - 4.b.a^2 + a^2$, $i = 0$; към 2.
2. Изчертаваме пикселът (x_i, y_i) и трите му симетрични относно координатните оси и центъра на координатната система; към 3.
3. Ако $a^2.y_i \square b^2.x_i$ към 6., иначе към 4.
4. Ако $d_i \square 0$, $d_{i+1} = d_i + 8.b^2.x_i - 8.a^2.y_i + 12.b^2 + 8.a^2$, $y_{i+1} = y_i - 1$, иначе $d_{i+1} = d_i + 8.b^2.x_i + 12.b^2$, $y_{i+1} = y_i$; към 5.
5. $x_{i+1} = x_i + 1$, $i = i + 1$; към 2.
6. $d_i = b^2.(2.x_i + 1)^2 + a^2.(2.y_i - 2)^2 - 4.a^2.b^2$; към 7.
7. Ако $d_i \square 0$, $d_{i+1} = d_i - 8.a^2.y_i + 12.a^2$, $x_{i+1} = x_i$, иначе $d_{i+1} = d_i + 8.b^2.x_i - 8.a^2.y_i + 8.b^2 + 12.a^2$, $x_{i+1} = x_i + 1$; към 8.
8. $y_{i+1} = y_i - 1$, $i = i + 1$; към 9.
9. Изчертаваме пикселът (x_i, y_i) и трите му симетрични относно координатните оси и центъра на координатната система; към 10.
10. Ако $y > 0$ към 7., иначе към 11.
11. Край.

10. Процедурно програмиране – основни информационни и алгоритмични структури (C++).

Основните принципи на структурното програмиране са принципа за **модулност** и принципа за **абстракция на данните**.
Съгласно принципа за модулност, програмата се разделя на подходящи взаимосвързани части, всяка от които се реализира чрез определени средства. Целта е промените в представянето на данните да не променят голям брой от модулите на програмата.
Функцията е самостоятелен фрагмент на програмата – отделна програмна единица, съдържаща описание на променливи и набор от оператори на езика. Те се затварят между фигурни скоби и се наричат тяло на функцията. Функцията има възможност да предава и получава информация към и от други функции. За да се предаде в извиканата функция стойност на една променлива, дефинирана в извикващата функция, е необходимо тази променлива да се включи в списъка на предаваните стойности. Този списък се нарича списък на аргументите. Обикновено след изпълнение на дадена функция в извикващата функция се

връщат резултатите от някакви изчисления. Тези резултати имат определен тип. Възможно е една функция да не връща резултат и тогава тя се нарича **процедура**.
Във функционално отношение функцията е част от програмата, с изпълнението на която се получават определени резултати. Чрез използването на функции една програма може да се раздели на отделни модули. Предимствата на модулния подход са следните:

- модулите могат да се програмират независимо един от друг;
- модулите могат много лесно да се тестват за грешки и да се модифицират;
- могат да се използват вече готови програми, оформени като модули;
- логиката на цялата програмата става по-разбираема.

Съгласно принципа за абстракция на данните, методите за използване на данните се отделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с абстрактни данни – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени конструктори, мутатори и функции за достъп (селектори), които реализират абстрактните данни по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракция:

1. Приложения в проблемната област.
2. Модули, реализиращи основните операции над данните.
3. Примитивни операции – конструктори, мутатори, селектори.
4. Избор на представянето на данните.

Реализацията на подхода трябва да е такава, че всяко ниво използва единствено средствата на непосредствено следващото го ниво. По този начин, промените, които възникват на едно ниво ще се отразят само на предходното ниво.

При изпълнение на компютърна програма се извършват определени действия над данните, дефинирани в програмата. Тези данни се съхраняват в **информационните структури**, допустими в съответния език за програмиране. Най-общо типовете информационни структури могат да бъдат разделени на два вида: вградени и абстрактни. Вградените типове са предварително дефинирани и се поддържат от самия език, а абстрактните типове се дефинират от програмиста. Друга класификация на типовете е следната: скаларни и съставни типове. Скаларните типове представят данни, които се състоят само от една компонента. При съставните типове данни, данните представляват редица от компоненти. Скаларните типове, поддържани в C++ са следните: булев, цял, реален, символен, изброен, указател, псевдоним. Съставните типове, поддържани в C++ са следните: масив, вектор, запис.
Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен **указател** към T. Множеството от стойностите на типа T* се състои

от адресите на променливите от тип T, заедно със специалната константа NULL (нулев указател), която може да бъде свързана с всеки указател, независимо от типа T. За променливите от тип указател се разпределят 4 байта памет.
Нека T е име или дефиниция на тип. За типа T, T& е тип, наречен **псевдоним** на T. Множеството от стойностите на типа T& се състои от всички вече дефинирани променливи от тип T. Дефинирането на променлива от тип T& задължително е с инициализация – дефинирана променлива от тип T. След това не е възможно променливата-псевдоним да стане псевдоним на друга променлива.
Променлива, това е място за съхранение на данни, което може да съдържа различни стойности от някакъв определен тип. Идентифицира се със зададено от потребителя име. Една променлива се дефинира като се зададат нейното име и типа на стойностите, които може да приема. Типът определя броя на байтовете, в които ще се съхранява променливата. Мястото в паметта, където е записана стойността на променливата се нарича **адрес** на тази променлива. По-точно адресът на променливата е адресът на първия байт от множеството байтове, отделени за променливата. Намирането на адреса на една променлива става чрез унарния префиксен оператор **&** (амперсанд) със следния синтаксис: &<променлива>, където <променлива> е име на вече дефинирана в програмата променлива.
Адресите на променливите от един тип T могат да се присвояват на променливите от тип T* - указател към T.
Извличането на съдържанието на един указател става чрез унарния префиксен оператор ***** със следния синтаксис: *<променлива_от_тип_указател>. С други думи, този оператор извлича стойността на адреса, записан в променливата от тип указател с име <променлива_от_тип_указател>. Ако променливата-указател е от тип T*, то резултатът от прилагането на оператора * е данна от тип T – всички операции допустими за типа T са допустими и за нея.

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, <, >, <=, >=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още **адресна аритметика**. Особеността се изразява в т.н. мащабиране. Ще го изясним с пример.
Да разгледаме следния фрагмент.

```
int *p;
double *q;
…
p = p + 1;
q = q + 1;
```

Операторът p = p+1; свързва p не с предишната стойност на p, увеличена с 1, а с предишната стойност на p, увеличена с 1*4,

където 4 е броят на байтовете, необходими за записването на данна от тип int (p е указател към int). Аналогично, q = q+1; увеличава стойността на q с 1*8, а не с 1, тъй като q е указател към тип double (8 байта са необходими за записване на данна от този тип). Общото правило е следното: ако p е указател от тип T*, стойността на p+i е p+i*sizeof(T), където sizeof (T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T. При изваждането нещата стоят по аналогичен начин.

В C++ е възможно да се дефинират указатели, които са константи, а също и указатели, които сочат към константи. И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента дефиниран като const (указателя или обекта, към който сочи) не може да се променя. Ще разгледаме пример.
int i, j, k;
int * const b = &i;
b е константен указател към тип int и той не може да променя стойността си, но чрез него може да се променя стойността на i;
const int *c = &j;
c е указател към константа от тип int и той може да променя стойността си, но чрез него не може да се променя стойността на j;
const int * const d = &k;
d е константен указател към константа от тип int – той не може да променя стойността си и чрез него не може да се променя стойността на k, може да служи единствено за извличане на стойността на k.

В C++ има интересна и полезна връзка между указателите и масивите. Тя се състои в това, че имената на масивите са указатели към техните “първи” елементи. Това позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив. Ще разгледаме пример. Нека a е масив, дефиниран по следния начин:
int a[100];
Тъй като a е указател към a[0], то *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a+1 е адресът на a[1],
a+2 е адресът на a[2] и т.н. a+99 е адресът на a[99]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, …, 99). Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните операции, приложими над указатели не могат да се приложат за масиви. Такива са ++, -- и присвояването на стойност. Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се

преобразуват в *(a+i), т.е. операторът за индексиране се обработва от компилатора чрез адресна аритметика.
Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще изясним с пример казаното. Нека a е двумерния масив, дефиниран така:
int a[10][20];
Тогава a е константен указател към първия елемент на едномерния масив a[0], a[1], …, a[9], като всяко a[i] (i = 0, 1, …, 9) е константен указател към a[i][0]. При това положение имаме следните еквивалентни записи: **a a[0][0], *a a[0], *(a+1) a[1], …, *(a+9) a[9], по-общо a[i] *(a + i) (i = 0, 1, …, 9), a[i][j] (*(a + i))[j] *(*(a + i) + j).

Основните алгоритмични структури в C++ са операторът за присвояване, разклонените алгоритмични структури – операторите за условен преход if в кратка форма, if-else в дълга форма и switch, оператор за безусловен преход goto и операторите за цикли, чрез които се пораждат циклични изчислителни процеси – for, while и do-while. Има и още един вид оператори – операторът break, който се използва за излизане от цикъл или от switch и оператор continue, чрез които безусловно се прескача към следващата итерация на цикъл.

Добавянето на нови оператори в приложенията, реализирани на езика C++, се осъществяват чрез **функциите**. Те са основни структурни единици, от които се изграждат програмите на езика. Всяка функция се състои от множество от оператори, оформени подходящо за да се използват като обобщено действие или операция. След като една функция бъде дефинирана, тя може да бъде изпълнявана многократно за различни входни данни.

Програмите на C++ се състоят от една или повече функции. Сред тях задължително трябва да има точно една с име **main**, която се нарича **главна функция**. Тя е първата функция, която се изпълнява при стартиране на програмата. Главната функция, от своя страна, може да се обръща към други функции. Нормалното изпълнение на програмата завършва с изпълнението на главната функция (възможно е изпълнението да завърши принудително по време на изпълнение на функция, различна от главната).
Ще разгледаме най-общото разпределение на оперативната памет за изпълнима програма на C++. Това разпределение зависи от изчислителната система, от типа на операционната система, а също от модела памет. Най-общо се състои от: програмен код, област на статичните данни, област на динамичните данни и програмен стек.
В частта **програмен код** е записан изпълнимият код на всички функции, изграждащи потребителската програма. В областта на **статичните данни** са записани глобалните обекти на програмата. За реализиране на динамични структури от данни се използват

средства за динамично разпределяне на паметта. Чрез тях се заделя и се освобождава памет в процеса на изпълнение на програмата, а не преди това (при нейното компилиране). Тази памет е областта на **динамичните данни**. **Програмният стек** съхранява данните на функциите на програмата. Неговите елементи са блокове от памет, съхраняващи данни, дефинирани в някаква функция, които се наричат **стекови рамки**.
Ще разгледаме примерен фрагмент, който изчислява най-големият общ делител на две естествени числа. За целта дефинираме две функции: gcd (x, y), която намира най-големият общ делител на x и y и main, която се обръща към gcd

```
int gcd (int x, int y)
{ int h ;
  while (y != 0)
   { h = x;
     x = y;
     y = h % y;
    }
  return x;
}
int main ()
 {  int a = 14, b = 21;
    int r = gcd (a, b);
  return 0;
 }
```

Описанието на функцията gcd прилича на това на функцията main. Заглавието определя, че gcd е двуаргументна целочислена функция с цели аргументи. Името е произволен идентификатор, в случая е направен мнемоничен. Запазената дума int пред името на функцията е нейният тип (по-точно е типът на резултата на функцията). В кръгли скоби и отделени със запетая са описани параметрите x и y на gcd. Те са различни идентификатори и се предшестват от типовете си. Наричат се **формални параметри** за функцията. Тялото на функцията е блок, реализиращ алгоритъма на Евклид за намиране на най-големия общ делител на естествените числа x и y. Завършва с оператора

```
return x;
```

чрез който се прекратява изпълнението на функцията като стойността на израза след return се връща като стойност на gcd в мястото, в което е направено обръщението към нея.
Ще опишем как се изпълнява фрагментът.
Дефинициите на функциите main и gcd се записват в областта на паметта, определена за програмния код. Изпълнението на програмата започва с изпълнението на функцията main. Чрез първия ред от тялото на main се дефинират целите променливи a и b, като на a се присвоява стойност 14 и на b стойност 21.
В тази последователност те се записват в дъното на програмния стек в стековата рамка на main. Чрез следващия оператор се дефинира цялата променлива r, като в стековата рамка на main,

веднага след променливата b се отделят четири байта, в които ще се запише резултатът от обръщението gcd (a, b) към функцията gcd. Променливите a и b се наричат **фактически параметри** за това обръщение. Забелязваме, че типът им е същия като на съответните формални параметри x и y.
Сега ще опишем как се изпълнява обръщението gcd (a, b).
В програмния стек се генерира нов блок памет – стекова рамка за функцията gcd. Обръщението се осъществява на два етапа. Първият етап е **свързването** на формалните с фактическите параметри. В стековата рамка на gcd се отделят по четири байта за формалните параметри x и y в обратен ред на реда, в който са записани в заглавието. В тази памет се откопират стойностите на съответните им фактически параметри. Отделят се също четири байта за т.н. return-адрес, адреса на мястото в main, където ще се върне резултатът, а също се отделя памет, в която се записва адресът на стековата рамка на извикващата функция (в случая main). И така на този етап формалните параметри x и y се свързват съответно със стойностите 14 (стойността на a) и 21 (стойността на b). Вторият етап е изпълнението на тялото на функцията gcd. На първата стъпка се заделя памет четири байта за дефинираната променлива h във върха на стековата рамка за gcd. Тъй като е в сила y != 0, стойността на h става 14, след това стойността на x става 21 и след това стойността на y става 14 % 21 = 14. С други думи, тук x и y размениха стойностите си. На следващата итерация на цикъла отново y != 0, така че стойността на h става 21, стойността на x става 14 и след това стойността на y става 21 % 14 = 7. Отново y != 0, така че отново се изпълнява тялото на цикъла – стойността на h става 14, стойността на x става 7, стойността на y става 14 % 7 = 0. Това е последната итерация на цикъла и той приключва. Изпълнението на оператора return x; преустановява изпълнението на gcd като връща в main в мястото на прекъсването (return-адреса) стойността 7 на обръщението gcd (a, b). Отделената за gcd стекова рамка се освобождава. Указателят към върха на стека сочи края на стековата рамка на main. Изпълнението на програмата продължава с инициализацията на променливата r. Резултатът от обръщението gcd (a, b) се записва в отделената за r памет. След това се изпълнява оператора return 0; след което се освобождава и стековата рамка на main.
Функцията gcd реализира най-простото и чисто дефиниране и използване на функции – получава входните си стойности единствено чрез формалните си параметри и връща резултата си чрез оператора return. Забелязваме, че обръщението gcd (a, b) работи с копия на стойностите на a и b, запомнени в x и y, а не със самите a и b. В процеса на изпълнение на тялото на gcd, стойностите на x и y се променят, но това не оказва влияние на стойностите на фактическите параметри a и b. Такова свързване на формалните с фактическите параметри се нарича **свързване по стойност** или още **предаване на параметрите по стойност**.

При него фактическите параметри могат да бъдат не само променливи, но и изрази от типове, съвместими с типовете на съответните формални параметри.
В редица случаи се налага функцията да получи входа си чрез някои от формалните си параметри и да върне резултат не по обичайния начин (чрез оператор return), а чрез същите или други параметри. Ще разгледаме примерен фрагмент, който разменя стойностите на две реални променливи. Идеята е следната – ако дефинираме функция swap (double *x, double *y), която разменя стойностите на реалните променливи, към които сочат указателите x и y, то обръщението swap (&a, &b) би разменило стойностите на a и b.

```
void swap (double *x, double *y)
{  double work = *x;
   *x = *y;
   *y = work;
   return;
}
int main () {
  double a = 1.5, b = 2.75;
  swap (&a, &b);
  return 0;
}
```

Функцията swap има подобна структура като на gcd. Но и заглавието и тялото и са по-различни. Типът на swap е указан чрез запазената дума void. Това означава, че функцията не връща стойност чрез оператора return. Затова в тялото на swap е пропуснат израза след return. Формалните параметри са указатели към double, а в тялото се работи със съдържанието на тези указатели. Забелязваме също, че обръщението към swap във функцията main не участва като аргумент на операция, а е оператор. Изпълнението се осъществява подобно на изпълнението на по-горния фрагмент с малки разлики – при първия етап на изпълнението на обръщението към swap в отделената памет за формалните параметри x и y се записват **адресите** на съответните им фактически параметри. Поради тази причина, промяната на стойностите, към които сочат формалните параметри x и y води до промяна на самите фактически параметри. Функцията swap получава входните си стойности чрез формалните си параметри и връща резултата си чрез тях. Забелязваме, че обръщението swap (&a, &b) работи не с копия на стойностите на a и b, а с техните адреси. В процеса на изпълнение на тялото на swap се променят стойностите на фактическите параметри a и b. Такова свързване на формалните с фактическите параметри се нарича **свързване по указател** или **предаване на параметрите по указател** или **свързване по адрес**. При този вид предаване на параметрите, фактическите параметри задължително са променливи указатели или адреси на променливи.

Освен тези два начина на предаване на параметри, в езика C++ има още един – **предаване на параметри по псевдоним**. Той е сравнително по-удобен от предаването по указател и се предпочита от програмистите. Ще го илюстрираме с фрагмент, еквивалентен на горния (т.е. изпълняващ същите действия), но реализиращ функцията swap, в която предаването на параметрите е по псевдоним.

```
void swap (double &x, double &y)
{  double work = x;
   x = y;
   y = work;
   return;
}
int main () {
  double a = 1.5, b = 2.75;
  swap (a, b);
  return 0;
}
```

Специфичното при изпълнението на фрагмента е следното – при първия етап от изпълнението на обръщението към swap, тъй като формалните параметри x и y са псевдоними на променливите a и b, за тях памет не се отделя в стековата рамка на swap. Това което се прави е параметърът x да се закачи за фактическия параметър a и аналогично параметърът y се закача за фактическия параметър b. Така всички действия с x и y в swap всъщност се изпълняват с фактическите параметри a и b от main съответно. Да забележим, че фактическите параметри, съответстващи на формални параметри-псевдоними при всички случаи са променливи. Тази реализация на swap е по-ясна от съответната с указатели. Тялото и реализира размяна на стойностите на две реални променливи без да се налага използването на адреси.

Възможно е някои параметри да се предават по стойност, други по псевдоним или указател, а също функцията да получи резултат и с оператора return.

Възможно е в една програма, към една функция да се извършват обръщение на място, което предшества нейната дефиниция. В този случай дефиницията на функцията, която извършва това обръщение трябва да се предшества от **декларацията** на използваната функция. Декларацията на една функция се състои от нейното заглавие, последвано от ';'. В него имената на формалните параметри могат да се пропуснат.

Типът на една функция е произволен без масив, но се допуска да е указател. Ако е пропуснат, се подразбира int.

Списъкът от формалните параметри на една функция (още се нарича **сигнатура**) може да е празен или void. В случай, че е непразен, имената на параметрите трябва да са различни. Те заедно с името определят еднозначно функцията. Възможно е

също един формален параметър да се специфицира със запазената дума const, което означава че неговата стойност няма да бъде променяна в рамките на тялото на функцията.
Операторът **return** връща резултата на функцията в мястото на извикването. Синтаксисът е следния: return [<израз>];. Тук return е запазена дума, <израз> е произволен израз от тип, съвместим с типа на функцията. Ако типът на функцията е void, <израз> се пропуска. Семантиката е следната: пресмята се стойността на <израз>, конвертира се до типа на функцията и връщайки получената стойност в мястото на извикването на функцията, прекратява нейното изпълнение. Да отбележим, че ако функцията не е void, тя задължително трябва да върне стойност. Това означава, че операторът return трябва да се намира във всички клонове на тялото.
Функциите могат да се дефинират в произволно място на програмата, но не и в други функции. Така за разлика от други езици за процедурно програмиране, в C++ не се допуска влагане на функции.
При стартирането на една програма се създава стековата рамка на главната функция, която се поставя в дъното на програмния стек. Във всеки момент от изпълнението на програмата, във върха на стека е стековата рамка на функцията, която се обработва в момента. Непосредствена под нея в стека стои стековата рамка на функцията, извършила обръщението към функцията, която се обработва в момента. Когато изпълнението на една функция завърши, стековата рамка на функцията, съответна на това обръщение се отстранява от стека.

Идентификаторите в C++ означават имена на константи, променливи, формални параметри, функции, класове. Най-общо казано има три вида области на идентификаторите – **глобална**, **локална** и **област на клас**. Областите се задават неявно – чрез позицията на дефиницията на идентификатора в програмата. Област на клас няма да разглеждаме.
Глобалните идентификатори са дефинираните извън дефинициите на функции променливи и константи и могат да се използват във всички функции на програмата, чиято дефиниция следва тяхната такава.
Повечето константи и променливи са локални идентификатори и имат локална област на действие. Те са дефинирани вътре във функциите и не са достъпни за кода в другите функции на модула. Областта им се определя според следното общо правило – започва от мястото на дефинирането и завършва в края на оператора (блока), в който е дефиниран идентификатора. Формалните параметри на функцията също имат локална видимост. Областта им на действие е цялото тяло на функцията. В различните области могат да се използват еднакви идентификатори. Ако областта на един идентификатор се съдържа в областта на друг идентификатор, първият се нарича нелокален за последния. В този

случай е в сила правилото: Локалният идентификатор скрива нелокалния в областта си.
Освен чрез механизма за свързване на формалните параметри с фактически и чрез предаване на резултат чрез оператора return, функциите могат да обменят помежду си данни чрез глобални променливи. Първите два начина са основни и са за предпочитане, тъй като с тях се постигат по-ясни и разбираеми програми. От друга страна, използването на глобални променливи е по-ефективно, тъй като не изисква заместване.
Използването на глобални променливи обикновено се счита за лоша практика поради тяхната нелокалност: една глобална променлива потенциално може да се модифицира от всяко място в програмата и всяка част от програмата може да зависи от нея. Въпреки това, в някои случаи, глобалните променливи са подходящи за употреба, тъй като те могат да се използват за избягване на предаването на често използвани променливи в рамките на няколко функции. Ще разгледаме пример с една проста програма.

```
#include <iostream.h>
int global = 3;
void ChangeGlobal ()
{ global = 5; }
int main ()
{ cout << global << endl;
  ChangeGlobal();
  cout << global << endl;
  return 0;
 }
```

Тъй като променливата global е глобална, няма нужда тя да се предава като параметър за да може да се използва от функции, различни от главната функция. Глобалната променлива може да се използва във всяка функция на програмата.
Изходът от изпълнението на програмата:

```
3
5
```

Използването на глобални променливи прави софтуерът по-труден за четене и за разбиране. Тъй като всяка част от кода на програмата може да се използва за промяна на стойността на глобална променлива, разбирането на използването на променливата може да включва разбирането на голяма част от самата програма.

Най-общо един обект е **рекурсивен**, ако се съдържа в себе или е дефиниран чрез себе си. В математиката има много примери за рекурсивни обекти, например функцията n!, която се дефинира така: 0! = 1, (n+1)! = n * n!.
Езикът C++ поддържа две конструкции, които позволяват да се реализират рекурсивни алгоритми – това са структури с рекурсия и рекурсивни функции.

Едно интересно свойство на структурата е това, че нейните елементи могат да имат тип указател и следователно тип указател към структура. Ще отбележим изрично, че не се допуска елемент на структура да бъде структура от същия тип. Тъй като указателят не дефинира променлива, а само сочи към такава, за него ограничението не е в сила. Така елементи на една структура могат да бъдат указатели към структура от същия тип. По този начин се получава рекурсивно използване на структури. Чрез него могат да се съставят сложни описания на данни като линейни списъци и дървовидни структури. Например, в C++ е валидна следната дефиниция на структура:

```
struct list {
  int value;
  list *next;
};
```

Указателят next, описан в тялото на структурата list сочи към променлива – структура от тип list. Тази структура може да се използва за реализация на едносвързан списък.

Известно е, че в тялото на всяка функция може да бъде извикана друга функция, която е дефинирана или декларирана до момента, в който се извиква. Освен това, в C++ е разрешено една функция да извиква в тялото си самата себе си. Функция, която пряко или косвено се обръща към себе си, се нарича **рекурсивна**.

Под **косвено** обръщение разбираме следното – функцията F_1 извиква функцията F_2, функцията F_2 извиква функцията F_3, …, функцията F_{n-1} извиква функцията F_n и след това функцията F_n извиква функцията F_1.

Сега ще разгледаме примерен фрагмент, който съдържа рекурсивна функция и ще проследим как той се изпълнява. Фрагментът ще пресмята m! за цяло число m, m ≥ 0.

```
int fact (int n)
 { if (n == 0) return 1;
   else return n * fact (n-1);
 }
int main ()
{ int m = 4;
  int n = fact (m);
  return 0;
}
```

В тази фрагмент е описана рекурсивна функция fact, която приложена към цяло неотрицателно число връща факториела на числото. При изпълнението на програмата интересно е извършването на обръщение към fact с фактически параметър m, който има стойност 4.

Ще проследим как се извършва това обръщение. Генерира се стекова рамка за обръщението към fact. В нея се отделят четири байта за формалния параметър n, в която памет се откопирва

стойността на фактическия параметър m (4) и след това започва изпълнение на тялото на функцията. Тъй като n е различно от 0, се изпълнява операторът в else-частта, при което се извършва обръщение към fact с фактически параметър n-1, т.е. fact (3). По такъв начин преди завършването на първото обръщение към fact се прави второ обръщение към тази функция. За целта се генерира нова стекова рамка за новото обръщение към функцията fact, в която за формалния параметър n се откопира стойност 3. Тялото на функцията fact започва да се изпълнява втори път (временно спира изпълнението на тялото на функцията fact, предизвикано от първото обръщение към нея). По аналогичен начин възникват още обръщения към функцията fact. При последното от тях, стойността на формалния параметър n е равна на 0 и тялото на fact се изпълнява напълно. В резултат, изпълнението на това обръщение завършва и за стойност на fact се връща 1. Тази стойност се връща в непосредствено предишното извикване на fact, което изчислява 1*1 = 1 и аналогично се извършва връщане назад по веригата от извикванията на fact, докато се стигне до първото извикване, което изчислява 4*6 = 24 и връща тази стойност в главната функция. Естествено, след завършването на всяко изпълнение на fact, стековата рамка, която отговаря за това изпълнение се освобождава.

В този конкретен случай, рекурсивното дефиниране на функцията факториел не е подходящо, тъй като съществува лесно итеративно решение. Ще отбележим, че тъй като в тялото на рекурсивната функция fact има само едно обръщение към fact, то тогава всяко извикване на fact поражда най-много едно извикване на fact. Затова функцията fact още се нарича **линейно рекурсивна**.
Когато в тялото на една рекурсивна функция има повече от едно обръщение към самата нея и при това съществува извикване на функцията, което поражда повече от едно извикване към същата функция, то рекурсивната функция се нарича **разклонено рекурсивна**. Пример е рекурсивната функция за изчисляване на n-тото число на Фибоначи (n естествено), която се дефинира така:

```
int fib (int n);
{ if (n == 0 || n == 1) return 1;
  else return fib (n-2) + fib (n-1);
}
```

В случая всяко извикване на fib поражда две независими извиквания на fib, което води до експоненциална сложност на рекурсивния алгоритъм. От друга страна, съществува елементарен итеративен алгоритъм, който решава задачата с линейна сложност.
Така стигаме до следната препоръка: ако за решаването на една задача може да се използва итеративен алгоритъм, той трябва да се реализира, не се препоръчва безпринципното използване на рекурсия, тъй като това води до загуба на време и памет.

Съществуват обаче задачи, които се решават трудно ако не се използва рекурсия.
Простотата и компактността на рекурсивните функции проличава особено при работа с динамичните рекурсивни структури данни (които се дефинират чрез рекурсивни структури): свързан списък, стек, опашка, дърво. Основен недостатък на рекурсивните програми е намаляването на бързодействието поради загуба на време за копиране на параметрите в стека. Освен това се изразходва повече памет, особено при по-дълбока степен на вложеност на рекурсията.
Едно общо правило при създаването на рекурсивни функции е следното: когато реализираме рекурсия, във функцията трябва да има гранични условия за излизане от рекурсията, които задължително се достигат след краен брой вложени извиквания на рекурсивната функция. В противен случай се получава безкрайно зацикляне и обикновено програмата приключва аварийно поради препълване на програмния стек. За повишаване на ефективността на рекурсивните функции, всички локални променливи и параметри във функцията, които се използват само преди първото рекурсивно обръщение могат да се изнесат извън функцията като глобални.
По този начин се избягва излишното им повторно създаване в новата стекова рамка при изпълнение на рекурсивното обръщение.
Ще отбележим, че за всеки итеративен алгоритъм лесно може да се построи рекурсивен алгоритъм, който решава същата задача. Обратното също е вярно, тъй като е възможно рекурсията да се моделира итеративно, но само чрез използване на достатъчно голям потребителски стек.

11. Обектно-ориентирано програмиране (C++): Основни принципи. Класове и обекти. Оператори. Шаблони на функции и класове.

При подхода **абстракция със структури данни** методите за използване на данните са разделени от методите за тяхното представяне. Програмите се конструират така, че да работят с абстрактни данни – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции: **конструктори**, **селектори** и **мутатори**, които реализират абстрактните данни по конкретен начин.
За да илюстрираме подхода ще създадем програма за рационално-числова аритметика. В програмата се дефинират функции за събиране, изваждане, умножение и деление на рационални числа по общоизвестните правила. Тези операции лесно могат да се реализират, ако има начин за конструиране на рационално число по зададени две цели числа, представящи съответно неговия числител и знаменател и ако има начини за извличане на числителя и знаменателя на рационално число.

Затова в програмата се дефинират следните функции:
void makerat(rat &r, int a, int b) – конструира рационално число r по зададени числител a и знаменател b;
int numer (rat &r) – извлича числителят на рационалното число r;
int denom (rat &r) – извлича знаменателят на рационалното число r.
В тези декларации rat е името на типа рационално число. Все още не знаем как са дефинирани трите функции, но ако предположим, че те са реализирани, то функциите за рационално-числова аритметика и процедурата за извеждане на рационално число могат лесно да се реализират. Например, функцията за събиране се реализира така:

```
rat sumrat (rat &r1, rat &r2)
{ rat r;
  makerat (r, numer (r1) * denom (r2) + numer (r2) * denom (r1),
              denom (r1) * denom (r2));
   return r;
}
```

Останалите операции subrat, multrat, quotrat за изваждане, умножение и деление на рационални числа се реализират аналогично. Функцията за извеждане се дефинира така:

```
void printrat (rat &r)
{ cout << numer (r) << '/' << denom (r) << endl; }
```

Сега ще се върнем към представянето на рационалните числа, а също към реализация на примитивните операции: конструктора makerat и селекторите numer и denom. Тъй като рационалните числа се представят чрез наредена двойка от цели числа, удобно е използването на структура:

```
struct rat {
  int num;
  int den;
};
```

Тогава примитивните функции, които реализират конструктора и селекторите имат вида:

```
void makerat (rat &r, int a, int b)
{ r.num = a;
  r.den = b;
}
int numer (rat &r)
{ return r.num; }
int denom (rat &r)
{ return r.den; }
```

След като са дефинирани всички функции можем да имаме следният програмен фрагмент в някоя функция:

```
…
rat r1, r2, r3;
makerat (r1, 1, 2);
makerat (r2, 3, 4);
r3 = sumrat (r1, r2);
printrat (r3);
```

…
След изпълнението на този фрагмент ще се изведе 10/8.
В примера се вижда, че реализирането на подхода абстракция със структури данни има четири нива на абстракция:

- използване на рационалните числа в проблемната област (намиране на сумата на 1/2 и 3/4);
- реализиране на правилата за рационално-числова аритметика (sumrat, subrat, multrat, quotrat, printrat);
- избор на представяне за рационалните числа и реализиране на примитивни конструктори и селектори (makerat, numer, denom);
- работа на ниво структура.

Използването на подхода абстракция със структури данни прави програмите по-лесни за описание и модификация. Недостатък на горната реализация е, че функциите не съкращават рационалните числа. За да се поправи този недостатък, обаче, трябва да се промени единствено функцията makerat. Да предположим, че функцията gcd намира най-големия общ делител на две естествени числа. Тогава новата makerat има вида:

```
void makerat (rat &r, int a, int b)
{ if (a == 0) { r.num = 0; r.den = 1; }
  else { int aa = (a > 0) ? a : -a;
         int ab = (b > 0) ? b : -b;
         int g = gcd (aa, ab);
         if ((a > 0 && b > 0) || (a < 0 && b < 0))
           { r.num = aa/g; r.den = ab/g; }
         else { r.num = – aa/g; r.den = ab/g; }
}
```

С това проблемът със съкращаването на рационални числа е решен. За неговото решаване се наложи малка модификация, която засегна само конструктора makerat. Така илюстрирахме лесната модифицируемост на програмите, реализиращи подхода абстракция със структури данни.
Изрично ще отбележим, че дефинираната структура rat, която представя рационално число не може да се използва като тип данни рационално число. Това е така, защото при типовете данни представянето на данните е скрито за потребителя.
За всеки тип данни е известно множество от допустимите стойности и множество от вградените функции и операции, допустими за този тип. Така възниква усещането, че представянето на рационално число като запис с две полета трябва да се обедини с примитивните операции (makerat, numer, denom). Последното е възможно, тъй като в C++ се допуска полетата на структура да бъдат функции. За да направим това в нашия пример извършваме следните стъпки:

1. Включваме декларациите на makerat, numer и denom в дефиницията на структурата rat, като елиминираме формалният параметър r и в трите функции.

Това води до следната дефиниция на структурата rat:

```
struct rat {
   int num;
   int den;
   void makerat (int a, int b);
   int numer();
   int denom();
};
```

2. Отразяваме промените в дефинициите на функциите makerat, numer и denom – премахваме формалният параметър r и пред името на всяка от тези функции поставяме името на структурата rat и операцията за разрешаване на достъп ::.

Така получаваме следните дефиниции:

```
void rat::makerat (int a, int b)
{ if (a == 0) { num = 0; den = 1; }
  else { int aa = (a > 0) ? a : -a;
         int ab = (b > 0) ? b : -b;
         int g = gcd (aa, ab);
         if ((a > 0 && b > 0) || (a < 0 && b < 0))
           { num = aa/g; den = ab/g; }
         else { num = – aa/g; den = ab/g; }
}
int rat::numer ()
{ return num; }
int rat::denom (rat r)
{ return den; }
```

Функциите makerat, numer, denom при тази модификация се наричат **член-функции** на структурата rat. Извикването им се осъществява като полета на структура, например (r е променлива от тип rat) r.makerat(1, 5), r.denom(), r.numer().

Сега забелязваме, че във функциите sumrat, subrat, multrat, quotrat и printrat не се използват полетата на записа num и den, но ако направим опит за използването им даже на ниво main, той ще бъде успешен. Последното може да се забрани, ако се използва етикетите private: пред дефиницията на полетата num и den и public: пред декларациите на член-функциите. Структурата rat приема вида:

```
struct rat {
 private:
   int num;
   int den;
 public:
   void makerat (int a, int b);
   int numer();
   int denom();
};
```

Опитът за използване на полетата num и den на структурата rat извън нейните член-функции води до грешка.

Сега ако заменим запазената дума struct със запазената дума class и запазим всички останали дефиниции получаваме еквивалентна програма. Единствената разлика е, че достъпът по подразбиране вътре в дефиницията в този случай е private, а не public.
Така естествено достигнахме до понятието **клас**.

Класовете са типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ потребителски тип или да представят нов тип данни. Както вече видяхме, класовете са подобни на структурите и в повечето отношения са идентични – всеки клас може да се разгледа като структура, на която са наложени някои ограничения по отношение на правата на достъп. Всеки клас съдържа данни, наречени **данни-елементи** (**член-данни**) на класа и набор от функции, наречени **функции-елементи** (**член-функции**), които обработват данните-елементи на класа. Понякога функциите-елементи в обектно-ориентираното програмиране се наричат **методи**.

Дефинирането на един клас се състои от две части:
- декларация на класа;
- дефиниция на неговите член-функции (методи).

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, последвана от името на класа. Тялото е заградено във фигурни скоби. След тези скоби стои списък от имена на обекти (може да е празен), разделени със запетаи и накрая се записва ‘;’. В тялото на класа са декларирани членовете на класа (член-данни и член-функции). Имената на членовете на класа са локални за този клас, т.е. в различните класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същи тип могат да се изредят, разделени със запетаи и предшествани от съответния тип. Изрично ще отбележим, че типът на член-данна на един клас не може да съвпада с името на този клас, но това не важи за типа на член-функциите и типовете на техните параметри. В тялото някои декларации могат да бъдат предшествани от **спецификаторите за достъп public**, **private**, **protected**. Областта на един спецификатор за достъп започва от самия него и продължава до следващия спецификатор или до края на декларацията на класа, ако няма следващ спецификатор. Подразбиращ се спецификатор за достъп е private. Един и същ спецификатор за достъп може да се използва повече от един път в декларацията на клас.
Достъпът до членовете на класовете може да се разглежда на следните две нива:
- ниво член-функции;
- ниво външни функции.

Член-функциите на един клас имат достъп до всички членове на този клас. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича **режим**

на пряк достъп. Режимът на достъп до членовете на класа за външните функции се определя от начина на дефиниране на членовете. Членовете на един клас, декларирани като private са видими само в рамките на класа и външните функции нямат достъп до тях. Чрез използване на членове на класа, декларирани като private, се постига скриването на тези членове за външната за класа среда. Този процес на скриване се нарича още **капсулиране на информацията**. Членовете на клас, които трябва да бъдат видими извън класа (т.е. достъпни за външни функции) трябва да бъдат декларирани като public. Освен като private и public, членовете на класа могат да бъдат декларирани като protected – тогава те имат същото поведение като private, но разликата е, че те могат да бъдат достъпни от производните класове на дадения клас.
След като един клас е деклариран, трябва да се дефинират неговите методи. Член-данните на един клас се дефинират в рамките на тялото на класа. За член-функциите има два варианта: дефинират се в тялото на класа или извън тялото на класа, като във втория случай в тялото на класа се посочват само техните прототипи. Когато една член-функция се дефинира извън тялото на съответния клас, нейното име трябва да се предшества от името на класа, към който принадлежи член-функцията, последвано от бинарната операция за разрешаване на достъп.
В случай, че една член-функция на клас е дефинирана вътре в тялото на класа, тя се разглежда като **вградена функция** (inline).
Ще отбележим, че в тялото на дефиницията на член-функция явно не се използва обектът, върху който тя ще се приложи. Той участва неявно – чрез член-данните на класа. Поради това той се нарича **неявен** параметър на член-функцията. Останалите параметри, които участват явно в дефиницията на член-функцията се наричат **явни**. Всяка член-функция има точно един неявен параметър и нула или повече явни.

Член-данните и член-функциите на класа имат **област на действие клас**. Външните функции, които не са елементи на клас имат област на действие файл. В областта на действие клас, елементите на класа са непосредствено достъпни за всички функции-елементи на този клас и те могат да се използват просто по име. Извън областта на действие клас, елементите на класа, които са декларирани като public, могат да се използват или с помощта на обект от този клас, или чрез псевдоним на обект от този клас, или чрез указател към обект от този клас. Компонентите на функциите-елементи имат област на действия функция – променливите, които са дефинирани в тялото на функцията-елемент са известни само на тази функция. Ако във функция-елемент се дефинира променлива с име, съвпадащо с името на член-данна от област на действие клас, то първата скрива втората. Такива скрити променливи могат да станат достъпни с помощта на бинарната операция за разрешаване на достъп с ляв операнд

името на класа и десен операнд името на съответната член-данна. За разлика от функциите, класовете могат да се дефинират на различни нива в програмата: **глобално** (в област на действие файл) и **локално** (вътре в тялото на функция или в тялото на клас). Областта на глобално деклариран клас започва от декларацията на класа и продължава до края на програмата. Ако клас е дефиниран в рамките на тялото на функция, всички негови член-функции трябва да са вградени, тъй като в противен случай би се получило недопустимото за C++ влагане на функции. Областта на клас, дефиниран във функция започва от мястото, където той е деклариран и завършва в края на блока, от който е част тази декларация. Обекти на такъв клас могат да се дефинират и използват само във въпросния блок в тялото на функцията.

След като един клас е дефиниран, могат да се създават негови екземпляри, наречени **обекти**. Връзката между клас и обект в C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество компоненти (член-данни и член-функции). При всяко дефиниране на обект на класа автоматично се извиква точно един от конструкторите на класа с цел да се инициализира обекта. Ако дефиницията е без явна инициализация, дефинираният обект се инициализира чрез конструктора по подразбиране. Явна инициализация може да се извърши по два начина:

- името на обекта в дефиницията е последвано от скоби, в които е поместен списък от начални стойности – тези стойности се предават като параметри в подходящ конструктор на класа;
- името на обекта е последвано от знак за равенство и друг, вече дефиниран обект от класа.

При компилиране на дефиницията на клас не се заделя никаква памет. Памет се заделя едва при дефиниране на конкретни обекти от този клас. Достъпът до компонентите на обекта (ако е възможен), се осъществява по два начина – пряко и косвено. При прекия достъп се използва името на обекта и името на данната или метода, разделени с ‘.’. При косвения достъп се използва името на указател към обекта и името на данната или метода, разделени с ‘->’. Двата метода за достъп са взаимнозаменяеми: x.a (&x) -> a, px -> a (*px).a.
При създаването на обекти на един клас, кодът на методите на този клас не се копира за всеки обект, а се намира само на едно място в паметта. Естествено, възниква въпросът по какъв начин методите на един клас разбират за кой обект на този клас са били извикани. Отговорът на този въпрос дава **указателят this**. Всяка член-функция на клас поддържа допълнителен неявен формален параметър указател this от тип <име_на_клас>*. Това става по следния начин: компилаторът преобразува всяка член-функция на клас до обикновена функция с уникално име и допълнителен параметър указателят this и вътре в тялото на тази функция

преките обръщения към членове на класа се преобразуват към косвени обръщения чрез this, съответно, след това, компилаторът преобразува всяко обръщение към член-функция на класа в програмата до обръщение към съответната обикновена функция, като за фактически параметър, отговарящ на формалния параметър this се използва адресът на обекта, за който е била извикана член-функцията.
Създаването на обекти е свързано със заделяне на памет, запомняне на текущо състояние, задаване на начални стойности и други дейности, които се наричат **инициализация** на обекта. В езика C++ тези действия се изпълняват от специален вид член-функции на класовете – **конструкторите**. Конструкторът е член-функция, която се отличава от останалите член-функции по следните характеристики:

- името на конструктора съвпада с името на класа;
- типът на резултата на конструктора не се указва явно в неговата декларацията – винаги се подразбира, че е типът на указателя this;
- конструкторът се изпълнява автоматично при създаване на обекти;
- конструкторът не може да се извиква явно чрез обект или косвено чрез указател към обект.

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.
В заглавието на един конструктор при неговата дефиниция, непосредствено след списъка с формалните параметри може да има инициализиращ списък. Чрез този списък член-данни на класа могат да се свържат с начални стойности.
Инициализиращият списък е съставен от имена на член-данни на класа, последвани от скоби, в които е записана началната стойност на съответната член-данна, като отделните елементи на списъка се разделят със запетаи и в началото му се записва символът ‘:’. В конструктора една член-данна може да се инициализира или чрез инициализиращия списък или вътре в тялото на конструктора. По обясними причини за константните член-данни е възможен само първият начин за инициализация.
В рамките на една програма може да се извършва **предефиниране** на функции, което означава използване на функции с еднакви имена в една и съща област на видимост. При осъществяване на извикване към предефинирана функция, компилаторът търси варианта на функцията с възможно най-добро съвпадение. Като критерии за добро съвпадане имаме следните нива на съответствие: точно съответствие (по брой и тип на формалните и фактическите параметри) и съответствие чрез разширяване на типа на някои от параметрите. В един клас може да се дефинират няколко конструктора. Всички те имат едно и също име (името на класа), така че трябва да се различават по броя и/или типа на своите параметри. При създаването на един обект се изпълнява точно един от предефинираните конструктори.

Кой конкретен конструктор да се изпълни се определя съгласно критерия за най-добро съвпадение.
В един клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Във втория случай автоматично се създава т.н. **конструктор по подразбиране**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др. В първия случай, т.е. когато за класа има дефиниран поне един конструктор, тогава конструктор по подразбиране не се създава автоматично и за да може да се дефинират обекти без да се инициализират, потребителят сам трябва да дефинира подразбиращ се конструктор – той няма параметри или всичките му параметри са подразбиращи се.
Инициализацията на новосъздаден обект от даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или скоби, в които се записва името на обекта-инициализатор. Самата инициализация в този случай се извършва от специален конструктор, наречен **конструктор за присвояване** (**конструктор за копие**). Този конструктор поддържа точно един параметър от тип const <име_на_клас> &. Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в първия момент когато се наложи новосъздаден обект от класа да се инициализира с обект от същия клас. Освен при инициализация чрез обект, конструкторът за присвояване се използва и при предаване по стойност на обект като аргумент на функция, а също и при връщане на обект като резултат от изпълнение на функция. Изрично ще отбележим, че конструкторът за присвояване се използва само в описаните случаи, но не и при присвояване на един обект на друг извън инициализация. В последния случай се използва предефинираната операция за присвояване.
Указателите към обекти се дефинират по същия начин както указател към стандартен тип данни. Например, ако в програмата е дефиниран класът MyClass, то са валидни следните дефиниции:
```
MyClass obj;
MyClass *ref = &obj;
```
Връзката между масиви и указатели е в сила и в случая, когато елементите на масива са обекти.
Елементите на един масив могат да са обекти, но разбира се от един и същи клас. Нека например е дефиниран класът MyClass, който има открита член-данна x и открита член-функция f без параметри.
Следната дефиниция е валидна:
```
MyClass obj[20];
```
Достъпът до елементите на масива е пряк и се осъществява по стандартния начин чрез индексиране: obj[0], obj[1], …, obj[19]. Тъй като obj[i], i = 0, 1, …, 19, е обект, към него са възможни обръщенията obj[i].x и obj[i].f().

Член-данна на един клас може да е масив. Например, да предположим, че MyClass има допълнителна открита член-данна y[5]. Нека obj е обект от клас MyClass. Тогава достъпът до елементите на масива y в обекта obj ще се осъществи по следния начин: obj.y[0], obj.y[1], …, obj.y[4]. Конструкторът по подразбиране играе важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програмата се инициализира по два начина: явно (чрез инициализиращ списък) или неявно (чрез извикване на конструктора по подразбиране за всеки обект – елемент на масива).
Всяка програма има няколко места за съхраняване на данни, едно от тях е областта за динамични данни (heap). Динамичните данни са такива, че техният брой и размер не е известен в момента на проектиране на програмата. Те се създават и разрушават по време на изпълнение на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва наново. Така паметта се използва по-ефективно.
Създаването и разрушаването на динамични данни в езика C++ се извършва чрез операцията new и оператора delete. Извикването на new заделя необходимата памет в heap-а и връща указател към нея. Извикването на delete освобождава паметта в heap-а, сочена от указател. На всяко извикване на new трябва да съответства извикване на delete, тъй като heap-ът не се чисти автоматично и в противен случай паметта рано или късно ще свърши. Синтаксисът на операцията new е следния:
new <име_на_тип> [[<размер>]] или
new <име_на_тип>(<инициализация>). Тук <име_на_тип> е име на някой от стандартните типове или е име на клас, <размер> е произволен израз, който може да се преобразува до цял и показва броя на компонентите от тип <име_на_тип>, за които да се задели памет в heap-а, <инициализация> е израз от тип <име_на_тип> или инициализация на обект според синтактиса на конструктор на класа, ако <име_на_тип> е име на клас. При това, <инициализация> не може да присъства, ако е зададен <размер>. Семантиката на операцията new е следната: в heap-а се заделят sizeof (<име_на_тип>) или sizeof (<име_на_тип>) * <размер> байта в зависимост от това дали не е указан или е указан <размер> и се инициализират чрез <инициализация>, ако такава има. Операцията връща като резултат указател към (първия елемент от) заделената памет. Ако в heap-а няма достатъчно памет, new връща NULL. Синтаксисът на оператора delete е следния:
delete <указател_към_динамичен_обект>. Тук <указател_към_динамичен_обект> е указател към динамична данна, създадена чрез new. Семантиката на delete е следната: разрушава данната, адресирана от указателя и паметта, заемана от тази данна се освобождава. Ако данната, адресирана от указателя е обект на клас преди да се разруши се извиква деструктора на класа. Ако динамичната данна е масив (в заделянето на памет за нея с new е бил указан <размер>

по-голям от 1), то delete трябва да се използва в следната форма: delete [] <указател>.
Разрушаването на обекти на класовете в някои случаи е свързано с извършването на определени действия, които се наричат заключителни. Най-често тези действия са свързани с освобождаване на заделената преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност тези действия да се извършат автоматично при разрушаване на обекта. Това се осъществява чрез **деструкторите** на класовете.
Деструкторът е член-функция, която се извиква при разрушаване на динамичен обект с оператора delete или при излизане от блока, в който е бил създаден обекта на класа. Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа '~'. Типът му е void и той явно не се указва. Освен това, деструкторът не може да има формални параметри. Ако конструкторът или някоя член-функция на един клас реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като трябва да се освободи заетата динамична памет.
Съществуват два начина за създаване на обекти: чрез дефиниция или чрез функциите за динамично управление на паметта.
В първия случай обектът се създава при достигане на неговата дефиниция и се разрушава при излизане от блока, в който е поместена дефиницията (или когато приключи изпълнението на програмата, ако обектът е глобален). Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се реализира чрез извикване на обикновен конструктор или на конструктора за присвояване. Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв е явно дефиниран.
Във втория случай създаването и разрушаването на обекта се управлява от програмиста. Създаването става чрез операцията new, а разрушаването чрез оператора delete. Операцията new използва конструкторите на класа, а оператора delete деструктора на класа.
Създаването на масиви от обекти става по два начина.
Първият начин е чрез обикновена дефиниция. При него масивът се създава при достигането на тази дефиниция и се разрушава при излизане от блока, в който е поместена дефиницията (или при приключване на изпълнението на програмата, ако масивът е глобален). При създаването на масива от обекти за всеки елемент на масива се извиква конструкторът по премълчаване, освен ако не е зададена явна инициализация на масива, която представлява списък от обръщения към конструктори на класа. При разрушававането на масива от обекти за всеки елемент на масива се извиква деструктора на класа.

Вторият начин е чрез функциите за динамично управление на паметта. Отново създаването става чрез new, при това се указва размерът на масива и унищожаването става чрез delete []. При това, при изпълнението на операцията new за всеки обект от новосъздадения масив се извиква конструктора по премълчаване на класа, а при изпълнението на оператора delete[] се извиква деструктора на класа за всеки един от елементите на масива. Както вече споменахме, за масивите, реализирани в динамичната памет не може явно да се задава инициализация.
Всеки оператор се характеризира с позиция на оператора, спрямо аргументите му, приоритет и асоциативност. Позицията на оператора спрямо аргументите му го определя като префиксен (поставя се пред аргументите), инфиксен (поставя се между аргументите) или постфиксен (поставя се след аргументите). Приоритетът определя реда на изпълнение на операторите в операторен терм. Операторите с по-висок приоритет се изпълняват преди тези с по-нисък. Асоциативността определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има ляво и дясно асоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните се изпълняват отдясно наляво.
В C++ не могат да се дефинират нови оператори, но всеки съществуващ оператор, с изключение на ::, ?:, ., .*, sizeof, може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас. Предефинирането се извършва чрез дефиниране на специален вид функции, наречени **операторни функции**. Последните имат синтаксиса на обикновени функции, но името им се състои от запазената дума operator, последвана от мнемоничното означение на предефинираната операция. Когато предефинираната операция изисква достъп до компонентите на класове, декларирани като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на тези класове. В противен случай, т.е. когато операторната дефиниция е външна функция неприятел на тези класове тя няма достъп до въпросните компоненти. Позицията спрямо аргументите, приоритета и асоциативността на една операция не може да се променят при нейното предефиниране. Също така, броят на нейните аргументи не може да бъде променен и операторните функции не могат да имат аргументи по премълчаване.
В различни случаи операторните функции е най-добре да бъдат приятелски функции или функции-елементи.
Ако левият операнд на предефинираната операция трябва задължително да бъде обект на клас или псевдоним на обект на клас, тогава тя се предефинира с функция елемент. Например операциите за извикване на функция ‘()’, за достъп до елемент на масив ‘[]’, указателната операция ‘->’ и операцията за присвояване ‘=’ винаги се предефинират с функция елемент на клас.

Ако левият операнд трябва да бъде обект от друг клас или от вграден тип, тогава операцията не може да бъде предефинирана с функция-елемент и тя се предефинира с външна функция, която е приятел на класа.
Унарна операция може да се предефинира с помощта на нестатична функция елемент без аргументи или с външна функция-приятел с един аргумент. Този аргумент трябва да бъде или обект на класа или псевдоним на обект на класа.
Бинарна операция може да се предефинира с помощта на нестатична функция елемент с един аргумент или с външна функция-приятел с два аргумента. Единият от тези аргументи трябва да бъде или обект от класа или псевдоним на обект от класа.
Операциите = и унарната & могат да се използват с обекти от всеки клас без те да са предефинирани. По премълчаване, унарната операция &, приложена към обект от кой да е клас, връща адреса на обекта в паметта. По премълчаване, присвояване (=) може да се извършва между обекти от един и същ клас и то се свежда до побитово копиране на данните-елементи на класа. Такова копиране е опасно за класове с данни-елементи, които сочат към динамично разпределена памет. След извършване на такова присвояване за два различни обекта от такъв клас, тези два обекта ще сочат към една и съща област от паметта. Изпълнението на деструктора на кой да е от тези обекти ще доведе до освобождаването на тази памет. Ако после, обаче, чрез другия обект се извърши обръщение към сочената от него вече освободена памет, резултатът ще бъде неопределен.
Поради тази причина, операцията = за такива класове задължително се предефинира.
Възниква необходимостта от средства, които реализират функции и класове, зависещи от параметри, които задават типове данни и при конкретни приложения параметрите да се конкретизират. Такива средства са **шаблоните**. Те позволяват създаването на функции и класове, използващи неопределени типове данни за своите аргументи. Така шаблоните на класове позволяват да бъдат описвани обобщени типове данни. Шаблоните на класове най-често се използват за изграждане на общоцелеви класове-контейнери (стекове, опашки, списъци и др.).
Шаблон на функция се дефинира като обикновена функция, но със следните разлики: заглавието на функцията се предшества от запазената дума template, последвана от списъка от формалните параметри на шаблона, заграден в ‘<’, ‘>’ – списък от идентификатори, предшествани от запазената дума class и разделени със запетаи. Формалните параметри на шаблона означават всеки вграден или потребителски тип и могат да се използват при указването на типове на параметрите на функцията, при указването на типа на резултата на функцията или при указването на тип вътре в тялото на функцията (например, при дефинирани на локални променливи).

Използването на дефиниран шаблон на функция става чрез обръщение към обобщената функция, която шаблонът дефинира, с параметри от конкретен тип. При такова обръщение, компилаторът генерира шаблонна функция, като замества параметрите на шаблона с типовете на съответните фактически параметри.
Декларацията на шаблон на клас изглежда като традиционно описание на клас със следната разлика: предшества се от заглавие, което започва с ключовата дума template, последвана от списъка на формалните параметри на шаблона, ограден в '<' и '>'. Този списък има аналогичен вид както при шаблоните на функции. Формалните параметри на шаблона на класа означават типове (вградени или потребителски) и могат да се използват навсякъде в декларацията на класа – като типове за член-данни, като типове на параметри или тип на резултат на член-функции, като типове на локални променливи на член-функции и т.н. Всяка дефиниция на функция-елемент извън декларацията на класа трябва да се предшества от заглавието на шаблона на класа. Дефиницията е стандартна с тази разлика, че винаги когато в нея се използва името на класа (с изключение на случая, когато то е име на конструктор), то трябва да се задава след него списъка от имената на формалните параметри на шаблона, ограден в '<' и '>'.
За разлика от шаблоните на функции, при създаване на обект на шаблонен клас изрично трябва да се зададат конкретните типове, за които ще се създава този обект, като имената на конкретните типове са подредени в списък, ограден с '<' и '>' след името на класа в дефиницията на обекта. Например, нека имаме следната декларация на шаблон на клас:

```
template <class T, class S>
class Stack {
…
};
```

Тогава са валидни следните дефиниции на обекти:

```
Stack <int, double> obj1;
Stack <double, int> obj2;
```

и т.н.
В шаблоните на функции и класове има възможност да се използват и нетипови параметри – те се задават заедно с типовите параметри в списъка на формалните параметри на шаблона. Фактическите стойности на нетиповите параметри трябва да са константни изрази, тъй като те се обработват по време на компилация.

12. Обектно-ориентирано програмиране (C++): Наследяване и полиморфизъм.

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране. Като се използва механизмът на наследяването от съществуващ клас може

да се създаде нов клас. Класът, от който се създава, се нарича **базов клас**, класът, който се създава се нарича **производен клас**. Производният клас може да наследи компонентите на един или на няколко базови класа. В първия случай наследяването се нарича **просто**, във втория случай се нарича **множествено**.
Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. Всеки производен клас може от своя страна да е базов при създаване на нови производни класове. Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като базови класове, а всяка от останалите части да се дефинира като производен клас на тези базови класове. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.
Производните класове се дефинират като обикновени класове с единствената разлика, че след името на производния клас в заглавието се поставя символът ':', последван от списък от двойки [<атрибут_за_област>] <име_на_базов_клас>, разделени със запетаи. Този списък определя кои са базовите класове. Атрибутът за област е незадължителен и може да бъде public, private или protected. Той определя областта на наследените компоненти в производния клас. Ако атрибутът за област е пропуснат, подразбира се private. Множеството от компонентите на производния клас се състои от компонентите на всички негови базови класове, заедно със собствените компоненти на производния клас.
Атрибутът за област на базов клас в декларацията на производния клас управлява механизма на наследяване и определя какъв да бъде режимът на достъп до наследените членове.
При атрибут за област public, елементите от тип public на базовия клас се наследяват като елементи от тип public на производния клас и елементите от тип protected на базовия клас се наследяват като елементи от тип protected на производния клас.
При атрибут за област protected, елементите от тип public и protected на базовия клас се наследяват като елементи от тип protected на производния клас.
При атрибут за област private, елементите от тип public и protected на базовия клас се наследяват като елементи от тип private на производния клас.
И при трите вида наследявания, производният клас няма достъп до елементите от тип private на базовия клас.
Външна функция, която не е приятел на производния клас чрез обект от производния клас има достъп само до компонентите от тип public на производния клас – това означава, че такава функция чрез обект на производния клас може да има достъп до наследени компоненти от базовия клас само ако те са от тип public в базовия клас и атрибутът за област е public.
Да отбележим, че член-функциите на базов клас нямат достъп до каквито и да е компоненти на производния клас. Причината е, че

когато базовият клас се дефинира, не е ясно какви производни класове ще произхождат от него.
Функциите приятели на производния клас имат същите права за достъп както член-функциите на производния клас – това са достъп до всички собствени компоненти на производния клас и достъп до наследените компоненти от тип public и protected на базовия клас. Декларацията за приятелство не се наследява – функция приятел на базов клас не е автоматично приятел на производния клас (освен ако не е изрично декларирана като такава в производния клас).
Базовият и производният клас могат да притежават компоненти с еднакви имена. В този случай, производният клас ще притежава компоненти с еднакви имена. Обръщението към такава компонента чрез обект от производния клас или от член-функция на производния клас извиква декларираната в производния клас компонента, т.е. името на собствената компонента скрива името на наследената компонента. За да се използва покритата компонента, трябва да се укаже нейното пълно име: <име_на_клас>::<име_на_компонента>, където <име_на_клас> е името на базовия клас.
По-нататък под обикновен конструктор ще разбираме конструктор, различен от конструктора за присвояване. Обикновените конструктори, конструкторът за присвояване, операторната функция за присвояване и деструкторът са методи за които не важат общите правила за наследяване. Тези методи (с някои малки изключения) не се наследяват от производния клас. Причината е, че ако се наследят, те биха се грижили само за наследените компоненти на производния клас, но не и за неговите собствени компоненти.
Основен въпрос е как да се реализира инициализирането на наследената част на производния клас. Най-естествено е това да се направи от конструктора на производния клас, но от друга страна този конструктор няма достъп до private компонентите на базовия клас. Затова конструкторите на производния клас инициализират само собствената част на този клас, а наследената част се инициализира от конструктор на базовия клас – това се осъществява като в дефиницията на конструктора на производния клас се укаже обръщение към съответен конструктор на базовия клас. Основен момент е, че обръщенията към конструктор на базови класове се осъществява посредством инициализиращ списък. Този списък се записва в дефиницията на конструктора на производния клас след името на този конструктор и знак ‘:’. Списъкът се състои от обръщения към конструктори на базовите класове, разделени със запетаи. При просто наследяване в този списък присъства най-много едно обръщение към конструктор на единствения базов клас. При множествено наследяване в списъка са указани няколко обръщения към конструктори на базови класове, най-много по едно обръщение към конструктор за даден базов клас. Имената на параметрите на

конструктора на производния клас могат да се използват като фактически параметри в обръщенията към конструкторите на базовите класове в инициализиращия списък.
Дефинирането на обект от производен клас предизвиква създаване на неявни обекти от базовите класове и добавяне към тях на декларираните в производния клас компоненти. Това означава, че първо се извикват конструкторите на базовите класове, а след това се извиква конструкторът на производния клас. Редът, в който се извикват конструкторите на базовите класове, е редът, в който тези класове са посочени в заглавието на производния клас. На този ред не влияе последователността, в която са посочени конструкторите на базовите класове в инициализиращия списък в дефиницията на конструктора на производния клас. Ако производният клас има член-данни, които са обекти, техните конструктори се извикват след изпълнението на конструкторите на базовите класове и преди изпълнението на тялото на конструктора на производния клас. Редът, в който се изпълняват конструкторите на обектите-елементи е редът, в който те са декларирани в тялото на производния клас.
Ако за базовия клас не е дефиниран конструктор, то за него се създава служебен конструктор по премълчаване, но независимо от това наследената част на производния клас остава неинициализирана. В този случай, в инициализиращия списък в дефиницията на конструктора на производния клас не се извършва обръщение към конструктора на този базов клас.
Ако за базовия клас е дефиниран поне един конструктор с параметри и за този клас не е дефиниран конструктор по премълчаване, то за производния клас задължително трябва да е дефиниран конструктор, който в своя инициализиращ списък извършва обръщение към един от дефинираните конструктори на въпросния базов клас. Ако това не е изпълнено, компилаторът съобщава за грешка.
Последният случай е когато за базовия клас е дефиниран поне един конструктор, включително конструктор по премълчаване. Тогава, ако в производния клас е дефиниран конструктор, то в неговия инициализиращ списък може да присъства обръщение към конструктора на базовия клас или да не присъства. Във втория случай се извиква конструкторът по премълчаване на базовия клас. Ако в производния клас не е дефиниран конструктор, тогава за него се създава служебен конструктор по премълчаване, който от своя страна извиква конструкторът по премълчаване на базовия клас. В последния случай, собствените членове на производния клас остават неинициализирани.
Деструкторите на един производен клас и на неговите базови класове се изпълняват в ред, обратен на реда на изпълнение на съответните конструктори. Най-напред се изпълнява деструкторът на производния клас, след това деструкторите на обектите-елементи на производния клас и най-накрая деструкторите на базовите класове.

С някои малки изключения, производният клас не наследява от базовия клас конструктора за присвояване и операторната функция за присвояване.
При конструкторите за присвояване се спазва същият принцип както при обикновените конструктори на производния и базовия клас. Конструкторът за присвояване на производния клас инициализира собствените член-данни на производния клас, а конструкторът за присвояване на базовия клас инициализира наследените член-данни.
Нека в производния клас не е дефиниран конструктор за присвояване, но в базовия клас има такъв. Тогава, за производния клас се създава служебен конструктор за присвояване, който от своя страна извиква конструктора за присвояване на базовия клас. Да отбележим, че при обикновените конструктори този случай ще предизвика грешка, ако за базовия клас не е дефиниран конструктор по премълчаване. Затова в случая се казва, че производният клас наследява конструкторът за присвояване от базовия клас.
Нека в производния клас и в базовия клас няма дефинирани конструктори за присвояване. Тогава и за двата класа се създават служебни конструктори за присвояване, като този на производния клас извиква този на базовия клас.
Нека в производния клас е дефиниран конструктор за присвояване. В неговия инициализиращ списък може да има, но може и да няма обръщение към конструктор (обикновен или за присвояване) на базовия клас. Препоръчва се в инициализиращия списък на конструктора за присвояване на производния клас да има обръщение към конструктора за присвояване на базовия клас, ако такъв е дефиниран. При това, фактическият параметър на това обръщение може да съвпада с фактическия параметър на обръщението към конструктора за присвояване на производния клас. Това е позволено, тъй като в случая обекта на производния клас може да се разглежда като обект на базовия клас (при работа с обекти във външни функции, това може да се счита само при наследяване от тип public). Ако в инициализиращия списък на конструктора за присвояване на производния клас не е указано обръщение към конструктор на базовия клас, то се извиква конструкторът по премълчаване на базовия клас. Ако базовия клас няма такъв конструктор, компилаторът ще съобщи за грешка.
Операторната функция за присвояване на производен клас трябва да указва как да става присвояването както на собствените, така и на наследените си член-данни. За разлика от конструкторите на производни класове, тя прави това в своето тяло (не поддържа инициализиращ списък).
Нека в производния клас не е дефинирана операторна функция за присвояване. Тогава компилаторът създава служебна такава. Тя се обръща към операторната функция за присвояване на базовия клас, чрез която инициализира наследената част, след това инициализира чрез обикновено присвояване и собствената част на

производния клас. Затова в този случай се казва, че операторът за присвояване на базовия клас се наследява, т.е. за наследените член-данни се използва служебният или предефинираният оператор за присвояване на базовия клас.
Нека в производния клас е дефинирана операторна функция за присвояване. Тогава тази функция трябва да се погрижи за присвояването на наследените член-данни. Налага се в нейното тяло да има обръщение към операторната функция за присвояване на базовия клас, ако има такава. Ако това обръщение не е направено явно, то стандартът на езика не уточнява как ще стане присвояването на наследените компоненти.
В случаите когато производният клас наследява повече от един базов клас се казва, че класът е с множествено наследяване. Този вид наследяване е мощен инструмент на обектно-ориентираното програмиране, тъй като чрез него се изграждат графовидни йерархични структури. Имената на базовите класове се задават в заглавието на производния клас след името му и символът ':', разделят се със запетаи, при това всеки от тях се предшества или не от съответен атрибут за област. За член-функциите от голямата четворка на производен клас с множествено наследяване са в сила същите правила, както при производен клас с просто наследяване – тези правила се прилагат независимо към всеки един базов клас.
Изпълнението на конструктор на производен клас с множествено наследяване става така: първо се извикват конструкторите на всички базови класове, в реда, указан в заглавието на производния клас – кой конструктор ще се извика зависи от това дали присъства или не обръщение в инициализиращия списък на конструктора на производния клас, второ се извикват конструкторите на собствените обекти-елементи на производния клас, в реда, в който те са описани в тялото на класа и най-накрая се изпълнява тялото на конструктора на производния клас.
Извикването на деструкторите става в ред, обратен на реда на съответните конструктори. Препоръчва се конструкторът за присвояване (ако е дефиниран) на производния клас да извършва обръщения чрез инициализиращия си списък към конструкторите за присвояване на всички базови класове. Операторната функция за присвояване на производен клас с множествено наследяване обикновено има следния вид:

```
<производен_клас>& <производен_клас>::operator= (const
<производен_клас>&r) {
        if (this != &r)
          { <базов_клас_1>::operator= (r);
            <базов_клас_2>::operator= (r);
            …
            <базов_клас_N>::operator= (r);
            Del();
            Copy (r);
          }
         return *this;
```

}
Тук функцията Del изтрива собствените компоненти на подразбиращия се обект, а функцията Copy (r) копира в подразбиращия се обект компонентите на обекта r, съответни на собствените член-данни на производния клас. Извършва се проверка за самоприсвояване и функцията връща псевдоним на обекта от лявата страна на присвояването, което позволява слепване на няколко присвоявания.
Виртуалните базови класове са механизъм за отмяна на стандартния наследствен механизъм. При реализиране на йерархии от класове с множествено наследяване е възможно един производен клас да наследи повече от един път даден базов клас. Например, възможно е A да е базов клас за B и C, които от своя страна са базови класове за класа D. Като производни на класа A, класовете B и C наследяват неговите компоненти. От друга страна, класовете B и C са базови за класа D, следователно класът D ще наследи два пъти компонентите на A – веднъж чрез класа B и веднъж чрез класа C. За член-функциите двойното наследяване не е от значение, тъй като за всяка член-функция се съхранява само едно копие. Член-данните, обаче, се дублират и обект на D ще наследи двукратно всяка член-данна, дефинирана в класа A. Многократното наследяване на клас води от една страна до затруднен достъп до многократно наследените членове, а от друга страна до поддържане на множество копия на член-данните на многократно наследения клас, което е неефективно.
Преодоляването на тези недостатъци се осъществява чрез използването на виртуални базови класове. Чрез тях се дава възможност да се поделят базови класове. Когато един клас е виртуален, независимо от многократното му участие като базов клас (пряк или косвен), се създава само едно негово копие.
В нашия пример, ако класът A се определи като виртуален за класовете B и C, класът D ще съдържа само един поделен базов клас A. Декларацията на базов клас като виртуален се осъществява като в декларацията на производния клас заедно с името и атрибута за област на базовия клас се укаже и запазената дума virtual. Виртуалното наследяване на един клас указва влияние само на наследниците на неговите непосредствени производни класове. То не указва влияние на тези непосредствени наследници. Дефинирането и използването на виртуални класове има редица особености. Една от тях касае дефинирането и използването на конструкторите на наследените класове. Нека A е виртуален базов клас за класа B, класът B е базов за класа D, който пък от своя страна е базов за класа E. Ако класът A има конструктор с параметри и няма конструктор по премълчаване, то този конструктор трябва да се извика в инициализиращия списък не само на класа B, но и на конструкторите D и E. С други думи, конструкторите с параметри на виртуални класове трябва да се извикват от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на непосредствените им

наследници. Друга особеност е промяната на реда на инициализиране. Инициализирането на виртуалните базови класове предхожда инициализирането на невиртуалните базови класове. Има и още едно уточнение. Нека класът A е базов виртуален клас за класовете B и C, които от своя страна са базови за класа D. Възможно е атрибутът за област на класа A да е public за класа B и private за класа C. Очевидно ситуацията е неоднозначна – външна функция чрез обект на D няма достъп до компонентите на A по пътя A-C-D, но има достъп по пътя A-B-D. Тази неоднозначност се преодолява с избора, че ако в някоя декларация виртуалният клас е наследен като public, счита се, че този клас се наследява като public навсякъде.
Възможно е да се използват функции с еднакви имена, в това число и методи на класове – така наречението **предефиниране** на функции. В случая на обикновени функции и собствени член-функции на един и същ клас при извикване на предефинирана функция, кой от нейните варианти се извиква се определя по следния механизъм: по време на компилация се сравняват формалните с фактическите параметри в обръщението и по правилото за най-доброто съвпадане се избира необходимата функция. При член-функциите на йерархия от класове, конфликтът между имената на наследените и собствените методи се разрешава също по време на компилация чрез правилото на локалния приоритет или чрез явно посочване на класа, към който принадлежи метода.
В тези два случая тъй като процесът на определяне на реализиране на обръщението към функцията приключва по време на компилация и не може да бъде променян по време на изпълнение на програмата се казва, че има **статично разрешаване на връзката** или **статично свързване**. Ще разгледаме пример, с дървовидна йерархия от три класа – клас Point2 за точка в равнината, клас Point3 за точка в пространството, който наследява пряко Point2 и клас ColPoint3 за оцветена точка в пространството, който наследява пряко Point3.

```
#include <iostream.h>
class Point2 {
  public:
        Point2 (int a = 0, int b = 0)
         { x = a; y = b;}
        void Print() const
         { cout << x << “, “ << y; }
  private:
        int x, y;
};
class Point3 : public Point2 {
  public:
        Point3 (int a = 0, int b = 0, int c = 0) : Point2 (a, b)
         { z = c; }
        void Print () const
```

```
        { Point2::Print();
          cout << “, “ << z;
        }
  private:
        int z;
};
class ColPoint3 : public Point3 {
  public:
        ColPoint3 (int a = 0, int b = 0, int c = 0, int o = 0)
        : Point3 (a, b, c)
        { col = o; }
        void Print() const
        { Point3::Print();
          cout << “colour: “ << col;
        }
  private:
        int col;
};
```

Сега ще дефинираме примерна функция main.

```
void main ()
 { Point2 p2(5, 10);
   Point3 p3(2, 4, 6);
   ColPoint3 p4 (12, 24, 36, 11);
   Point2 *ptr1 = &p3;
   ptr1 -> Print();
   cout << endl;
   Point2 *ptr2 = &p4;
   ptr2 -> Print();
   cout << endl;
  }
```

Резултатът от изпълнението на тази функция е:

2 4

12 24

Изрично ще отбележим, че инициализирането на указателите, които са от типа на базовия клас чрез адреси на обекти от производните класове е възможно, тъй като наследяването и при двата производни класа е от тип public.

И в трите класа е дефинирана функция Print без параметри от тип void. Обръщението ptr1 -> Print(); извежда първите две координати на точката p3, а обръщението ptr2 -> Print(); извежда първите две координати на точката p4, т.е. изпълнява се метода Print() на класа Point2 и в двата случая. Още по време на компилация, член-функцията Print() на Point2 е определена като функция на двете обръщения. Определянето става от типа на двата указателя – той е Point2. Връзката е определена статично и не може да се промени по време на изпълнение на програмата. Ако искаме след свързването на ptr1 с адреса на p3 да се изпълни член-функцията Print() на Point3, а също след свързването на ptr2

с адреса на p4 да се изпълни член-функцията Print() на ColPoint3, са небходими явни преобразувания от следния вид:
((Point3 *)ptr1) -> Print();
((ColPoint3 *)ptr2) -> Print();
В този случай връзките отново са разрешени статично.
При статичното свързване по време на създаването на класа трябва да се предвидят възможните обекти, чрез които ще се викат неговите член-функции. При сложни йерархии от класове това е не само трудно, но понякога и невъзможно. Езикът C++ поддържа още един механизъм, прилаган върху специален вид член-функции, наречен **динамично свързване**. При него изборът на функцията, която трябва да се изпълни, става по време на изпълнение на програмата. Динамичното свързване капсулира детайлите в реализацията на йерархията. При него не се налага проверка на типа. Разширяването на йерархията не създава проблеми. Това обаче е с цената на забавяне на процеса на изпълнение на програмата. Прилагането на механизма на динамичното свързване се осъществява върху специални член-функции на класове, наречени **виртуални член-функции** или само **виртуални функции**. Виртуалните методи се декларират чрез поставяне на запазената дума virtual пред декларацията им, т.е. virtual <тип_на_резултата> <име_на_метод> (параметри);
Да предположим, че е в горния пример член-функцията Print() и в трите класа е декларирана като виртуална. Тогава при обръщенията ptr1 -> Print(); и ptr2 -> Print(); коя функция ще бъде извикана се определя по време на изпълнение на програмата. Определянето е в зависимост от типа на обекта, към който сочи указателя, а не от класа към който е указателя. В случая, указателят ptr1 е към класа Point2, но сочи обекта p3, който е от класа Point3 и затова обръщението ptr1 -> Print(); води до изпълнение на функцита Print() от класа Point3. Аналогично, указателят ptr2 е към класа Point2, но сочи обекта p4, който е от класа ColPoint3 и затова обръщението ptr2 -> Print(); води до изпълнение на функцията Print() от класа ColPoint3.
Ще отбележим някои важни особености на виртуалните функции:

1. Само член-функциите на класове могат да се декларират като виртуални. По технически причини конструкторите не могат да се дефинират като виртуални, но е възможно да има виртуални деструктори.
2. Ако в даден клас е декларирана виртуална функция, декларираните член-функции със същия прототип (име, параметри и тип) в производните на класа класове автоматично стават виртуални, дори ако запазената дума virtual не присъства в тяхните декларации.
3. Ако в производен клас е дефинирана функция със същото име като определена вече в базов клас като виртуална член-функция, но с други параметри или тип, това ще бъде друга функция, която може да бъде или да не бъде декларирана като виртуална.

4. Възможно е виртуална функция да се дефинира извън клас. Запазената дума virtual обаче присъства само в декларацията на функцията в тялото на класа, но не и в нейната дефиниция.
5. Виртуалните функции се наследяват като останалите компоненти на класовете.
6. Виртуалната функция, която в действителност се изпълнява зависи от типа на аргумента.
7. Виртуалните функции не могат да бъдат декларирани като приятели на други класове.

Всяка член-функция на клас, в който е дефинирана виртуална функция има пряк достъп до виртуалната функция, т.е. на локално ниво достъпът се определя по традиционните правила. На глобално ниво достъпът се определя от вида на секцията, в която е дефинирана виртуалната функция на класа към който сочи указателят, чрез който се активира функцията, а не от вида на секцията, в която е предефинирана виртуалната функция на класа, към който принадлежи обекта, сочен от въпросния указател.

Съществуват три случая при които обръщението към виртуална функция се разрешава статично (по време на компилация):

1. Виртуалната функция се извиква чрез обект на класа, в който е дефинирана.
2. Виртуалната функция се извиква чрез указател към обект, но явно, чрез бинарната операция за разрешаване на достъп :: е посочена конкретната функция.
3. Виртуалната функция се активира с тялото на конструктор или деструктор на базов клас.

Основно предимство на виртуалните функции е, че чрез тях могат да се реализират полиморфни действия. **Полиморфизмът** е важна характеристика на обектно-ориентираното програмиране и тя се изразява в това, че едни и същи действия се реализират по различни начини в зависимост от обектите, към които се прилагат, т.е. действията са полиморфни (с много форми). Полиморфизмът е свойство на член-функциите на обектите и в C++ се реализира чрез виртуални функции. За да се реализира полиморфно действие, класовете върху които то ще се прилага трябва да имат общ родител или прародител, т.е. да бъдат производни на един и същи базов клас. В този клас трябва да бъде дефиниран виртуален метод, съответстващ на полиморфното действие. Във всеки от производните класове този метод може да бъде предефиниран съобразно особеностите на този клас. Активирането на полиморфното действие става чрез указател към базовия клас, на който могат да се присвоят адресите на обекти на който и да е от производните класове от йерархията. Ще бъде изпълнен методът на съответния обект, т.е. в зависимост от обекта, към който сочи указателят ще бъде изпълняван един или друг метод. Ако класовете, в които трябва да се дефинират виртуални методи

нямат общ родител, такъв може да бъде създаден изкуствено чрез дефиниране на така наречен абстрактен клас.
Възможно е виртуалните функции да имат само декларация без да имат дефиниция. Такива виртуални член-функции се наричат **чисти**. За да се определи една виртуална функция като чиста се използва следния синтаксис:
virtual <тип><име_на_функция>(<параметри>) = 0;
Клас, който съдържа поне една чисто виртуална функция се нарича **абстрактен клас**. Основното свойство на абстрактните класове е, че не е възможно от тях да се създават обекти. Чистите виртуални функции могат да се предефинират в производните класове със същите прототипи, а може и да не се предефинират. Когато производния клас наследи чисто виртуална функция без да я предефинира, той също става абстрактен. В случай, че производния клас предефинира всички чисто виртуални функции на своя базов клас, той става **конкретен** и от него могат да се създават обекти.
Абстрактните класове са предназначени да служат като базови на други класове. Чрез тях се обединяват в обща структура различни йерархии. Обикновено, абстрактните базови класове определят интерфейса за различни типове обекти в йерархията на класовете. Всички обработки в йерархията могат да прилагат един и същ интерфейс, използвайки полиморфизъм – дефинират се указатели от абстрактния базов клас и след това те се използват за полиморфно опериране с обектите на производните конкретни класове.

13. Структури от данни. Стек, опашка, списък, дърво. Основни операции върху тях. Реализация.

Под **структура от данни** се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на програма. За да се определи една структура от данни е необходимо да се направи:

- логическо описание на структурата, което я описва на базата на декомпозицията и на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции;
- физическо представяне на структурата, което дава метода за представяне на структурата в паметта на компютъра.

Стекът е линейна динамична структура от данни. Стекът е крайна редица от елементи от един и същи тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича **връх** на стека. Възможен е пряк достъп само до елемента, който се намира във върха на стека. При тази организация на логическите операции, последният включен елемент се изключва пръв. Затова стекът се определя още като структура “последен влязъл – пръв излязъл”

(last in – first out, LIFO).
Широко се използват два основни начина за представяне на стек: **последователно** и **свързвано**. При последователното представяне, предварително в паметта се запазва блок, вътре в който се помества стекът и той там расте и се съкращава. При включване на елементи в стека, те се поместват в последователни адреси в неизползваната част веднага след върха на стека.
При свързаното представяне последователните елементи се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в стека, стойността на този указател трябва да бъде някаква различима стойност (например NULL).
За задаване на стека е достатъчен указател към върха на стека.
В общия случай елементите на стека при свързаното представяне се състоят от две полета – inf (данните, записани в елемента) и link (указател към следващия елемент).
Сега ще дефинираме клас, който реализира свързаното представяне на стек.
Първо ще дефинираме помощен клас Item, който реализира двойната кутия, чрез която се представят елементите на стека.
За по-голяма общност, дефинираме класовете като шаблони.

```
template <class T> class Stack;
template <class T>
class Item {
  friend class Stack <T>;
  private:
     Item (const T& x = 0)
      { inf = x;
        link = NULL;
      }
   T inf;
   Item *link;
};
```

Тъй като класът Item използва класът Stack в декларацията си, затова прототипът на класа Stack предшества декларацията на класа Item. Всички членове на Item са капсулирани. Чрез декларацията за приятелство, класът Item позволява само на класа Stack да създава и обработва негови обекти. Сега вече сме готови да дефинираме класът Stack. Тъй като стекът се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
class Stack {
 public:
  Stack (const T&);
  Stack ();
```

```
  ~Stack ();
  Stack (const Stack &);
  Stack& operator= (const Stack &);
  void push (const T&);
  bool pop (T&);
  bool top(T&) const;
  bool empty () const;
private:
  Item<T> *start;
  void delStack();
  void copyStack (const Stack &);
};
template <class T>
Stack<T>::Stack (const T& x)
{ start = new Item<T> (x); }
template <class T>
Stack<T>::Stack ()
{ start = NULL; }
template <class T>
Stack<T>::~Stack ()
{ delStack(); }
template <class T>
Stack<T>::Stack (const Stack<T> &r)
{ copyStack(r); }
template <class T>
Stack<T>& operator= (const Stack<T> &r)
{ if (this != &r)
   { delStack ();
     copyStack (r);
   }
  return *this;
}
template <class T>
void Stack<T>::delStack ()
 {  Item<T> *p;
    while (start)
     { p = start;
       start = start -> link;
       delete p;
     }
  }
template <class T>
void Stack<T>::copyStack (const Stack<T> &r)
{ if (!r.start) start = NULL;
  else {
     Item<T> *p = r.start, *q, *s;
     start = new Item<T>(p -> inf);
     q = start;
     p = p -> link;
```

```
    while (p)
      { s = new Item<T> (p -> inf);
        q -> link = s;
        q = s;
        p = p -> link;
      }
  }
}
template <class T>
void Stack<T>::push (const T& x)
{ Item<T> *p = new Item<T>(x);
  p -> link = start;
  start = p;
}
template <class T>
bool Stack<T>::pop (T& x)
 { if (!start) return false;
   Item<T> *p = start;
   x = start -> inf;
   start = start -> link;
   delete p;
   return true;
}
template <class T>
bool Stack<T>::top (T& x) const
{  if (!start) return false;
   x = start -> inf;
   return true;
}
template <class T>
bool Stack<T>::empty () const
{ return start == NULL; }
```

Опашката е крайна редица от елементи от един и същи тип. Операцията включване е допустима за елементите от единия край на редицата, който се нарича **край** на опашката, а операцията изключване на елемент – само за елементите от другия край на редицата, който се нарича **начало** на опашката. Възможен е пряк достъп само до елемента, намиращ се в началото на опашката. При тази организация на логическите операции, пръв се изключва най-отдавна включеният елемент. Затова опашката се определя още като структура от данни "пръв влязъл – пръв излязъл" (first in – first out, FIFO).
Широко се използват два основни начина за физическо представяне на опашка: **последователно** и **свързано**.
При последователното представяне първоначално в паметта се запазва блок, вътре в който опашката да расте и да се съкращава. Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част

веднага след края на опашката. Обикновено се счита, че блокът от памет е цикличен – когато краят на опашката достигне края на разпределения блок памет, но има освободена памет в неговото начало, там може да се извърши включване на елементи.
При свързаното представяне последователните елементи на опашката се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в опашката, стойността на този указател трябва да бъде някаква различима стойност (например NULL). За задаване на опашката са достатъчни указатели към началото и края на опашката. В общия случай елементите на опашката при свързаното представяне се състоят от две полета – inf (данните, записани в елемента) и link (указател към следващия елемент). Сега ще дефинираме клас, който реализира свързаното представяне на опашка.
Първо ще дефинираме помощен клас Item, който реализира двойната кутия, чрез която се представят елементите на опашката.
За по-голяма общност, дефинираме класовете като шаблони.

```
template <class T> class Queue;
template <class T>
class Item {
  friend class Queue <T>;
  private:
     Item (const T& x = 0)
      { inf = x;
        link = NULL;
      }
   T inf;
   Item *link;
};
```

Тъй като класът Item използва класът Queue в декларацията си, затова прототипът на класа Queue предшества декларацията на класа Item. Всички членове на Item са капсулирани. Чрез декларацията за приятелство, класът Item позволява само на класа Queue да създава и обработва негови обекти. Сега вече сме готови да дефинираме класът Queue. Тъй като опашката се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
class Queue {
 public:
   Queue(const T&);
   Queue();
   ~Queue();
   Queue (const Queue &);
```

```
   Queue& operator= (const Queue &) ;
   void InsertElem (const T&);
   bool DeleteElem (T &);
   bool ViewElem(T &) const;
   bool empty() const;
 private:
   Item<T> *front, *rear;
   void delQueue();
   void copyQueue (const Queue &);
} ;
template <class T>
Queue<T>::Queue(const T& x)
{ front = new Item<T>(x);
  rear = front;
}
template <class T>
Queue<T>::Queue() {
 front = rear = NULL;
}
template <class T>
Queue<T>::~Queue() {
 delQueue();
}
template <class T>
Queue<T>::Queue (const Queue<T> &r)
{ copyQueue (r); }
template <class T>
Queue<T>& Queue<T>::operator= (const Queue<T> &r)
{ if (this != &r)
   { delQueue();
     copyQueue(r);
   }
  return *this;
}
template <class T>
void Queue<T>::delQueue()
 { T x;
   while (DeleteElem(x));
 }
template <class T>
void Queue<T>::copyQueue(const Queue<T> &r)
{  front = rear = NULL;
   if (r.front) {
      Item <T> *p = r.front;
      while (p)
       { InsertElem (p -> inf);
         p = p -> link;
       }
   }
```

```
}
template <class T>
void Queue<T>::InsertElem (const T& x)
{ Item <T> *p = new Item<T>(x);
  if (front) { rear->link = p; rear = p; }
  else front = rear = p;
}
template <class T>
bool Queue<T>::DeleteElem (T &x)
{ if (!front) return false;
  Item <T> *p = front;
  x = p -> inf;
  if (front == rear) front = rear = NULL;
  else front = front -> link;
  delete p;
  return true;
}
template <class T>
bool Queue<T>::ViewElem(T &x) const
{ if (!front) return false;
  x = front -> inf;
  return true;
}
template <class T>
bool Queue<T>::empty() const
{ return front == NULL; }
```

Списъкът е крайна редица от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими на произволно място в редицата. Възможен е достъп до всеки елемент на редицата (пряк или непряк в зависимост от реализацията на списъка).

Има два основни начина за представяне на списък в паметта на компютъра: **свързано** представяне **с една връзка**, **свързано** представяне **с две връзки** и **последователно** представяне.

При последователното представяне списъкът се представя чрез масив, т.е. елементите на списъка са последователно разположени в паметта, но това представяне не се използва, тъй като включването и изключването на елемент е неефективно и освен това заради добре развитите средства за динамично разпределение на паметта.

При свързаното представяне с една връзка последователните елементи на списъка се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в списъка, стойността на този указател трябва да бъде някаква различима стойност (например NULL). За задаване на списъка е достатъчен указател към неговото начало. За удобство при реализирането на

операциите включване, изключване и обхождане се въвеждат още указатели към края и към текущ елемент на списъка. В общия случай елементите на списъка при свързаното представяне с една връзка се състоят от две полета – inf (данните, записани в елемента) и link (указател към следващия елемент).
При свързаното представяне с две връзки последователните елементи на списъка се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзките между отделните елементи се осъществяват чрез два указателя към следващия и към предхождащия елемент. Ако елементът е последен в списъка, стойността на указателя към следващия елемент трябва да бъде някаква различима стойност (например NULL), аналогично за указателя към предхождащия елемент за елемента в началото на списъка. За задаване на списъка е достатъчен указател към неговото начало. За удобство при реализирането на операциите включване, изключване и обхождане се въвеждат още указатели към края и указател към текущ елемент на списъка. В общия случай елементите на списъка при свързаното представяне с две връзки се състоят от три полета – inf (данните, записани в елемента), next (указател към следващия елемент) и prev (указател към предхождащия елемент).

Сега ще дефинираме клас, който реализира свързаното представяне на списък с една връзка.
Първо ще дефинираме помощна структура Item, която реализира двойната кутия, чрез която се представят елементите на списъка. За по-голяма общност, дефинираме структурата Item и класа LList като шаблони.

```
template <class T>
struct Item {
   T inf;
   Item *link;
};
```

Сега вече сме готови да дефинираме класът LList. Тъй като едносвързаният списък се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
class LList {
 public:
   LList(const T&);
   LList();
   ~LList();
   LList(const LList &);
   LList& operator= (const LList &);
   void IterStart (Item <T>* = NULL);
   Item <T>* Iter();
   bool IterView(T &) const;
```

```
   void InsertToEnd (const T&);
   void InsertToBegin (const T&);
   void InsertAfter (Item <T>*, const T&);
   void InsertBefore (Item <T>*, const T&);
   void DeleteAfter (Item <T>*, T &);
   void DeleteBefore (Item <T>*, T &);
   void DeleteElement (Item <T>*, T &);
   int length() const;
   void print() const;
   void concat (const LList &);
   void reverse();
   bool empty() const;
 private:
   Item<T> *start, *end, *current;
   void delLList();
   void copyLList(const LList &);
};
template <class T>
LList<T>::LList (const T& x)
 { start = new Item<T> ;
   start -> inf = x;
   start -> link = NULL;
   end = start;
 }
template <class T>
LList<T>::LList()
{ start = end = NULL; }
template <class T>
LList<T>::~LList()
{ delLList(); }
template <class T>
LList<T>::LList (const LList<T> &r)
{ copyLList (r); }
template <class T>
LList<T>& LList<T>::operator= (const LList<T> &r)
{ if (this != &r)
   { delLList();
     copyLList(r);
   }
 return *this;
}
template <class T>
void LList<T>::delLList()
{ Item <T> *p;
  while (start)
   { p = start;
     start = start -> link;
     delete p;
   }
```

```
  end = NULL;
}
template <class T>
void LList<T>::copyLList(const LList<T> &r)
{ start = end = NULL;
  Item <T> *p = r.start;
  while (p)
   { InsertToEnd (p -> inf);
     p = p -> link;
   }
}
template <class T>
void LList<T>::IterStart (Item <T> *p)
{ if (p) current = p;
  else current = start;
}
template <class T>
Item <T>* LList<T>::Iter()
{ Item <T> *p = current;
  if (current) current = current -> link;
  return p;
}
template <class T>
bool LList<T>::IterView(T &x)
{ if (!current) return false;
  x = current -> inf;
  return true;
}
template <class T>
void LList<T>::InsertToEnd(const T& x)
{ Item <T> *p = new Item<T>;
  p -> inf = x;
  p -> link = NULL;
  if (!start) start = end = p;
  else { end -> link = p; end = p; }
}
template <class T>
void LList<T>::InsertToBegin(const T& x)
{ Item <T> *p = new Item<T>;
  p -> inf = x;
  p -> link = start;
  start = p;
  if (!end) end = p;
}
template <class T>
void LList<T>::InsertAfter (Item <T> *p, const T& x)
{ Item <T> *q = new Item<T>;
  q -> inf = x;
  q -> link = p -> link;
```

```
  p -> link = q;
  if (p == end) end = q;
}
template <class T>
void LList<T>::InsertBefore (Item <T> *p, const T& x)
{ Item <T> *q = new Item<T>;
  *q = *p;
  p -> inf = x;
  p -> link = q;
  if (end == p) end = q;
}
template <class T>
void LList<T>::DeleteBefore (Item <T> *p, T &x)
{ Item <T> *q = start;
  if (q -> link == p)
   { x = q -> inf;
     start = start -> link;
     delete q;
   }
  else {
   while (q -> link -> link != p) q = q -> link;
   DeleteAfter (q, x);
  }
}
template <class T>
void LList<T>::DeleteAfter (Item <T> *p, T &x)
{ Item <T> *q = p -> link;
  x = q -> inf;
  p -> link = q -> link;
  if (end == q) end = p;
  delete q;
}
template <class T>
void LList<T>::DeleteElement (Item <T> *p, T &x)
{ if (p == start)
   { x = p -> inf;
     if (start == end) start = end = NULL;
     else start = start -> link;
     delete p;
    }
  else if (p == end) {
    Item <T> *q = start;
    while (q -> link != end) q = q -> link;
    q -> link = NULL;
    delete end;
    end = q;
   }
  else DeleteBefore(p -> link, x);
}
```

```
template <class T>
void LList<T>::print () const
{ Item <T> *p = start;
  while (p)
   { cout << p -> inf << ' ';
     p = p -> link;
   }
  cout << endl;
}
template <class T>
int LList<T>::length () const
{ Item <T> *p = start;
  int n = 0;
  while (p)
   { n++; p = p -> link; }
  return n;
}
template <class T>
void LList<T>::concat (const LList<T> &r)
{ Item <T> *p = r.start;
  while (p)
    { InsertToEnd (p -> inf);
      p = p -> link;
    }
}
template <class T>
void LList<T>::reverse ()
{ Item <T> *p, *q, *temp;
  if (start == end) return;
  p = start;
  q = NULL;
  start = end;
  end = p;
  while (p)
   { temp = p -> link;
     p -> link = q;
     q = p;
     p = temp;
   }
}
template <class T>
bool LList<T>::empty () const
{ return start == NULL; }
```

Функциите Iter, IterStart и IterView реализират т.н. **итератор**. Итераторът е указател към текущ елемент на редица. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:

++ - приложена към итератор, премества този итератор да сочи към следващия елемент на редицата;

-- - приложена към итератор, премества този итератор да сочи към предишния елемент на редицата;
* - приложена към итератор, дава елемента, към който сочи итератора.
В шаблона LList, итераторът е current. Функцията IterStart инициализира итератора да сочи към началото на списъка, ако за нея не е указан аргумент или е указан аргумент NULL. Ако за нея е указан друг аргумент, той е началната стойност на итератора. Операцията ++ се реализира от функцията Iter, която между другото връща стария итератор. Операцията -- не е реализирана, тъй като списъкът е само с една връзка и тя би била неефективна. Операцията * се реализира от функцията IterView.
Сега ще дефинираме клас, който реализира свързаното представяне на списък с две връзки.
Първо ще дефинираме помощна структура Item, която реализира тройната кутия, чрез която се представят елементите на списъка. За по-голяма общност, дефинираме структурата Item и класа DLList като шаблони.

```
template <class T>
struct Item {
   T inf;
   Item *pred;
   Item *succ;
};
```

Сега вече сме готови да дефинираме класът DLList. Тъй като двусвързаният списък се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
class DLList {
 public:
   DLList(const T&);
   DLList();
   ~DLList();
   DLList(const DLList &);
   DLList& operator= (const DLList &);
   void IterStart (Item <T>* = NULL);
   void IterEnd (Item <T>* = NULL);
   Item <T>* IterPred();
   Item <T>* IterSucc();
   bool IterView(T &) const;
   void InsertToEnd (const T&);
   void InsertToBegin (const T&);
   void InsertAfter (Item <T>*, const T&);
   void InsertBefore (Item <T>*, const T&);
   void DeleteAfter (Item <T>*, T &);
   void DeleteBefore (Item <T>*, T &);
   void DeleteElement (Item <T>*, T &);
```

```
   int length() const;
   void print() const;
   void concat (const DLList &);
   void reverse();
   bool empty () const;
 private:
   Item<T> *start, *end, *current;
   void delDLList();
   void copyDLList(const DLList &);
};
template <class T>
DLList<T>::DLList (const T &x)
{ start = new Item <T>;
  start -> inf = x;
  start -> pred = start -> succ = NULL;
  end = start;
}
template <class T>
DLList<T>::DLList()
{ start = end = NULL; }
template <class T>
DLList<T>::~DLList()
{ delDLList(); }
template <class T>
DLList<T>::DLList (const DLList <T> &r)
{ copyDLList(r); }
template <class T>
DLList<T>& DLList<T>::operator= (const DLList<T> &r)
{ if (this != &r)
   { delDLList();
     copyDLList(r);
   }
  return *this;
}
template <class T>
void DLList<T>::delDLList()
{ Item <T> *p;
  while (start)
   { p = start;
     start = start -> succ;
     delete p;
   }
  end = NULL;
}
template <class T>
void DLList<T>::copyDLList (const DLList<T> &r)
{ start = end = NULL;
  Item <T> *p = r.start;
  while (p)
```

```
   { InsertToBegin(p -> inf);
     p = p -> link;
   }
}
template <class T>
void DLList<T>::IterStart (Item <T> *p)
{ if (p) current = p;
  else current = start;
}
template <class T>
void DLList<T>::IterEnd (Item <T> *p)
{ if (p) current = p;
  else current = end;
}
template <class T>
Item <T> * DLList<T>::IterPred ()
{ Item <T> *p = current;
  if (current) current = current -> pred;
  return p;
}
template <class T>
Item <T> * DLList<T>::IterSucc ()
{ Item <T> *p = current;
  if (current) current = current -> succ;
  return p;
}
template <class T>
bool DLList<T>::IterView (T &x) const
{ if (!current) return false;
  x = current -> inf;
  return true;
}
template <class T>
void DLList<T>::InsertToEnd (const T x)
{ Item <T> *p = new Item <T>;
  p -> inf = x;
  p -> pred = end;
  p -> succ = NULL;
  if (!start) { start = end = p; }
  else { end -> succ = p; end = p; }
}
template <class T>
void DLList<T>::InsertToBegin (const T x)
{ Item <T> *p = new Item <T>;
  p -> inf = x;
  p -> pred = NULL;
  p -> succ = start;
  if (!end) { start = end = p; }
  else { start -> pred = p; start = p; }
```

```
}
template <class T>
void DLList<T>::InsertAfter (Item <T> *p, const T x)
{ Item <T> *q = new Item <T>;
  q -> inf = x;
  q -> pred = p;
  q -> succ = p -> succ;
  p -> succ = q;
  if (q -> succ) q -> succ -> pred = q;
  else end = q;
}
template <class T>
void DLList<T>::InsertBefore (Item <T> *p, const T x)
{ Item <T> *q = new Item <T>;
  q -> inf = x;
  q -> succ = p;
  q -> pred = p -> pred;
  p -> pred = q;
  if (q -> pred) q -> pred -> succ = q;
  else start = q;
}
template <class T>
void DLList<T>::DeleteAfter (Item <T> *p, T &x)
{ Item <T> *q = p -> succ;
  x = q -> inf;
  p -> succ = q -> succ;
  if (q -> succ) q -> succ -> pred = p;
  else end = p;
  delete q;
}
template <class T>
void DLList<T>::DeleteBefore (Item <T> *p, T &x)
{ Item <T> *q = p -> pred;
  x = q -> inf;
  p -> pred = q -> pred;
  if (q -> pred) q -> pred -> succ = p;
  else start = p;
  delete q;
}
template <class T>
void DLList<T>::DeleteElement (Item  <T> *p, T &x)
{ x = p -> inf;
  if (start == end) { start = end = NULL; }
  else if (p == start)
  { start = start -> succ;
    start -> pred = NULL;
  }
 else if (p == end)
  { end = end -> pred;
```

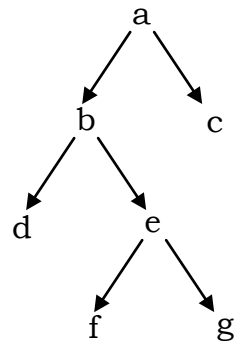
```
    end -> succ = NULL;
  }
 else {
  p -> pred -> succ = p -> succ;
  p -> succ -> pred = p -> pred;
  }
 delete p;
}
template <class T>
int DLList<T>::length () const
{ Item <T> *p = start;
  int n = 0;
  while (p)
   { n++; p = p -> succ; }
  return n;
}
template <class T>
void DLList<T>::print () const
{ Item <T> *p = start;
  while (p)
   { cout << p -> inf << ' ';
     p = p -> succ;
   }
  cout << endl;
}
template <class T>
void DLList<T>::concat (const DLList<T> &r)
{ Item <T> *p = r.start;
  while (p)
    { InsertToEnd (p -> inf);
      p = p -> succ;
    }
}
template <class T>
void DLList<T>::reverse ()
{ Item <T> *p, *temp;
  if (start == end) return;
  p = start;
  start = end;
  end = p;
  while (p)
   { temp = p -> succ;
     p -> succ = p -> pred;
     p -> pred = temp;
     p = temp;
   }
}
template <class T>
bool DLList<T>::empty() const
```

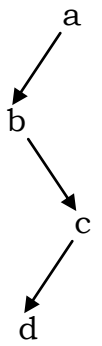
{ return start == NULL; }
Функциите IterPred, IterSucc, IterStart, IterEnd и IterView реализират т.н. **итератор**. Итераторът е указател към текущ елемент на редица. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:
++ - приложена към итератор, премества този итератор да сочи към следващия елемент на редицата;
-- - приложена към итератор, премества този итератор да сочи към предишния елемент на редицата;
* - приложена към итератор, дава елемента, към който сочи итератора.
В шаблона DLList, итераторът е current. Функцията IterStart инициализира итератора да сочи към началото на списъка, ако за нея не е указан аргумент или е указан аргумент NULL. Ако за нея е указан друг аргумент, той е началната стойност на итератора. Аналогично, функцията IterEnd инициализира итератора да сочи към края на списъка, ако за нея не е указан аргумент или е указан аргумент NULL. Ако за нея е указан друг аргумент, той е началната стойност за итератора. Операцията ++ за итератора се реализира от функцията IterSucc, която между другото връща стария итератор. Операцията -- се реализира от функцията IterPred, която между другото връща стария итератор. Операцията * се реализира от функцията IterView.
Двоично дърво от тип T е рекурсивна структура от данни, която е или празна или е образувана от:

- данна от тип T, наречена **корен** на двоичното дърво;
- двоично дърво от тип T, наречено **ляво поддърво** на двоичното дърво;
- двоично дърво от тип T, наречено **дясно поддърво** на двоичното дърво.

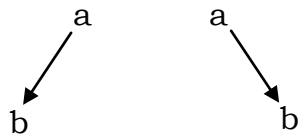
Множеството на **върховете** (**възлите**) на едно двоично дърво се определя рекурсивно: празното двоично дърво няма върхове, а върховете на непразно дърво са неговият корен и върховете на двете му поддървета.
Ще разгледаме примери. Нека a, b, c, d, e, f и g са данни от тип T. Тогава следните графични представяния определят двоични дървета от тип T.

a





Посоката на линиите, свързващи върховете с поддървета позволява да се различи ляво от дясно поддърво. Следните две двоични дървета са различни:



В първия случай дясното поддърво е празно, а в другия случай дясното поддърво не е празно.
Листата на двоичното дърво са върховете с две празни поддървета. Например, на първата рисунка листата на трите дървета са a; d, f, g, c; d съответно.
Вътрешни върхове на двоичното дърво са върховете, различни от корена и листата. Например, на първата рисунка първото дърво няма вътрешни върхове, вътрешните върхове на второто дърво са b, e и на третото дърво са b, c.
Ляв наследник на един връх е коренът на лявото му поддърво (ако то е непразно). **Десен наследник** на един връх е коренът на дясното му поддърво (ако то е непразно). Ясно е, че листата на едно дърво са точно онези върхове, които нямат нито ляв, нито десен наследник. Ако a е наследник на b (ляв или десен), казваме, че b е **родител** (**баща**) на a. Ясно е, че всеки връх освен корена има точно един баща.
На всеки връх в дървото може да се съпостави **ниво**. Коренът на дървото има ниво 1 (или 0). Ако един връх има ниво *i*, то неговите наследници имат ниво *i*+1. С други думи, нивото на един връх е броят на върховете, които трябва да бъдат обходени за да се стигне до този връх от корена. Максималното ниво на едно дърво се нарича негова **височина**. Над структурата от данни двоично дърво са възможни следните операции:

- достъп до връх – възможен е пряк достъп до корена и непряк достъп до останалите върхове;
- включване и изключване на връх – възможни са на произволно място в двоичното дърво, резултатът трябва отново да е двоично дърво от същия тип;

- обхождане – това е метод, позволяващ да се осъществи достъп до всеки връх на дървото един единствен път.

Обхождането е рекурсивна процедура, която се осъществява чрез изпълнение на следните три действия, в някакъв фиксиран ред:

- обхождане на корена;
- обхождане на лявото поддърво;
- обхождане на дясното поддърво.

Най-разпространени са **смесеното обхождане** (първо лявото поддърво, после корена и най-накрая дясното поддърво), **низходящото обхождане** (първо корена, после лявото поддърво и най-накрая дясното поддърво) и **възходящо обхождане** (първо лявото поддърво, после дясното поддърво и най-накрая корена).

Основно се използват три начина за представяне на двоично дърво – **свързано**, **верижно** и **чрез списък на бащите**.

Свързаното представяне се реализира чрез указател към кутия с три полета – информационно, съдържащо стойността на корена и две адресни, съдържащи представянията на лявото и дясното поддърво.

При верижното представяне се използват три масива – a[N], b[N] и c[N]. Тук N е броят на върховете в дървото, върховете са номерирани от 0 до N-1. Елементът a[i] на масива a съдържа стойността на i-тия връх на дървото. Елементът b[i] на масива b съдържа индекса на левия наследник на i-тия връх (-1, ако той няма ляв наследник), елементът c[i] на масива c съдържа индекса на десния наследник на i-тия връх (-1, ако той няма десен наследник). Отделно се пази и индекса на корена на двоичното дърво.

При представянето чрез списък на бащите се използва един масив p[N]. Тук N е броят на върховете в дървото, върховете са номерирани от 0 до N-1. Елементът p[i] е единствения баща на i-тия връх на дървото (-1, ако този връх е коренът).

Сега ще дефинираме клас, който реализира свързаното представяне на двоично дърво.

Първо ще дефинираме помощна структура Node, която реализира тройната кутия, чрез която се представят върховете на дървото. За по-голяма общност, дефинираме структурата Node и класа Tree като шаблони.

```
template <class T>
struct Node {
  T inf;
  Node *left;
  Node *right;
};
```

Сега вече сме готови да дефинираме класът Tree. Тъй като двоичното дърво се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
```

```
class Tree {
 public:
   Tree(const T&);
  Tree();
  ~Tree();
  Tree(const Tree &);
  Tree& operator= (const Tree &);
  bool empty () const;
  bool RootTree (T&) const;
  bool LeftTree (Tree &) const;
  bool RightTree (Tree &) const;
  void Create();
  void Create3(const T&, const Tree &, const Tree &);
  void PreOrder() const;
  void InOrder() const;
  void PostOrder() const;
 private:
  Node <T> *root;
  void DeleteTree (Node<T> *&);
  void CopyTree (Node <T> * &, const Node<T> *);
  void Preord (const Node <T> *);
  void Inord (const Node <T> *);
  void Postord (const Node <T> *);
  void CreateTree (Node <T> *&);
};
template <class T>
Tree<T>::Tree (const T& x)
{ root = new Node <T>;
  root -> inf = x;
  root -> left = root -> right = NULL;
}
template <class T>
Tree<T>::Tree ()
{ root = NULL; }
template <class T>
Tree<T>::~Tree ()
{ DeleteTree (root); }
template <class T>
Tree<T>::Tree (const Tree<T> &t)
{ CopyTree (root, t.root); }
template <class T>
Tree<T>& Tree<T>::operator= (const Tree<T> &t)
{ if (this != &t)
   { DeleteTree (root);
     CopyTree (root, t.root);
   }
  return *this;
}
template <class T>
```

```
void Tree<T>::DeleteTree (Node <T> * &t)
{ if (t) {
   DeleteTree (t -> left);
   DeleteTree (t -> right);
   delete t;
   t = NULL;
 }
}
template <class T>
void Tree<T>::CopyTree (Node <T> * &p, const Node <T> *t)
{ if (t)
   { p = new Node <T>;
     p -> inf = t -> inf;
     if (t -> left) CopyTree (p -> left, t -> left);
     else p -> left = NULL;
     if (t -> right) CopyTree (p -> right, t -> right);
     else p -> right = NULL;
   }
}
template <class T>
bool Tree<T>::empty () const
{ return root == NULL; }
template <class T>
bool Tree<T>::RootTree (T &x) const
{ if (!root) return false;
  x = root -> inf;
  return true;
}
template <class T>
bool Tree<T>::LeftTree (Tree <T> &t) const
{ if (!root) return false;
  DeleteTree (t.root);
  CopyTree (t.root, root -> left);
  return true;
}
template <class T>
bool Tree<T>::RightTree (Tree <T> &t) const
{ if (!root) return false;
  DeleteTree (t.root);
  CopyTree (t.root, root -> right);
  return true;
}
template <class T>
void Tree<T>::PreOrder () const
{ Preord (root);
  cout << endl;
}
template <class T>
void Tree<T>::InOrder () const
```

```
{ Inord (root);
  cout << endl;
}
template <class T>
void Tree<T>::PostOrder () const
{ Postord (root);
  cout << endl;
}
template <class T>
void Tree<T>::Preord (const Node <T> *t)
{ if (t)
   { cout << t -> inf << ' ';
     Preord (t -> left);
     Preord (t -> right);
   }
}
template <class T>
void Tree<T>::Inord (const Node <T> *t)
{ if (t)
   { Inord (t -> left);
     cout << t -> inf << ' ';
     Inord (t -> right);
   }
}
template <class T>
void Tree<T>::Postord (const Node <T> *t)
{ if (t)
   { Postord (t -> left);
     Postord (t -> right);
     cout << t -> inf << ' ';
   }
}
template <class T>
void Tree<T>::Create3(const T&rt, const Tree<T> &l, const Tree<T> &r)
 { DeleteTree (root);
   root = new Tree<T>;
   root -> inf = rt;
   CopyTree (root -> left, l.root);
   CopyTree (root -> right, r.root);
 }
template <class T>
void Tree<T>::Create()
{ DeleteTree (root);
  CreateTree (root);
}
template <class T>
void Tree<T>::CreateTree (Node <T> * &t)
{ T x; char c;
  cout << "root: ";
```

```
  cin >> x;
  t = new Node <T>;
  t -> inf = x;
  cout << "Left tree of: " << x << "  (y/n)? ";
  cin >> c;
  if (c == 'y' || c == 'Y') CreateTree (t -> left); else t -> left = NULL;
  cout << "Right tree of: " << x << "  (y/n)? ";
  cin >> c;
  if (c == 'y' || c == 'Y') CreateTree (t -> right); else t -> right = NULL;
}
```

Предполагаме, че за елементите на типа T е установена линейна наредба.

Двоично наредено дърво от тип T се дефинира рекурсивно по следния начин: празното двоично дърво от тип T е наредено и непразно двоично дърво от тип T е наредено тогава и само тогава, когато всички върхове на лявото му поддърво са по-малки от корена и всички върхове на дясното му поддърво са по-малки от корена и освен това, лявото и дясното му поддърво са двоични наредени дървета от тип T.

Нека t е двоично наредено дърво от тип T. **Включването** на елемента a от тип T в t се осъществява по следния начин:

- ако t е празното дърво, новото двоично наредено дърво е с корен елемента a и празни ляво и дясно поддървета;
- ако t е непразно и a е по-малко от корена му, a се включва в лявото поддърво на t;
- ако t е непразно и a е по-голям от корена му, a се включва в дясното поддърво на t.

Ще използваме този начин за включване на елементи за да създаваме двоичните наредени дървета. В този случай, те притежават следното свойство: смесеното обхождане сортира във възходящ ред елементите от върховете на дървото.

Изтриването на връх елемент a от двоичното наредено дърво t се извършва по следната схема:

- ако коренът на t е по-голям от a, изтриваме a от лявото поддърво на t;
- ако коренът на t е по-малък от a, изтриваме a от дясното поддърво на t;
- ако коренът на t съвпада с a и t има празно ляво поддърво, то t се заменя с дясното си поддърво;
- ако коренът на t съвпада с a и t има празно дясно поддърво, то t се заменя с лявото си поддърво;
- ако коренът на t съвпада с a и t има непразни ляво и дясно поддърво, то се намира най-големият елемент в лявото поддърво (това става като се спуснем по най-десния клон до достигане на връх с празно дясно поддърво), разменя се неговата стойност със стойността на корена (която е a) и въпросният връх се изтрива (той има празно дясно поддърво, така че влизаме в предния случай).

Сега ще дефинираме клас, който реализира свързаното представяне на двоично наредено дърво, в който са включени гореописаните операции. Ще отбележим, че не допускаме в дървото да има върхове с еднакви стойности (това ограничение, обаче, лесно може да се премахне).
Първо ще дефинираме помощна структура Node, която реализира тройната кутия, чрез която се представят върховете на дървото. За по-голяма общност, дефинираме структурата Node и класа BinOrdTree като шаблони.

```
template <class T>
struct Node {
   T inf;
   Node *left;
   Node *right;
};
```

Сега вече сме готови да дефинираме класът BinOrdTree. Тъй като двоичното наредено дърво се реализира в динамичната памет, за този клас трябва изрично да се реализира каноничното представяне – деструктор, обикновени конструктори, конструктор за присвояване и операторна функция за присвояване.

```
template <class T>
class BinOrdTree {
 public:
   BinOrdTree (const T&);
   BinOrdTree ();
   ~BinOrdTree ();
   BinOrdTree (const BinOrdTree &);
   BinOrdTree& operator= (const BinOrdTree &);
   bool empty () const;
   bool RootTree (T&) const;
   bool LeftTree (BinOrdTree &) const;
   bool RightTree (BinOrdTree &) const;
   void PrintSorted () const;
   void AddNode (const T&);
   void DeleteNode (const T&);
   void Create();
private:
   Node<T> *root;
   void DeleteTree (Node <T> *&);
   void Del (Node <T> *&, const T&);
   void CopyTree (Node <T> *&, const Node <T>*);
   void Print (const Node <T> *);
   void Add (Node <T> *&, const T&);
};
template <class T>
BinOrdTree<T>::BinOrdTree (const T& x)
{ root = new Node <T>;
  root -> inf = x;
  root -> left = root -> right = NULL;
```

```
}
template <class T>
BinOrdTree<T>::BinOrdTree ()
{ root = NULL; }
template <class T>
BinOrdTree<T>::~BinOrdTree ()
{ DeleteTree (root); }
template <class T>
BinOrdTree<T>::BinOrdTree (const BinOrdTree &t)
{ CopyTree (root, t.root); }
template <class T>
BinOrdTree<T>& BinOrdTree<T>::operator= (const BinOrdTree &t)
{ if (this != &t)
   { DeleteTree (root);
     CopyTree (root, t.root);
   }
  return *this;
}
template <class T>
void BinOrdTree<T>::DeleteTree (Node <T> * &t)
{ if (t) {
   DeleteTree (t -> left);
   DeleteTree (t -> right);
   delete t;
   t = NULL;
  }
}
template <class T>
void BinOrdTree<T>::CopyTree (Node <T> *&p, const Node <T> *t)
{ if (t)
   { p = new Node <T>;
     p -> inf = t -> inf;
     if (t -> left) CopyTree (p -> left, t -> left);
     else p -> left = NULL;
     if (t -> right) CopyTree (p -> right, t -> right);
     else p -> right = NULL;
   }
}
template <class T>
bool BinOrdTree<T>::empty () const
{ return root == NULL; }
template <class T>
bool BinOrdTree<T>::RootTree (T &x) const
{ if (!root) return false;
  x = root -> inf;
  return true;
}
template <class T>
bool BinOrdTree<T>::LeftTree (BinOrdTree <T> &t) const
```

```
{ if (!root) return false;
  DeleteTree (t.root);
  CopyTree(t.root, root -> left);
  return true;
}
template <class T>
bool BinOrdTree<T>::RightTree (BinOrdTree <T> &t) const
{ if (!root) return false;
  DeleteTree (t.root);
  CopyTree(t.root, root -> right);
  return true;
}
template <class T>
void BinOrdTree<T>::PrintSorted () const
{ Print (root);
  cout << endl;
}
template <class T>
void BinOrdTree<T>::Print (const Node <T> *t)
{ if (t)
   { Print (t -> left);
     cout << t -> inf << ' ';
     Print (t -> right);
   }
}
template <class T>
void BinOrdTree<T>::AddNode (const T& x)
{ Add (root, x); }
template <class T>
void BinOrdTree<T>::Add (Node <T> * &t, const T& x)
{ if (!t)
   { t = new Node <T>;
     t -> inf = x;
     t -> left = t -> right = NULL;
   }
  else if (x < t -> inf) Add (t -> left, x);
  else if (x > t -> inf) Add (t -> right, x);
  else cout << "Duplicate cannot be inserted!" << endl;
}
template <class T>
void BinOrdTree<T>::DeleteNode (const T x)
{ Del (root, x); }
template <class T>
void BinOrdTree<T>::Del (Node <T> * &t, const T& x)
{ if (!t) return;
  if (x < t -> inf) Del (t -> left, x);
  else if (x > t -> inf) Del (t -> right, x);
  else {
    Node <T> *p;
```

```
    if (!(t -> left)) { p = t; t = t -> right; delete p; }
    else if (!(t -> right)) { p = t; t = t -> left; delete p; }
    else {  p = t -> left;
            while (p -> right) p = p -> right;
            t -> inf = p -> inf;
            Del (t -> left, p -> inf);
    }
  }
}
template <class T>
void BinOrdTree<T>::Create ()
{ DeleteTree (root);
  T x; char c;
  do {
    cout << "Enter element: ";
    cin >> x;
    AddNode (x);
    cout << "Enter more (y/n)? ";
    cin >> c;
  } while (c == 'y' || c == 'Y');
}
```

14. Основни конструкции в езиците за функционално програмиране. Дефиниране и използване на функции. Списъци. Функции от по-висок ред за работа със списъци. Потоци.

Във функционалното програмиране програмата се изгражда в дескриптивен стил – описват се свойствата на желания резултат, докато алгоритъмът за неговото намиране не е съществен. Най-често програмата е поредица от дефиниции на функции, които имат за цел да пресмятат някакъв резултат. Тези функции не предизвикват странични ефекти подобно на програмите при процедурното програмиране. Това води до по лесно намиране и отстраняване на грешки. В програмите няма операции за цикли, за условен или безусловен преход, операции за присвояване. Итерациите се реализират чрез рекурсия. При функционалното програмиране има две основни дейности – дефиниции на функции и прилагане на функции за извличане на резултат.
Тук ще разглеждаме езика Scheme, който е пример за нестрог функционален език и е диалект на езика Lisp.
Математически основи на езика Lisp се основава на ℓ-смятането на Чърч. Основната структура в този език са списъците и чрез тях се моделират обекти от реалния свят.

Езикът Scheme притежава три механизма за изразяване:

- примитивни изрази;
- средства за комбинация;

- средства за абстракция.

Основна синтактична единица в езика Scheme е **изразът**.
Неформално работният цикъл на интерпретатора на Scheme е следния:
- чете израз (read);
- оценява израз (evaluate);
- извежда оценката (print).

Изразите могат да бъдат **примитивни** или **съставни**.
Примитивните изрази още се наричат **атомарни изрази** или **атоми**, а съставните се наричат **комбинации**.

Примитивните изрази се делят на:
- **числа** – с фиксирана или плаваща точка – на практика с неограничен размер;
- **символни низове** – поредица от знаци, заградени в кавички (“ “);
- **вградени функции** – например +, -, *, / и др.;
- **символни атоми** – в процедурните езици са известни като идентификатори – поредица от букви или цифри, която трябва да започва с буква.

Оценката на примитивен израз директно се свързва с неговото име:
- оценката на число е самото число;
- оценката на символен низ е самия низ;
- оценката на вградена функция е така наречения **функционален (процедурен) обект** – двойка съдържаща указател към среда и поредица от машинни инструкции, които реализират съответната функция;
- за да има оценка един символен атом, той трябва да означава име на променлива или функция, т.е. предварително трябва да е дефиниран, ако не е дефиниран оценката на символния атом е неопределена.

Списък в Scheme е редица от обекти, оградени в кръгли скоби и разделени с един или повече интервали, допуска се влагане на списъци и празен списък, който се означава с ().
Семантично списъците биват няколко вида и оценката им зависи от техния вид, ако искаме независимо от вида на списъка оценката му да бъде самия списък, използваме цитат по следния начин: (quote <списък>) или ‘<списък>.

Съставните изрази (комбинациите) са обръщения към функции. Функциите могат да бъдат вградени или дефинирани от потребителя. Съставните изрази синтактично се означават чрез списък по следния начин:
(<функция> <ФактП 1> <ФактП 2> … <ФактП n>).
Числото n е броят на допустимите аргументи на функцията. Често първият аргумент в съставен израз се нарича **оператор**, а фактическите параметри се наричат **операнди**.

Оценяването на комбинация се извършва по следното общо правило:

- отляво надясно се оценяват отделните подизрази на комбинацията;
- оценката на оператора, която задължително е функция, се прилага върху оценките на операндите, които трябва да бъдат коректни по брой и тип.

Има изключение от общото правило при така наречените **специални форми**. Всяка специална форма си има собствено правило за оценяване.

Чрез средствата за абстракция, обектите на езика могат да се именуват и обработват като едно цяло. Основно такова средство е специалната форма **define**. Тя има две разновидности: за дефиниране на променливи и за дефиниране на процедури.
Синтаксисът за дефиниране на променливи е следния:
(define <име_на_променлива> <израз>).
Тук <име_на_променлива> е произволен символен атом, <израз> е произволен израз. Оценката на израза е <име_на_променлива>, но като страничен ефект се получава свързване на символния атом <име_на_променлива> с оценката на <израз> в специална работна памет, наречена **среда**. След като този израз е оценен, оттам нататък оценката на <име_на_променлива> ще бъде оценката на <израз>.
Особеността на define е, че нейният втори аргумент не се оценява, затова тя е специална форма.
Синтаксисът за дефиниране на процедури е следния:
(define (<име_на_процедура> <ФормП1> …<ФормПn>) <тяло>).
Тук <име_на_процедура> и <ФормПi>, i = 1, …, n са символни атоми, <тяло> е редица от изрази, разделени с един или повече интервали. Оценката на израза е <име_на_процедура>, но като страничен ефект в средата, в която се извършва свързването се създава нов процедурен обект – той съдържа указател към текущата среда и код. Кодът се състои от параметрите (<ФормП1>, …, <ФормПn>) и тяло (изразите в редицата <тяло>).

За да може да се реализира именуването е нужно да се поддържа памет, където да се съхраняват двойки от вида (име, обект).
В езика Scheme при оценяването се използва т.н. **модел на средите**. При него, паметта, която служи за съхраняване на двойките (име, обект) се нарича **среда**. В началото на оценяването имаме дефинирана само една среда – наричаме я **глобална**. В процеса на оценяване могат да възникнат и други среди. Когато се създава нова среда тя винаги се явява разширение на вече съществуваща среда. Когато се извършва оценяване, то винаги се прави в контекста на някоя среда. Например, нека имаме средите E_0, E_1, …, E_n, като E_0 е глобалната среда и средата E_i е разширение на средата E_{i-1}, i = 1, …, n.

Да предположим, че търсим оценката на името A в E_n. В модела на средите това търсене се извършва по следния начин: започвайки от E_n, последователно обхождаме средите в обратен ред на разширяването (от E_i към E_{i-1}), до първия момент в който достигнем до среда E_k, в която е дефинирана двойка (A, обект). Тогава процесът на търсене спира и оценката на A е съответният обект. При това е възможно да е дефинирана двойка (A, обект1) в среда E_j при $j < k$, но това не променя нищо (локалните имена скриват глобалните).

Както вече казахме, оценяването на един съставен израз (комбинация) представлява обръщения към процедура.
Нека имаме следната комбинация:
(<процедура> <ФактП1> …<ФактПn>).
Ще опишем как се извършва нейното оценяване в модела на средите. Нека оценяването на комбинацията се извършва в средата E. <процедура> е израз, който определя процедурата, към която се обръщаме, <ФактПi>, i = 1, …, n са фактическите параметри на обръщението. Първото което се прави е в средата E да се оценят <процедура> и <ФактПi>, i = 1, …, n. За да е коректно оценяването, оценката на <процедура> задължително трябва да е процедурен обект и фактическите параметри трябва да съответстват на формалните параметри по брой, тип и смисъл. Нека указателят в процедурния обект, който е оценка на <процедура> сочи към средата E□. Тогава се създава нова среда E□□, която е разширение на E□ и в E□□ всички формалните параметри на процедурния обект се свързва с оценките на съответните им фактически параметри. След това в контекста на средата E□□ се извършва последователно оценка на изразите от тялото на процедурния обект. Оценката на последния израз от тялото е оценка на първоначалната комбинация в средата E.
Описаният модел на средите реализира т.н. **апликативен модел** на оценяване (неговата същност е, че първо се оценяват операндите и след това към получените оценки се прилага оценката на оператора). Съществува и друг модел на оценяване – т.н. **нормален модел**. При него, при извършване на процедурно обръщение се оценява само оператора, но не и операндите – те се заместват в тялото на оценката на оператора на мястото на съответните им формални параметри без да се оценяват предварително и след това се оценява тялото на оценката на оператора.
Може да се докаже, че ако процесът на оценяване по апликативния модел завършва, то процесът на оценяване по нормалния модел също завършва и в резултат се получава една и съща оценка. Съществуват, обаче, много програми, които завършват изпълнението си по нормалния модел на оценяване, но не завършват по апликативния. Затова казваме, че нормалният модел е по-общ от апликативния. Езикът Scheme използва моделът на средите, в основата на който стои апликативният модел.

В някои случаи, обаче, например при работа с отложени безкрайни обекти Scheme използва нормалният модел.

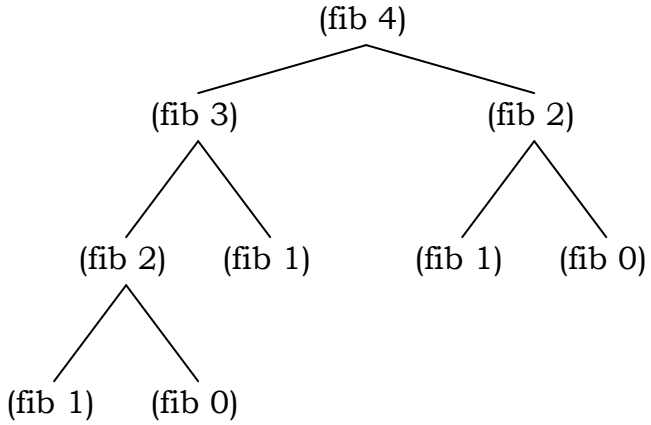
На всяка дефинирана процедура може да се гледа както декларативно, така и императивно (като процес, който ще доведе до получаване на търсения резултат). **Процес** – това е абстрактно понятие, описващо развитието на пресмятането във времето, т.е. преобразуването на входните данни стъпка по стъпка до получаване на резултата. Можем да класифицираме процесите в два основни класа – **рекурсивни** и **итеративни**. Рекурсивните процеси се делят на **линейно рекурсивни** и **дървовидно рекурсивни**. Итеративните се делят на **линейни итеративни** и **нелинейни итеративни** (**логаритмични** и др.).
За да сравним линейно рекурсивните и линейно итеративните процеси ще реализираме по два начина функцията факториел.

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))
(define (fact n)
  (define (iterfact result m)
    (if (> m n) result (iterfact (* result m) (+ m 1))))
  (iterfact 1 1)
)
```

Оценяването на (fact 3) поражда следните процеси:
(fact 3) □ (* 3 (fact 2)) □ (* 3 (* 2 (fact 1))) □ (* 3 (* 2 (* 1 (fact 0)))) □
□ (* 3 (* 2 (* 1 1))) □ (* 3 (* 2 1)) □ (* 3 2) □ 6,
(fact 3) □ (iterfact 1 1) □ (iterfact 1 2) □ (iterfact 2 3) □ 6.
Забелязваме, че в първия случай имаме разширяване и след това свиване на изчислителния процес, като при това имаме редица отложени операции. Порядъка на отложените операции е линейна функция на размера на входа и затова процесът в този случай процесът се нарича линейно рекурсивен.
При втория случай нямаме разширяване и свиване. Изчислителният процес напълно се описва от краен брой променливи на състоянието, правила които описват как тези променливи се променят при преминаване на едно състояние в друго и условие за край. Порядъка на броя на смените на състояния е линейна функция на размера на входа и затова процесът в този случай се нарича линейно итеративен.
Когато в своята дефиниция една процедура извиква себе си повече от веднъж, то забелязваме че броят на отложените операции става експоненциален. Процесите, които пораждат такива процедури наричаме дървовидно рекурсивни. Добър пример е процедурата за намиране на n-тото число на Фибоначи:

```
(define (fib n)
  (if (= n 0) 1
    (if (= n 1) 1
      (+ (fib (- n 1)) (fib (- n 2))))))
```

Оценяването на (fib 4) поражда следния процес:



Вижда се, че това е твърде неефективна реализация, тъй като се правят излишни изчисления – (fib 2) се оценява 2 пъти, (fib 1) се оценява 3 пъти, (fib 0) се оценява 2 пъти.
Много по-ефективно е следното решение:

```
(define (fib n)
        (define (iterfib a b counter)
                (if (= counter n) a
                    (iterfib b (+ a b) (+ counter 1))))
        (iterfib 1 1 0))
```

Оценяването на комбинацията (fib 4) се извършва по следния начин: (fib 4) □ (iterfib 1 1 0) □ (iterfib 1 2 1) □ (iterfib 2 3 2) □ (iterfib 3 5 3) □ (iterfib 5 8 4) □ 5. Процесът е линейно итеративен.

Всяка процедура, която манипулира с процедурни обекти се нарича **процедура от по-висок ред**. По-специално процедурите от по-висок ред получават процедурни обекти като параметри или връщат процедурни обекти като резултат.
Предаването на процедурите като параметри дава възможност за осъществяване на по-голямо ниво на абстракция в програмите на Scheme. Синтактично, то се осъществява по начина, по който се предават обикновени параметри. Като пример ще реализираме абстракцията f (a) + f (next (a)) … + f (b). Тук чрез next се получава следващата стъпка, сумирането започва от a и продължава до b.

```
(define (sum a b next f)
   (if (> a b) 0
      (+ (f a) (sum (next a) b next f))))
```

По-ефективен е следният линейно итеративен вариант:

```
(define (sum a b next f)
   (define (itersum i result)
     (if (> i b) result
        (itersum (next i) (+ result (f i)))))
   (itersum a 0)
)
```

Примерно използване: (sum 1 5 (lambda (x) (+ x 1)) (lambda (x) x)) се оценява до 1+2+3+4+5 = 15.
Оценката на една комбинация може да бъде процедурен обект. Това е съвсем лесно за реализация – оценката на последният израз

в тялото трябва да е процедурен обект. Като пример ще дефинираме процедура, която има един аргумент процедура и връща нейната n-та експонента (функцията, приложена n-пъти върху аргумента си):

```
(define (exponent f n)
  (if (= n 0) (lambda (x) x)
   (lambda (x) (f ((exponent f (- n 1)) x)))))
```

В случая е използвана специалната форма lambda, която не оценя аргументите си и се използва за създаване на анонимен процедурен обект в текущата среда.
Примерно използване:
((exponent (lambda (x) (* x x)) 2) 3) се оценява до $(3^2)^2 = 81$.

Основното предназначение на езика за функционално програмиране Scheme е работа **със списъци**. Списъците се реализират чрез **точкови двойки**. Точковата двойка е съставен тип данни в Scheme и тя представлява указател към двойна клетка, състояща се от лява и дясна част. Тези части сами по себе си могат да са точкови двойки или атоми. За да можем да работим с точкови двойки са необходими операция за конструиране на точкова двойка, операция за извличане на лявата част на точкова двойка и операция за извличане на дясната част на точкова двойка. Въпреки, че програмистите могат сами да си дефинират такива операции, за по-голяма ефективност те обикновено са вградени. В Scheme имаме:
(cons a b) – конструира точкова двойка с лява част оценката на a и дясна част оценката на b;
(car x) – извлича лявата част на оценката на x, която трябва задължително да е точкова двойка;
(cdr x) – извлича дясната част на оценката на x, която трябва задължително да е точкова двойка.
Списъкът е крайна редица от обекти от произволен тип, възможно празна. В Scheme списъците се дефинират посредством точкови двойки по следния начин: празният списък () е списък и ако L е списък и a е произволен обект, то точковата двойка с лява част a и дясна част L е списък. С вградените операции на Scheme списъкът $<a_1>, <a_2>, \ldots, <a_n>$ може да се конструира така:
(cons $<a_1>$ (cons $<a_2>$ …(cons $<a_n>$ ())…)). Понякога не е удобно да се конструира списък по този начин и тогава може да се използва примитивната процедура **list**: (list $<a_1>$ $<a_2>$ …$<a_n>$). Също може да се използва и специалната форма **quote**:
(quote ($<a_1>$ $<a_2>$ … $<a_n>$)) или в съкратен вариант
‘($<a_1>$ $<a_2>$ … $<a_n>$).
Отделните елементи на списъците се извличат с помощта на подходящо комбиниране на car и cdr. Примери: (предполагаме, че оценката на <списък> е някакъв списък)
(car <списък>) – връща първия елемент на списъка;
(cdr <списък>) – връща списъка с премахнат първи елемент.

И в двата случая, ако оценката на <списък> е (), то оценката на комбинацията е неопределена. Други примери:
(car (cdr <списък>)) – връща втория елемент на списъка;
(car (cdr (cdr <списък>))) – връща третия елемент на списъка.
За синтактично удобство се допускат специални форми от вида c*xxxx*r, където *x* е една от буквите a или d. Например:
(cadr <списък>) (car (cdr <списък>));
(caddr <списък>) (car (cdr (cdr <списък>))).
В езика Scheme има множество вградени операции за работа със списъци, но за пълнота ще разгледаме реализациите на някои операции.
Намиране на дължината на списък:
```
(define (length l)
  (if (null? l) 0 (+ 1 (length (cdr l)))))
```
Вграденият предикат null? приема един аргумент и връща #t, ако този аргумент е празният списък и #f, в противен случай.
Намиране на k-тия елемент на списък, като номерацията започва от 1 (операцията има смисъл когато k не надвишава дължината на списъка, в частност операцията няма смисъл за празен списък):
```
(define (kth_el l k)
 (if (= k 1) (car l) (kth_el (cdr l) (- k 1))))
```
Конкатенация на списъци:
```
(define (append l1 l2)
  (if (null? l1) l2 (cons (car l1) (append (cdr l1) l2))))
```
Обръщане на списък (вариант с append):
```
(define (reverse l)
  (if (null? l) l
    (append (reverse (cdr l)) (list (car l)))))
```
Обръщане на списък (итеративен вариант без append):
```
(define (reverse l)
  (define (helprev l1 result)
    (if (null? l1) result (helprev (cdr l1) (cons (car l1) result))))
  (helprev l ()))
```
Проверка за принадлежност към списък:
```
(define (member x l)
  (cond ((null? l) #f)
        ((equal? x (car l)) #t)
        (else (member x (cdr l)))))
```
Вграденият предикат equal? приема два аргумента и връща стойност #t, ако тези два аргумента имат еднакво представяне и #f, в противен случай.
Под списък с вложения разбираме списък, елементите на който може да са списъци. Още казваме, че работим със списъци на произволно ниво.
Намиране на броя на атомите в списък на произволно ниво: (вариант, в който празният списък не се брои за атом):
```
(define (deep_count_atoms l)
   (cond ((null? l) 0)
         ((atom? l) 1)
```

```
        (else (+ (deep_count_atoms (car l)) (deep_count_atoms (cdr l))))))
```
(вариант, в който празният списък се брои за атом):
```
(define (deep_count_atoms l)
  (cond ((null? l) 0)
        ((atom? (car l)) (+ 1 (deep_count_atoms (cdr l))))
        (else (+ (deep_count_atoms (car l)) (deep_count_atoms (cdr l))))))
```
Разликата е следната: например, (deep_count_atoms '(()()())) в първия вариант се оценява с 0, а във втория вариант с 3.
Вграденият предикат atom? приема един аргумент и връща стойност #t, ако този аргумент не е точкова двойка (т.е. е атом) и #f, в противен случай (ако не е атом).
Обръщане на списък на произволно ниво:
```
(define (deep_reverse l)
  (if (atom? l) l
    (append (deep_reverse (cdr l)) (list (deep_reverse (car l))))))
```
Ще разгледаме някои процедури от по-висок ред за работа със списъци.
Първата процедура извършва комбиниране (акумулиране) на елементите на списък:
```
(define (accum l combiner init_value)
  (if (null? l) init_value
    (combiner (car l) (accum (cdr l) combiner init_value))))
```
С помощта на тази процедура можем да намерим сумата на елементите на списък по следния начин: (accum l + 0), произведението на елементите на списък по следния начин: (accum l * 1), можем да създадем копие на списък по следния начин: (accum l cons ()) и освен това можем да реализираме функцията append с променлив брой аргументи:
(accum '(<l_1> <l_2> …<l_n>) append ()).
Втората процедура е за трансформиране на списък чрез прилагане на една и съща процедура върху елементите му.
```
(define (map f l)
  (if (null? l) l (cons (f (car l)) (map f (cdr l)))))
```
Третата процедура е за филтриране на елементите на списък, т.е. за образуване на подсписък от елементите на списъка, които удоволетворяват някакво условие (филтър).
Ще я реализираме в два варианта – чрез рекурсивен процес и чрез използване на вече дефинираните функции за акумулиране и за трансформация.
Рекурсивен вариант:
```
(define (filter l pred)
  (cond ((null? l) l)
        ((pred (car l)) (cons (car l) (filter (cdr l) pred)))
        (else (filter (cdr l) pred))))
```
Решение чрез трансформация:
```
(define (filter l pred)
  (accum (map (lambda (x) (if (pred x) (list x) ())) l) append ()))
```
Ако един елемент минава през филтъра, той се замества със списък от себе си, в противен случай се замества с празен списък.

След това върху елементите на получения списък се изпълнява append с променлив брой аргументи.

Потокът е съставна структура от данни, която се явява обобщение на структурата списък. Потокът представлява крайна или безкрайна редица от елементи от произволен тип. Възможен е пряк достъп само до първия елемент на потока. До останалите елементи достъпът е последователен и се осъществява чрез **форсиране**.

Нека редицата a_1, a_2, …, a_n, … е поток. В Scheme тази редица се представя като точкова двойка, която има за лява част представянето на a_1 – **глава** на потока и за дясна част т.н. **отложен обект**, съдържащ информация, необходима за намиране на **опашката** на потока – a_2, a_3, …, a_n, …. По този начин в началото е видим само първият елемент на потока, а останалите елементи са отложени (неконструирани). Ако е необходимо, отложената част на потока се форсира. Чрез еднократно форсиране вторият елемент става видим, а останалата част от потока остава невидима, т.е. отложена. По този начин една част от потока е отложена и остава такава докато не възникне необходимост тя да се конструира.
В Scheme потоците се определят с помощта на двуаргументен конструктор **cons-stream** и едноаргументни селектори **head** и **tail**. За тях е изпълнено следното: (head (cons-stream a b)) се оценява с оценката на a, (tail (cons-stream a b)) се оценява с оценката на b. Конструкторът cons-stream създава поток с първи елемент оценката на първия аргумент и опашка – отложен обект, свързан с втория аргумент, оценката на който трябва да е поток. Селекторът head извлича първият елемент на потока, селекторът tail извлича опашката на потока. Празният поток се означава с атома **the-empty-stream**. Предикатът **empty-stream?** се използва за проверка за празен поток.
Пример (дефиниране на поток с елементи 1 и 2):
(define stream (cons-stream 1 (cons-stream 2 the-empty-stream))).
Дотук съществува известна прилика между потоците и списъците, но както ще видим по-нататък разликата е принципна. Възможно е, например, да създаваме потоци с безкрайно много елементи.

Основната идея в реализацията на потоците е те да не се конструират изцяло. С други думи, при конструиране на поток се пресмята само първия елемент на потока, а конструирането на останалата част от потока се отлага докато не се поиска достъп до нея. Това се постига с помощта на така наречената **отложено оценяване**, което се осигурява чрез специалната форма **delay** и вградената функция **force**.
Специалната форма delay има следния синтаксис: (delay <израз>). Тя не оценява аргумента си. В резултат на оценяване на обръщение към delay в текущата среда се създава “пакетирана”

форма на израза, която позволява той да се оцени чак когато е необходимо.
Може да се счита, че delay е такава специална форма, че извикването (delay <израз>) е еквивалентно на извикването (lambda () <израз>).
Вградената функция force има следния синтаксис:
(force <отложен_израз>). Тя връща оценката на израза, чието оценяване преди това е било отложено с delay. Казваме още, че отложеният израз се **форсира**. В съответствие с горното съображение за delay, извикването (force <отложен_израз>) е еквивалентно на извикването (<отложен_израз>).
При тези дефиниции оценката на (force (delay x)) винаги съвпада с оценката на x.

Конструкторът cons-stream може да се реализира като специална форма, която не оценява втория си аргумент и обръщението (cons-stream a b) е еквивалентно на обръщението (cons a (delay b)).
При това положение селекторите head и tail се дефинират по следния начин:

```
(define (head stream) (car stream))
(define (tail stream) (force (cdr stream)))
```

Като пример ще дефинираме някои полезни процедури за работа с потоци, включително и такива от по-висок ред.

Конкатениране на два потока:

```
(define (append-streams s1 s2)
  (if (empty-stream? s1) s2
     (cons-stream (head s1) (append-streams (tail s1) s2))))
```

Тази процедура е приложима само когато s1 е краен поток, s2 може да е безкраен поток.

Извеждане на поток:

```
(define (print-stream s)
  (if (empty-stream? s) (newline)
     (begin
        (print (head s))
        (display " ")
        (print-stream (tail s)))))
```

Тази процедура е приложима само когато s е краен поток.

Филтриране на елементите на поток
(действа подобно на filter при списъци):

```
(define (filter-stream s pred)
   (cond ((empty-stream? s) s)
         ((pred (head s)) (cons-stream (head s)
                                       (filter-stream (tail s) pred)))
         (else (filter-stream (tail s) pred))))
```

Тази процедура е приложима дори когато s е безкраен поток.

Сумиране на два целочислени потока (допълнени с нули, ако е необходимо):

```
(define (add-streams s1 s2)
   (cond ((empty-stream? s1) s2)
         ((empty-stream? s2) s1)
         (else (cons-stream (+ (head s1) (head s2))
               (add-streams (tail s1) (tail s2))))))
```

Процедурата е приложима дори когато и двата потока s1 и s2 са безкрайни.

Функция за съставяне на краен поток от естествените числа в даден интервал:

```
(define (enum-stream a b)
  (if (> a b) the-empty-stream
     (cons-stream a (enum-stream (+ a 1) b))))
```

Потоците се използват тогава, когато е нужна структура, от която трябва да се използва някаква предварително неизвестна част. Формирането на цялата структура би било неефективно, ако тази структура има голям брой елементи и дори невъзможно, ако структурата е с безброй много елементи.

При конструиране на безкрайни потоци се използват два основни подхода – **неявен** (**индиректен**) подход и **явен** (**директен**) подход.

При първия подход потокът се генерира чрез рекурсивна процедура, при втория подход потокът се генерира чрез рекурсия по отношение на генерираната негова част.

Ще разгледаме два примера за неявно генериране на безкраен поток.

Генериране на безкраен поток от естествените числа:

```
(define (ints_from_n n)
       (cons-stream n (ints_from_n (+ n 1))))
(define integers (ints_from 0))
```

Генериране на безкраен поток от числата на Фибоначи:

```
(define (fibgen a b)
       (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 1 1))
```

И в двата случая не се достига до безкрайна рекурсия, поради реализацията на потоците чрез отложено оценяване.

Накрая ще разгледаме примери за явно генериране на безкраен поток.

Генериране на безкраен поток от единици:

```
(define ones (cons-stream 1 ones))
```

Генериране на безкраен поток от естествените числа:

```
(define ints (cons-stream 0 (add-streams ones ints)))
```

Тук идеята е, че потокът (tail ints) се получава като към всеки елемент на ints се добави единица.

Генериране на безкраен поток от числата на Фибоначи:

```
(define fibs (cons-stream 1
                  (cons-stream 1
```

(add-streams (tail fibs) fibs))))

Тук идеята е, че потокът (tail (tail fibs)) се получава като се съберат поелементно потоците fibs и (tail fibs).

15. Семантична характеризация на логическите формули и програми.

Дадена е една азбука, наречена базисна, която съдържа измежду своите символи отварящата кръгла скоба (ляв ограничител) **(**, затварящата кръгла скоба (десен ограничител) **)** и запетаята (разделител) **,**. Фиксираме множество □ от думи над тази азбука, което е обединение на следните множества:

- □ (множество на **променливите**) – изброимо безкрайно множество от думи;
- за всяко цяло n □ 0, множество Φ_n от думи – множество на **n-местните функционални символи**;
- за всяко цяло n □ 0, множество Π_n от думи – множество на **n-местните предикатни символи**;
- непразно множество **Λ** на **логическите символи**.

Налице са следните ограничения:

- множеството **Λ** се състои точно от 7 различни думи, които ще означаваме с not, true, fail, and, or, for_all, for_some;
- **Λ** □ □ = □;
- **Λ** □ Φ_n = □ за всяко n □ 0;
- **Λ** □ Π_n = □ за всяко n □ 0;
- □ □ Φ_n = □ за всяко n □ 0;
- □ □ Π_n = □ за всяко n □ 0;
- Φ_n □ Π_n = □ за всяко n □ 0;
- думите от всички множества **Λ**, □, Φ_n (n □ 0) Π_n (n □ 0) не съдържат скоби и запетаи, откъдето следва, че префиксните изрази над □ имат еднозначен синтактичен анализ.

0-местните функционални символи още наричаме **константи**.

Нека D е дадено непразно множество.
Интерпретация на символите от Φ_n в D наричаме съответствие, което на всяка дума от Φ_n съпоставя n-местна операция в множеството D, т.е. тотално изображение на декартовата степен D^n в D. При това, под 0-местна операция разбираме елемент на D.
Интерпретация на символите от Π_n в D наричаме съответствие, което на всяка дума от Π_n съпоставя n-местен предикат в множеството D, т.е. тотално изображение на декартовата степен D^n в { 0, 1}. При това, под 0-местен предикат разбираме елемент на { 0, 1}.
Под **структура** разбираме наредена тройка от непразно множество D, което наричаме **носител** на структурата,

редица { φ_n }, където за всяко цяло n $\geq$ 0 φ_n е интерпретация на символите от $\mathbf{\Phi_n}$ в D и редица { $\prec_n$}, където за всяко цяло n $\geq$ 0 $\prec_n$ е интерпретация на символите от $\mathbf{\Pi_n}$ в D.
Нека **S** = < D, { φ_n }, { $\prec_n$} > е структура.
Ако f $\in$ $\mathbf{\Phi_n}$, то вместо φ_n (f) ще пишем $f^{\mathbf{S}}$.
Ако p $\in$ $\mathbf{\Pi_n}$, то вместо $\prec_n$ (p) пишем $p^{\mathbf{S}}$.
Въведеното означение не е съвсем точно – например една дума може да е едновременно функционален и предикатен символ (на различен брой аргументи). Затова понякога е удобно да се въведе допълнително ограничение - всяка дума от Σ да попада най-много в едно от множествата $\mathbf{\Phi_n}$ и $\mathbf{\Pi_n}$, n = 0, 1, …, но ние засега няма да въвеждаме такова ограничение – ще считаме, че интерпретацията се подразбира от контекста.

Провеждаме индуктивна дефиниция за понятието **терм**.
База: Всяка константа е терм. Всяка променлива е терм.
Предположение: Нека f $\in$ $\mathbf{\Phi_n}$, n > 0 и T_1, T_2, …, T_n са термове.
Стъпка: Тогава думата f (T_1, T_2, …, T_n) е терм.

Ясно е, че всеки терм е префиксен израз над Σ, което осигурява еднозначния синтактичен анализ на термовете. Това е съществено за коректността на всички следващи дефиниции с рекурсия по построението на терма.

За всеки терм T определяме VAR (T) – **множеството на променливите** на T с индукция по построението на T.
База: Ако T е константа, определяме VAR (T) = $\emptyset$. Ако T е променлива, определяме VAR (T) = { T}.
Предположение: Нека T = f (T_1, T_2, …, T_n), където T_1, T_2, …, T_n са термове и VAR (T_1), VAR (T_2), …, VAR (T_n) са дефинирани.
Стъпка: Тогава дефинираме
VAR (T) = VAR (T_1) $\cup$ VAR (T_2) $\cup$ … $\cup$ VAR (T_n).

Твърдение: За всеки терм T, VAR (T) е крайно множество.
Доказателство: Тривиална индукция по построението на T.

Казваме, че термът T е **затворен терм**, ако VAR (T) = $\emptyset$.
Ще дадем още една индуктивна дефиниция за затворен терм.
База: Всички константи са затворени термове.
Предположение: Нека f $\in$ $\mathbf{\Phi_n}$, n > 0 и T_1, …, T_n са затворени термове.
Стъпка: Тогава f (T_1, …, T_n) е затворен терм.

Твърдение: Двете дефиниции за затворен терм са еквивалентни.
Доказателство: За едната посока с индукция се показва, че всеки затворен в смисъл на индуктивната дефиниция терм има празно множество от променливи, за другата посока с индукция се

показва, че всеки терм е или с непразно множество от променливи или е затворен в смисъла на индуктивната дефиниция.

Фиксираме структура **S** с носител D и интерпретации Φ_0, Φ_1, …; $\prec_0$, $\prec_1$, …; съответно на функционалните и на предикатните символи.
На всеки затворен терм T съпоставяме елемент от D, който наричаме **стойност** на затворения терм T в **S**.
Означаваме стойността на T в **S** с T^{S}.
Дефиницията е с индукция по построението на T.
База: Ако T е константа, то $T^{S} = \Phi_0(T)$.
Предположение: Нека $T = f(T_1, T_2, \ldots, T_n)$, където $T_1, \ldots, T_n$ са затворени термове и $T_1^{S}, \ldots, T_n^{S}$ са дефинирани.
Стъпка: Тогава $T^{S} = f^{S}(T_1^{S}, \ldots, T_n^{S})$.

Под **оценка на променливите** в **S** разбираме тотално изображение на множеството на променливите Σ в D.
Нека **v** е оценка на променливите в **S**. На всеки терм T съпоставяме елемент от D, който наричаме **стойност** на терма T в **S** при оценката **v**.
Означаваме стойността на T в **S** при оценката **v** с $T^{S,v}$.
Дефиницията е с индукция по построението на T.
База: Ако T е константа, $T^{S,v} = \Phi_0(T) = T^{S}$. Ако T е променлива, $T^{S,v} = \mathbf{v}(T)$.
Предположение: Нека $T = f(T_1, T_2, \ldots, T_n)$ и $T_1^{S,v}, T_2^{S,v}, \ldots, T_n^{S,v}$ са дефинирани.
Стъпка: Тогава $T^{S,v} = f^{S}(T_1^{S,v}, T_2^{S,v}, \ldots, T_n^{S,v})$.

Твърдение: Ако T е затворен терм, то $T^{S,v} = T^{S}$ за всяка оценка **v**.
Доказателство: Тривиална индукция по T.

Твърдение: Ако T е терм, **v** и **u** са оценки, които се съгласуват по променливите от VAR (T), то $T^{S,v} = T^{S,u}$;
Доказателство: Тривиална индукция по T.

Дефинираме понятието **атомарна формула**.
Ако $p \in \Pi_n$, $n > 0$ и $T_1, T_2, \ldots, T_n$ са термове, то $p(T_1, T_2, \ldots, T_n)$ е атомарна формула. Ако $p \in \Pi_0$, то p е атомарна формула.

Ясно е, че всяка атомарна формула е префиксен израз над Σ, което осигурява еднозначен синтактичен анализ на атомарните формули. Освен това, атомарните формули и термове съществено се различават, тъй като $\Phi_n \cap \Pi_n = \emptyset$ за всяко $n \neq 0$.

За всяка атомарна формула A определяме VAR (A) – **множеството на променливите** на A. Ако $A = p(T_1, T_2, \ldots, T_n)$,
$VAR(A) = VAR(T_1) \cup VAR(T_2) \cup \ldots \cup VAR(T_n)$.
Ако $A \in \Pi_0$, $VAR(A) = \emptyset$.

Очевидно, VAR (A) е крайно множество.

Една атомарна формула наричаме **затворена**, ако VAR (A) = □ .
Очевидно е, че атомарната формула A е затворена □
A □ $\mathbf{\Pi_0}$ или A = p (T_1, T_2, …, T_n), където T_1, T_2, …, T_n са затворени термове.

Фиксираме структура **S** с носител D и интерпретации □$_0$, □$_1$, …; $\prec_0$, $\prec_1$, …; съответно на функционалните и на предикатните символи.

На всяка затворена атомарна формула A съпоставяме елемент от { 0, 1}, наречен **стойност** на A в **S**.
Стойността на A бележим с $A^{\mathbf{S}}$.
По дефиниция, ако A = p (T_1, …, T_n), то $A^{\mathbf{S}} = p^{\mathbf{S}}$ ($T_1^{\mathbf{S}}$, …, $T_n^{\mathbf{S}}$).
Ако A □ $\mathbf{\Pi_0}$, то $A^{\mathbf{S}} = \prec_0$ (A).
Ако $A^{\mathbf{S}}$ = 1, казваме че A е **вярна** в **S**,
ако $A^{\mathbf{S}}$ = 0, казваме че A **не е вярна** в **S**.
Нека **v** е оценка на променливите. На всяка атомарна формула A съпоставяме елемент от { 0, 1}, наречен **стойност** на A в **S** при оценката **v**. Бележим стойността на A в **S** при оценката **v** с $A^{\mathbf{S},\,\mathbf{v}}$.
По дефиниция, ако A = p (T_1, T_2, …, T_n),
то $A^{\mathbf{S},\,\mathbf{v}} = p^{\mathbf{S}}$ ($T_1^{\mathbf{S},\,\mathbf{v}}$, $T_2^{\mathbf{S},\,\mathbf{v}}$, …, $T_n^{\mathbf{S},\,\mathbf{v}}$). Ако A □ $\mathbf{\Pi_0}$, то $A^{\mathbf{S},\,\mathbf{v}} = \prec_0$ (A) = $A^{\mathbf{S}}$.

Твърдение: Ако A е затворена атомарна формула, то $A^{\mathbf{S},\,\mathbf{v}} = A^{\mathbf{S}}$ за всяка оценка **v**.
Доказателство: Прилагаме съответното твърдение от термове.

Твърдение: Ако A е атомарна формула, а **v** и **u** са оценки, които се съгласуват върху променливите от VAR (A), то $A^{\mathbf{S},\,\mathbf{v}} = A^{\mathbf{S},\,\mathbf{u}}$.
Доказателство: Прилагаме съответното твърдение от термове.

Дефинираме индуктивно **логическа формула**.
База: Всяка атомарна формула е логическа формула.
Думите true, fail са логически формули.
Предположение: Нека F е логическа формула, n > 1 и
F_1, F_2, …, F_n са логически формули.
Стъпка: Думата not(F) е логическа формула – нарича се **отрицание** на F. Думата and(F_1, F_2, …, F_n) е логическа формула – нарича се **конюнкция** на формулите F_1, F_2, …, F_n. Думата or(F_1, F_2, …, F_n) е логическа формула – нарича се **дизюнкция** на формулите F_1, F_2, …, F_n. Ако *x* е променлива, думата for_all(*x*, F) е логическа формула – нарича се **генерализация** на F по *x*. Ако *x* е променлива, думата for_some(*x*, F) е логическа формула – нарича се **екзистенциализация** на F по *x*.

Ще възприемем и други означения:
логическата формула not(F) ще означаваме с □F,

логическата формула and(F_1, F_2, …, F_n) ще означаваме с
F_1 & F_2 & … & F_n,
логическата формула or(F_1, F_2, …, F_n) ще означаваме с
$F_1 \vee F_2 \vee \ldots \vee F_n$,
логическата формула for_all(x, F) ще означаваме с $\forall x$ F,
логическата формула for_some(x, F) ще означаваме с $\hat{} \; x$ F.

Въпросът за еднозначност на синтактичния анализ отново се решава от факта, че логическите формули са префиксни изрази над Ω. Атомарните формула се прочитат еднозначно, тъй като предикатните символи са различни от логическите знаци. Също не е възможно логическа формула да е терм, тъй като функционалните символи и променливите са различни от логическите знаци.

Нека **S** е структура. За всяка формула F дефинираме **множество на променливите** на F – VAR (F) с индукция по построението.
База: Ако F е атомарна формула, то VAR (F) е вече дефинирано.
Ако F = true или F = fail, дефинираме VAR (F) = $\emptyset$.
Предположение: Нека F, F_1, …, F_n (n > 1) са формули и VAR (F), VAR (F_1), …, VAR (F_n) са вече дефинирани.
Стъпка: Дефинираме VAR ($\neg$F) = VAR (F),
VAR (F_1 & F_2 & …& F_n) = VAR (F_1) $\cup$ VAR (F_2) $\cup$ …$\cup$ VAR (F_n),
VAR ($F_1 \vee F_2 \vee \ldots \vee F_n$) = VAR ($F_1$) $\cup$ VAR (F_2) $\cup$ …$\cup$ VAR (F_n),
VAR ($\forall x$ F) = VAR (F) $\setminus$ { x}, VAR ($\hat{} \; x$ F) = VAR (F) $\setminus$ { x}.

С индукция по построението тривиално се проверява, че за всяка формула F, VAR (F) е крайно множество.

Нека **S** е структура, F е формула.
За формулата F дефинираме **множество на свободните променливите** на F – FVAR (F) с индукция по построението.
База: Ако F е атомарна формула, то FVAR (F) = VAR (F).
Ако F = true или F = fail, дефинираме FVAR (F) = $\emptyset$.
Предположение: Нека F, F_1, …, F_n (n > 1) са формули и FVAR (F), FVAR (F_1), …, FVAR (F_n) са вече дефинирани.
Стъпка: Дефинираме FVAR ($\neg$F) = FVAR (F),
FVAR (F_1 & F_2 & … F_n) = FVAR (F_1) $\cup$ FVAR (F_2) $\cup$ …$\cup$ FVAR (F_n),
FVAR ($F_1 \vee F_2 \vee \ldots \vee F_n$) = FVAR ($F_1$) $\cup$ FVAR (F_2) $\cup$ …$\cup$ FVAR (F_n),
FVAR ($\forall x$ F) = FVAR (F)$\setminus$ { x}, FVAR ($\hat{} \; x$ F) = FVAR (F)$\setminus$ { x}.

Казваме, че една формула е **затворена**, ако FVAR (F) = $\emptyset$.
Ако A е атомарна формула, то FVAR (A) = VAR (A), така че за атомарни формули тази дефиниция съвпада с предишната дефиниция за затвореност.

Ще дадем индуктивна дефиниция за **безкванторни формули**.

База: Всяка атомарна формула е безквантorна формула.
Думите true, fail са безквантorни формули.
Предположение: Нека F е безквантorна формула, n > 1 и $F_1, F_2, \ldots, F_n$ са безквантorни формули.
Стъпка: Тогава $\neg F, F_1 \& F_2 \& \ldots \& F_n, F_1 \vee F_2 \vee \ldots \vee F_n$ са безквантorни формули.

Твърдение: Ако F е безквантorна формула, то FVAR (F) = VAR (F).
Доказателство: Тривиална индукция по F.

Фиксираме структура **S** с носител D и интерпретации $\mathcal{I}_0, \mathcal{I}_1, \ldots$; $\prec_0, \prec_1, \ldots$; съответно на функционалните и на предикатните символи.

Нека **v** е оценка, x е променлива и $d \in D$.
Дефинираме оценка **u** по следния начин:
u (y) = d, ако $y = x$, **u** (y) = **v** (y) при $y \neq x$.
Оценката **u** наричаме **модификация** на **v** върху x чрез d.
Модификацията бележим по следния начин: **v**[x : d].

Нека **v** е произволна оценка. На всяка логическа формула F съпоставяме елемент от { 0, 1}, наречен **стойност** на F в **S** при оценката **v**. Стойността на F в **S** при оценка **v** означаваме с $F^{\mathbf{S},\mathbf{v}}$.
Дефинираме индуктивно стойност на формулата F при произволна оценка **v**.
База: Ако F е атомарна формула, то при произволна оценка **v** $F^{\mathbf{S},\mathbf{v}}$ е вече дефинирано.
Ако F = true, то при произволна оценка **v**, $F^{\mathbf{S},\mathbf{v}} = 1$.
Ако F = fail, то при произволна оценка **v**, $F^{\mathbf{S},\mathbf{v}} = 0$.
Предположение: Нека $F^{\mathbf{S},\mathbf{v}}, F_1^{\mathbf{S},\mathbf{v}}, F_2^{\mathbf{S},\mathbf{v}}, \ldots, F_n^{\mathbf{S},\mathbf{v}}$ са дефинирани за произволна оценка **v**.
Стъпка: Тогава за произволна оценка **v** дефинираме:
$(\neg F)^{\mathbf{S},\mathbf{v}} = 1 - F^{\mathbf{S},\mathbf{v}}$,
$(F_1 \& F_2 \& \ldots \& F_n)^{\mathbf{S},\mathbf{v}} = \min \{ F_1^{\mathbf{S},\mathbf{v}}, F_2^{\mathbf{S},\mathbf{v}}, \ldots, F_n^{\mathbf{S},\mathbf{v}}\}$,
$(F_1 \vee F_2 \vee \ldots \vee F_n)^{\mathbf{S},\mathbf{v}} = \max \{ F_1^{\mathbf{S},\mathbf{v}}, F_2^{\mathbf{S},\mathbf{v}}, \ldots, F_n^{\mathbf{S},\mathbf{v}}\}$,
$(\forall x\, F)^{\mathbf{S},\mathbf{v}} = \min \{ F^{\mathbf{S},\mathbf{v}[x:d]} \mid d \in D\}$,
$(\exists x\, F)^{\mathbf{S},\mathbf{v}} = \max \{ F^{\mathbf{S},\mathbf{v}[x:d]} \mid d \in D\}$.

Казваме, че F е **вярна** в **S** при оценката **v**, ако $F^{\mathbf{S},\mathbf{v}} = 1$.
Това записваме още по следния начин: **S**, **v** ⊨ F.
Казваме, че F е **невярна** в **S** при оценката **v**, ако $F^{\mathbf{S},\mathbf{v}} = 0$.
Това записваме още по следния начин: **S**, **v** ⊭ F.

Твърдение: Нека **S** е структура с носител D. За произволна формула F, ако две оценки **v** и **u** се съгласуват по променливите от FVAR (F), то $F^{\mathbf{S},\mathbf{v}} = F^{\mathbf{S},\mathbf{u}}$.
Доказателство: Провеждаме индукция по построението на F.

База: Ако F е атомарна формула, то FVAR (F) = VAR (F) и твърдението е вярно (по-горе). Ако F = true, то $F^{S, v} = F^{S, u} = 1$. Ако F = fail, то $F^{S, v} = F^{S, u} = 0$.
Предположение: Нека твърдението е изпълнено за формулите G, $F_1, F_2, \ldots, F_n$, n > 1 за произволни оценки **v**, **u**, съвпадащи върху съответните множества.
Стъпка: Нека **v** и **u** са произволни оценки, които съвпадат върху FVAR (F). Случаите F = □G, $F = F_1 \& F_2 \& \ldots \& F_n$ и $F = F_1 \square F_2 \square \ldots \square F_n$ следват директно от индукционното предположение.
Нека F = □*x* G. **v** и **u** съвпадат ърху FVAR (F) = FVAR (G)\{ *x*}.
Имаме $F^{S, v} = \min \{ G^{S, v[x : d]} \mid d \in D \}$, $F^{S, u} = \min \{ G^{S, u[x : d]} \mid d \in D \}$.
Ясно е, че за всяко d ∈ D имаме, че **v**[*x* : d], **u**[*x* : d] съвпадат върху FVAR (G). Така по индукционното предположение $G^{S, v[x : d]} = G^{S, u[x : d]}$ за всяко d ∈ D ⇒ $F^{S, v} = F^{S, u}$.
Аналогично се разсъждава при F = ^ *x* G.

Следствие: Ако F е затворена формула, то за произволни оценки **u** и **v** имаме $F^{S, v} = F^{S, u}$.
Доказателство: Произволни оценки **v** и **u** се съгласуват по променливите от FVAR (F) = ∅.

Ако F е затворена формула, по дефиниция $F^{S} = F^{S, v}$ за коя да е оценка **v**. Ще казваме, че F е **вярна** в **S**, ако $F^{S} = 1$.
Означаваме **S** ‘ F. Ще казваме, че F **не е вярна** в **S**, ако $F^{S} = 0$.
Означаваме **S** □F.

Нека **S** е структура с носител D.

Казваме, че формулата F е **тъждествено вярна** в **S**, ако F е вярна в **S** при всяка оценка на променливите.
Казваме, че формулата F е **изпълнима** в **S**, ако съществува оценка **v**, такава че $F^{S, v} = 1$.

Ясно е, че ако F е затворена формула, то
F е тъждествено вярна в **S** ⇔ F е вярна в **S** ⇔ F е изпълнима в **S**.

Нека **S** е структура с носител D, F е формула, *x* ∈ ∅.

Твърдение: F е тъждествено вярна в **S** ⇔ □*x* F е тъждествено вярна в **S**. F е изпълнима в **S** ⇔ ^ *x* F е изпълнима в **S**.
Доказателство:
Нека F е тъждествено вярна в **S**. Нека **v** е произволна оценка. Тогава $\square x\ F^{S, v} = \min \{ F^{S, v[x : d]} \mid d \in D\}$. Тъй като F е тъждествено вярна в **S**, то $F^{S, v[x : d]} = 1$ за всяко d ∈ D ⇒ $\min \{ F^{S, v[x : d]} \mid d \in D\} = 1$ ⇒ □*x* F е тъждествено вярна в **S**.

Нека $\forall x$ F е тъждествено вярна в **S**. Нека **v** е произволна оценка. Тогава за всяко d $\in$ D имаме $F^{\mathbf{S}, \mathbf{v}[x : d]} = 1$ при d = **v** (x) получаваме $F^{\mathbf{S}, \mathbf{v}[x : \mathbf{v}(x)]} = F^{\mathbf{S}, \mathbf{v}} = 1 \Rightarrow$ F е тъждествено вярна в **S**.
Нека F е изпълнима в **S**. Нека **v** е оценка, такава че $F^{\mathbf{S}, \mathbf{v}} = 1$. Тогава $\exists x F^{\mathbf{S}, \mathbf{v}} = \max \{ F^{\mathbf{S}, \mathbf{v}[x : d]} \mid d \in D\} = 1$, тъй като $F^{\mathbf{S}, \mathbf{v}[x : \mathbf{v}(x)]} = F^{\mathbf{S}, \mathbf{v}} = 1 \Rightarrow \exists x$ F е изпълнима в **S**.
Нека $\exists x$ F е изпълнима в **S**. Тогава съществува оценка **v**, такава че $\exists x F^{\mathbf{S}, \mathbf{v}} = 1 \Rightarrow \max \{ F^{\mathbf{S}, \mathbf{v}[x : d]} \mid d \in D\} = 1 \Rightarrow$ съществува d $\in$ D, така че $F^{\mathbf{S}, \mathbf{v}[x : d]} = 1 \Rightarrow$ F е изпълнима в **S**.

Следствие: Нека F е формула и FVAR (F) = { x_1, x_2, …, x_n}. Тогава F е тъждествено вярна в **S** $\Leftrightarrow \forall x_1 \forall x_2 \ldots \forall x_n$ F е тъждествено вярна в **S**, т.е. $\forall x_1 \forall x_2 \ldots \forall x_n$ F е вярна в **S**, тъй като тя е затворена формула. Формулата $\forall x_1 \forall x_2 \ldots \forall x_n$ F наричаме **универсално затваряне** на F, то е определено с точност до подредбата на кванторите.

Следствие: Нека F е формула и FVAR (F) = { x_1, x_2, …, x_n}. Тогава F е изпълнима в **S** $\Leftrightarrow \exists x_1 \exists x_2 \ldots \exists x_n$ F е изпълнима в **S**, т.е. $\exists x_1 \exists x_2 \ldots \exists x_n$ F е вярна в **S**, тъй като тя е затворена формула. Формулата $\exists x_1 \exists x_2 \ldots \exists x_n$ F наричаме **екзистенциално затваряне** на F, то е определено с точност до подредбата на кванторите.

Нека **S** е структура, F е произволна формула. В сила е:
F е тъждествено вярна в **S** $\Leftrightarrow \neg$F не е изпълнима в **S**,
F е изпълнима в **S** $\Leftrightarrow \neg$F не е тъждествено вярна в **S**.

Казваме, че една формула F е **тъждествено вярна**, ако F е тъждествено вярна във всяка структура **S**.
Казваме, че една формула F е **изпълнима**, ако F е изпълнима в някоя структура **S**.

Ясно е, че F е тъждествено вярна $\Leftrightarrow \neg$F не е изпълнима,
F е изпълнима $\Leftrightarrow \neg$F не е тъждествено вярна.

Ще отбележим, че не съществува алгоритъм, който разпознава дали произволна формула е тъждествено вярна (това нетривиално твърдение е известно като теорема на Чърч за неразрешимост).

Субституция наричаме изображение σ на множеството на променливите Π в множеството на термовете, такова че σ (x) $\neq x$ най-много за краен брой променливи.
Ако x_1, x_2, …, x_n са две по две различни променливи и u_1, u_2, …, u_n са термове, то със [x_1/u_1, x_2/u_2, …, x_n/u_n] означаваме субституцията σ, определена по следния начин: σ (x_i) = u_i, i = 1, 2, …, n, σ (x) = x за $x \notin$ { x_1, x_2, …, x_n}. Всяка субституция може да се зададе с краен израз по този начин. **Тъждествената**

субституция бележим с □, □ (*x*) = *x* за всяко *x* □ □ , □ = [*x*/*x*] за произволна променлива *x*.

Нека T е терм, ← е субституция. Дефинираме индуктивно термът T←, който ще наричаме **резултат от прилагането** на ← към T.
База: Ако T □ **Ф0**, то T← = T. Ако T □ □ , T← = ← (T).
Предположение: Нека T = f (T_1, T_2, …, T_n) и $T_{1←}$, $T_{2←}$, …, $T_{n←}$ са дефинирани.
Стъпка: Тогава T← = f ($T_{1←}$, $T_{2←}$, …, $T_{n←}$).

Твърдение: Нека T е терм. Ако ← и → са субституции, които се съгласуват по променливите от VAR (T), то T← = T→.
Доказателство: Индукция по построението на T.

Твърдение: Нека T е терм. Тогава T□ = T.
Доказателство: Тривиална индукция.

Твърдение: Ако T е затворен терм, то T← = T за всяка субституция ←.
Доказателство: Нека ← е произволна субституция. Тогава ← и □ съвпадат върху VAR (T) = □ □T← = T□ = T.

Твърдение: Нека T е терм, ← е субституция.
Тогава VAR (T←) = $\bigcup_{x \in VAR(T)}$ VAR (← (*x*)).
Доказателство: Индукция по построението на T.

Нека F е безквантorна формула, ← е субституция. Дефинираме индуктивно безквантorната формула F←, която ще наричаме **резултат от прилагането** на ← към F.
База: Ако F е атомарна формула, F = p (T_1, …, T_n) дефинираме F← = p ($T_{1←}$, …, $T_{n←}$), ако F □ **П0**, то F← = F.
Ако F = true или F = fail, дефинираме F← = F.
Предположение: Нека G, F_1, …, F_n (n > 1) са безквантorни формули и G←, $F_{1←}$, …, $F_{n←}$ са дефинирани.
Стъпка: Ако F = □G, дефинираме F← = □(G←).
Ако F = F_1 & F_2 & … & F_n, дефинираме F← = $F_{1←}$ & $F_{2←}$ & …& $F_{n←}$.
Ако F = F_1 □ F_2 □ … □ F_n, дефинираме F← = $F_{1←}$ □ $F_{2←}$ □ …□ $F_{n←}$.

Твърдение: Нека F е безквантorна формула. Ако ← и → са субституции, които съвпадат върху VAR (F), то F← = F→.
Доказателство: Индукция по построението на F, използваме и съответното твърдение за термове.

Твърдение: Нека F е безквантorна формула. Тогава F□ = F.

Доказателство: Тривиална индукция по F, използваме и съответното твърдение за термове.

Твърдение: Ако F е безквантворна формула и σ е субституция, то $VAR(F_\sigma) = \bigcup_{x \in VAR(F)} VAR(\sigma(x))$.
Доказателство: Индукция по построението на F, използваме и съответното твърдение за термове.

Нека **S** е структура, σ е субституция. Дефинираме $\sigma^{\mathbf{S}}$ – **оператор за присвояване**, съответен на σ в **S**, който съпоставя на всяка оценка **v** на променливите в **S** друга оценка $\sigma^{\mathbf{S}}(\mathbf{v})$ на променливите в **S**, така че $(\sigma^{\mathbf{S}}(\mathbf{v}))(x) = \sigma(x)^{\mathbf{S},\mathbf{v}}$.

Теорема: Нека **S** е структура, σ е субституция, **v** е оценка на променливите в **S**. Ако E е терм или безквантворна формула, то $(E_\sigma)^{\mathbf{S},\mathbf{v}} = E^{\mathbf{S},\mathbf{u}}$, където $\mathbf{u} = \sigma^{\mathbf{S}}(\mathbf{v})$.
Доказателство: С отделни случаи, първо когато E е терм и след това когато E е безквантворна формула.

Нека F е произволна безквантворна формула. **Частни случаи** на формулата F наричаме всички формули от вида F_σ, където σ е произволна субституция.

Например, всяка безквантворна формула F е частен случай на себе си, тъй като $F = F_\varepsilon$. Ако F е затворена безквантворна формула, то F е единственият частен случай на F, тъй като $F_\sigma = F$ за всяка субституция σ. Напротив, ако F не е затворена, то е ясно, че F има и други частни случаи, дори безброй много.

Твърдение: Нека **S** е структура и F е безквантворна формула.
В сила е:

1. Ако F е тъждествено вярна в **S**, то всички частни случаи на F са тъждествено вярни в **S**.
2. Ако някой частен случай на F е изпълним в **S**, то F е изпълнима в **S**.

Доказателство:
Нека F е тъждествено вярна в **S**. Нека σ е произволна субституция. Тогава за произволна оценка **v** в **S** имаме: $(F_\sigma)^{\mathbf{S},\mathbf{v}} = F^{\mathbf{S},\mathbf{u}}$, където $\mathbf{u} = \sigma^{\mathbf{S}}(\mathbf{v})$. Имаме $F^{\mathbf{S},\mathbf{u}} = 1 \Rightarrow (F_\sigma)^{\mathbf{S},\mathbf{v}} = 1$. Така F_σ е тъждествено вярна в **S**, т.е. всеки частен случай на F е тъждествено верен в **S**.
Нека F_σ е частен случай на F, който е изпълним в **S** (σ е някаква субституция). Нека **v** е оценка, такава че $(F_\sigma)^{\mathbf{S},\mathbf{v}} = 1$. Имаме $1 = (F_\sigma)^{\mathbf{S},\mathbf{v}} = F^{\mathbf{S},\mathbf{u}}$, където $\mathbf{u} = \sigma^{\mathbf{S}}(\mathbf{v})$. Така F е изпълнима в **S**.

Обратното твърдение също е вярно, тъй като всяка формула F е частен случай на себе си.

Ясно е, че $F_{\leftarrow}$ е затворен частен случай на F □
$_{\leftarrow}$ (x) е затворен терм за всяко x □ VAR (F).

Следствие: Нека **S** е структура, F е безквантворна формула.
В сила е:

1. Ако F е тъждествено вярна в **S**, то всички затворени частни случаи на F са верни в **S**.
2. Ако някой затворен частен случай на F е верен в **S**, то F е изпълнима в **S**.

Доказателство: Следствието е непосредствено.
Обратното твърдение на следствието не е вярно в общия случай.
Нека **S** е структура. Казваме, че **S има термално породен носител**, ако всеки елемент на носителя на **S** е стойност в **S** на някой затворен терм.

Твърдение: Нека **S** е структура с термално породен носител.
Нека F е безквантворна формула. В сила е:

1. Ако всички затворени частни случаи на F са вярни в **S**, то F е тъждествено вярна в **S**.
2. Ако F е изпълнима в **S**, то съществува затворен частен случай на F, който е верен в **S**.

Доказателство:
Нека всеки затворен частен случай на F е верен в **S**.
Нека **v** е произволна оценка. Ще покажем, че $F^{\mathbf{S},\mathbf{v}} = 1$.
Ако F е затворена формула, то тя е затворен частен случай на себе си □ F е вярна в **S**, т.е. F е тъждествено вярна в **S**.
Нека VAR (F) = { x_1, x_2, …, x_n}, n □ 1. За i = 1, 2, …, n
v (x_i) е елемент на носителя на **S**, така че съществува затворен терм T_i, такъв че $T_i^{\mathbf{S}} = \mathbf{v}$ (x_i). Разглеждаме следната субституция $_{\leftarrow} = [x_1/T_1, x_2/T_2, \ldots, x_n/T_n]$. Ясно е, че $F_{\leftarrow}$ е затворен частен случай на F, тъй като $_{\leftarrow}$ (x) е затворен терм за всяко x □ VAR (F).
Така $(F_{\leftarrow})^{\mathbf{S}} = 1$ □ $(F_{\leftarrow})^{\mathbf{S},\mathbf{v}} = 1$ □ $F^{\mathbf{S},\mathbf{u}} = 1$, където
$\mathbf{u} = {}_{\leftarrow}{}^{\mathbf{S}}(\mathbf{v})$. Остава да се съобрази, че **u** = **v**. Така $F^{\mathbf{S},\mathbf{v}} = F^{\mathbf{S},\mathbf{u}} = 1$.
Нека F е изпълнима в **S**. Нека **v** е оценка в **S**, такава че $F^{\mathbf{S},\mathbf{v}} = 1$. Ако F е затворена формула, то тя е затворен частен случай на себе си и тя е търсения верен затворен частен случай.
Нека VAR (F) = { x_1, x_2, …, x_n}, n □ 1. **v** (x_i) е елемент на носителя на **S**, така че съществува затворен терм T_i, така че $\mathbf{v}(x_i) = T_i^{\mathbf{S}}$, i = 1, 2, …, n. Нека $_{\leftarrow} = [x_1/T_1, x_2/T_2, \ldots, x_n/T_n]$. Тъй като T_1, T_2, …, T_n са затворени термове, то $F_{\leftarrow}$ е затворен частен случай на F.
Ще покажем, че $(F_{\leftarrow})^{\mathbf{S},\mathbf{v}} = 1$. Имаме $(F_{\leftarrow})^{\mathbf{S},\mathbf{v}} = F^{\mathbf{S},\mathbf{u}}$, където $\mathbf{u} = {}_{\leftarrow}{}^{\mathbf{S}}(\mathbf{v})$.
Както по-горе **u** = **v**. Така $(F_{\leftarrow})^{\mathbf{S}} = (F_{\leftarrow})^{\mathbf{S},\mathbf{v}} = F^{\mathbf{S},\mathbf{u}} = F^{\mathbf{S},\mathbf{v}} = 1$ и $F_{\leftarrow}$ е търсеният верен затворен частен случай на F.

Оттук нататък предполагаме допълнително изискване Φ_0 □ □ .

Множеството от всички затворени термове наричаме **ербранов универсум** – бележим го с **H**. При направеното изискване, **H** ≠ ∅, така че **H** може да е носител на структура.
Казваме, че една структура **S** е **ербранова**, ако е изпълнено:

1. Носителят на **S** е **H**.
2. Ако c ∈ $\mathbf{\Phi_0}$, то $c^{\mathbf{S}}$ = c.
3. Ако f ∈ $\mathbf{\Phi_n}$, n > 0, то $f^{\mathbf{S}}$ (T_1, T_2, …, T_n) = f (T_1, T_2, …, T_n) са всяка наредена n-торка T_1, T_2, …, T_n от елементи на **H**.

Твърдение: Ако **S** е ербранова структура, то за всеки затворен терм T имаме $T^{\mathbf{S}}$ = T.
Доказателство: Индукция по построението на T.

Следствие: Всяка ербранова структура **S** има термално породен носител.
Доказателство: Действително, всеки затворен терм от **H** е стойност на същия затворен терм в **S**.

Нещо повече, всеки елемент на носителя на ербранова структура е стойност на точно един затворен терм.

Твърдение: Нека M е множество от затворени атомарни формули. Тогава съществува единствена ербранова структура **S**, такава че измежду всички затворени атомарни формули в **S** са вярни точно онези, които принадлежат на M.
Доказателство: Ще построим ербрановата структура **S**. Тя има носител **H** и функционалните символи са интерпретирани по дефиницията за ербранова структура. Нека p е n-местен предикатен символ.
Ако n > 0 определяме $p^{\mathbf{S}}$ (T_1, T_2, …, T_n) = 1 ⇔ p (T_1, T_2, …, T_n) ∈ M за всяка наредена n-торка T_1, T_2, …, T_n от елементи на **H**.
При n = 0 определяме $p^{\mathbf{S}}$ = 1 ⇔ p ∈ M.
Ще покажем, че описаното свойство е в сила.
Нека A е затворена атомарна формула.
Ако A = p (T_1, T_2, …, T_n), то
$A^{\mathbf{S}}$ = $p^{\mathbf{S}}$ ($T_1^{\mathbf{S}}$, $T_2^{\mathbf{S}}$, …, $T_n^{\mathbf{S}}$) = $p^{\mathbf{S}}$ (T_1, T_2, …, T_n) = 1 ⇔
p (T_1, T_2, …, T_n) ∈ M, т.е. $A^{\mathbf{S}}$ = 1 ⇔ A ∈ M.
Ако A = p ∈ $\mathbf{\Pi_0}$, то $A^{\mathbf{S}}$ = $p^{\mathbf{S}}$ = 1 ⇔ p ∈ M, т.е. $A^{\mathbf{S}}$ = 1 ⇔ A ∈ M.
Очевидно е, че ербрановата структура с описаното свойство е единствена.

По този начин една ербранова структура се определя еднозначно, ако се зададат атомарните формули, които са вярни в нея.

Нека M е множество от формули. **Модел** на M наричаме всяка структура **S**, в която всички формули от M са тъждествено вярни.

Едно множество M от формули се нарича **изпълнимо**, ако съществува структура **S** и оценка **v**, такива че всяка формула от M вярна в **S** при оценката **v**.

Едно множество M от формули се нарича **силно изпълнимо**, ако съществува структура **S**, която е модел на M.

Ясно е, че за множества от затворени формули понятията изпълнимост и силна изпълнимост са еквивалентни.

Твърдение: Нека M е множество от безквантворни формули. Нека M□ е множеството на всички затворени частни случаи на формулите от M. Тогава следните условия са еквивалентни:

1. M е силно изпълнимо.
2. M□ е изпълнимо.
3. Съществува ербранова структура, която е модел на M.

Доказателство: Извършваме го по следната схема от импликации: 1 □ 2 □ 3 □ 1.

1 □ 2 Тъй като M е силно изпълнимо, то M има модел, т.е. структура **S**, в която всички формули от M са тъждествено вярни. При това положение, всеки затворен частен случай на формула от M е верен в **S** □ **S** е модел на M□ □ M□ е изпълнимо.

2 □ 3 Тъй като M□ е изпълнимо, съществува структура **S**, в която всяка формула от M□ е вярна. Нека M_0 е множеството от затворените атомарни формули, които са вярни в структурата **S**. Съществува ербранова структура $\mathbf{S_0}$, в която вярните атомарни формули са точно формулите от M_0. С индукция по построението се показва, че $F^{S} = F^{S_0}$ за всяка безквантворна формула F. В частност, тъй като $F^{S} = 1$ за всяка F □ M□ □ $F^{S_0} = 1$ за всяка F □ M□. С други думи, всички затворени частни случаи на формулите от M са вярни в $\mathbf{S_0}$ и тъй като $\mathbf{S_0}$ е с термално породен носител, всяка формула от M е тъждествено вярна в $\mathbf{S_0}$, т.е. $\mathbf{S_0}$ е ербранова структура, която е модел на M.

3 □ 1 Тривиално.

Следствие: Ако едно множество от универсални затваряния на безквантворни формули е изпълнимо, то съществува ербранова структура, която е негов модел.

Доказателство: Да разгледаме множеството M от безквантворните части на формулите от даденото множество. Очевидно моделите на M и на даденото множество са едни и същи. Тогава M е силно изпълнимо и от твърдението съществува ербранова структура, която е модел за M. Тази ербранова структура е модел и за първоначалното множество.

16. Операционна семантика на логическите програми.

Субституция наричаме изображение $\leftarrow$ на множеството на променливите $\square$ в множеството на термовете, такова че $\leftarrow(x) \square x$ най-много за краен брой променливи.
Ако $x_1, x_2, \ldots, x_n$ са две по две различни променливи и $u_1, u_2, \ldots, u_n$ са термове, то със $[x_1/u_1, x_2/u_2, \ldots, x_n/u_n]$ означаваме субституцията $\leftarrow$, определена по следния начин: $\leftarrow(x_i) = u_i$, $i = 1, 2, \ldots, n$, $\leftarrow(x) = x$ за $x \square \{x_1, x_2, \ldots, x_n\}$.
Очевидно, всяка субституция се представя чрез такъв краен израз и то по безброй много начини.
Тъждествената субституция бележим с $\square$, $\square(x) = x$ за всяко $x \square \square$.
Например, $\square = [x/x]$, където x е произволна променлива.

Нека T е терм, $\leftarrow$ е субституция. Дефинираме индуктивно термът $T_{\leftarrow}$, който ще наричаме **резултат от прилагането** на $\leftarrow$ към T.
База: Ако $T \square \Phi_0$, то $T_{\leftarrow} = T$. Ако $T \square \square$, $T_{\leftarrow} = \leftarrow(T)$.
Предположение: Нека $T = f(T_1, T_2, \ldots, T_n)$ и $T_{1\leftarrow}, T_{2\leftarrow}, \ldots, T_{n\leftarrow}$ са дефинирани.
Стъпка: Тогава $T_{\leftarrow} = f(T_{1\leftarrow}, T_{2\leftarrow}, \ldots, T_{n\leftarrow})$.

Твърдение: Нека T е терм. Ако $\leftarrow$ и $\rightarrow$ са субституции, които съвпадат върху VAR (T), то $T_{\leftarrow} = T_{\rightarrow}$.
Доказателство: Индукция по построението на T.

Твърдение: Нека T е терм. Тогава $T\square = T$.
Доказателство: Индукция по построението на T.

Твърдение: Ако T е затворен терм, то $T_{\leftarrow} = T$ за всяка субституция $\leftarrow$.
Доказателство: Нека $\leftarrow$ е произволна субституция. Тогава $\leftarrow$ и $\square$ съвпадат върху VAR (T) = $\square$ $\square T_{\leftarrow} = T\square = T$.

Твърдение: Нека T е терм, $\leftarrow$ е субституция.
Тогава $VAR(T_{\leftarrow}) = \bigcup_{x \square VAR(T)} VAR(\leftarrow(x))$.
Доказателство: Индукция по построението на T.

Следствие: Нека T е терм, $\leftarrow$ е субституция. Тогава термът $T_{\leftarrow}$ е затворен $\square$ термът $\leftarrow(x)$ е затворен терм за всяко $x \square$ VAR (T).
Доказателство: Директно от твърдението.

Нека F е безквантворна формула, $\leftarrow$ е субституция. Дефинираме индуктивно безквантворната формула $F_{\leftarrow}$, която ще наричаме **резултат от прилагането** на $\leftarrow$ към F.
База: Ако F е атомарна формула, $F = p(T_1, \ldots, T_n)$ дефинираме

$F_\leftarrow = p(T_{1\leftarrow}, \ldots, T_{n\leftarrow})$. Ако $F \in \mathbf{\Pi_0}$, то $F_\leftarrow = F$.
Ако F = true или F = fail, дефинираме $F_\leftarrow = F$.
Предположение: Нека G, F_1, …, F_n (n > 1) са безкванторни формули и $G_\leftarrow$, $F_{1\leftarrow}$, …, $F_{n\leftarrow}$ са дефинирани.
Стъпка: Ако $F = \neg G$, дефинираме $F_\leftarrow = \neg(G_\leftarrow)$.
Ако $F = F_1 \& F_2 \& \ldots \& F_n$, дефинираме $F_\leftarrow = F_{1\leftarrow} \& F_{2\leftarrow} \& \ldots \& F_{n\leftarrow}$.
Ако $F = F_1 \vee F_2 \vee \ldots \vee F_n$, дефинираме $F_\leftarrow = F_{1\leftarrow} \vee F_{2\leftarrow} \vee \ldots \vee F_{n\leftarrow}$.

Твърдение: Нека F е безкванторна формула. Ако $\leftarrow$ и $\rightarrow$ са субституции, които съвпадат върху VAR (F), то $F_\leftarrow = F_\rightarrow$.
Доказателство: Индукция по построението на F.

Твърдение: Нека F е безкванторна формула. Тогава $F\forall = F$.
Доказателство: Индукция по построението на F.

Твърдение: Ако F е безкванторна формула и $\leftarrow$ е субституция, то $VAR(F_\leftarrow) = \bigcup_{x \in VAR(F)} VAR(\leftarrow(x))$.
Доказателство: Индукция по построението на F.

Следствие: Нека F е безкванторна формула, $\leftarrow$ е субституция. Тогава формулата $F_\leftarrow$ е затворена $\Leftrightarrow$ термът $\leftarrow(x)$ е затворен терм за всяко $x \in VAR(T)$.
Доказателство: Директно от твърдението.

Твърдение: Нека E е терм или безкванторна формула. Ако $\leftarrow$ и $\rightarrow$ са такива субституции, че $E_\leftarrow = E_\rightarrow$, то $\leftarrow$ и $\rightarrow$ съвпадат върху VAR (E).
Доказателство: Ще разгледаме само случая когато E е терм. За безкванторните формули разсъжденията са абсолютно аналогични. Провеждаме индукция по постронието на E.
База: Нека $\leftarrow$ и $\rightarrow$ са такива субституции, че $E_\leftarrow = E_\rightarrow$.
Ако E е константа, то очевидно $\leftarrow$ и $\rightarrow$ съвпадат върху $VAR(E) = \emptyset$. Ако $E \in \Pi_0$, то $E_\leftarrow = \leftarrow(E)$, $E_\rightarrow = \rightarrow(E)$ $\Rightarrow \leftarrow(E) = \rightarrow(E)$ $\Rightarrow$ $\leftarrow$ и $\rightarrow$ съвпадат върху $VAR(E) = \{E\}$.
Предположение: Нека $E = f(T_1, T_2, \ldots, T_n)$ и твърдението е изпълнено за термовете $T_1, T_2, \ldots, T_n$.
Стъпка: Нека $\leftarrow$ и $\rightarrow$ са субституции, такива че $E_\leftarrow = E_\rightarrow$.
Имаме $E_\leftarrow = f(T_{1\leftarrow}, T_{2\leftarrow}, \ldots, T_{n\leftarrow})$, $E_\rightarrow = f(T_{1\rightarrow}, T_{2\rightarrow}, \ldots, T_{n\rightarrow})$. От еднозначния синтактичен анализ на термовете получаваме $T_{1\leftarrow} = T_{1\rightarrow}$, $T_{2\leftarrow} = T_{2\rightarrow}$, …, $T_{n\leftarrow} = T_{n\rightarrow}$. Използваме индукционното предположение за термовете $T_1, T_2, \ldots, T_n$ и получаваме, че $\leftarrow$ и $\rightarrow$ съвпадат върху $VAR(T_1)$, $VAR(T_2)$, …, $VAR(T_n)$. Тогава $\leftarrow$ и $\rightarrow$ съвпадат върху $VAR(T_1) \cup VAR(T_2) \cup \ldots \cup VAR(T_n) = VAR(E)$.

Нека σ_1 и σ_2 са субституции. Дефинираме функция σ (x) от множеството на променливите в множеството от всички термове по следния начин: σ (x) = ($x\sigma_1$)σ_2 за всяко x .
Ще покажем, че σ (x) е субституция, т.е. σ (x) x най-много за краен брой променливи x. За целта ще покажем, че за всяко x ако σ (x) x, то σ_1 (x) x или σ_2 (x) x. Да допуснем противното – σ (x) x и σ_1 (x) = x, σ_2 (x) = x за някоя променлива x . Тогава σ (x) = ($x\sigma_1$)σ_2 = $x\sigma_2$ = x, което е противоречие. Ясно е, че σ_1 (x) x или σ_2 (x) x е изпълнено само за краен брой променливи x, тъй като σ_1 и σ_2 са субституции. Така σ (x) x само за краен брой променливи x, т.е. σ (x) е субституция. Тази субституция наричаме **произведение** на субституциите σ_1 и σ_2, означаваме σ = $\sigma_1\sigma_2$. Така по дефиниция x($\sigma_1\sigma_2$) = ($x\sigma_1$)σ_2.

Ще покажем, че σ = σ = σ за всяка субституция σ.
Действително, за всяко x имаме x(σ) = ($x\sigma$) = $x\sigma$ (от съответното твърдение за термове) σ = σ. Също, за всяко x имаме x(σ) = (x)σ = $x\sigma$ $\sigma\sigma$ = σ.
Ясно е, че умножението на субституции не е комутативно.

Твърдение: Ако E е терм или безквантorна формула и σ_1, σ_2 са субституции, то E($\sigma_1\sigma_2$) = (Eσ_1)σ_2.
Доказателство: Разглеждат се отделно случаи първо за терм и след това за безквантorна формула, като и в двата случая се провежда тривиална индукция.

Твърдение (асоциативност на умножението на субституции): За всеки три субституции σ_1, σ_2, σ_3 е изпълнено: ($\sigma_1\sigma_2$)σ_3 = σ_1($\sigma_2\sigma_3$).
Доказателство: Нека x .
Тогава x (($\sigma_1\sigma_2$)σ_3) = (x($\sigma_1\sigma_2$))σ_3 = (($x\sigma_1$)σ_2)σ_3.
Също x (σ_1($\sigma_2\sigma_3$)) = ($x\sigma_1$)($\sigma_2\sigma_3$) = (($x\sigma_1$)σ_2)σ_3. Тук използваме горното твърдение за терма ($x\sigma_1$).
Така x (($\sigma_1\sigma_2$)σ_3) = x (σ_1($\sigma_2\sigma_3$)) за всяко x ($\sigma_1\sigma_2$)σ_3 = σ_1($\sigma_2\sigma_3$).

Нека F е произволна безквантorна формула. **Частни случаи** на формулата F наричаме всички формули от вида Fσ, където σ е произволна субституция.
Например, всяка безквантorна формула F е частен случай на себе си, тъй като F = F. Ако F е затворена безквантorна формула, то F е единственият частен случай на F, тъй като Fσ = F за всяка субституция σ. Напротив, ако F не е затворена, то е ясно, че F има и други частни случаи, дори безброй много.
Релацията “е частен случай на” е транзитивна: Ако G е частен случай на F и H е частен случай на G, то H е частен случай на F.
Действително по условие G = Fσ_1, H = Gσ_2 H = (Fσ_1)σ_2 = F ($\sigma_1\sigma_2$), т.е. H е частен случай на F.

Нека F е безквантворна формула. **Вариант** на F наричаме такъв частен случай G на F, такъв че F е частен случай на G.
С други думи, релацията “е вариант на” е симетричното затваряне на релацията “е частен случай на”. В такъв случай е очевидно, че релацията “е вариант на” в множеството от всички безквантворни формули е релация на еквивалентност.

Ако F е затворена формула, то F е единственият вариант на F, тъй като F е единственият частен случай на F.
Напротив, ако F е безквантворна формула, която не е затворена, то тя има безброй много варианти, както се вижда от следното

Твърдение: Нека F е безквантворна формула, която не е затворена.
Нека VAR (F) = { x_1, x_2, …, x_n}, n □ 1, x_i □ x_j при i □ j □ { 1, 2, …, n}.
Нека y_1, y_2, …, y_n са две по две различни променливи.
Ако ← = [x_1/y_1, x_2/y_2, …, x_n/y_n], то G = F← е вариант на F.
Доказателство:
Очевидно G е частен случай на F. Ще покажем, че F е частен случай на G. Да разгледаме субституцията
←□ = [y_1/x_1, y_2/x_2, …, y_n/x_n] – тя е коректно зададена, тъй като y_i □ y_j при i □ j □ { 1, 2, …, n}. Ще покажем, че F = G←□.
Имаме G←□ = (F←)←□ = F (←←□).
За i = 1, 2, …, n имаме: x_i (←←□) = (x_i←)←□ = y_i←□ = x_i.
Така ←←□ и □ съвпадат върху VAR (F) □ F (←←□) = F□ = F.
Така G←□ = F и G е вариант на F.

Тъй като множеството □ е безкрайно, по посочения начин можем да построим безброй много варианти на F. Дори можем да построим вариант на F, който не съдържа променливи от дадено крайно подмножество на □ .

Твърдение: Нека F е безквантворна формула, която не е затворена.
Нека VAR (F) = { x_1, x_2, …, x_n}, n □ 1, x_i □ x_j при i □ j □ { 1, 2, …, n}.
Нека G е произволен вариант на F. Тогава
G = F[x_1/y_1, x_2/y_2, …, x_n/y_n], за някои променливи y_1, y_2, …, y_n, които са две по две различни помежду си.
Доказателство: Тъй като G е вариант на F, то G = F←, F = G←□ за някои субституции ← и ←□. Тогава F = G←□ = (F←)←□ = F (←←□) □
□ и ←←□ съвпадат върху VAR (F), т.е. x_i (←←□) = x_i, i = 1, 2, …, n.
Да допуснем, че x_i← е съставен терм. Тогава x_i (←←□) = (x_i←)←□ също е съставен терм, тъй като след прилагането на субституцията ←□, (x_i←)←□ отново ще съдържа функционални символи. Така x_i = (x_i←)←□ е съставен терм, което е противоречие тъй като съставните термове са различни от променливите.
Да допуснем, че x_i← е константа. Тогава x_i = (x_i←)←□ = x_i←, което е противоречие, тъй като множествата **Ф0** и □ не се пресичат.

Така $x_{i\leftarrow}$ е променлива, нека $x_{i\leftarrow} = y_i \in$ **.** Тогава
$\leftarrow$ и $[x_1/y_1, x_2/y_2, \ldots, x_n/y_n]$ съвпадат върху VAR (F) $\cup$
$G = F_{\leftarrow} = F\,[x_1/y_1, x_2/y_2, \ldots, x_n/y_n]$. Ако допуснем, че за някои
$i \neq j \in \{1, 2, \ldots, n\}$ имаме $y_i = y_j \Rightarrow x_{i\leftarrow} = x_{j\leftarrow} \Rightarrow$
$(x_{i\leftarrow})_{\leftarrow\cup} = (x_{j\leftarrow})_{\leftarrow\cup} \Rightarrow x_i = x_j$, което е противоречие. Така $y_1, y_2, \ldots, y_n$ са две по две различни помежду си.

Нека A и B са произволни атомарни формули. Въпросът, който ще разглеждаме е съществува ли атомарна формула, която е частен случай както на A, така и на B, т.е. съществуват ли субституции $\mapsto\updownarrow$, такива че $A\mapsto\updownarrow = B\mapsto\updownarrow$.

Унификатор на атомарните формули A и B наричаме субституция $\leftarrow$, такава че $A_{\leftarrow} = B_{\leftarrow}$. Ако A и B притежават унификатор, казваме че A и B са **унифицируеми**.

Ако A и B са унифициеруеми, очевидно съществува атомарна формула, която е частен случай на A и на B. Обратното в общия случай не е вярно. Вярно е, обаче, следното

Твърдение: Нека A и B са атомарни формули, такива че $VAR(A) \cap VAR(B) = \emptyset$. Ако съществува атомарна формула, която е частен случай на A и на B, то A и B са унифицируеми.
Доказателство: По условие съществуват субституции $\mapsto\updownarrow$ такива че $A\mapsto\updownarrow = B\mapsto\updownarrow$. Разглеждаме изображението $\leftarrow$ от $\cup$ в множеството на всички термове, $\leftarrow(x) = \mapsto(x)$, ако $x \in VAR(A)$, $\leftarrow(x) = \updownarrow(x)$, ако $x \in \cup \setminus VAR(A)$. Ясно е, че $\leftarrow$ е субституция, тъй като $\leftarrow(x) \neq x$ най-много за променливите от $VAR(A) \cap VAR(B)$, които са краен брой. Тъй като $\leftarrow$ и $\mapsto$ съвпадат върху VAR (A), то $A\mapsto\updownarrow = A_{\leftarrow}$.
Също, тъй като $VAR(A) \cap VAR(B) = \emptyset \Rightarrow VAR(B) \subseteq \cup \setminus VAR(A) \Rightarrow$
$\leftarrow$ и $\updownarrow$ съвпадат върху $VAR(B) \Rightarrow B\updownarrow = B_{\leftarrow}$.
Така $\leftarrow$ е унификатор на A и B.

Нека A и B са произволни атомарни формули. Както знаем, съществува вариант B' на B, такъв че $VAR(A) \cap VAR(B') = \emptyset$. Ясно е, че B' и B имат едни и същи частни случаи. Тогава една атомарна формула е частен случай на A и на $B \Leftrightarrow$ тя е частен случай на A и на B'. От горното твърдение, тъй като $VAR(A) \cap VAR(B') = \emptyset$, то съществува атомарна формула, която е частен случай на A и на $B \Leftrightarrow$ A и B' са унифицируеми.
От доказателството на твърдението заключаваме, че общият вид на атомарните формули, които са частни случаи на A и на B е $A_{\leftarrow}$, където $\leftarrow$ е унификатор на A и на B'.

Нека A и B са атомарни формули.

Ако A и B имат различни предикатни символи или предикатни символи с различен брой аргументи, то очевидно A и B не са унифицируеми.

Уравнение между термове наричаме формално равенство от вида T = U, където T и U са термове. **Решение** на това уравнение наричаме субституция $\leftarrow$, такава че $T\leftarrow = U\leftarrow$.

Под **система** от уравнения между термове разбираме крайно множество от уравнения: $T_1 = U_1$, $T_2 = U_2$, …, $T_n = U_n$, T_i, U_i са термове. Една субституция $\leftarrow$ наричаме **решение** на системата, ако тя е решение на всяко едно от уравненията, т.е.
$T_1\leftarrow = U_1\leftarrow$, $T_2\leftarrow = U_2\leftarrow$, …, $T_n\leftarrow = U_n\leftarrow$.
Празното множество от уравнения също е система и нейните решения са всички субституции.

Твърдение: Нека $A = p(T_1, T_2, \ldots, T_n)$ и $B = p(U_1, U_2, \ldots, U_n)$ са атомарни формули с един и същ предикатен символ с един и същ брой аргументи. Една субституция $\leftarrow$ е унификатор на A и B $\Leftrightarrow$ $\leftarrow$ е решение на системата $T_1 = U_1$, $T_2 = U_2$, …, $T_n = U_n$.
Доказателство: Нека $\leftarrow$ е произволна субституция.
Имаме $A\leftarrow = p(T_1\leftarrow, T_2\leftarrow, \ldots, T_n\leftarrow)$, $B\leftarrow = p(U_1\leftarrow, U_2\leftarrow, \ldots, U_n\leftarrow)$.
От еднозначния синтактичен анализ имаме:
$A\leftarrow = B\leftarrow \Leftrightarrow T_1\leftarrow = U_1\leftarrow, T_2\leftarrow = U_2\leftarrow, \ldots, T_n\leftarrow = U_n\leftarrow$, т.е. $\leftarrow$ е унификатор на A и B $\Leftrightarrow$ $\leftarrow$ е решение на системата $T_1 = U_1$, $T_2 = U_2$, …, $T_n = U_n$.

Твърдението е вярно дори при n = 0 (по-точно при A = p = B, където p е 0-местен предикатен символ) – тогава унификаторите на A и B са всички субституции, а решенията на празната система също са всички субституции.

Нека $\leftarrow_1$ и $\leftarrow_2$ са субституции. Казваме, че $\leftarrow_2$ е **частен случай** на $\leftarrow_1$, ако съществува субституция σ, такава че $\leftarrow_2 = \leftarrow_1\sigma$.
Ясно е, че релацията “е частен случай на” върху субституциите е рефлексивна и транзитивна.

Нека $\leftarrow_1$ и $\leftarrow_2$ са субституции. Казваме, че $\leftarrow_2$ е **вариант** на $\leftarrow_1$, ако $\leftarrow_2$ е частен случай на $\leftarrow_1$, такъв че $\leftarrow_1$ е частен случай на $\leftarrow_2$.
С други думи, както при безквънторни формули, релацията “е вариант на” е симетрично затваряне на релацията “е частен случай на” за субституции. Естествено, в такъв случай релацията “е вариант на” за субституциите е релация на еквивалентност.

Изисква се малко труд за да се покаже, че вариантите на една субституция $\leftarrow$ се изчерпват със $\leftarrow\sigma$, където σ е субституция, която е биекция на σ върху σ.

Ще казваме, че една система е в **решен вид**, ако тя има вида $x_1 = U_1, x_2 = U_2, \ldots, x_n = U_n$, където $x_1, x_2, \ldots, x_n$ са две по две различни помежду си променливи, $U_1, U_2, \ldots, U_n$ са термове и $x_i \notin VAR(U_j)$ за всеки $i, j \in \{1, 2, \ldots, n\}$.
От тази дефиниция, по тривиални съображения, празната система е в решен вид.
Тъй като $x_1, x_2, \ldots, x_n$ са две по две различни помежду си можем да построим субституцията $\sigma_0 = [x_1/U_1, x_2/U_2, \ldots, x_n/U_n]$
($\sigma_0 = \varepsilon$ при $n = 0$).

Твърдение: Субституцията σ_0 е решение на горната система в решен вид и ако σ е произволно решение на тази система, $\sigma = \sigma_0\sigma$.
Доказателство: За $i = 1, 2, \ldots, n$ имаме $x_i\sigma_0 = U_i$, $U_i\sigma_0 = U_i\varepsilon = U_i$, тъй като σ_0 и ε съвпадат върху $VAR(U_i)$. Така $x_i\sigma_0 = U_i = U_i\sigma_0$, $i = 1, 2, \ldots, n$ $\Rightarrow \sigma_0$ е решение на системата.
Нека σ е произволно решение на системата, т.е. $x_i\sigma = U_i\sigma$, $i = 1, 2, \ldots, n$. Тогава $x_i(\sigma_0\sigma) = (x_i\sigma_0)\sigma = U_i\sigma = x_i\sigma$, $i = 1, 2, \ldots, n$.
Също за всяко $x \in V \setminus \{x_1, x_2, \ldots, x_n\}$, $x(\sigma_0\sigma) = (x\sigma_0)\sigma = x\sigma$. Така $\sigma = \sigma_0\sigma$.

Нека $T_1 = U_1, T_2 = U_2, \ldots, T_n = U_n$ е система от уравнения между термове. Нека σ е решение на системата. Тогава всички частни случаи на σ също са решения. Действително, нека τ е произволна субституция. За $i = 1, 2, \ldots, n$ имаме $T_i(\sigma\tau) = (T_i\sigma)\tau = (U_i\sigma)\tau = U_i(\sigma\tau)$ $\Rightarrow \sigma\tau$ е решение на системата.

Под **най-общо решение** на системата $T_1 = U_1, T_2 = U_2, \ldots, T_n = U_n$ разбираме решение σ, такова че всяко друго решение на системата е частен случай на σ. По-долу ще покажем, че ако една система има решение, то тя има най-общо решение σ и тогава всички решения на системата се изчерпват с частните случаи на σ. В случая когато системата е в решен вид, т.е. има вида $x_1 = U_1, x_2 = U_2, \ldots, x_n = U_n$, където $x_1, x_2, \ldots, x_n$ са две по две различни помежду си променливи, които не участват в никое U_i, от горното твърдение субституцията $\sigma_0 = [x_1/U_1, x_2/U_2, \ldots, x_n/U_n]$ ($\sigma_0 = \varepsilon$ при $n = 0$) е най-общо решение на тази система, така че решенията на тази система се изчерпват със $\sigma_0\tau$, където τ е произволна субституция.

Да разгледаме уравнение от вида $T = U$, където $T \notin V$, $U \notin V$ и T и U имат различни функционални символи или функционални символи с различен брой аргументи. Ясно е, че за всяка субституция σ имаме $T\sigma \neq U\sigma$, т.е. това уравнение няма решение.

Да разгледаме уравнение от вида $x = U$, където $x \notin V$, U е терм, $U \neq x$ и $x \notin VAR(U)$.

Твърдение: При горните предположения за всяка субституция σ имаме $x\sigma \neq U\sigma$, т.е. уравнението $x = U$ няма решение.
Доказателство: С индукция по построението ще покажем следното свойство: За всеки терм W, ако $x \in VAR(W)$ и $x \neq W$, то за всяка субституция σ, $|W\sigma| > |x\sigma|$.
База: Ако W е константа, свойството е тривиално изпълнено, тъй като една от предпоставките ($x \in VAR(W)$) не е удоволетворена. Ако W е променлива, свойството отново е тривиално изпълнено, тъй като двете препоставки ($x \in VAR(W)$, $W \neq x$) не могат да са изпълнени едновременно.
Предположение: Нека $W = f(W_1, W_2, \ldots, W_k)$ и свойството е изпълнено за термовете $W_1, W_2, \ldots, W_k$.
Стъпка: Нека σ е произволна субституция. Имаме $W\sigma = f(W_1\sigma, W_2\sigma, \ldots, W_k\sigma)$. Ясно е, че $|W\sigma| > |W_i\sigma|$ за всяко $i = 1, 2, \ldots, n$. Тъй като $x \in VAR(W)$, то $x \in VAR(W_i)$ за някое $i \in \{1, 2, \ldots, k\}$. Ако $W_i \neq x$, то по индукционното предположение $|W_i\sigma| > |x\sigma| \Rightarrow |W\sigma| > |W_i\sigma| > |x\sigma|$.
Ако $W_i = x$, то $|W\sigma| > |W_i\sigma| = |x\sigma|$.
Така и в двата случая $|W\sigma| > |x\sigma|$.
Сега, ако допуснем, че σ е решение на системата $x = U$ ще получим, че $|x\sigma| = |U\sigma|$, което е противоречие.

Уравнения от горните два вида наричаме **явно нерешими**.
Казваме, че една система е **явно нерешима**, ако тя съдържа явно нерешимо уравнение. Ясно е, че такава система няма решение.

Две системи от уравнения между термове наричаме **еквивалентни**, ако множествата от решенията им съвпадат.
Ще посочим четири еквивалентни преобразувания на системи, т.е. преобразувания чрез които една система преминава в еквивалентна на нея система.
Нека $T_1 = U_1, T_2 = U_2, \ldots, T_n = U_n$ е система от уравнения между термове.
1. **Премахване на тъждество** – ако за някое $i \in \{1, 2, \ldots, n\}$ имаме $T_i = U_i$ премахваме това уравнение от системата. Системата очевидно преминава в еквивалентна на нея, тъй като премахнатото уравнение се удоволетворява от всяка субституция.
2. **Разпадане** – ако за някое $i \in \{1, 2, \ldots, n\}$ имаме $T_i = f(V_1, V_2, \ldots, V_m)$, $U_i = f(W_1, W_2, \ldots, W_m)$, $m > 0$ и $T_i \neq U_i$, заменяме уравнението $T_i = U_i$ с уравненията $V_1 = W_1, V_2 = W_2, \ldots, V_m = W_m$. Системата преминава в еквивалентна на нея, тъй като σ е решение на $T_i = U_i$, т.е. $T_i\sigma = U_i\sigma \Rightarrow$ $f(V_1\sigma, V_2\sigma, \ldots, V_m\sigma) = f(W_1\sigma, W_2\sigma, \ldots, W_m\sigma) \Rightarrow$ (от еднозначния синтактичен анализ) $V_1\sigma = W_1\sigma, V_2\sigma = W_2\sigma, \ldots, V_m\sigma = U_m\sigma$.
Изисква се $T_i \neq U_i$, тъй като при $T_i = U_i$ е удачно да се извърши премахване на тъждество.

3. **Обръщане** – ако за някое $i \in \{1, 2, \ldots, n\}$ имаме $T_i \notin \mathbb{Z}$, $U_i \notin \mathbb{Z}$, то заменяме уравнението $T_i = U_i$ с $U_i = T_i$. Очевидно системата преминава в еквивалентна на нея.
4. **Заместване** – ако за някое $i \in \{1, 2, \ldots, n\}$ имаме $T_i \notin \mathbb{Z}$, $T_i \notin VAR(U_i)$ и $T_i \notin VAR(U_j) \cup VAR(T_j)$ за някое $j = 1, 2, \ldots, n$, то заменяме системата със следната:
$T_1[T_i/U_i] = U_1[T_i/U_i], \ldots, T_{i-1}[T_i/U_i] = U_{i-1}[T_i/U_i],$
$T_i = U_i, T_{i+1}[T_i/U_i] = U_{i+1}[T_i/U_i], \ldots, T_n[T_i/U_i] = U_n[T_i/U_i].$
Ще покажем, че двете системи са еквивалентни.
Да означим $\leftarrow_0 = [T_i/U_i]$. Нека $\leftarrow$ е произволно решение на изходната система. Тъй като $\leftarrow$ е решение на уравнението $T_i = U_i$, $T_i \notin \mathbb{Z}$ и $T_i \notin VAR(U_i)$, то от по-предното твърдение, $\leftarrow = \leftarrow_0\leftarrow$.
За $j \in i \in \{1, 2, \ldots, n\}$ $T_j\leftarrow = T_j(\leftarrow_0\leftarrow) = (T_j\leftarrow_0)\leftarrow$ и $U_j\leftarrow = U_j(\leftarrow_0\leftarrow) = (U_j\leftarrow_0)\leftarrow \in (T_j\leftarrow_0)\leftarrow = (U_j\leftarrow_0)\leftarrow$. Така $\leftarrow$ е решение и на преобразуваната система.
Нека $\leftarrow$ е произволно решение на преобразуваната система. Отново $\leftarrow$ е решение на уравнението $T_i = U_i$, така че $\leftarrow = \leftarrow_0\leftarrow$.
За $j \in i \in \{1, 2, \ldots, n\}$ $T_j\leftarrow = T_j(\leftarrow_0\leftarrow) = (T_j\leftarrow_0)\leftarrow$ и
$U_j\leftarrow = U_j(\leftarrow_0\leftarrow) = (U_j\leftarrow_0)\leftarrow \Rightarrow T_j\leftarrow = U_j\leftarrow$, т.е. $\leftarrow$ е решение на изходната система.

Теорема: Всяка система от уравнения между термове може да се приведе чрез четирите еквивалентни преобразувания до система в решен вид или до явно нерешима система.
Доказателство:
Ще посочим алгоритъм за преобразуване на произволна система чрез еквивалентните преобразувания до система в решен вид или до явно нерешима система. Алгоритъмът се състои от следните стъпки:

1. Ако системата не е в решен вид или не е явно нерешима към 2., иначе край.
2. Към системата прилагаме някое от четирите еквивалентни преобразувания (което е възможно) и отиваме към 1.

Ще докажем, че алгоритъмът е коректен.
За целта трябва да покажем две неща:
1. Ако една система не е в решен вид и не е явно нерешима, то е възможно да се извърши някое от четирите преобразувания.
2. Алгоритъмът приключва след краен брой стъпки.
Нека $T_1 = U_1, T_2 = U_2, \ldots, T_n = U_n$ е система от уравнения между термове, която не е в решен вид и не е явно нерешима.
Да допуснем, че не е възможно да извършим никое от четирите преобразувания. Преди всичко, $T_i \notin U_i$ за $i \in \{1, 2, \ldots, n\}$, тъй като в противен случай бихме могли да извършим премахване на тъждество. Първо ще покажем, че $T_1, T_2, \ldots, T_n$ са променливи.
Да допуснем, че за някое $i \in \{1, 2, \ldots, n\}$ T_i не е променлива.
Ако U_i е променлива, тогава можем да извършим обръщане – противоречие. Ако U_i е константа, то уравнението $T_i = U_i$ е явно нерешимо, тъй като $T_i \notin U_i$ системата е явно нерешима –

противоречие. Ако U_i е съставен терм и T_i, U_i имат еднакви функционални символи с един и същ брой аргументи, то можем да извършим разпадане – противоречие. Ако U_i е съставен терм и T_i, U_i имат различни функционални символи или функционални символи с различен брой аргументи, то уравнението $T_i = U_i$ е явно нерешимо системата е явно нерешима – противоречие.
Във всички случаи получихме противоречие с допускането, че T_i не е променлива $T_1, T_2, \ldots, T_n$ са променливи.
По предположение системата не е явно нерешима и тъй като $T_i \in U_i$, то $T_i \in VAR(U_i)$, $i = 1, 2, \ldots, n$.
Ако допуснем, че за някои $i \neq j \in \{1, 2, \ldots, n\}$ $T_i \in VAR(U_j) \in \{T_j\}$ ще получим противоречие, тъй като бихме могли да извършим заместване. Така променливите $T_1, T_2, \ldots, T_n$ са две по две различни и $T_i \notin VAR(U_j)$ за всеки $i, j \in \{1, 2, \ldots, n\}$, т.е. системата е в решен вид, което е противоречие с допускането, че системата не е в решен вид. Така към всяка система от уравнения между термове, която не е в решен вид и не е явно нерешима може да се приложи някое от четирите еквивалентни преобразувания.
Ще покажем, че алгоритъмът завършва след краен брой стъпки.
Нека T е терм. Дефинираме височина на терма h (T) с индукция по построението.
База: Ако T е променлива, дефинираме h (T) = 1.
Ако T е константа, дефинираме h (T) = 2.
Предположение: Нека $T = f(T_1, T_2, \ldots, T_n)$ и $h(T_1), h(T_2), \ldots, h(T_n)$ са дефинирани.
Стъпка: Дефинираме $h(T) = h(T_1) + h(T_2) + \ldots + h(T_n) + 1$.
Съвсем лесно с индукция по построението се проверява, че височината на всеки терм е положително цяло число. При това термовете, които не са променливи имат височина по-голяма от 1.
Под височина на системата от уравнения между термове $T_1 = U_1, T_2 = U_2, \ldots, T_n = U_n$ ще разбираме числото $h(T_1) + h(T_2) + \ldots + h(T_n)$. Ще изследваме как се променя височината на системата при извършване на еквивалентните преобразувания:
1. При премахване на тъждество височината намалява, тъй като от лявата страна премахваме терм и височината на всеки терм е положителна.
2. При разпадане от лявата страна премахваме съставен терм $f(V_1, V_2, \ldots, V_m)$ и прибавяме термовете $V_1, V_2, \ldots, V_m$.
От дефиницията за височина на терм височината на системата намалява с 1.
3. При обръщане от лявата страна премахваме терм, който не е променлива и го заменяме с променлива. Тъй като всеки терм, който не е променлива има височина по-голяма от 1, а променливите имат височина 1, то височината на системата намалява.

4. При заместване лявата страна или не се променя или променливи се заместват с термове, така че височината на системата или не се променя или нараства.
Така при първите три преобразувания височината на системата намалява, така че алгоритъмът може да извърши само краен брой пъти тези преобразувания едно след друго, т.е. без извършване на заместване.
Ще казваме, че системата от уравнения между термове $T_1 = U_1, T_2 = U_2, \ldots, T_n = U_n$ е решена относно променливата x, ако $x = T_i$ за някое $i \in \{1, 2, \ldots, n\}$, $x \notin VAR(T_j)$ $\in VAR(U_j)$, $j \neq i \in \{1, 2, \ldots, n\}$ и $x \notin VAR(U_i)$, т.е. x не се среща никъде другаде в системата освен като лява част на точно едно уравнение.
Например, ако системата е в решен вид, то тя е решена относно променливите $T_1, T_2, \ldots, T_n$.
Нека системата е решена относно променливата x, $x = T_i$.
Ще покажем, че при всяко от еквивалентните преобразувания системата остава решена относно x:
1. При премахване на тъждество не можем да премахнем $T_i = U_i$, тъй като $x \notin VAR(U_i)$ $\Rightarrow T_i \neq U_i$. Така системата остава решена относно x.
2. При разпадане уравнението $T_i = U_i$ не се променя, тъй като T_i не е съставен терм. Ясно е, че в новодобавените уравнения x не присъства, тъй като x не присъства в уравнението, което се разпада. Така системата остава решена относно x.
3. При обръщане уравнението $T_i = U_i$ не се променя, тъй като T_i е променлива. Системата очевидно остава решена относно x.
4. При заместване не можем да използваме уравнението $T_i = U_i$, тъй като променливата x не се среща на никое друго място в системата. Така заместваме променлива, различна от x (с термове, които не съдържат x) и получената системата ще остане решена относно x.
Нека Δ_0 е множеството от всички променливи, които се срещат в първоначалната система. Очевидно, Δ_0 е крайно множество.
Разглеждаме броят на променливите от Δ_0, относно които системата не е решена. Съгласно горните разсъждения при извършване на кое да е от четирите еквивалентни преобразувания този брой не може да нарастне. Освен това, при заместване този брой намалява с 1. Действително, ако заместваме променливата T_i, то тази система не е решена относно T_i, тъй като T_i се среща поне на още едно място в системата. Също $T_i \notin VAR(U_i)$ и след заместване на T_i с U_i в другите уравнения ще получим система, която е решена относно T_i.
От направените разсъждения получаваме, че алгоритъмът може да извърши само краен брой пъти заместване, тъй като Δ_0 е крайно множество.
Така алгоритъмът извършва само краен брой пъти заместване и само краен брой пъти последователно останалите три преобразувания, така че той приключва след краен брой стъпки.

След приключване на алгоритъма получената система очевидно е еквивалентна на изходната система. Ако получената система е явно нерешима, то изходната система няма решение. Ако получената система е в решен вид, най-общото решение на изходната система се дава с най-общото решение на получената система в решен вид. В частност, всяка система която има поне едно решение има най-общо решение и решенията на системата са частните случаи на най-общото решение.

Твърдение: Ако A и B са две унифицируеми атомарни формули, то измежду техните унификатори има най-общ, т.е. съществува унификатор σ на A и B, така че всички унификатори на A и B са $\sigma\theta$, където θ е произволна субституция.
Доказателство: Тъй като A и B са унифицируеми,
то $A = p(T_1, T_2, \ldots, T_n)$, $B = p(U_1, U_2, \ldots, U_n)$, $n \geq 1$ (p е n-местен предикатен символ) или $A = p = B$ (p е n-местен предикатен символ за $n = 0$). Тогава унификаторите на A и B (и в двата случая) са решенията на системата $T_1 = U_1, \ldots, T_n = U_n$. Тази система има решение, тъй като A и B са унифицируеми, така че тя притежава най-общо решение σ и всички други решения на системата са $\sigma\theta$, където θ е произволна субституция. Така σ е най-общ унификатор на A и на B.

Ще отбележим, че най-общият унификатор на две унифицируеми атомарни формули е определен с точност до вариант, т.е. най-общите унификатори на две унифицируеми атомарни формули са вариантите на кой да е от техните най-общи унификатори.

17. Пространство на състоянията – основни понятия и задачи. Стратегии и алгоритми за неинформирано и информирано търсене на път до определена цел. Представяне и използване на знания – основни понятия и формализми.

Решаването на много задачи, традиционно смятани за интелектуални, може да бъде сведено до последователно преминаване от едно описание (формулировка) на задачата към друго, еквивалентно на първото или по-просто от него, докато се стигне до това, което се смята за решение на задачата.
Състояние е едно описание (формулировка) на задачата в процеса на нейното решаване. Видовете състояния са начално, междинно и крайно (целево).
Оператор е начин (правило, алгоритъм, функция, връзка и т.н.) по който от едно състояние се получава друго.
Пространство на състоянията е съвкупността от всички възможни състояния, които могат да се получат от началните състояния посредством операторите.

Най-естественият начин за представяне на пространството на състоянията е чрез ориентиран граф с възли – състоянията и ориентирани дъги – операторите.
Основните действия свързани с пространството на състоянията са следните:

1. Генериране на състояния – генериране на един от наследниците или на всички наследници на дадено състояние.
2. Оценяване на състояния – оценките може да са двоични (например, true/false за целево/нецелево състояние) или числови в определен интервал (оценката на целевите състояния съвпада с единия край на интервала).

Основните типове задачи свързани с пространството на състоянията са:

1. Генериране на цялото пространство на състоянията.
2. Решаване на задачи върху генерирано пространство.
3. Комбинирана задача – едновременно постепенно генериране и оценяване на генерираните до момента състояния.

Стратегиите за решаване на тези задачи са сходни и затова обикновено се говори само за търсене (а се подразбира и/или генериране).
Три са основните типове задачи за търсене в пространството на състоянията:

1. Търсене на път до определена цел – търси се път от някакво начално състояние до определено целево състояние (или кое да е състояние от определено множество от целеви състояния), вариант на това е търсене на оптимален (по някакъв критерий) път, пример е задачата за търговския пътник.
2. Търсене на печеливша стратегия (при игра за двама играчи) – примери са шахмат, кръстчета и нулички и др.
3. Търсене на цел при ограничителни условия – примери са задачата за осемте царици, решаване на криптограми и др.

За решаване на задачата търсене на път до цел (или по-точно за генериране и обхождане на необходимата част от пространството на състоянията) се използват два основни типа методи:

1. **Неинформирано**, **сляпо** търсене – извършва се при липса на допълнителна информация за спецификата на решаваната задача, обикновено се осъществява чрез пълно изчерпване по фиксирана стратегия, крайно неефективно е често дори практически невъзможно поради получаване на комбинаторен взрив и изчерпване на компютърните ресурси.
2. **Информирано**, **евристично** търсене – извършва се при наличие на допълнителна информация за задачата, осъществява се чрез пълно изчерпване по гъвкава стратегия или чрез търсене с отсичане на част от графа на състоянията, което води до повишаване на ефективността на търсенето, с цената на известен риск – пренебрегват се

части от пространството на състоянията и не винаги се достига до оптимален път, а понякога въобще не се достига до решение.

Обикновено в задачата за търсене на печеливша стратегия се разглеждат интелектуални игри с пълна информация, които се играят от двама души и върху хода на които не оказват влияние случайни фактори. Двамата играчи играят последователно и всеки от тях има пълна информация за игровата ситуация. В определен момент от развитието на играта става ясно, че някой от играчите печели или играта е равна. Очевидно е, че задачата се свежда до намиране на методи за търсене на най-добър пръв ход на играча, който трябва да направи текущия ход. Пространството на състоянията при таза задача е съвкупността от всички възможни позиции в резултат от възможните ходове на двамата играчи. Известни са различни методи за решаване на задачата, като най-популярни са **минимаксната** и **алфа-бета** процедурите. Минимаксната процедура изисква построяване на цялото пространство на състоянията и намиране на оценките на състоянията без наследници. След това оценките се разпространяват отдолу нагоре, като всеки от възлите, съответни на ход на максимизиращия играч получава оценка, равна на максималната от оценките на неговите преки наследници, а всеки от възлите, съответни на ход на минимизиращия играч получава оценка, равна на минималната от оценките на неговите преки наследници.

При алфа-бета процедурата състоянията без наследници се оценяват веднага след генерирането им и при първа възможност се пресмятат съответните оценки на възлите от по-горните нива или поне се намират подходящи горни граници на оценките на възлите от минимизиращите нива и долни граници на оценките на възлите от максимизиращите нива. Тази процедура изисква генериране на пространството на състоянията в дълбочина, при което се преценява безполезността от генерирането и оценяването на някои клонове, които не оказват влияние върху резултата.

При търсенето на цел при ограничителни условия са дадени множество от променливи със съответни дефиниционни области и множество от ограничения (допустими/недопустими) за тези променливи. Целевите състояния са множествата от стойности на променливите, които не нарушават ограниченията. Основният алгоритъм тук е разпространяване на ограниченията. Първоначалните ограничения се разширяват, като в тях се включват и ограниченията, които са логически следствия от входните. Този процес се нарича разпространяване на ограниченията. След това, ако не е намерено решение, се прави предположение за някой от неуточнените параметри. По такъв начин ограниченията се засилват, след което новополучените ограничения отново се разпространяват и т.н. Целта е да се достигне до противоречие (тогава се осъществява връщане назад и

се прави ново предположение за съответния параметър, ако това е възможно и т.н.) или до целево състояние.

Формулировка на задачата за търсене до определена цел:
Даден е граф и част от върховете му са обявени за начални върхове и част за целеви върхове. Решение на задачата е път от начално до крайно състояние. Пътят може да се търси под формата на списък от възли или списък от дъги в графа на състоянията. Общият алгоритъм за решаване на тази задача е следния: Последователно се изследват пътищата от началното състояние. Поддържа се фронт от пътищата, които са били изследвани. По време на процеса на търсене фронтът се разширява в посока към неизследваните възли, докато се достигне до целеви възел. Начинът, по който фронтът се разширява дефинира стратегията на търсене. Общата схема на алгоритъма е следната:

1. Инициализация. Определят се начално състояние s, целево състояние g и генерираща функция gen, която генерира списък от преките наследници на дадено състояние, създават се списъци unexplored (на неизследваните състояния) и explored (на вече изследваните състояния), като се присвояват начални стойности на двата списъка: unexplored = (s) и explored = ().
2. Последователно изпълнение на процедурата search:
    a. Ако unexplored == (), към стъпка 3.
    b. cs = първият елемент на списъка unexplored.
    c. Изтриване на cs от списъка unexplored.
    d. Ако cs == g, към стъпка 3.
    e. Изпълняване на gen за cs.
    f. Наследниците на cs, които не се съдържат в списъка explored се добавят в списъка unexplored.
    g. Добавяне на cs към списъка explored.
    h. Обратно към a.
3. Извеждане на резултата. Ако cs == g се използва списъкът explored за да се изведе пътя до целта g, в противен случай алгоритъмът завършва с неуспех (път не е намерен).

Списъкът explored съдържа обходените вече върхове в реда, в който са били обходени (или в обратния ред). Той се използва за получаване на пътя до целта, а също и за проверяване дали текущото състояние вече не е било изследвано.

Първият вид стратегии за търсене са стратегиите за неинформирано (сляпо) търсене.
При **търсенето в широчина** се започва от нулевото ниво, което се състои от началните състояния и след това се преминава последователно през всички възли от следващото ниво, което се състои от преките наследници на възлите от предишното ниво. Това се прави докато не се изчерпят всички възли на пространството на състоянията. Търсенето в широчина е пълно,

но е скъпо (експоненциално). Схемата на търсенето в широчина се получава чрез следната модификация на общата схема: в стъпка 2.f. генерираните необходени наследници на cs се добавят в края на списъка unexplored.
При **търсенето в дълбочина** се обхожда изцяло даден клон и се преминава към следващия, като се започне от най-скоро обходен връх, който има поне един необходен наследник. Търсенето в този случай е евтино (линейно), но не е пълно (при краен граф е пълно). Схемата на търсенето в дълбочина се получава чрез следната модификация на общата схема: в стъпка 2.f. генерираните необходени наследници на cs се добавят
в началото на списъка unexplored.
Съществуват някои модификации на двата метода. При метода **търсене в ограничена дълбочина**, който е модификация на метода за търсене в дълбочина, се изследват само върхове до определена, предварително зададена дълбочина. Схемата на този метод е аналогична със схемата на търсенето в дълбочина с една модификация: в стъпка 2.f. генерираните необходени наследници на cs не се добавят в списъка unexplored, ако дълбочината на възела cs е по-голяма или равна на определената максимална възможна дълбочина. При метода **итеративно търсене по нива** се изпълнява последователно търсене в ограничена дълбочина, като се започне от максимална дълбочина 1 и на всяка итерация максималната дълбочина се увеличава с 1 до достигане до някаква горна граница (например, броят на дъгите в графа).
Ще опишем реализация на четирите метода за неинформирано търсене на Prolog.
Графът представяме чрез множество от факти от вида
arc (Node1, Node2, Cost) – означава, че съществува ребро от върха Node1 до върха Node2 с цена Cost.

```
breadth_first ([[Goal|Path]|_], Goal, FinalPath) :–
        reverse ([Goal|Path], FinalPath).
breadth_first ([Path|List], Goal, FinalPath) :–
        extend (Path, NewPaths),
        append (List, NewPaths, NewList),
        breadth_first (NewList, Goal, FinalPath).
extend ([Node|Path], NewPaths) :–
        findall ([NewNode, Node|Path], (arc (Node, NewNode, _),
                \+ member (NewNode, [Node|Path])), NewPaths).
```

Предикатът breadth_first реализира търсене в широчина. Първият аргумент на breadth_first е списък от пътища, които започват в началния връх и завършват в някакъв междинен връх (този списък се обработва като опашка), вторият аргумент на breadth_first е целевият възел, в третия аргумент се получава търсеният път.

Предикатът depth_first, който реализира търсене в дълбочина се реализира абсолютно аналогично на предиката breadth_first с една единствена разлика: редът append (List, NewPaths, NewList) се заменя с append (NewPaths, List, NewList), т.е. списъкът първи аргумент на depth_first се обработва като стек.

Предикатът depth_bound, който реализира търсене в ограничена дълбочина се реализира подобно на предиката depth_first със следните разлики: добавя се допълнителен аргумент Depth, който не се променя в процеса на изпълнение (той задава максималната допустима дълбочина) и освен това редът extend (Path, NewPaths) се заменя с реда extend1 (Depth, Path, NewPaths), като предиката extend1 се дефинира така:

```
extend1 (Depth, Path, NewPaths) :–
        length (Path, Len),
        Len < Depth, !,
        extend (Path, NewPaths).
extend1 (_, _, []).

iterative_deepening (List, Goal, FinalPath) :–
        findall (arc (X, Y, Z), arc (X, Y, Z), Graph),
        length (Graph, N),
        iterative_deepening1 (1, List, Goal, FinalPath, N).

iterative_deepening1 (Depth, List, Goal, FinalPath, N) :–
        depth_bound (Depth, List, Goal, FinalPath).
iterative_deepening1 (Depth, List, Goal, FinalPath, N) :–
        Depth1 is Depth+1,
        Depth1 <= N, !,
        iterative_deepening1 (Depth1, List, Goal, FinalPath, N).
```

Предикатът iterative_deepening реализира итеративно търсене по нива. Първият аргумент е списък от пътища, започващи в началния връх и завършващи в някакъв междинен връх, вторият аргумент е целевият връх, а в третия аргумент се получава резултата.

Вторият вид стратегии за търсене са стратегиите за информирано (евристично) търсене. Както вече споменахме, те са приложими при наличие на допълнителна специфична информация за предметната област. Тази информация трябва да е достатъчна, за да се построи **оценяваща функция** (**евристика**), която представлява числова оценка (в някакъв интервал) на възлите. Тази оценка може да служи, например, за мярка на близостта на оценяваното състояние до целевото състояние или на необходимия ресурс за достигане от оценяваното състояние до целевото състояние (в целевото състояние евристиката е 0).
При **метода на най-доброто спускане (best-first search)** евристиката се използва не за отсичане, а за осигуряване на

гъвкавост при търсенето. При него, на всяка стъпка от изпълнението от списъка на необходените възли се избира най-добрия (тук се намесва евристиката). След това се генерират наследниците на избрания връх, необходените от тях се добавят към списъка на необходените върхове, отново се избира най-добрия възел и т.н. до достигане на целеви възел. Методът е ефективен, но нито е пълен (при краен граф е пълен), нито оптимален. Схемата на този метод за търсене се получава от общата схема със следната модификация: добавя се нова стъпка между 2.f. и 2.g., в която се сортира списъка unexplored по нарастване на евристиките на възлите в него.

Търсенето в ограничена широчина (beam search) е аналогично на метода на най-доброто спускане, но след добавяне на необходените наследниците на избрания връх към списъка на необходените възли от този списък се премахват всички възли, с изключение на първите n най-добри възела (в съответствие с евристиката), където n е параметър. Този метод не е пълен дори при краен граф и не е оптимален. Схемата на метода се получава от общата схема със следната модификация: добавят се две нови стъпки между 2.f. и 2.g., в които първо се сортира списъка unexplored по нарастване на евристиките на възлите и след това от него се премахват всички възли, освен първите n възела.

Методът на най-бързото изкачване (hill climbing) може да се разглежда като частен случай на търсенето в ограничена широчина при n = 1, с тази разлика, че новият връх задължително трябва да е по-добър от предишния (в съответствие с евристиката). При този метод търсенето е еднопосочно, без възможност за възврат. Това е най-икономичният метод относно памет. Методът не е пълен дори при краен граф, нито пък е оптимален. В някои случаи методът ще приключи без да намери решение, дори такова да съществува. Това например ще стане, ако алгоритъмът достигне до локален екстремум – нецелеви възел, който е по-добър от всичките си необходени наследници или достигне до плато – текущият връх и необходените му наследници имат еднакви оценки. Схемата на метода се получава от общата схема със следната модификация: добавят се две нови стъпки между 2.f. и 2.g., като в първата стъпка се сортира списъка unexplored по нарастване на евристичните оценки на възлите, а във втората се отрязват всички върхове от списъка unexplored освен първия и ако останалия единствен възел няма по-добра евристична оценка от cs, алгоритъмът приключва.

Последният метод за евристично търсене е **търсене с минимизиране на общата цена на пътя (A*)**. Той може да се разглежда като частен случай на метода на най-доброто спускане, при който се използва специална оценяваща функция. Тази функция се получава като сума на два компонента – точен компонент, който дава цената на изминатия път от началния възел до оценявания връх и евристичен компонент, който дава приблизителна стойност на цената на оставащата част от пътя от

оценявания връх до целевия връх. Методът е пълен дори при безкрайни графи (естествено, с крайна разклоненост), стига теглата на ребрата в графа да са положителни и е оптимален, ако евристичната компонента в оценяващата функция е приемлива, т.е. никога не надценява действителната оптимална цена на оставащата част от пътя от оценявания връх до целевия връх.
В частния случай когато евристичната компонента е 0 получаваме неинформиран метод за търсене, който се нарича **търсене с равномерна цена на пътя (uniform cost search)**. Той винаги е оптимален и е пълен за безкрайни графи, ако теглата на ребрата са положителни.
Сега ще опишем реализация на всички описани методи на Prolog. Графът отново представяме както по-горе. Евристичната функция се описва с факти от вида distance (Node, Goal, D) – означава, че евристичната оценка на възела Node на оставащата част от пътя до възела Goal е D.

```
best_first ([[Goal|Path],_], Goal, FinalPath) :–
        reverse ([Goal|Path], FinalPath).
best_first ([Path|List], Goal, FinalPath) :–
        extend (Path, NewPaths),
        append (NewPaths, List, NewList),
        sort (NewList, Goal, SortedList),
        best_first (SortedList, Goal, FinalPath).
sort ([], _, []).
sort (L, Goal, [MinPath|SortedL]) :–
        min (L, MinPath, Goal, LeftL),
        sort (LeftL, Goal, SortedL).
min ([P|List], Q, Goal, [P|List1]) :–
        min (List, Q, Goal, List1),
        h (P, Goal, HP), h (Q, Goal, HQ),
        HP > HQ, !.
min ([P|List], P, _, List).
h ([Node|Path], Goal, H) :– distance (Node, Goal, H).
```

Предикатът best_first реализира метода на най-доброто спускане. Реализацията на extend е както по-горе. Предикатът sort сортира списъка от пътища, предаден като първи аргумент по метода на пряката селекция. Той има допълнителен аргумент целевият възел Goal, тъй като той участва при изчисляване на евристичните оценки (функцията h).

Предикатът beam_search, който реализира търсене в ограничена широчина се реализира почти по същия начин както предиката best_first със следните модификации: добавя се допълнителен аргумент Beam, който не се променя по време на изпълнението (задава параметърът n, който определихме в описанието на метода – максималният брой пътища в списъка от пътища), между редовете sort(…) и best_first (…) се добавя следния ред:

```
        trim (Beam, SortedList, SortedList1)
```

и първият аргумент на последващото извикване на best_first се променя от SortedList на SortedList1. Предикатът trim се реализира по следния начин:

```
trim (N, [X|L], [X|L1]) :– N > 0, !, N1 is N-1, trim (N1, L, L1).
trim (_, _, []).
```

Предикатът a_star, който реализира търсене с минимизиране на общата цена на пътя се реализира аналогично на предиката best_first с една разлика: в дефиницията на предиката min, редът

```
        h (P, Goal, HP), h (Q, Goal, HQ),
```

се заменя с реда

```
        f (P, Goal, FP), f (Q, Goal, FQ).
```

Предикатът f се дефинира така:

```
f ([Node|Path], Goal, F) :–
        reverse_path_cost ([Node|Path], G),
        distance (Node, Goal, H),
        F is G+H.
reverse_path_cost ([A, B], Cost) :– arc (B, A, Cost), !.
reverse_path_cost ([A, B|Path], Cost) :–
        arc (B, A, Cost1),
        reverse_path_cost ([B|Path], Cost2),
        Cost is Cost1+Cost2.
```

За да получим неинформираният метод за търсене с равномерна цена на пътя, променяме само реализацията на f:

```
f ([Node|Path], Goal, F) :–
        reverse_path_cost ([Node|Path], F).
```

И накрая описваме предикатът hill_climbing, който реализира методът на най-бързото изкачване.

```
hill_climbing ([Goal|Path], Goal, FinalPath) :–
        reverse ([Goal|Path], FinalPath).
hill_climbing (Path, Goal, FinalPath) :–
        extend (Path, NewPaths),
        bestpath (NewPaths, Goal, BestPath),
        betterpath (BestPath, Path, Goal, BestPath),
        hill_climbing (BestPath, Goal, FinalPath).
bestpath ([P], _, P) :- !.
bestpath ([P|Rest], Goal, Q) :–
        bestpath (Rest, Goal, P1),
        betterpath (P, P1, Goal, Q).
betterpath ([Node1|P1], [Node2|P2], Goal, [Node1|P1]) :–
        h (Node1, Goal, H1), h (Node2, Goal, H2),
        H1 < H2, !.
betterpath (_, P, _, P).
```

Знанията и **данните** изразяват две страни на категорията информация. Знанията са общата, относително постоянната и неизменяема част от информацията. Данните са по-динамичната

част и могат да се разглеждат като допълнение на знанията, тъй като носят конкретна информация за всеки обект. Рязка граница между данни и знания не съществува – не съществуват чисти знания и чисти данни. Това, което е знания в един момент от работата на системата, може да е данни в друг момент, както и данните могат да се интепретират като знания в определени случаи. Основните различия между данните и знанията в системите с изкуствен интелект са:

- различие в степента на общност и постоянност/изменяемост;
- различие в начина на ползване;
- наличие на класификационно-йерархична структура на знанията, основана на родово-видови отношения от тип обект-клас, клас-суперклас, тип-подтип, ситуация-подситуация и т.н.

Считаме, че знанията са по-общи и се предполага, че се променят рядко. Данните, от друга страна, могат да се изменят бързо и като правило дефинират конкретно решаваната задача.

Съществуват три типа знания в системите с изкуствен интелект:

1. Знания за обектите и фактите в предметната област (фактологически знания от най-ниско равнище).
2. Знания за връзките (отношенията, релациите) между обектите и фактите в предметната област.
3. Метазнания на много равнища (знания за самите знания, за структурата им, за връзките между тях, за начините на използването им и т.н.).

Всеки формализъм за представяне на знания може да бъде разгледан в два аспекта: синтактичен и логически. Синтактичният аспект се отнася до начина, по който явно зададените знания се съхраняват в системата, а логическият аспект до начина, по който явно зададените знания могат да се използват, за да бъдат извлечени допълнителни знания, които неявно се съдържат в тях. Съществуват два основни типа формализми за представяне и използване на знания: формализми от **процедурен** (алгоритмичен) тип и формализми от **декларативен** тип. Основната разлика между двата типа формализми се свежда до това дали знанията са представени във вид, който позволява лесен и естествен отговор на въпроса какво? (декларативни формализми) или са във вид, който позволява лесен и естествен отговор на въпроса как? (процедурни формализми). При декларативните формализми основна тежест пада върху представянето на знанията. При процедурните формализми съществен е начинът на използване на знанията. При декларативните формализми знанията се представят явно, в декларативен формат, а при процедурните те се съдържат в процедурите на някаква програмна система. При процедурното представяне не съществуват толкова обособени база знания и интерпретатор, както е при декларативното представяне. Процедурните формализми дават възможност за по-голяма

ефективност при използването на знанията, но при тези формализми измененията на знанията се правят по-трудно.

Първият формализъм за представяне и използване на знания, който ще разгледаме е **представяне и използване на знания чрез процедури**. Това е един типичен процедурен формализъм. Знанията се представят чрез системи от малки по обем процедури-демони, в които са кодирани знания за определени характерни ситуации и всяка от които се активира сама при настъпване на съответната ситуация. Обикновено демоните са от изчислителен характер и например пресмятат релации. За базите от въпросните процедури-демони е типична хетерархична структура, за разлика от процедурите в стандартните програмни системи.
При създаването на една процедура, в нея трябва да се вложат знанията за конкретна ситуация и да се създадат необходимите връзки с останалите процедури от базата знания. За разлика от декларативните представяния, където знанията и връзките между тях са разделени, тук както самите знания, така и връзките са заложени в процедурите. Знанията се изразяват чрез семантиката на процедурите, а връзките между тях, чрез връзки между отделните процедури, т.е. чрез обръщения между процедури и предаване на параметри. Изменението на знанията е основен проблем при този формализъм. Изменението само на една процедура може да доведе след себе си до необходимост от изменение на голяма част от процедурите на системата (например, при добавяне на допълнителен параметър на процедура). Добавянето на нови процедури от своя страна не се изчерпва с механичното им долепяне към базата от знания, а води до изменението на редица от съществуващите процедури, за да се създадат необходимите връзки.
Основните предимства на този формализъм са неговите относително по-високи ефективност и бързодействие, което се дължи на факта, че управлението се предава директно от процедура на процедура в зависимост от текущите данни. Директните връзки, които са обръщения към други процедури са заложени в самите процедури, което води до директен извод. Формализмът чрез процедури предоставя редица удобства при моделирането на процеси, тъй като дадена ситуация се обработва от конкретна процедура и управлението се предава директно точно на тази процедура, която притежава знания за съответната ситуация. Този формализъм е много удобен и при представяне на евристични знания. От друга страна, връзките са относително принудителни, което води до липса на модулност в системата и до трудности в управлението на процеса отвън. Друг недостатък е, че трудно се реализират системи за обяснение на резултата. Представянето и използването на знания чрез процедури има голямо приложение при управление на роботи, управление на процеси и др., където преобладават алгоритмичните знания.

Вторият формализъм, който разглеждаме е представяне и използване на знания **чрез средствата на математическата логика**. Това е един от най-рано възникналите формализми с ясно изразен декларативен характер. Представянето на знанията в термините на дадена формална логическа система се извършва с помощта на правилно построените формули в тази система, а използването на знанията се осъществява посредством валидните за логическата система правила и методи за извод. Най-често използваната формална логическа система е предикатното смятане от първи ред. Съществените му недостатъци са неговата полуразрешимост и недостатъчната му изразителна сила. Полуразрешимостта означава, че съществува алгоритъм, който с помощта на краен брой стъпки определя дали една формула е логическо следствие на краен брой други формули, ако въпросната формула действително е следствие, но не съществува подходящ краен алгоритъм за получаване на отрицателния отговор. Изразителната сила е недостатъчна, тъй като в системите от първи ред кванторите се отнасят само за променливи (пробягват само елементите на универсума), но не и за функции (или произволни релации). За преодоляване на този проблем могат да се използват логики от по-висок ред. Друга особеност на предикатното смятане от първи ред е неговата монотонност – в процеса на логически извод могат само да се добавят формули към началното множество, но не и да се изключват. Това свойство пречи да се задават знания или свойства, които се смятат за верни по премълчаване, т.е. знания, които се смятат за верни, докато не стане известно противното. За представяне и използване на знания, които се смятат за верни по премълчване, се използват специален тип логики, наречени немонотонни логики. В случаите на представяне на знания, които се променят във времето могат да се използват темпорални логики. Общият вид на правилно построените формули от предикатното смятане от първи ред затруднява представянето с тяхна помощ на твърдения, които съдържат модалности от типа необходимо, възможно, случайно, невъзможно и др. За представянето на такива твърдения се използват модални логики. Често в практиката се работи с твърдения, които съдържат неточни определения като много, малко, повечето и др. Неточни знания от този тип обикновено се представят с помощта на размита логика. Най-съществено предимство на логическите системи е ясната семантика.

Третият формализъм, който разглеждаме е представяне и използване на знания чрез **системи от продукционни правила**. Това е един декларативен формализъм с елементи на процедурност на по-ниско равнище. Неговата основна характеристика е декомпозирането на знанията на малки части (правила) от тип условие-следствие (ситуация-действие).

Всяка система с изкуствен интелект, основана на правила има три основни компонента: работна памет, база от правила и интерпретатор на правилата.
Работната памет (контекст) съдържа данните за решаваната задача, установени към текущия момент. Тези данни се формират от два източника. Първият източник е потребителя – той вмъква първоначалните данни в системата или вмъква допълнителни данни като отговор на допълнителен въпрос на системата. Вторият източник е интерпретатора на правилата – в зависимост от неговата реализация могат както да се вмъкват междинни резултати от логическия извод, така и да се изключват или модифицират съществуващи данни от работната памет.
Представянето на данните в работната памет зависи от синтаксиса на правилата и най-често то е чрез тройки от вида обект-атрибут-стойност (например, тройката (A, B, C) означава, че обектът A има атрибут B със стойност C) или чрез тройки от вида атрибут-релация-стойност (например, тройката (A, R, B) означава, че стойността на атрибута A е в релация R със стойността B).
Базата от правила е съвкупност от правила за дадена предметна област, които са наредени по определен начин. Общият вид на правилата е следния: (if <условие1> <условие2> …<условиеN> then <следствие1> <следствие2> … <следствиеM>). Семантиката на правилата е следната: ако условията са вярни, то са вярни и следствията.
Интепретаторът е машината за извод на съответната система. Това е програмна система, чиято основно предназначение е да приложи вложените в правилата знания в системата върху данните за конкретната задача. Често интерпретаторът се използва за да направи опит да удоволетвори някаква зададена от потребителя цел. Условно работата му може да се раздели на две фази: избор на правило и изпълнение на правилото. По същество задачата в първата фаза се свежда до търсене в базата от правила по някакъв образец. Според образеца на търсене се различават два типа системи от продукционни правила – прави и обратни.
В зависимост от това също се говори за два типа алгоритми за действие на интерпретатора – прав извод (извод, управляван от данните) и обратен извод (извод, управляван от целите).
При първия тип системи, интерпретаторът в първата фаза от работата си търси правило, чиито условия се удоволетворяват от наличните данни в контекста, след което във втората фаза изпълнява следствията на намереното правило. Ако съществува повече от едно правило, чиито условия се удоволетворяват от контекста (т.е. трябва да се извърши конфликтна резолюция), се използват различни методи за избор на правило (например: първото срещнато правило; правило, което не е прилагано до момента; правило, което използва най-скоро записани в контекста данни). Изпълнението на следствията на правилото води до изменение на контекста, което от своя страна прави възможно

изпълнението на друго правило, ако не е достигната съответната цел и т.н.
При втория тип системи, търсенето се извършва върху следствията на правилата, като за образец служи някаква зададена цел. След това се анализира лявата част на намереното правило, като неизвестните все още параметри се задават като нови цели. След това се прави опит да се удоволетворят по аналогичен начин новите цели и т.н. Този процес продължава, докато текущите цели се удоволетворят от данните в контекста или докато стане ясно, че те не могат да бъдат удоволетворени. При необходимост се извършва връщане назад, докато се удоволетворят целите или се изследват всички неизследвани възможности.
Основните предимства на системите с продукции са няколко. Първото предимство е естественост на представянето на експертни знания, което ги прави широко приложими при изграждането на експертни системи.
Второто предимство е е модулност, която се изразява в няколко аспекта – постоянните знания са отделени от временните (първите са в базата от правила, вторите са в работната памет), различните правила са семантично независими, единственият начин по който правилата общуват е свързан с промяната на съдържанието на работната памет, интерпретаторът работи независимо от смисъла на знанията и данните, които са кодирани в базата от правила и в работната памет.
Третото предимство на системите с продукции е ограниченият синтаксис, въпреки че по-нататък ще посочим и някои недостатъци, които произтичат от него. Ограниченият синтаксис позволява лесно да се изграждат програми-редактори на системи от продукции и лесно да се реализират средства за генериране и обяснение на резултата на естествен език.
Съществуват и недостатъци на системите с продукции.
Всъщност, въпреки че системите с продукции дават възможност за обясняване на резултатите от извода, в тяхния класически вид има видове знания като стратегическите и поддържащите знания, които не могат да се представят в явен вид в базата от правила, което понякога прави генерираните обяснения недостатъчно съдържателни.
Друг недостатък, който произтича от ограничения синтаксис е невъзможността за представяне на дизюнктивни знания в следствията и на отрицателни знания въобще (освен в някои специални частни случаи).
Последният недостатък е, че съществува потенциална неефективност на работата на интерпретатора, източникът на която са определянето на конфликтното множество от правила и изборът на правило от него (конфликтна резолюция).
За преодоляване на този недостатък има различни подходи – уточняване на условията в правилата, пренареждане и

ограничаване на конфликтното множество и използване на архитектура от тип “черна дъска”.

Четвъртият формализъм, който разглеждаме е представяне и използване на знания чрез **семантични мрежи**. Това е декларативен формализъм и спада към класа на структурните формализми, при които знанията се представят чрез множество от структурни единици, свързани помежду си чрез явно зададени връзки, подобно на организацията на знанията в човешкия мозък. В семантичните мрежи знанията се представят с помощта на мрежа от възли и дъги, които ги свързват (с други думи семантичната мрежа е ориентиран граф). Възлите обозначават обектите (понятията, класове и т.н.) от предметната област. Дъгите представляват връзките (отношенията, релациите) между обектите. Възлите са два вида – релационни константи, които представляват таксономични категории или стойности на свойства и обектни константи, които представят предмети или обекти от предметната област. Връзките са три вида – връзки от тип “подмножество”, които описват релации от тип клас-суперклас, връзки от тип “елемент”, които описват релации от тип обект-клас и връзки тип “функция”, които описват свойства на обектите и класовете.
Изрично ще отбележим, че дъгите изразяват бинарни отношения между възлите.
Използването на знания, представени чрез семантични мрежи обикновено е свързано с извършване на изводи за обектите и намиране на техни свойства, които не са представени в явен вид. Основният механизъм за извод върху семантична мрежа е свързан с проследяване на връзките между отделните възли в мрежата. Най-често се използват две стратегии за извод: разпространяваща се активност и наследяване.
Най-съществените предимства на семантичните мрежи са простота, естественост, нагледност и яснота на представянето. Семантичните мрежи имат доста недостатъци.
Първият недостатък е трудности при представянето на произволни n-арни релации. Този недостатък може частично (с цената на нарушаване на предимствата) да се избегне, например чрез метода на Саймънс. При него за n-арната релация се въвежда отделен възел и обектите-аргументи на релацията се свързват към новия възел с бинарни връзки.
Друг недостатък е трудности при представянето на знания, които съдържат квантори. И тук има решения, например използването на разделени семантични мрежи, но отново се нарушава простотата и естествеността на представянето. Разделената семантична мрежа е йерархична съвкупност от подмрежи, всяка от които съответства на областта на действие на една или няколко променливи. Някои специални дъги в мрежата сочат не към възли, а към цели подмрежи.

Трети недостатък е неясната семантика – една семантична мрежа често може да бъде тълкувана по различни начини.
Четвърти недостатък са проблемите, които възникват при извършването на различни операции върху семантичните мрежи – например, дублиране на информация за съвпадащи обекти при обединяване на две семантични мрежи.
И последен недостатък са проблемите при управляване на наследяването – не винаги е целесъобразно да се извършва наследяване на свойства, а трудно се задава частична забрана за наследяване.

Последният формализъм, който ще разгледаме е представяне и използване на знания чрез **фреймове**. Фреймовете са едно от структурните представяния на знания. При тях са съчетани декларативни и процедурни знания. Част от знанията са включени в статична декларативна структура фрейм. Освен това, има и допълнителни процедурни знания за това как да се използват декларативните знания от фрейма, как да се свързват тези знания със знанията от други фреймове и др.
От информационна гледна точка, фреймът представлява йерархична структура от данни с три равнища. На най-високо равнище стои идентифициращата информация, която обикновено включва името на фрейма. На следващото равнище се съдържат полета с описания на различни характеристики, качества или свойства на обекта или класа обекти, представян от фрейма (слотове или атрибути). На най-ниско равнище се намират фасетите. Всяка фасета се прикрепя към някой слот и се използва за записване на допълнителни данни за описваната в слота характеристика. Най-често използваните имена на фасети са:

- value – съдържа стойността на характеристиката, описвана от съответния слот;
- default – съдържа стойността по премълчаване (която се използва, ако във value не е зададена конкретна стойност) на характеристиката, описвана от съответния слот;
- if-needed – съдържа име или тяло на процедура, която да се изпълни при изчисляване на стойността на характеристиката, описвана от съответния слот, ако тази стойност не е дадена в явен вид и не се подразбира;
- if-added (if-removed) – съдържат имена или тела на процедури, които се активират, ако бъде добавена (изтрита) стойност на съответната характеристика.

Имената на слотовете обикновено се избират така, че да съответстват на установените термини за описваните от тях характеристики. От гледна точка на процеса на наследяване, най-важна роля играе слотът с примерно име a-k-o (a kind of). Фасетата value към този слот съдържа списък от имена на фреймове, описващи класовете (суперкласовете), към които се причислява обектът (класът), описван от дадения фрейм.

Използването на знанията, представени чрез фреймове, се свежда най-вече до две основни дейности: съпоставяне с дадено описание и логически извод.
При първата дейност, целта е по дадено описание на някаква ситуация да се намерят съществуващи в базата от знания фреймове, които съответстват в задоволителна степен на това описание.
Втората дейност се свежда до наследяване на свойства от класовете, към които принадлежи даден обект (клас).
Ако ни интересува стойността, свързана с даден слот на даден фрейм, можем най-напред да потърсим тази стойност последователно във фасетите value, default и if-needed на дадения слот. Ако и на трите места не открием нищо подходящо, можем да използваме йерархията между фреймовете, дефинирана чрез слотовете a-k-o. В слота a-k-o на дадения фрейм е записано името на съответния суперфрейм и търсенето може да продължи в съответните фасети на суперфрейма, след това – при суперфрейма на суперфрейма и т.н. Възможно е слотовете a-k-o на някои фреймове да са записани имената на повече от един суперфрейм. Така в общия случай, предходниците на един фрейм образуват дърво и при търсене трябва да се възприеме някаква стратегия за обхождане на това дърво (например, в широчина или в дълбочина). Независимо от тази стратегия, търсенето на необходимата стойност може да се извършва по два начина: Z-търсене и N-търсене.
При Z-търсенето, най-напред се обхождат фасетите value, default, if-needed на разглеждания слот на дадения фрейм, след това се преминава към първия предходник на фрейма (съгласно стратегията за обхождане на предходниците) и за същия слот на този фрейм се преглеждат отново трите въпросни фасети и т.н.
При N-търсенето най-напред се преглеждат фасетите value на дадения фрейм и на всички негови предходници, след това се преглеждат фасетите default на фрейма и на всички негови предходници и най-накрая се преглеждат фасетите if-needed на фрейма и на всички негови предходници.
Основните предимства на фреймовете са естественост на представянето, модулност и йерархична структура на базата от знания и добри възможности за използване на свойства, които са валидни по премълчаване.
При използването на фреймове възникват някои проблеми. Един от тях е неясната семантика на фреймовете – знанията, представени чрез дадена система от фреймове, често могат да бъдат тълкувани по различни начини. Друг проблем е управлението на наследяването на свойства – в случаите, когато се допуска задаване на повече от един клас, директен предходник на даден обект или клас. И третият проблем е недостатъчната изразителна сила на фреймовете в сравнение с предикатното смятане – трудно се изразяват твърдения, чиято формулировка

съдържа квантори, а също и непълни знания, които съдържат дизюнкция от свойства на различни обекти.

18. Симетрични оператори в крайномерни евклидови пространства. Основни свойства. Теорема за диагонализация.

Нека A = ($\mapsto_{ij}$) $\in$ $\mathbb{R}^{n\times n}$. A е **симетрична** матрица $\Leftrightarrow$
$A^t = A \Leftrightarrow \mapsto_{ij} = \mapsto_{ji}$ за всеки i, j $\in$ { 1, 2, …, n}.

A $\in$ $\mathbb{R}^{n\times n}$ е **ортогонална** матрица $\Leftrightarrow$ е изпълнено равенството:
$A.A^t = E \Leftrightarrow$ A е обратима и $A^{-1} = A^t$.

Твърдение: Ако A е ортогонална матрица, то A^{-1} също е ортогонална матрица.
Доказателство: Имаме $A^{-1}.(A^{-1})^t = A^t.(A^{-1})^t = (A^{-1}.A)^t = E^t = E \Rightarrow$
A^{-1} е ортогонална матрица.

Твърдение: Ако A, B са ортогонални матрици, то A.B също е ортогонална матрица.
Доказателство: Имаме $(A.B).(A.B)^t = A.B.B^t.A^t = A.E.A^t = A.A^t = E \Rightarrow$
A.B е ортогонална матрица.

Твърдение: A $\in$ $\mathbb{R}^{n\times n}$ е ортогонална матрица $\Leftrightarrow$ векторите-редове (векторите-стълбове) на A са ортонормирана система вектори в $\mathbb{R}^n$.
Доказателство: Нека A = ($\mapsto_{ij}$) $\in$ $\mathbb{R}^{n\times n}$.
$A.A^t = E \Leftrightarrow \mapsto_{i1}.\mapsto_{j1} + \mapsto_{i2}.\mapsto_{j2} + \ldots + \mapsto_{in}.\mapsto_{jn} = \delta_{ij}$
за всеки i, j $\in$ { 1, 2, …, n }.
Ако $a_1 = (\mapsto_{11}, \mapsto_{12}, \ldots, \mapsto_{1n})$, $a_2 = (\mapsto_{21}, \mapsto_{22}, \ldots, \mapsto_{2n})$, …,
$a_n = (\mapsto_{n1}, \mapsto_{n2}, \ldots, \mapsto_{nn})$ са векторите-редове на матрицата A, тези равенства показват, че $(a_i, a_j) = \delta_{ij}$ за всеки i, j $\in$ { 1, 2, …, n }, т.е. системата вектори $a_1, a_2, \ldots, a_n$ е ортонормирана.
Аналогично като използваме, че
$A.A^t = E \Leftrightarrow A^t.(A^t)^t = E$, получаваме, че векторите-стълбове на A са ортонормирана система вектори.
Твърдение: Ако A е ортогонална матрица, то $\det A = \pm 1$.
Доказателство: $A.A^t = E \Rightarrow \det A.\det A^t = \det E \Rightarrow (\det A)^2 = 1 \Rightarrow$
$\det A = \pm 1$.

Нека **V** е крайномерно евклидово пространство, n = dim**V**.
Нека E = ($e_1, e_2, \ldots, e_n$) е един базис на **V** и F = ($f_1, f_2, \ldots, f_n$) е друг базис на **V**. Нека T = (τ_{ij}) $\in$ $\mathbb{R}^{n\times n}$ е матрицата на прехода от E към F,

т.е. $f_i = \tau_{1i}.e_1 + \tau_{2i}.e_2 + \ldots + \tau_{ni}.e_n = \sum_{k=1}^{n} \tau_{ki}.e_k$.

Разглеждаме $(f_i, f_j) = (\sum_{k=1}^{n} \tau_{ki}.e_k, \sum_{m=1}^{n} \tau_{mj}.e_m) = \sum_{k=1}^{n}\sum_{m=1}^{n} \tau_{ki}.\tau_{mj}.(e_k, e_m)$,

i, j ∈ { 1, 2, …, n}.
Нека базисът E е ортонормиран ⇔ $(e_k, e_m) = \delta_{km}$ за всеки
k, m ∈ { 1, 2, …, n}.
Тогава $(f_i, f_j) = \sum_{k=1}^{n}\sum_{m=1}^{n} \alpha_{ki}.\alpha_{mj}.\delta_{km} = \sum_{k=1}^{n} \alpha_{ki}.\alpha_{kj}$ за i, j ∈ { 1, 2, …, n }.

1. Ако базисът F е ортонормиран, тогава $(f_i, f_j) = \delta_{ij}$ за
   i, j ∈ { 1, 2, …, n} ⇔ $\sum_{k=1}^{n} \alpha_{ki}.\alpha_{kj} = \delta_{ij}$ ⇔ $T^t.T = E$ ⇔ T е ортогонална
   матрица. И така матрицата на прехода между два ортонормирани базиса е ортогонална.
2. Ако матрицата T е ортогонална, тогава $\sum_{k=1}^{n} \alpha_{ki}.\alpha_{kj} = \delta_{ij}$
   за i, j ∈ { 1, 2, …, n } ⇔ $(f_i, f_j) = \delta_{ij}$ ⇔ F е система от n на брой ненулеви ортогонални вектори с дължина 1 ⇔ F е ортонормиран базис. И така всяка ортогонална матрица е матрица на прехода от един ортонормиран базис към друг ортонормиран базис.

Нека **V** е крайномерно евклидово пространство и φ ∈ **Hom (V)**.
Казваме, че φ е **симетричен линеен оператор**, ако
за всеки два вектора x, y ∈ **V** имаме (φ (x), y) = (x, φ (y)).
Ясно е, че ако $e_1, e_2, …, e_n$ е базис на **V** достатъчно е
(φ $(e_i), e_j) = (e_i,$ φ $(e_j))$ за всеки i, j ∈ { 1, 2, …, n} поради линейността на φ и линейността на скаларното произведение по двата си аргумента.

Твърдение: Нека **V** е крайномерно евклидово пространство, dim**V** = n ∈ ℕ. Нека $e_1, e_2, …, e_n$ е <u>ортонормиран базис</u> на **V**.
Нека φ ∈ **Hom (V)** и φ има матрица A спрямо този базис. Тогава
φ е симетричен линеен оператор ⇔ A е симетрична матрица.
Доказателство: Нека $A = (\alpha_{ij}) \in \mathbb{R}^{n \times n}$.
Тогава φ $(e_i) = \alpha_{1i}.e_1 + \alpha_{2i}.e_2 + … + \alpha_{ni}.e_n$. Операторът φ е симетричен
⇔ (φ $(e_i), e_j) = (e_i,$ φ $(e_j))$ за всеки i, j ∈ { 1, 2, …, n}

$$\varphi\left(\sum_{k=1}^{n} \alpha_{ki}.e_k, e_j\right) = \left(e_i, \sum_{k=1}^{n} \alpha_{kj}.e_k\right) \Leftrightarrow \sum_{k=1}^{n} \alpha_{ki}(e_k, e_j) = \sum_{k=1}^{n} \alpha_{kj}(e_i, e_k)$$

$$\Leftrightarrow \sum_{k=1}^{n} \alpha_{ki}\delta_{kj} = \sum_{k=1}^{n} \alpha_{kj}.\delta_{ik} \Leftrightarrow \alpha_{ji} = \alpha_{ij} \text{ за всеки } i, j \in \{ 1, 2, …, n \} \Leftrightarrow$$

A е симетрична матрица.

Твърдение: Ако A е реална симетрична матрица ($A \in \mathbb{R}^{n \times n}$, $A^t = A$), то всички характеристични корени на A са реални числа.
Доказателство: Нека характеристичните корени на A са
$\ell_1, \ell_2, …, \ell_n, \ell_i \in \mathbb{C}$, и нека ℓ е кой да е от тях.
Нека $A = (\alpha_{ij}), \alpha_{ij} \in \mathbb{R}$ и $\alpha_{ij} = \alpha_{ji}$ за всяко i, j ∈ { 1, 2, …, n }.
Разглеждаме системата:

$$
\begin{cases}
(a_{11} - \ell).x_1 + a_{12}.x_2 + \ldots + a_{1n}.x_n = 0 \\
a_{21}.x_1 + (a_{22} - \ell).x_2 + \ldots + a_{2n}.x_n = 0 \\
\ldots \\
a_{n1}.x_1 + a_{n2}.x_2 + \ldots + (a_{nn} - \ell).x_n = 0
\end{cases} \quad (*)
$$

Детерминантата на (*) е $f_A(\ell) = 0$ $\Rightarrow$ (*) има ненулево решение $(\alpha_1, \alpha_2, \ldots, \alpha_n) \neq (0, 0, \ldots, 0)$, $\alpha_i \in \mathbb{C}$ за всяко i = 1, 2, …, n, тъй като коефициентите на (*) са комплексни.

За всяко i = 1, 2, …, n имаме:

$a_{i1}.\alpha_1 + a_{i2}.\alpha_2 + \ldots + (a_{ii} - \ell).\alpha_i + \ldots + a_{in}.\alpha_n = 0 \Leftrightarrow$

$a_{i1}.\alpha_1 + a_{i2}.\alpha_2 + \ldots + a_{in}.\alpha_n = \ell.\alpha_i$. Умножаваме двете страни по $\overline{\alpha_i}$ и получаваме: $a_{i1}.\alpha_1.\overline{\alpha_i} + a_{i2}.\alpha_2.\overline{\alpha_i} + \ldots + a_{in}.\alpha_n.\overline{\alpha_i} = \ell.|\alpha_i|^2$, т.е.

$\sum_{j=1}^{n} a_{ij}.\alpha_j.\overline{\alpha_i} = \ell.|\alpha_i|^2$. Последното равенство сумираме по i = 1, 2, …, n и получаваме: $\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}.\alpha_j.\overline{\alpha_i} = \ell.\sum_{i=1}^{n}|\alpha_i|^2$.

Нека $v = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}.\alpha_j.\overline{\alpha_i}$, $u = \sum_{i=1}^{n}|\alpha_i|^2 \Rightarrow v = \ell.u$. Имаме $u \in \mathbb{R}$, $u \neq 0$ (дори $u > 0$), тъй като $(\alpha_1, \alpha_2, \ldots, \alpha_n) \neq (0, 0, \ldots, 0)$. Освен това,

$\overline{v} = \overline{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}.\alpha_j.\overline{\alpha_i}} = \sum_{i=1}^{n}\sum_{j=1}^{n} \overline{a_{ij}}.\overline{\alpha_j}.\overline{\overline{\alpha_i}} = \sum_{i=1}^{n}\sum_{j=1}^{n} \overline{a_{ij}}.\overline{\alpha_j}.\alpha_i = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ji}.\overline{\alpha_j}.\alpha_i =$

$= \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}.\alpha_j.\overline{\alpha_i} = v$. Използвали сме $a_{ij} \in \mathbb{R}$, $a_{ij} = a_{ji}$ за всеки i, j $\in$ {1, 2, …, n}, а в предпоследното равенство разменихме индексите, което е позволено, тъй като те се менят в еднакви множества.

Получихме, че $\overline{v} = v \Rightarrow v \in \mathbb{R}$. Така u, v $\in \mathbb{R} \Rightarrow \ell = \frac{v}{u} \in \mathbb{R}$.

Твърдение: Нека **V** е крайномерно евклидово пространство, $\dim\mathbf{V} = n \geq 1$ и φ е симетричен линеен оператор на **V**. Тогава всичките n характеристични корени на φ са реални числа, т.е. φ има n (не непременно различни) собствени стойности.

Доказателство: Нека $e_1, e_2, \ldots, e_n$ е ортонормиран базис на **V** и A е матрицата на φ спрямо този базис. Съгласно по-предното твърдение A е симетрична матрица. От предното твърдение всичките характеристични корени $\ell_1, \ell_2, \ldots, \ell_n$ на A са реални $\Rightarrow$ всичките са собствени стойности на оператора φ.

Твърдение: Нека **V** е крайномерно евклидово пространство, $\dim\mathbf{V} = n \geq 1$ и φ е симетричен линеен оператор на **V**.

Ако x, y са собствени вектори, отговарящи на различни собствени стойности ℓ, $\mp$ на φ, то x и y са ортогонални.

Доказателство: Имаме $\varphi(x) = \ell.x$, $\varphi(y) = \mp.y$. φ е симетричен оператор $\Rightarrow (\varphi(x), y) = (x, \varphi(y)) \Rightarrow (\ell.x, y) = (x, \mp.y) \Rightarrow \ell.(x, y) = \mp.(x, y) \Rightarrow (\ell - \mp).(x, y) = 0$ и тъй като $\ell \neq \mp \Rightarrow (x, y) = 0$, т.е. x и y са ортогонални вектори.

Теорема: Нека **V** е крайномерно (ненулево) евклидово пространство и φ е симетричен линеен оператор на **V.** Тогава съществува ортонормиран базис $f_1, f_2, \ldots, f_n$ на **V** в който матрицата на φ е диагонална, т.е. от вида $\begin{pmatrix} \ell_1 & & 0 \\ & \ddots & \\ 0 & & \ell_n \end{pmatrix}$.

Доказателство: Нека $n = \dim\mathbf{V}$, $n \in \mathbb{N}$, $n \geq 1$.
Провеждаме индукция по n.
База: При $n = 1$ в кой да е базис матрицата на φ е $(\ell)_{1x1}$, т.е. тя е диагонална.
Стъпка: Нека твърдението е вярно за n-1 ($n \geq 2$). Нека ℓ_1 е собствена стойност на φ и e_1 е собствен вектор, отговарящ на тази собствена стойност. Нека $f_1 = \frac{1}{|e_1|}.e_1$. Тогава

$\varphi(f_1) = \varphi(\frac{1}{|e_1|}.e_1) = \frac{1}{|e_1|}.\varphi(e_1) = \frac{1}{|e_1|}.\ell.e_1 = \ell.f_1$, т.е. f_1 също е собствен вектор, отговарящ на собствената стойност ℓ_1.
Нека $\mathbf{U} = \ell(f_1)$. Тогава $\mathbf{U} \subseteq \mathbf{V}$ и $\dim\mathbf{U} = 1$. Разглеждаме ортогоналното допълнение $\mathbf{U}^\perp$. Имаме $\mathbf{U}^\perp \subseteq \mathbf{V}$ и $\mathbf{V} = \mathbf{U} \oplus \mathbf{U}^\perp \Rightarrow \dim\mathbf{U} + \dim\mathbf{U}^\perp = \dim\mathbf{V} \Rightarrow \dim\mathbf{U}^\perp = n - 1$. Ще покажем, че $\mathbf{U}^\perp$ е φ-инвариантно подпространство на **V**. Действително, за всяко $x \in \mathbf{U}^\perp$ имаме
(φ е симетричен) $(\varphi(x), f_1) = (x, \varphi(f_1)) = (x, \ell_1.f_1) = \ell_1.(x, f_1) = 0 \Rightarrow \varphi(x) \in \mathbf{U}^\perp$. Разглеждаме φ като линеен оператор на евклидовото пространство $\mathbf{U}^\perp$ (разглеждаме ограничението на φ върху $\mathbf{U}^\perp$). Очевидно φ е симетричен оператор в $\mathbf{U}^\perp$. По индукционното предположение твърдението е вярно за оператора φ: $\mathbf{U}^\perp \to \mathbf{U}^\perp$, т.е. съществува ортонормиран базис $f_2, f_3, \ldots, f_n$ на $\mathbf{U}^\perp$, в който операторът φ има матрица $\begin{pmatrix} \ell_2 & & 0 \\ & \ddots & \\ 0 & & \ell_n \end{pmatrix}$.

Разглеждаме векторите $f_1, f_2, \ldots, f_n$. Имаме $f_2, \ldots, f_n \in \mathbf{U}^\perp \Rightarrow f_2, \ldots, f_n$ са ортогонални на $f_1 \Rightarrow f_1, f_2, \ldots, f_n$ са ортонормирана система от n вектора с дължина 1 $\Rightarrow$ те образуват ортонормиран базис на **V**. За всяко $i = 1, 2, \ldots, n$ имаме $\varphi(f_i) = \ell_i.f_i \Rightarrow$

матрицата на в този базис е точно $\begin{pmatrix} \ell_1 & & 0 \\ & \ddots & \\ 0 & & \ell_n \end{pmatrix}$.

Следствие: За всяка симетрична матрица A $\mathbb{R}^{n \times n}$ съществува ортогонална матрица T $\mathbb{R}^{n \times n}$, такава че $T^{-1}.A.T = \begin{pmatrix} \ell_1 & & 0 \\ & \ddots & \\ 0 & & \ell_n \end{pmatrix}$,

където $\ell_1, \ldots, \ell_n$ са точно характеристичните корени на A.
Доказателство: Нека **V** е крайномерно евклидово пространство с размерност n (например **V** = $\mathbb{R}^n$). Нека E = (e_1, e_2, …, e_n) е ортонормиран базис на **V**. Тогава съществува единствен линеен оператор   **Hom (V)**, такъв че в базиса E матрицата на е A. Тъй като A е симетрична матрица и базисът E е ортонормиран, е симетричен линеен оператор на **V**. От теоремата съществува ортонормиран базис F = (f_1, f_2, …, f_n) на **V**, спрямо който матрицата на е D = $\begin{pmatrix} \ell_1 & & 0 \\ & \ddots & \\ 0 & & \ell_n \end{pmatrix}$. Нека T е матрицата на прехода от базиса E към базиса F. Тогава от по-горе T е ортогонална матрица и от формулата за смяна на базиса имаме, че $D = T^{-1}.A.T$. Тъй като матриците A и D са подобни, те имат едни и същи характеристични корени, т.е. характеристичните корени $\ell_1, \ldots, \ell_n$ на D са точно характеристичните корени на A.

19. Алгебрическа затвореност на полето на комплексните числа. Следствия.

Теорема: Нека **F** е поле, f **F[x]**, deg f 1. Тогава съществуват разширение **K** **F** и   **K**, такива че f () = 0.

Следствие: Нека **F** е поле, f **F[x]**, n = deg f 1, a_0 **F** е старшият коефициент на f. Тогава съществуват разширение **L** **F** и $\alpha_1, \alpha_2, \ldots, \alpha_n$ **L**, такива че f (x) = a_0.(x - α_1).(x - α_2). ….(x - α_n).
В означенията на следствието, нека **M** е сечението на всички подполета на полето **L**, които съдържат **F** и $\alpha_1, \alpha_2, \ldots, \alpha_n$ **L**. Ясно е, че **M** е подполе на **L** и **M** е най-малкото такова подполе, което съдържа **F** и $\alpha_1, \alpha_2, \ldots, \alpha_n$. **M** се нарича **поле на разлагане** на f над полето **F**.

Може да се покаже, че всеки две полета на разлагане на полином над едно и също поле са изоморфни, т.е. неформално, полето на разлагане е независимо от разширението, което съдържа корените на полинома.

Нека $f \in \mathbf{F[x]}$, $f(x) = a_0.x^n + a_1.x^{n-1} + \ldots + a_{n-1}.x + a_n$, $n \in \mathbb{N}$, $a_0 \in \mathbf{F^*}$.
В подходящо разширение $\mathbf{L} \supset \mathbf{F}$ имаме:
$f(x) = a_0.(x - \alpha_1).(x - \alpha_2). \ldots .(x - \alpha_n)$.
Като приравним коефициентите пред x^{n-1}, x^{n-2}, …, x^1, x^0 в равенството получаваме **формулите на Виет**:

$$\alpha_1 + \ldots + \alpha_n = -\frac{a_1}{a_0},$$

$$\alpha_1.\alpha_2 + \ldots + \alpha_{n-1}.\alpha_n = \frac{a_2}{a_0},$$

…

$$\alpha_1.\alpha_2. \ldots .\alpha_k + \ldots + \alpha_{n-k+1}.\alpha_{n-k+2}. \ldots .\alpha_n = (-1)^k.\frac{a_k}{a_0},$$

…

$$\alpha_1.\alpha_2. \ldots .\alpha_n = (-1)^n.\frac{a_n}{a_0}.$$

Тук лявата страна на k-тата формула ($1 \le k \le n$) съдържа сумата от всевъзможните произведения на k елемента от $\alpha_1, \alpha_2, \ldots, \alpha_n$, т.е. съдържа $\binom{n}{k}$ събираеми.

Нека **A** е комутативен пръстен с единица.
Нека $f = f(x_1, x_2, \ldots, x_n) \in \mathbf{A[x_1, \ldots, x_n]}$. Казваме, че f е **симетричен полином**, ако $f(x_{i_1}, x_{i_2}, \ldots, x_{i_n}) = f(x_1, x_2, \ldots, x_n)$ (равенство на полиноми) за произволна пермутация $i_1, i_2, \ldots, i_n$ на числата 1, 2, …, n.
Следните симетрични полиноми

$\sigma_1 = x_1 + x_2 + \ldots + x_n$

$\sigma_2 = x_1.x_2 + x_1.x_3 + \ldots + x_{n-1}.x_n$

…

$\sigma_n = x_1.x_2. \ldots .x_n$

се наричат **елементарни симетрични полиноми**.

Теорема: Нека **A** е област, $f = f(x_1, x_2, \ldots, x_n) \in \mathbf{A[x_1, x_2, \ldots, x_n]}$ е симетричен полином. Тогава съществува единствен полином g на n променливи с коефициенти от **A**, такъв че
$g(\sigma_1, \sigma_2, \ldots, \sigma_n) = f(x_1, x_2, \ldots, x_n)$.
Следствие: Нека **F** е поле, $f(x) \in \mathbf{F[x]}$, $n = \deg f \ge 1$ и нека $\alpha_1, \alpha_2, \ldots, \alpha_n$ са корените на $f(x)$, $\alpha_i \in \mathbf{K} \supset \mathbf{F}$. Ако $h(x_1, x_2, \ldots, x_n)$ е симетричен полином с коефициенти от полето **F**, то $h(\alpha_1, \alpha_2, \ldots, \alpha_n) \in \mathbf{F}$.
Доказателство: Нека $f(x) = a_0.x^n + a_1.x^{n-1} + \ldots + a_n$, $a_i \in \mathbf{F}$, $a_0 \in \mathbf{F^*}$.
От теоремата съществува полином g на n променливи с коефициенти от полето **F**, такъв че
$h(x_1, x_2, \ldots, x_n) = g(\sigma_1, \sigma_2, \ldots, \sigma_n)$. За $k = 1, 2, \ldots, n$ имаме:

$\sigma_k(\alpha_1, \alpha_2, \ldots, \alpha_n) = (-1)^k.\frac{a_k}{a_0} \in \mathbf{F}$ (от формулите на Виет). Тогава

$h(\alpha_1, \alpha_2, \ldots, \alpha_n) = g(\frac{a_1}{a_0}, \frac{a_2}{a_0}, \ldots, (-1)^n.\frac{a_n}{a_0}) \in \mathbf{F}$, тъй като g е с коефициенти от полето **F**.

Казваме, че едно поле **F** е **алгебрически затворено**, ако всеки полином $f \in \mathbf{F[x]}$, $\deg f \geq 1$ има поне един корен в **F**.
Еквивалентна дефиниция е: За всеки полином $f \in \mathbf{F[x]}$, $\deg f(x) = n \geq 1$, съществуват $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbf{F}$, такива че $f(x) = a_0.(x - \alpha_1).(x - \alpha_2). \ldots.(x - \alpha_n)$, където $a_0 \in \mathbf{F^*}$ е старшият коефициент на f.
Действително, ако $\alpha_1 \in \mathbf{F}$ е корен на f (x), то $f(x) = (x - \alpha_1).g(x)$, $g(x) \in \mathbf{F[x]}$. По същия начин, ако $\alpha_2 \in \mathbf{F}$ е корен на g (x), то $g(x) = (x - \alpha_2).h(x)$, $h(x) \in \mathbf{F[x]}$ и $f(x) = (x - \alpha_1).(x - \alpha_2).h(x)$ …и т.н. след точно n стъпки f ще се разложи в линейни множители.

Теорема: Полето $\mathbb{C}$ е алгебрически затворено.
Доказателство:
Ще отбележим, че няма чисто алгебрично доказателство на тази теорема, т.е. което не използва поне малка част от анализа – например всеки полином $f(x) \in \mathbb{R}\mathbf{[x]}$ е непрекъсната функция от $\mathbb{R}$ в $\mathbb{R}$.
Стъпка 1:
Ако $f(x) \in \mathbb{R}\mathbf{[x]}$, $\deg f$ е нечетно число, f (x) има поне един реален корен. Действително, нека $n = \deg f \in \mathbb{N}$, $n \geq 1$, $a_0 \neq 0$ е старшият коефициент на f. Използваме, че n е нечетно число:
Ако $a_0 > 0$, то $\lim\limits_{x \to +\infty} f(x) = +\infty$, $\lim\limits_{x \to -\infty} f(x) = -\infty$.
Ако $a_0 < 0$, то $\lim\limits_{x \to +\infty} f(x) = -\infty$, $\lim\limits_{x \to -\infty} f(x) = +\infty$.
И в двата случая съществуват $a, b \in \mathbb{R}$, $a < b$, такива че $f(a).f(b) < 0$. Тогава от теоремата на Болцано-Коши, тъй като f (x) е непрекъсната функция, съществува $c \in [a, b]$, такова че $f(c) = 0$.
Стъпка 2:
Ако $f(x) \in \mathbb{C}\mathbf{[x]}$ и $\deg f \geq 1$, то f (x) има поне един корен в $\mathbb{C}$.
Нека $n = \deg f$, $n \geq 1$, $n = 2^k.m$, $k \geq 0$, $m \geq 1$, m е нечетно.
Провеждаме индукция по k.
База: k = 0. Тогава n = m е нечетно и от стъпка 1 f има дори реален корен.
Стъпка: Нека твърдението е изпълнено за k – 1, $k \geq 1$.
Нека $\alpha_1, \alpha_2, \ldots, \alpha_n$ са корените на f (x), $\alpha_{n+1} \in \mathbf{K}$ – поле на разлагане на f (x) над $\mathbb{C}$. Нека $r \in \mathbb{R}$. Означаваме $\beta_{ij} = \alpha_i\alpha_j + r.(\alpha_i + \alpha_j)$ за всеки i, j, $1 \leq i < j \leq n$. Естествено, $\beta_{ij} \in \mathbf{K}$.
Броят на β_{ij} е $n_1 = \frac{n.(n-1)}{2} = \frac{2^k.m.(2^k.m-1)}{2} = 2^{k-1}.m.(2^k.m-1) = 2^{k-1}.m'$, където $m' \in \mathbb{N}$ и m' е нечетно. Разглеждаме полиномът

$g(x) = \prod_{1 \le i < j \le n} x - \updownarrow_{ij}$, $g(x) \in \mathbf{K[x]}$. Нека $g(x) = x^{n_1} + b_1.x^{n_1-1} + \ldots + b_{n_1}$,

засега е ясно, че $b_t \in \mathbf{K}$. За всяко $t = 1, 2, \ldots, n_1$ имаме:
$b_t = (-1)^t.\leftarrow_t (\ldots, \updownarrow_{ij}, \ldots)$. При произволна пермутация $\mapsto_{p_1}, \mapsto_{p_2}, \ldots, \mapsto_{p_n}$ на $\mapsto_1, \mapsto_2, \ldots, \mapsto_n$ имаме, че $\updownarrow_{ij} = \mapsto_i.\mapsto_j + r.(\mapsto_i + \mapsto_j)$ преминава в $\mapsto_{p_i}.\mapsto_{p_j} + r.(\mapsto_{p_i} + \mapsto_{p_j}) = \updownarrow_{p_i p_j}$. С други думи, произволна пермутация на $\mapsto_1, \mapsto_2, \ldots, \mapsto_n$ води до пермутация на $\updownarrow_{ij}$. Но $\leftarrow_t (\ldots, \updownarrow_{ij}, \ldots)$ е симетричен полином на $\updownarrow_{ij}$, така че той не се променя при пермутация на $\updownarrow_{ij}$. И така $b_t = (-1)^t.\leftarrow_t (\ldots, \updownarrow_{ij}, \ldots)$ е симетричен полином с реални коефициенти на $\mapsto_1, \mapsto_2, \ldots, \mapsto_n$ и от следствието по-горе $b_t \in \mathbb{R}$ за всяко $t = 1, 2, \ldots, n_1$. Така имаме:
$g(x) \in \mathbb{R}\mathbf{[x]}$, $\deg g = 2^{k-1}.m$, където m е нечетно. По индукционното предположение полиномът $g(x)$ има комплексен корен.
Тъй като $\mathbb{C} \supset \mathbf{K}$, това означава, че за всяко $r \in \mathbb{R}$ съществуват индекси i, j, $1 \le i < j \le n$, такива че $\updownarrow_{ij} = \mapsto_i.\mapsto_j + r.(\mapsto_i + \mapsto_j) \in \mathbb{C}$.
Но двойките i, j, $1 \le i < j \le n$ са краен брой, реалните числа са безброй много $\Rightarrow$ съществуват реални числа r_1, r_2, $r_1 \ne r_2$, такива че
$\mapsto_i.\mapsto_j + r_1.(\mapsto_i + \mapsto_j) = c \in \mathbb{C}$,
$\mapsto_i.\mapsto_j + r_2.(\mapsto_i + \mapsto_j) = d \in \mathbb{C}$ за една и съща двойка $\mapsto_i, \mapsto_j$, $1 \le i < j \le n$.
В такъв случай, $\mapsto_i + \mapsto_j = \dfrac{c - d}{r_1 - r_2} = a \in \mathbb{C}$, $\mapsto_i.\mapsto_j = \dfrac{d.r_1 - c.r_2}{r_1 - r_2} = b \in \mathbb{C}$.
Така $\mapsto_i, \mapsto_j$ са корените на полинома $x^2 - a.x + b = 0$, т.е.

$\mapsto_i, \mapsto_j = \dfrac{a \pm \sqrt{a^2 - 4.b}}{2} \in \mathbb{C}$, тъй като $\mathbb{C}$ е затворено относно коренуване (което се вижда лесно с формулата на Моавър).
И така $\mapsto_i, \mapsto_j \in \mathbb{C}$, т.е. f наистина има комплексен корен.
<u>Стъпка 3:</u>
Ако $f(x) \in \mathbb{C}\mathbf{[x]}$, $\deg f \ge 1$, то $f(x)$ има поне един корен в $\mathbb{C}$.
Нека $f(x) = a_0.x^n + a_1.x^{n-1} + \ldots + a_n$, $a_i \in \mathbb{C}$, $a_0 \in \mathbb{C}^*$, $n \ge 1$.

Нека $\overline{f}(x) = \overline{a_0}.x^n + \overline{a_1}.x^{n-1} + \ldots + \overline{a_{n-1}}.x + \overline{a_n} \in \mathbb{C}\mathbf{[x]}$.

За всяко $\mapsto \in \mathbb{C}$ имаме: $\overline{f}(\overline{\mapsto}) = \overline{f(\mapsto)}$. Действително,

$\overline{f}(\overline{\mapsto}) = \overline{a_0}.\overline{\mapsto}^n + \overline{a_1}.\overline{\mapsto}^{n-1} + \ldots + \overline{a_{n-1}}.\overline{\mapsto} + \overline{a_n} =$

$= \overline{a_0.\mapsto^n + a_1.\mapsto^{n-1} + \ldots + a_{n-1}.\mapsto + a_n} = \overline{f(\mapsto)}$.

Разглеждаме $h(x) = f(x).\overline{f}(x) \in \mathbb{C}\mathbf{[x]}$,
$h(x) = c_0.x^{2.n} + c_1.x^{2.n-1} + \ldots + c_{2.n-1}.x + c_{2.n}$, $c_t \in \mathbb{C}$.

За всяко $t = 0, 1, 2, \ldots, 2.n$, $c_t = \sum_{i+j=t} a_i.\overline{a_j}$. Имаме

$\overline{c_t} = \overline{\sum_{i+j=t} a_i.\overline{a_j}} = \sum_{i+j=t} \overline{a_i}.\overline{\overline{a_j}} = \sum_{i+j=t} \overline{a_i}.a_j = \sum_{i+j=t} a_i.\overline{a_j} = c_t$ (в предпоследното равенство сменихме ролите на индексите, което е позволено, тъй

като те участват симетрично). Получихме, че $\overline{c_t} = c_t \Rightarrow c_t \in \mathbb{R}$ и така показахме, че $h(x) \in \mathbb{R}[x]$. Тогава от стъпка 2 съществува $\alpha \in \mathbb{C}$, такова че $h(\alpha) = 0$, т.е. $f(\alpha).\overline{f}(\alpha) = 0$. Тъй като $\mathbb{C}$ е поле, то $f(\alpha) = 0 \Rightarrow \alpha$ е корен на f или $\overline{f}(\alpha) = 0 \Rightarrow \overline{f}(\overline{\alpha}) = 0 \Rightarrow \overline{f(\alpha)} = 0 \Rightarrow f(\overline{\alpha}) = 0 \Rightarrow \overline{\alpha}$ е корен на f(x).

И така за всеки полином $f(x) \in \mathbb{C}[x]$, $n = \deg f \geq 1$, съществуват $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{C}$, такива че $f(x) = a_0.(x - \alpha_1).(x - \alpha_2). \ldots .(x - \alpha_n)$, където $a_0 \in \mathbb{C}^*$ е старшият коефициент на f.
Така неразложими над $\mathbb{C}$ полиноми с коефициенти от $\mathbb{C}$ са само полиномите от степен 1.

Твърдение: Нека $f(x) \in \mathbb{C}[x]$. Ако $\alpha \in \mathbb{C}$ е корен на f(x), то $\overline{\alpha} \in \mathbb{C}$ също е корен на f(x), при това със същата кратност.
Доказателство: Ако $f(x) \in \mathbb{C}[x]$, то

$\overline{f}(x) = \overline{a_0}.x^n + \overline{a_1}.x^{n-1} + \ldots + \overline{a_{n-1}}.x + \overline{a_n} \in \mathbb{C}[x]$ и по-горе проверихме, че за всяко $\alpha \in \mathbb{C}$ имаме: $\overline{f}(\overline{\alpha}) = \overline{f(\alpha)}$.

В частност, ако $f(x) \in \mathbb{R}[x]$, то $\overline{f}(x) = f(x)$ и от $f(\alpha) = 0 \Rightarrow$

$\Rightarrow f(\overline{\alpha}) = \overline{f}(\overline{\alpha}) = \overline{f(\alpha)} = \overline{0} = 0 \Rightarrow \overline{\alpha}$ също е корен на f(x).

Ако $\alpha \in \mathbb{R}$ и α е корен на f(x), то $\overline{\alpha} = \alpha$ и очевидно $\overline{\alpha}$ и α имат една и съща кратност. Нека $\alpha \in \mathbb{C} \setminus \mathbb{R}$, т.е. $\overline{\alpha} \neq \alpha$ Имаме $(x - \alpha, x - \overline{\alpha}) = 1$, $x - \alpha \mid f(x)$, $x - \overline{\alpha} \mid f(x) \Rightarrow (x - \alpha).(x - \overline{\alpha}) \mid f(x)$.

Нека $\varphi(x) = (x - \alpha).(x - \overline{\alpha})$, т.е. $\varphi(x) = x^2 - (\alpha + \overline{\alpha}).x + \alpha.\overline{\alpha} \in \mathbb{R}[x]$.
Нека k е най-голямото естествено число, такова че $(\varphi(x))^k \mid f(x)$, $k \geq 1$, $f(x) = (\varphi(x))^k.g(x)$, $g(x) \in \mathbb{R}[x]$.

Ако $g(\alpha) = 0 \Rightarrow g(\overline{\alpha}) = 0 \Rightarrow \varphi(x) = (x - \alpha).(x + \overline{\alpha}) \mid g(x) \Rightarrow$
$(\varphi(x))^{k+1} \mid f(x)$, което е противоречие с избора на k $\Rightarrow g(\alpha) \neq 0$.

Аналогично $g(\overline{\alpha}) \neq 0$. Тогава $x - \alpha \nmid g(x)$, $x - \overline{\alpha} \nmid g(x) \Rightarrow \alpha$ и $\overline{\alpha}$ са корени на f(x) с една и съща кратност k.
От горното твърдение получаваме, че всеки полином $f(x) \in \mathbb{R}[x]$ се записва еднозначно във вида:
$f(x) = a_0.(x - \alpha_1)^{k_1}. \ldots .(x - \alpha_t)^{k_t}.(x^2 + p_1.x + q_1)^{l_1}. \ldots .(x^2 + p_s.x + q_s)^{l_s}$,
където $a_0 \in \mathbb{R}^*$ е старшият коефициент на f(x), $\alpha_i \in \mathbb{R}$, $p_j, q_j \in \mathbb{R}$, $p_j^2 - 4.q_j < 0$, $t, s \in \mathbb{N}$, $k_i, l_j \in \mathbb{N}$, $k_i \geq 1$, $l_j \geq 1$.
И така неразложими полиноми над $\mathbb{R}$ полиноми с коефициенти от $\mathbb{R}$ са полиномите от степен 1 и полиномите от степен 2 с отрицателна дискриминанта.

20. Теореми за средните стойности (Рол, Лагранж, Коши). Формула на Тейлър.

Навсякъде разглеждаме реалнозначни функции на реален аргумент.

Нека f (x) е дефинирана в околност M на точката $x_0 \in$.
Казваме, че f (x) има **локален максимум** в точката x_0, ако съществува околност $U \subset M$ на x_0, такава че
$f(x_0) \geq f(x)$ за всяко $x \in U$.
Нека f (x) е дефинирана в околност M на точката $x_0 \in$.
Казваме, че f (x) има **локален минимум** в точката x_0, ако съществува околност $U \subset M$ на x_0, такава че
$f(x_0) \leq f(x)$ за всяко $x \in U$.

Локалните минимуми и максимуми се наричат събирателно **локални екстремуми**.

Теорема (Ферма): Нека f (x) е дефинирана в околност на точката x_0 и f (x) е диференцируема в точката x_0. Ако f (x) има локален екстремум в точката x_0, то $f'(x_0) = 0$.
Доказателство: Нека за определеност $f(x_0)$ е локален максимум за f (x) (разсъждението за локален минимум е абсолютно аналогично). Съществува околност U на x_0, в която f (x) е дефинирана и $f(x) \leq f(x_0)$ за всяко $x \in U$.
Разглеждаме диференчното частно $\frac{f(x_0+h)-f(x_0)}{h}$.
Нека $h \to 0$ с положителни стойности, така че $x_0 + h$ остава в U.
Тогава $f(x_0 + h) - f(x_0) \leq 0$, $h > 0 \Rightarrow \frac{f(x_0+h)-f(x_0)}{h} \leq 0 \Rightarrow$ след граничен преход $f'(x_0) \leq 0$.
Нека $h \to 0$ с отрицателни стойности, така че $x_0 + h$ остава в U.
Тогава $f(x_0 + h) - f(x_0) \leq 0$, $h < 0 \Rightarrow \frac{f(x_0+h)-f(x_0)}{h} \geq 0 \Rightarrow$ след граничен преход $f'(x_0) \geq 0$.
От двете неравенства получаваме $f'(x_0) = 0$.

Теорема (Вайерщрас): Нека f (x) е дефинирана и непрекъсната в крайния затворен интервал [a, b]. Тогава тя притежава най-голяма и най-малка стойност сред стойностите си за $x \in [a, b]$.

Теорема (Рол): Нека f (x) е дефинирана и непрекъсната в крайния затворен интервал [a, b]. Освен това, нека f (x) е диференцируема в отворения интервал (a, b) и f (a) = f (b). В такъв случай, съществува поне една точка $\in$ (a, b), за която $f'(\ ) = 0$.

Доказателство: Функцията f (x) е непрекъсната в [a, b] □ от теоремата на Вайерщрас f (x) притежава най-големи и най-малки стойности в [a, b]. Нека m = f (x_1) е най-малката стойност на f (x), x_1 □ [a, b] и M = f (x_2) е най-голямата стойност на f (x), x_2 □ [a, b].
Ако m = M, от m □f (x) □M за x □ [a, b] □ f (x) е константа в [a, b] □ f □(x) = 0 за всяко x □ (a, b) и теоремата е доказана.
Нека m < M. Поне една от двете точки x_1 и x_2 е вътрешна за интервала [a, b] – в противен случай ще получим, че m = M = f (a) = f (b). Ако x_1 □ (a, b), тогава x_1 е локален екстремум □ f □(x_1) = 0 по теоремата на Ферма. Ако x_1 = a или x_1 = b, тогава x_2 □ (a, b) е локален екстремум и f □(x_2) = 0 по теоремата на Ферма.

Теорема (Лагранж): Нека f (x) е дефинирана и непрекъсната в крайния затворен интервал [a, b] и диференцируема в отворения интервал (a, b). Тогава съществува □ □ (a, b), такова че f (b) – f (a) = f □(□).(b – a).
Доказателство:
Да отбележим, че при f (a) = f (b) получаваме теоремата на Рол, която е частен случай на теоремата на Лагранж.
Разглеждаме функцията g (x) = f (x) – k.x, където k = $\frac{f(b) - f(a)}{b - a}$. В такъв случай, g (a) = g (b). Действително, g (a) = g (b) □ f (a) – k.a = f (b) – k.b □ f (b) – f (a) = k.(b – a), което е изпълнено. Освен това, g (x) е дефинирана и непрекъсната в затворения интервал [a, b] и g (x) е диференцируема в отворения интервал (a, b). Налице са условията в теоремата на Рол □ съществува □ □ (a, b), такова че g□ (□) = 0. Но g□ (x) = f □(x) – k, така че 0 = g□ (□) = f □(□) – k = f □(□) – $\frac{f(b) - f(a)}{b - a}$ □ f (b) – f (a) = f □(□).(b – a).

Следствие: Нека функцията f (x) е дефинирана и диференцируема в интервал ƛ. Ако f □(x) = 0 за всяко x □ ƛ □ f (x) е константа в ƛ.
Доказателство: Фиксираме две точки x_1 < x_2 □ ƛ. В затворения интервал [x_1, x_2] очевидно са изпълнени условията на теоремата на Лагранж □ съществува точка □ □ (x_1, x_2), такава че f (x_2) – f (x_1) = f □(□).(x_2 – x_1), но f □(□) = 0 за всяко □ □ ƛ □ f (x_1) = f (x_2). Точките x_1 и x_2 са избрани произволно □ f (x) е константа в ƛ.

Следствие: Нека f (x) е дефинирана и диференцируема в интервал ƛ. Ако f □(x) □ 0 (f □(x) □0) за всяко x □ ƛ, то f (x) е монотонно растяща (намаляваща) в ƛ. Ако f □(x) > 0 (f □(x) < 0) за всяко x □ ƛ, то f (x) е строго монотонно растяща (намаляваща) в ƛ.

Доказателство: Фиксираме две точки $x_1 < x_2 \in \lambda$. В затворения интервал $[x_1, x_2]$ очевидно са изпълнени условията на теоремата на Лагранж $\Rightarrow$ съществува точка $\xi \in (x_1, x_2)$, такава че $f(x_2) - f(x_1) = f'(\xi).(x_2 - x_1)$, но $f'(\xi) \geq 0$ $(f'(\xi) \leq 0)$ за всяко $\xi \in \lambda \Rightarrow$ $f(x_1) \leq f(x_2)$ $(f(x_1) \geq f(x_2))$. Точките $x_1 < x_2$ са избрани произволно $\Rightarrow$ $f(x)$ е монотонно растяща (намаляваща) в λ.
Ако неравенството е строго, т.е. $f'(\xi) > 0$ $(f'(\xi) < 0) \Rightarrow f(x_1) < f(x_2)$ $(f(x_1) > f(x_2))$ и функцията $f(x)$ е строго монотонно растяща (намаляваща).

Следствие: Нека функцията $f(x)$ е дефинирана и диференцируема в интервал λ. Ако $f'(x)$ е ограничена в λ, то $f(x)$ е равномерно непрекъсната в λ.
Доказателство: Тъй като $f'(x)$ е ограничена в $\lambda \Rightarrow$ съществува $C \in \mathbb{R}^+$, такова че $|f'(x)| < C$ за всяко $x \in \lambda$.
Фиксираме две точки $x_1 < x_2 \in \lambda$. В затворения интервал $[x_1, x_2]$ очевидно са изпълнени условията на теоремата на Лагранж $\Rightarrow$ съществува $\xi \in (x_1, x_2)$, такова че $f(x_2) - f(x_1) = f'(\xi).(x_2 - x_1)$, но $|f'(\xi)| < C \Rightarrow |f(x_2) - f(x_1)| < C.|x_2 - x_1|$.
Фиксираме $\varepsilon > 0$. Избираме $\delta = \varepsilon/C$. Тогава за всеки $x, y \in \lambda$, такива че $|x - y| < \delta$ имаме $|f(x) - f(y)| < C.|x - y| < C.\varepsilon/C = \varepsilon \Rightarrow$ $f(x)$ е равномерно непрекъсната в λ.

Теорема (Коши): Нека функциите $f(x)$ и $g(x)$ са дефинирани и непрекъснати в затворения интервал $[a, b]$ и диференцируеми в отворения интервал (a, b). Нека освен това $g'(x) \neq 0$

за всяко $x \in (a, b)$. В такъв случай, съществува

$\xi \in (a, b)$, такова че $\dfrac{f(b)-f(a)}{g(b)-g(a)} = \dfrac{f'(\xi)}{g'(\xi)}$.

Доказателство: Теоремата на Лагранж е частен случай на теоремата на Коши, при нея $g(x) = x$. Преди всичко, условието е коректно, т.е. $g(a) \neq g(b)$. Действително, ако допуснем $g(a) = g(b)$, тогава за функцията $g(x)$ ще са изпълнени условията на теоремата на Рол $\Rightarrow$ съществува $\xi \in (a, b)$, такова че $g'(\xi) = 0$, което е противоречие с условието.

Разглеждаме функцията $\varphi(x) = f(x) - k.g(x)$, където
$k = \dfrac{f(b)-f(a)}{g(b)-g(a)}$. В такъв случай, $\varphi(a) = \varphi(b)$. Действително,
$\varphi(a) = \varphi(b) \Leftrightarrow f(a) - k.g(a) = f(b) - k.g(b) \Leftrightarrow$
$f(b) - f(a) = k.(g(b) - g(a))$, което е изпълнено.
Освен това, $\varphi(x)$ е дефинирана и непрекъсната в затворения интервал $[a, b]$ и е диференцируема в отворения интервал (a, b). Налице са условията в теоремата на Рол $\Rightarrow$ съществува $\xi \in (a, b)$, такова че $\varphi'(\xi) = 0$. Но $\varphi'(x) = f'(x) - k.g'(x) \Rightarrow 0 = \varphi'(\xi) =$

$= f'(\xi) - k.g'(\xi) = f'(\xi) - \frac{f(b)-f(a)}{g(b)-g(a)}.g'(\xi) \Rightarrow \frac{f(b)-f(a)}{g(b)-g(a)} = \frac{f'(\xi)}{g'(\xi)}$.

Теорема (формула на Тейлър): Нека f (x) е дефинирана и n+1 пъти диференцируема в околност U на точката x_0, $n \in \mathbb{N}$.

Тогава за всяка точка x от U и всяко $p \in \mathbb{N}$, $p \neq 0$ имаме, че съществува t_0 между x и x_0, такова че

$$f(x) = f(x_0) + \frac{x-x_0}{1!}.f'(x_0) + \frac{(x-x_0)^2}{2!}.f''(x_0) + \ldots + \frac{(x-x_0)^n}{n!}.f^{(n)}(x_0) + R_n,$$

където $R_n = \frac{(x-x_0)^p.(x-t_0)^{n+1-p}.f^{(n+1)}(t_0)}{p.n!}$.

Доказателство: Фиксираме $x \neq U$, $p \in \mathbb{N}$, $p \neq 0$. Ако $x = x_0$ въпросното равенство очевидно е изпълнено при $t_0 = x = x_0$.

Нека за определеност $x_0 < x$. Разглеждаме функцията: $\varphi(t) =$

$$= f(x) - (f(t) + \frac{x-t}{1!}.f'(t) + \frac{(x-t)^2}{2!}.f''(t) + \ldots + \frac{(x-t)^n}{n!}.f^{(n)}(t)) - \ell.(x-t)^p,$$

дефинирана за $t \in U$ (ℓ е параметър, който ще уточним).

Ясно е, че $\varphi(x) = f(x) - f(x) = 0$. Избираме ℓ така, че $\varphi(x_0) = 0$, т.е.

$$f(x) - (f(x_0) + \frac{x-x_0}{1!}.f'(x_0) + \frac{(x-x_0)^2}{2!}.f''(x_0) + \ldots + \frac{(x-x_0)^n}{n!}.f^{(n)}(x_0)) -$$

$$- \ell.(x-x_0)^p = 0 \Rightarrow$$

$$\ell = \frac{f(x) - \left[f(x_0) + \frac{x-x_0}{1!}.f'(x_0) + \frac{(x-x_0)^2}{2!}.f''(x_0) + \ldots + \frac{(x-x_0)^n}{n!}.f^{(n)}(x_0)\right]}{(x-x_0)^p}.$$

Тъй като f е n+1 пъти диференцируема в U, то φ е диференцируема в U. Непосредствено пресмятаме,

$$\varphi'(t) = -(f'(t) + (-1).f'(t) + \frac{x-t}{1!}.f''(t) + (-1).\frac{x-t}{1!}.f''(t) + \frac{(x-t)^2}{2!}.f'''(t) + \ldots + (-1).\frac{(x-t)^{n-1}}{(n-1)!}.f^{(n)}(t) + \frac{(x-t)^n}{n!}.f^{(n+1)}(t)) - (-1).\ell.p.(x-t)^{p-1} =$$

$$= \ell.p.(x-t)^{p-1} - \frac{(x-t)^n}{n!}.f^{(n+1)}(t).$$ Функцията $\varphi(t)$ е диференцируема, а следователно и непрекъсната в интервала $[x_0, x]$. Сега за нея можем да приложим теоремата на Рол в интервала $[x_0, x]$ $\Rightarrow$ съществува $t_0 \in (x_0, x)$, такова че $\varphi'(t_0) = 0$, т.е.

$$\ell.p.(x-t_0)^{p-1} - \frac{(x-t_0)^n}{n!}.f^{(n+1)}(t_0) = 0 \Rightarrow \ell.(x-x_0)^p = R_n$$ и

от горния вид на ℓ получаваме окончателно

$$f(x) = f(x_0) + \frac{x - x_0}{1!}.f'(x_0) + \frac{(x - x_0)^2}{2!}.f''(x_0) + \ldots + \frac{(x - x_0)^n}{n!}.f^{(n)}(x_0) + R_n.$$

R_n се нарича **остатъчен член** във формулата на Тейлър.

При p = n+1 получаваме **форма на Лагранж** за остатъчния член и тя е : $R_n = \frac{(x - x_0)^{n+1}.f^{(n+1)}(t_0)}{(n+1)!}$.

При p = 1 получаваме **форма на Коши** за остатъчния член и тя е: $R_n = \frac{(x - x_0).(x - t_0)^n.f^{(n+1)}(t_0)}{n!}$.

21. Определен интеграл. Дефиниция и свойства. Интегруемост на непрекъснатите функции. Теорема на Нютон-Лайбниц.

Фиксиране е функцията $f : [a, b] \to \mathbb{R}$.

Под **разбиване** на интервала [a, b] разбираме редица от точки $x_0 = a, x_1, x_2, \ldots, x_n = b$, такива че $x_0 < x_1 < x_2 \ldots < x_n$. Под **диаметър** на разбиването разбираме числото $d(\{x_i\}) = \max_{1 \le i \le n}(x_i - x_{i-1})$, т.е. дължината на най-големия подинтервал, определен от разбиването.

Сумата $\sigma(\{x_i\}, \{t_i\}) = \sum_{i=1}^{n} f(t_i).(x_i - x_{i-1})$, където $\{x_i\}$ е някакво разбиване на интервала [a, b], а t_i са **междинни точки**, $t_i \in [x_{i-1}, x_i]$ за i = 1, 2, …, n, се нарича **Риманова интегрална сума** на функцията f, определена от разбиването $\{x_i\}$ и междинните точки $\{t_i\}$.

Геометрично е ясно е, че Римановата интегрална сума представлява сума от лица на правоъгълници, която апроксимира лицето на криволинейния трапец, определен от графиката на f.

Казваме, че функцията f е **интегруема в Риманов смисъл** в интервала [a, b], ако съществува число I, такова че Римановите интегрални суми клонят равномерно към I, при условие, че диаметърът на използваните разбивания клони към 0, т.е. за всяко $\varepsilon > 0$, съществува $\delta > 0$, такова че за всяко разбиване $\{x_i\}$ с $d(\{x_i\}) < \delta$ и всеки избор на междинните точки t_i, $|\sigma(\{x_i\}, \{t_i\}) - I| < \varepsilon$.

Ако f е интегруема в Риманов смисъл, лесно се вижда (както единствеността на границата на една сходяща редица), че числото

I е еднозначно определено и то се нарича **Риманов** (**определен**) **интеграл** на f от a до b и се означава $I = \int_a^b f(x)dx$.

Твърдение: Ако f е интегруема в Риманов смисъл в интервала [a, b], то f е ограничена в [a, b].

Доказателство: Нека ε е произволно положително число, $I = \int_a^b f(x)dx$. Образуваме разбиване $\{x_i\}$ на [a, b], такова че да имаме $|\sigma(\{x_i\}, \{t_i\}) - I| < \varepsilon$ при всеки избор на междинните точки t_i. За всеки две числа x, y е в сила неравенството $|x - y| \geq |x| - |y|$, което лесно следва от неравенството на триъгълника ($|x + y| \leq |x| + |y|$ за всеки две числа x, y).

В такъв случай, $\varepsilon > |\sigma(\{x_i\}, \{t_i\}) - I| = |\sum_{i=1}^{n} f(t_i).(x_i - x_{i-1}) - I| =$

$$= |f(t_k).(x_k - x_{k-1}) + \sum_{i=1}^{k-1} f(t_i).(x_i - x_{i-1}) + \sum_{i=k+1}^{n} f(t_i).(x_i - x_{i-1}) - I| =$$

$$= |f(t_k).(x_k - x_{k-1}) - (I - \sum_{i=1}^{k-1} f(t_i).(x_i - x_{i-1}) - \sum_{i=k+1}^{n} f(t_i).(x_i - x_{i-1}))| \geq$$

$$\geq |f(t_k).(x_k - x_{k-1})| - |I - \sum_{i=1}^{k-1} f(t_i).(x_i - x_{i-1}) - \sum_{i=k+1}^{n} f(t_i).(x_i - x_{i-1})| \geq$$

$$|f(t_k)| \geq \frac{-\varepsilon + \left|I - \sum_{i=1}^{k-1} f(t_i).(x_i - x_{i-1}) - \sum_{i=1}^{k-1} f(t_i).(x_i - x_{i-1})\right|}{x_k - x_{k-1}}.$$

Сега фиксираме t_i при $i \neq k$ и оставяме t_k да се мени в $[x_{k-1}, x_k]$.

Полученото неравенство ни учи, че функцията f е ограничена в подинтервала $[x_{k-1}, x_k]$, определен от разбиването $\{x_i\}$. Тъй като можем да даваме на k стойности 1, 2, …, n заключаваме, че f е ограничена в целия интервал [a, b].

Предполагаме, че е фиксирана функция $f : [a, b] \to \mathbb{R}$, която е ограничена в целия интервал [a, b].

Нека имаме разбиване $\{x_i\}$ на интервала [a, b].

Полагаме $m_i = \inf_{x \in [x_{i-1}, x_i]} f(x)$, i = 1, 2, …, n.

Полагаме $M_i = \sup_{x \in [x_{i-1}, x_i]} f(x)$, i = 1, 2, …, n.

Ще отбележим, че числата m_i и M_i са добре дефинирани, тъй като f е ограничена в [a, b], а следователно и във всеки подинтервал на [a, b], определен от разбиването $\{x_i\}$.

Сумата $s(\{x_i\}) = \sum_{i=1}^{n} m_i.(x_i - x_{i-1})$ наричаме **малка сума на**

Дарбу на функцията f, определена от разбиването $\{x_i\}$.

Сумата $s = \sum_{i=1}^{n} M_i.(x_i - x_{i-1})$ наричаме **голяма сума на**

Дарбу на функцията f, определена от разбиването $\{x_i\}$.

Очевидна е следната верига от неравенства:

$s(\{x_i\}) \leq s(\{x_i\}, t_i) \leq S(\{x_i\})$, която следва от $m_i \leq f(t_i) \leq M_i$ за всяко

$t_i \in [x_{i-1}, x_i]$, $i = 1, 2, \ldots, n$.

Геометрично е ясно е, че голямата сума на Дарбу представлява лице на фигура съставена от правоъгълници, която е описана около криволинейния трапец, определен от графиката на f, а малката сума на Дарбу представлява лице на фигура съставена от правоъгълници, която е вписана в криволинейния трапец, определен от графиката на f.

Твърдение: Нека функцията f е дефинирана и ограничена в интервала [a, b]. Нека τ е разбиване, определено от точките $\{x_i\}$ и x' е точка $\neq x_i$ за $i = 1, 2, \ldots, n$. Нека $\tau^* = \tau \cup \{x'\}$. Тогава

1. $s(\tau^*) \geq s(\tau)$.
2. $S(\tau^*) \leq S(\tau)$.
3. Ако $M = \sup_{x \in [a, b]} f(x)$, то $S(\tau) - S(\tau^*) \leq 2.M.d(\tau)$, $s(\tau^*) - s(\tau) \leq 2.M.d(\tau)$.

Доказателство: Нека $x' \in (x_{i-1}, x_i)$ (такова i има единствено).
Нека $m' = \inf_{x \in [x_{i-1}, x']} f(x)$, $m'' = \inf_{x \in [x', x_i]} f(x)$. Ясно е, че $m' \geq m_i$, $m'' \geq m_i$.
Получаваме:
$s(\tau^*) - s(\tau) = m'.(x' - x_{i-1}) + m''.(x_i - x') - m_i.(x_i - x_{i-1}) =$
$= (m' - m_i).(x' - x_{i-1}) + (m'' - m_i).(x_i - x') \geq 0.$
Тъй като $m' - m_i \leq 2.M$, $m'' - m_i \leq 2.M$, $x' - x_{i-1} < x_i - x_{i-1} \leq d(\tau)$,
$x_i - x' < x_i - x_{i-1} \leq d(\tau) \Rightarrow s(\tau^*) - s(\tau) \leq 2.M.d(\tau)$.
Аналогично, нека $M' = \sup_{x \in [x_{i-1}, x']} f(x)$, $M'' = \sup_{x \in [x', x_i]} f(x)$.
Ясно е, че $M' \leq M_i$, $M'' \leq M_i$. Получаваме:
$S(\tau) - S(\tau^*) = M_i.(x_i - x_{i-1}) - M'.(x' - x_{i-1}) - M''.(x_i - x') =$
$= (M_i - M').(x' - x_{i-1}) + (M_i - M'').(x_i - x') \geq 0.$
Тъй като $M_i - M' \leq 2.M$, $M_i - M'' \leq 2.M$, $x' - x_{i-1} < x_i - x_{i-1} \leq d(\tau)$,
$x_i - x' < x_i - x_{i-1} \leq d(\tau) \Rightarrow S(\tau) - S(\tau^*) \leq 2.M.d(\tau)$.

По-общо, ако τ^* се получава от τ чрез добавяне на k нови точки, то $0 \leq S(\tau) - S(\tau^*) \leq 2.k.M.d(\tau)$ и $0 \leq s(\tau^*) - s(\tau) \leq 2.k.M.d(\tau)$, което лесно следва от твърдението като се използва факта, че при добавяне на нови точки в разбиването, неговият диаметър не нараства.

Твърдение: Всяка малка сума на Дарбу не надминава коя да е голяма сума на Дарбу.
Доказателство: Нека s ($\{x_i\}$) е малка сума на Дарбу, определена от разбиването $\{x_i\}$. Нека S ($\{y_j\}$) е голяма сума на Дарбу, определена от разбиването $\{y_j\}$. Нека τ е обединението на двете разбивания $\{x_i\}$ и $\{y_j\}$. Тогава τ се получава от $\{x_i\}$ с добавяне на точки $\Rightarrow$ s ($\{x_i\}$) $\leq$ s (τ). Също, τ се получава от $\{y_j\}$ с добавяне на точки $\Rightarrow$ S (τ) $\leq$ S ($\{y_j\}$). От друга страна е изпълнено неравенството s (τ) $\leq$ S (τ) $\Rightarrow$ s ($\{x_i\}$) $\leq$ s (τ) $\leq$ S (τ) $\leq$ S ($\{y_j\}$).

От горното твърдение получаваме, че множеството от всички големи суми на Дарбу за f е ограничено отдолу (от коя да е малка сума на Дарбу за f), освен това е непразно, така че то има точна долна граница. Точната долна граница на големите суми на Дарбу за f се нарича **горен интеграл** на функцията f в интервала [a, b] и се означава с $\overline{I}$.
Аналогично, от горното твърдение получаваме, че множеството от всички малки суми на Дарбу за f е ограничено отгоре (от коя да е голяма сума на Дарбу за f), освен това е непразно, така че то има точна горна граница. Точната горна граница на малките суми на Дарбу за f се нарича **долен интеграл** на функцията f в интервала [a, b] и се означава с $\underline{I}$.
Казваме, че f е **интегруема в смисъл на Дарбу**, ако горният интеграл на f е равен на долния, т.е. $\underline{I} = \overline{I}$.
Ще отбележим, че за произволна ограничена функция f е в сила неравенството $\underline{I} \leq \overline{I}$. Действително, нека s е малка сума на Дарбу за f. Тогава s е долна граница на големите суми на Дарбу за f $\Rightarrow$ s $\leq \overline{I}$ и това е за всяка малка сума на Дарбу s за f, т.е. $\overline{I}$ е горна граница на малките суми на Дарбу за f $\Rightarrow \underline{I} \leq \overline{I}$.

Твърдение: Нека f е дефинирана и ограничена в интервала [a, b]. Тогава горният интеграл $\overline{I}$ на f в интервала [a, b] е равен на границата на големите суми на Дарбу, когато диаметърът на разбиването на интервала [a, b] клони към 0, т.е. за всяко $\varepsilon > 0$, съществува $\delta > 0$, такова че за всяко разбиване τ на интервала [a, b] с d (τ) < δ имаме S (τ) – $\overline{I}$ < ε.
Доказателство: Фиксираме $\varepsilon > 0$. Тогава $\overline{I} + \frac{\varepsilon}{2}$ вече не е точна долна граница на големите суми на Дарбу $\Rightarrow$ съществува разбиване τ_0 на [a, b], такова че S (τ_0) < $\overline{I} + \frac{\varepsilon}{2}$.
Нека τ_0 има k делящи точки. Да изберем числото δ така, че

$0 < \varepsilon < \frac{\varepsilon}{4.M.k}$, където $M = \sup_{x \in [a, b]} f(x)$. Нека τ е разбиване, такова че $d(\tau) < \delta$. Като използваме бележката след по-предното твърдение получаваме: $S(\tau) - S(\tau \cup \tau_0) \leq 2.M.k.d(\tau) < 2.M.k.\delta < \frac{\varepsilon}{2}$.

Освен това, $S(\tau \cup \tau_0) \leq S(\tau_0) < \overline{I} + \frac{\varepsilon}{2} \Rightarrow S(\tau \cup \tau_0) - \overline{I} < \frac{\varepsilon}{2}$.

Като съберем двете неравенства получаваме:

$S(\tau) - \overline{I} < \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon$.

Твърдение: Нека f е дефинирана и ограничена в интервала [a, b]. Тогава долният интеграл $\underline{I}$ на f в интервала [a, b] е равен на границата на малките суми на Дарбу, когато диаметърът на разбиването на интервала [a, b] клони към 0, т.е. за всяко $\varepsilon > 0$, съществува $\delta > 0$, такова че за всяко разбиване τ на интервала [a, b] с $d(\tau) < \delta$ имаме $\underline{I} - s(\tau) < \varepsilon$.

Доказателство: Аналогично на горното твърдение.

Твърдение: Нека f е дефинирана и ограничена в интервала [a, b]. Тогава f е интегруема в смисъл на Дарбу $\Leftrightarrow$ за всяко $\varepsilon > 0$ съществува разбиване τ, такова че $S(\tau) - s(\tau) < \varepsilon$.

Доказателство: $\Rightarrow$. Нека f е интегруема в смисъл на Дарбу и $I = \overline{I} = \underline{I}$. Избираме $\varepsilon > 0$. От дефиницията за горен и долен интеграл съществува разбиване τ_1, такова че $S(\tau_1) - I < \varepsilon/2$ и разбиване τ_2, такова че $I - s(\tau_2) < \varepsilon/2$. Нека $\tau = \tau_1 \cup \tau_2$. Тогава:
$S(\tau) \leq S(\tau_1) \Rightarrow S(\tau) - I < \varepsilon/2$ и
$s(\tau) \geq s(\tau_2) \Rightarrow I - s(\tau) < \varepsilon/2 \Rightarrow S(\tau) - s(\tau) < \varepsilon$.

$\Leftarrow$. Нека $\varepsilon > 0$ и да вземем разбиване τ, такова че $S(\tau) - s(\tau) < \varepsilon$. Изпълнени са неравенствата: $s(\tau) \leq \underline{I} \leq \overline{I} \leq S(\tau)$
$\Rightarrow \overline{I} - \underline{I} \leq S(\tau) - s(\tau) < \varepsilon$. Последното неравенство е изпълнено за всяко $\varepsilon > 0 \Rightarrow \underline{I} = \overline{I} \Rightarrow$ f е интегруема в смисъл на Дарбу.

Теорема: Нека f е дефинирана в интервала [a, b].
Тогава f е интегруема в Риманов смисъл $\Leftrightarrow$ f е ограничена и f е интегруема в смисъл на Дарбу.

Доказателство:

$\Rightarrow$.Нека f е ограничена и интегруема в смисъл на Дарбу.
Нека $I = \overline{I} = \underline{I}$. В сила са неравенствата:
$s(\{x_i\}) \leq \sigma(\{x_i\}, t_i) \leq S(\{x_i\})$ за всяко разбиване $\{x_i\}$. В тях извършваме граничен преход при $d(\{x_i\}) \to 0$, използваме двете дуални твърдения от по-горе и получаваме, че
$s(\{x_i\}) \to I$, $S(\{x_i\}) \to I \Rightarrow \sigma(\{x_i\}, t_i) \to I \Rightarrow$ f е интегруема по Риман.

□. Нека f е интегруема в Риманов смисъл. Тогава от първото твърдение, f е ограничена в [a, b]. Фиксираме □ > 0.
Избираме □ > 0, такова че за всяко разбиване { x_i} с d ({ x_i}) < □ да имаме |⌐ ({ x_i}, t_i) - I| < □/6 □ I - $\frac{□}{6}$ < ⌐ ({ x_i}, t_i) < I + $\frac{□}{6}$ за всеки избор на междинните точки t_i.
Нека ↦≥ $\frac{□}{3.(b-a)}$ > 0. Тъй като M_i е супремумът на f (x) в интервала [x_{i-1}, x_i], тогава M_i - ↦не е супремум □ съществуват междинни точки t_i□, такива че f (t_i□) > M_i - ↦за всяко i. Образуваме съответната риманова сума със същото разбиване { x_i} и междинните точки t_i□.
Имаме: ⌐ ({ x_i}, t_i□) > S ({ x_i}) - ↦(b – a) = S ({ x_i}) - $\frac{□}{3}$ □
S ({ x_i}) - $\frac{□}{3}$ < I + $\frac{□}{6}$ □ S ({ x_i}) < I + $\frac{□}{2}$. Аналогично, тъй като m_i е инфимум на f (x) в интервала [x_{i-1}, x_i], тогава m_i + ↦не е инфимум □ съществуват междинни точки t_i□□, такива че f (t_i□□) < m_i + ↦за всяко i. Образуваме съответната риманова сума със същото разбиване { x_i} и междинните точки t_i□□.Имаме:
⌐ ({ x_i}, t_i□□) < s ({ x_i}) + ↦(b – a) = s ({ x_i}) + $\frac{□}{3}$ □ s ({ x_i}) + $\frac{□}{3}$ > I – $\frac{□}{6}$ □
□ s ({ x_i}) > I – $\frac{□}{2}$ □ – s ({ x_i}) < $\frac{□}{2}$ – I. Като съберем това неравенство с полученото неравенство по-горе получаваме
S ({ x_i}) – s ({ x_i}) < I + $\frac{□}{2}$ + $\frac{□}{2}$ – I = □. От горното твърдение □ f е интегруема в смисъл на Дарбу.

От горните теорема и твърдение получаваме, че следните условия за една функция f, дефинирана в [a, b] са еквивалентни:

1. f е интегруема в Риманов смисъл.
2. f е ограничена и е интегруема в смисъл на Дарбу.
3. f е ограничена и за всяко □ > 0 съществува разбиване ⌐, такова че S (⌐) – s (⌐) < □.

Теорема (Кантор): Ако f (x) е дефинирана и непрекъсната в крайния затворен интервал [a, b], то f (x) е равномерно непрекъсната в [a, b].

Теорема: Нека f (x) е дефинирана и непрекъсната в крайния затворен интервал [a, b]. Тогава f (x) е интегруема в Риманов смисъл.
Доказателство: Преди всичко от теоремата на Вайерщрас f е ограничена, така че е достатъчно да покажем, че за всяко □ > 0 съществува разбиване ⌐, такова че S (⌐) – s (⌐) < □.

От теорема на Кантор f (x) е равномерно непрекъсната в [a, b].
Фиксираме > 0. Да изберем > 0, така че

от x, x [a, b] и |x - x| < да следва |f (x) – f (x)| < $\frac{\varepsilon}{(b-a)}$.

Образуваме разбиване { x_i} на [a, b], така че d ({ x_i}) < .
Разглеждаме интервалът [x_{i-1}, x_i]. От теоремата на Вайерщрас,
M_i = f (x), m_i = f (x) за някои x, x [x_{i-1}, x_i].

Тъй като x, x [x_{i-1}, x_i], то |x - x| < f (x) – f (x) < $\frac{\varepsilon}{(b-a)}$

 $M_i - m_i < \frac{\varepsilon}{(b-a)}$. В такъв случай, S ({ x_i}) – s ({ x_i}) =

$= \sum_{i=1}^{n} (M_i - m_i).(x_i - x_{i-1}) < \sum_{i=1}^{n} \frac{\varepsilon}{(b-a)}.(x_i - x_{i-1}) = \varepsilon$.

Така f е интегруема в Риманов смисъл.

Ще изброим без доказателство основните свойства на Римановия интеграл.

Свойство 1.: $\int_a^b C\,dx = C.(b - a)$.

Свойство 2.: Ако $\int_a^b f(x)dx$ и $\int_b^c f(x)dx$ съществуват, то съществува $\int_a^c f(x)dx$ и е изпълнено: $\int_a^c f(x)dx = \int_a^b f(x)dx + \int_b^c f(x)dx$.

Свойство 3: Ако $\int_a^b f(x)dx$ и $\int_a^b g(x)dx$ съществуват, то съществува $\int_a^b (f(x)+g(x))dx$ и е изпълнено: $\int_a^b (f(x)+g(x))dx = \int_a^b f(x)dx + \int_a^b g(x)dx$.

Свойство 4: Ако $\int_a^b f(x)dx$ съществува, то съществува $\int_a^b \ell.f(x)dx$ и е изпълнено: $\int_a^b \ell.f(x)dx = \ell.\int_a^b f(x)dx$.

Свойство 5: Нека f и g са интегруеми функции. Ако f (x) g (x) за всяко x [a, b], то е изпълнено: $\int_a^b f(x)dx \le \int_a^b g(x)dx$.

Теорема: Нека f е дефинирана и непрекъсната в затворения интервал [a, b]. Тогава съществува t [a, b], такова че

$\int_a^b f(x)dx = f(t).(b - a)$.

Доказателство: По теоремата на Вайерщрас f е ограничена.

Нека $m = \inf_{x \in [a, b]} f(x)$, $M = \sup_{x \in [a, b]} f(x)$. За всяко $x \in [a, b]$ имаме:

$m \le f(x) \le M \Rightarrow \int_a^b m\,dx \le \int_a^b f(x)dx \le \int_a^b M\,dx \Rightarrow$

$\Rightarrow m.(b - a) \le \int_a^b f(x)dx \le M.(b - a) \Rightarrow m \le \frac{\int_a^b f(x)dx}{b-a} \le M.$

По теоремата на Вайерщрас $m = f(x_1)$, $M = f(x_2)$ за някои $x_1, x_2 \in [a, b]$ и по теоремата за междинните стойности съществува

$t \in [x_1, x_2]$, такова че $f(t) = \frac{\int_a^b f(x)dx}{b-a} \Rightarrow \int_a^b f(x)dx = f(t).(b - a)$.

Теорема (Лайбниц – Нютон): Нека $f : [a, b] \to \mathbb{R}$ е непрекъсната функция. Тогава функцията $F(x) = \int_a^x f(t)\,dt$ е диференцируема и нейната производна е $f(x)$ за всяко $x \in [a, b]$.

Доказателство: Нека $x \in [a, b]$. Записваме диференчното частно на $F(x)$: $(x+h \in [a, b])$ $\frac{F(x+h)-F(x)}{h} = \frac{\int_a^{x+h} f(t)\,dt - \int_a^x f(t)\,dt}{h} = \frac{\int_x^{x+h} f(t)\,dt}{h}$.

Съгласно горната теорема съществува точка

$u_h \in [x, x+h]$, такава че $\frac{\int_x^{x+h} f(t)\,dt}{h} = f(u_h)$.

Сега като извършим граничен преход при $h \to 0$, получаваме че $u_h \to x \Rightarrow f(u_h) \to f(x)$, тъй като функцията f е непрекъсната $\Rightarrow \frac{F(x+h)-F(x)}{h} \to f(x) \Rightarrow F$ е диференцируема и $F'(x) = f(x)$ за всяко $x \in [a, b]$.

Да предположим, че трябва да пресметнем интегралът $\int_a^b f(x)dx$, където $f(x)$ е непрекъсната функция. Образуваме функцията $F(x) = \int_a^x f(t)\,dt$. Тогава очевидно търсеното число е $F(b)$.

Нека $\Phi(x)$ е примитивна на $f(x)$ в $[a, b]$. От теоремата на Лайбниц-Нютон $F(x)$ също е примитивна на $f(x)$ в $[a, b] \Rightarrow$ съществува константа C, така че $F(x) = \Phi(x) + C$ за всяко x в $[a, b]$. При $x = a$ получаваме $F(a) = \Phi(a) + C$, т.е. $0 = \Phi(a) + C \Rightarrow C = -\Phi(a)$. Така $F(x) = \Phi(x) - \Phi(a)$ за всяко x в $[a, b] \Rightarrow$

□ F (b) = Φ (b) - Φ (a). Окончателно, $\int_a^b f(x)dx = \Phi(x)\Big|_a^b$, където Φ е произволна примитивна на f в [a, b].

22. Диференцируеми функции на много променливи. Диференциране на съставни функции.

Нека е дадена функцията f (x, y) с дефиниционна област M □ □^{2} и нека (x_0, y_0) е вътрешна точка за M, т.е. съществува число □, такова че околността { (x, y) | |x – x_0|< □, |y – y_0|< □ } □ M.
За стойности на x, такива че |x – x_0|< □, точките (x, y_0) попадат в M. Нека разглеждаме функцията f (x, y) за такива точки – получаваме някаква функция □ (x) зависеща само от x, т.е. □ (x) = f (x, y_0), |x – x_0|< □. Ако съществува производната на □ (x) в точката x_0, казваме че функцията f (x, y) е **диференцируема частно** относно x в точката (x_0, y_0), а самата производна □ □(x_0) се нарича **частна производна** на f относно x в точката (x_0, y_0) и се означава с $\frac{\partial f}{\partial x}(x_0, y_0)$ или f_x (x_0, y_0). Аналогично, при фиксиране на втората променлива, т.е. изхождайки от функцията □ (y) = f (x_0, y), |y – y_0|< □, получаваме дефиниция на частна производна на f относно y в точката (x_0, y_0) – това е числото □ □(y_0), ако то съществува и се бележи с $\frac{\partial f}{\partial y}(x_0, y_0)$ или f_y (x_0, y_0).

Дефиницията може да се префразира и по следния начин:

$f_x(x_0, y_0) = \lim_{h \to 0} \frac{f(x_0+h, y_0) - f(x_0, y_0)}{h}$ и

$f_y(x_0, y_0) = \lim_{h \to 0} \frac{f(x_0, y_0+h) - f(x_0, y_0)}{h}$.

Ясно е как се обобщава дефиницията за частна производна на произволна функция f (x_1, x_2, …, x_n) на n променливи. Именно, частната производна на f относно променливата x_i в точката $(x_1^0, \ldots, x_n^0)$ е $f_{x_i}(x_1^0, \ldots, x_n^0) \square \lim_{h \to 0} \frac{f(x_1^0, \ldots, x_i^0+h, \ldots, x_n^0) - f(x_1^0, \ldots, x_n^0)}{h}$, ако тази граница съществува.

Нека f (x, y) е дефинирана в някаква околност на точката (x_0, y_0). Казваме, че f е **диференцируема** в точката (x_0, y_0), ако при (x, y) □ (x_0, y_0) е в сила представянето

f (x, y) = f (x_0, y_0) + a.(x – x_0) + b.(y – y_0) + $o\left(\sqrt{(x-x_0)^2+(y-y_0)^2}\right)$, където a, b □ □ са числа, които не зависят от x и y. Еквивалентен запис е

$$\lim_{(x,y) \to (x_0,y_0)} \frac{f(x,y) - f(x_0,y_0) - a.(x-x_0) - b.(y-y_0)}{\sqrt{(x-x_0)^2+(y-y_0)^2}} = 0.$$

Теорема: Нека f (x, y) е дефинирана в околност на точката (x_0, y_0) и е диференцируема в тази точка, т.е. при (x, y) $\in (x_0, y_0)$ е в сила горното представяне. Тогава f притежава частни производни относно x и y и a = $\frac{\partial f}{\partial x}(x_0, y_0)$, b = $\frac{\partial f}{\partial y}(x_0, y_0)$.

Доказателство: В представянето заместваме y = y_0 и получаваме f (x, y_0) = f (x_0, y_0) + a.(x – x_0) + *o* ($|x - x_0|$) при x $\to x_0$. С други думи, $\lim\limits_{x \to x_0} \frac{f(x, y_0) - f(x_0, y_0) - a.(x - x_0)}{|x - x_0|}$ = 0. При x $\to x_0 + 0$ имаме

$\lim\limits_{x \to x_0 + 0} \frac{f(x, y_0) - f(x_0, y_0) - a.(x - x_0)}{x - x_0}$ = 0, т.е. $\lim\limits_{x \to x_0 + 0} \frac{f(x, y_0) - f(x_0, y_0)}{x - x_0}$ = a.

При x $\to x_0 - 0$ имаме

$\lim\limits_{x \to x_0 - 0} \frac{f(x, y_0) - f(x_0, y_0) - a.(x - x_0)}{x_0 - x} \Leftrightarrow \lim\limits_{x \to x_0 - 0} \frac{a.(x - x_0) - (f(x, y_0) - f(x_0, y_0))}{x - x_0}$ = 0,

т.е. a = $\lim\limits_{x \to x_0 - 0} \frac{f(x, y_0) - f(x_0, y_0)}{x - x_0}$. С други думи, лявата и дясната граница на диференчното частно $\frac{f(x, y_0) - f(x_0, y_0)}{x - x_0}$ при x $\to x_0$ съществуват и са равни на a $\Leftrightarrow$ f е диференцируема частно относно x в точката (x_0, y_0) и $\frac{\partial f}{\partial x}(x_0, y_0)$ = a. Съвсем аналогично се показва, че $\frac{\partial f}{\partial y}(x_0, y_0)$ съществува и $\frac{\partial f}{\partial y}(x_0, y_0)$ = b.

Теорема: Нека f (x, y) е дефинирана и притежава частни производни относно x и y в някоя околност на точката (x_0, y_0), които са непрекъснати в тази точка. Тогава f е диференцируема в точката (x_0, y_0).

Доказателство: Имаме f (x, y) – f (x_0, y_0) - $\frac{\partial f}{\partial x}(x_0, y_0).(x - x_0)$ –

- $\frac{\partial f}{\partial y}(x_0, y_0).(y - y_0)$ = f (x, y) – f (x_0, y) + f (x_0, y) – f (x_0, y_0) –

- $\frac{\partial f}{\partial x}(x_0, y_0).(x - x_0) - \frac{\partial f}{\partial y}(x_0, y_0).(y - y_0)$. Функциите φ (x) = f (x, y) и ψ (y) = f (x_0, y) са непрекъснати и диференцируеми между x и x_0 и y и y_0, съответно, тъй като f притежава частни производни относно x и y. По теоремата за крайните нараствания на Лагранж съществуват числа x_1 между x и x_0 и y_1 между y и y_0, такива че

φ (x) - φ (x_0) = $\varphi'(x_1).(x - x_0)$, т.е. f (x, y) – f (x_0, y) = $\frac{\partial f}{\partial x}(x_1, y).(x - x_0)$ и

ψ (y) - ψ (y_0) = $\psi'(y_1).(y - y_0)$, т.е. f (x_0, y) – f (x_0, y_0) = $\frac{\partial f}{\partial y}(x_0, y_1).(y - y_0)$.

Така горното равенство приема вида f (x, y) – f (x_0, y_0) –

$$- \frac{\partial f}{\partial x}(x_0, y_0).(x - x_0) - \frac{\partial f}{\partial y}(x_0, y_0).(y - y_0) = \frac{\partial f}{\partial x}(x, y).(x - x_0) +$$

$$+ \frac{\partial f}{\partial y}(x_0, y).(y - y_0) - \frac{\partial f}{\partial x}(x_0, y_0).(x - x_0) - \frac{\partial f}{\partial y}(x_0, y_0).(y - y_0) =$$

$$= \left(\frac{\partial f}{\partial x}(x, y) - \frac{\partial f}{\partial x}(x_0, y_0)\right).(x - x_0) + \left(\frac{\partial f}{\partial y}(x_0, y) - \frac{\partial f}{\partial y}(x_0, y_0)\right).(y - y_0).$$

Така $\frac{f(x,y) - f(x_0,y_0) - \frac{\partial f}{\partial x}(x_0,y_0).(x-x_0) - \frac{\partial f}{\partial y}(x_0,y_0).(y-y_0)}{\sqrt{(x-x_0)^2 + (y-y_0)^2}}$ ≤

$$\leq \frac{\left|\frac{\partial f}{\partial x}(x, y) - \frac{\partial f}{\partial x}(x_0, y_0)\right|.|x - x_0| + \left|\frac{\partial f}{\partial y}(x_0, y) - \frac{\partial f}{\partial y}(x_0, y_0)\right|.|y - y_0|}{\sqrt{(x - x_0)^2 + (y - y_0)^2}}.$$

При $(x, y) \to (x_0, y_0)$ имаме $x \to x_0$, $y \to y_0$ и тъй като $\frac{\partial f}{\partial x}$ и $\frac{\partial f}{\partial y}$ са непрекъснати в (x_0, y_0), то $\left|\frac{\partial f}{\partial x}(x, y) - \frac{\partial f}{\partial x}(x_0, y_0)\right| \to 0$ и $\left|\frac{\partial f}{\partial y}(x_0, y) - \frac{\partial f}{\partial y}(x_0, y_0)\right| \to 0$, освен това $\frac{x - x_0}{\sqrt{(x - x_0)^2 + (y - y_0)^2}}$, $\frac{y - y_0}{\sqrt{(x - x_0)^2 + (y - y_0)^2}}$ са ограничени ⇒ последният израз клони към 0,

т.е. $\lim_{(x,y) \to (x_0,y_0)} \frac{f(x,y) - f(x_0,y_0) - \frac{\partial f}{\partial x}(x_0,y_0).(x-x_0) - \frac{\partial f}{\partial y}(x_0,y_0).(y-y_0)}{\sqrt{(x-x_0)^2 + (y-y_0)^2}} = 0,$

което доказва диференцируемост на f в точката (x_0, y_0).

Теорема: Нека f (x, y) е дефинирана и притежава първи частни производни в околност на точката (x_0, y_0). Нека $\frac{\partial f}{\partial x}$ и $\frac{\partial f}{\partial y}$ са непрекъснати в тази точка. Нека функцията $\varphi(t)$ е дефинирана в околност на t_0 и диференцируема в t_0, също така $\varphi(t_0) = x_0$. Нека функцията $\psi(t)$ е дефинирана в околност на t_0 и диференцируема в t_0, също така $\psi(t_0) = y_0$. При тези условия, функцията $g(t) = f(\varphi(t), \psi(t))$ е дефинирана в околност на t_0, диференцируема в точката t_0 и $g'(t_0) = \frac{\partial f}{\partial x}(x_0, y_0).\varphi'(t_0) + \frac{\partial f}{\partial y}(x_0, y_0).\psi'(t_0)$.

Доказателство: Преди всичко, g е дефинирана в околност на t_0, тъй като функциите φ и ψ са диференцируеми ⇒ непрекъснати в t_0 и при стойности на t близки до t_0, $\varphi(t)$ е близко до x_0 и $\psi(t)$ е близко до y_0. Разгледаме диференчното частно на g (t) в точката t_0, имаме:

$$\frac{g(t_0+h)-g(t_0)}{h} = \frac{f\left(\varphi(t_0+h),\psi(t_0+h)\right)-f\left(\varphi(t_0),\psi(t_0)\right)}{h} =$$

$$= \frac{f\left(\varphi(t_0+h),\psi(t_0+h)\right)-f\left(\varphi(t_0+h),\psi(t_0)\right)+f\left(\varphi(t_0+h),\psi(t_0)\right)-f\left(\varphi(t_0),\psi(t_0)\right)}{h}.$$

Функцията p (y) = f (φ (t0 + h), y) е диференцируема (и следователно непрекъсната) в околност на y0, тъй като f (x, y) е диференцируема частно относно y в околност на точката (x0, y0). Прилагаме теоремата за крайните нараствания на Лагранж и получаваме:

$f\left(\varphi(t_0+h),\psi(t_0+h)\right)-f\left(\varphi(t_0+h),\psi(t_0)\right)=$

$= \left(\psi(t_0+h)-\psi(t_0)\right).\frac{\partial f}{\partial y}\left(\varphi(t_0+h),\theta\right)$, където θ е число между ψ (t0) и ψ (t0 + h)]. Аналогично функцията q (x) = f (x, ψ (t0)) е диференцируема (и следователно непрекъсната) в околност на x0, тъй като f (x, y) е диференцируема частно относно x в околност на точката (x0, y0). Прилагаме теоремата за крайните нараствания на Лагранж и получаваме: $f\left(\varphi(t_0+h),\psi(t_0)\right)-f\left(\varphi(t_0),\psi(t_0)\right)=$

$= \left(\varphi(t_0+h)-\varphi(t_0)\right).\frac{\partial f}{\partial x}\left(\eta,\psi(t_0)\right)$, където η е число между φ (t0) и φ (t0 + h). Така диференчното частно на g в t0 приема вида:

$$\frac{\left(\varphi(t_0+h)-\varphi(t_0)\right).\frac{\partial f}{\partial x}\left(\eta,\psi(t_0)\right)}{h} \to \frac{\left(\psi(t_0+h)-\psi(t_0)\right).\frac{\partial f}{\partial y}\left(\varphi(t_0+h),\theta\right)}{h}.$$

Извършваме граничен преход при h → 0. Тъй като функциите φ (t) и ψ (t) са диференцируеми и следователно непрекъснати в t0, то φ (t0 + h) → φ (t0), ψ (t0 + h) → ψ (t0), така че η → φ (t0), θ → ψ (t0). Също така $\frac{\varphi(t_0+h)-\varphi(t_0)}{h} \to \varphi'(t_0)$, $\frac{\psi(t_0+h)-\psi(t_0)}{h} \to \psi'(t_0)$.

И накрая, частните производни $\frac{\partial f}{\partial x}$ и $\frac{\partial f}{\partial y}$ са непрекъснати в точката (x0, y0) = (φ (t0), ψ (t0)) → $\frac{\partial f}{\partial x}(\eta,\psi(t_0)) \to \frac{\partial f}{\partial x}(\varphi(t_0),\psi(t_0))$ =

$= \frac{\partial f}{\partial x}(x_0,y_0)$ и $\frac{\partial f}{\partial y}\left(\varphi(t_0+h),\theta\right) \to \frac{\partial f}{\partial y}\left(\varphi(t_0),\psi(t_0)\right) = \frac{\partial f}{\partial y}(x_0,y_0)$.

В такъв случай, границата на диференчното частно при h → 0 съществува, т.е. g (t) е диференцируема в точката t0 и

$g'(t_0) = \frac{\partial f}{\partial x}(x_0,y_0).\varphi'(t_0) + \frac{\partial f}{\partial y}(x_0,y_0).\psi'(t_0)$.

23. Уравнения на права и равнина. Формули за разстояния и ъгли.

Нека в равнината разглеждаме права g и нека M0 е дадена точка от g. Нека $\vec{p}$ е ненулев вектор, колинеарен с g. Ако някаква

точка M лежи на правата g, то векторите $\overrightarrow{M_0M}$ и $\vec{p}$ са колинеарни и тогава съществува единствено реално число ℓ, такова че $\overrightarrow{M_0M} = \ell . \vec{p}$. Обратно, ако за точката M в равнината е изпълнено, че $\overrightarrow{M_0M} = \ell . \vec{p}$ за някое реално число ℓ, то M лежи на g. Нека $K = O\overrightarrow{e_1}\overrightarrow{e_2}$ е афинна координатна система в равнината. Имаме $\overrightarrow{M_0M} = \overrightarrow{OM} - \overrightarrow{OM_0}$ и тогава горното равенство се записва във вида $\overrightarrow{OM} = \overrightarrow{OM_0} + \ell . \vec{p}$ или $\vec{r} = \vec{r_0} + \ell . \vec{p}$, където сме положили $\overrightarrow{OM} = \vec{r}$ и $\overrightarrow{OM_0} = \vec{r_0}$.

Последното равенство се нарича **векторно параметрично уравнение** на правата g, определена от точката M_0 и вектора $\vec{p}$. Нека сега спрямо K имаме M_0 (x_0, y_0), M (x, y) и $\vec{p}$ (a, b). Тогава равенството записваме във вида $\begin{vmatrix} x = x_0 + \ell .a \\ y = y_0 + \ell .b \end{vmatrix}$ и наричаме **скаларни параметрични уравнения** на правата g, определена от точката M_0 и вектора $\vec{p}$. Ако правата g е зададена с две различни точки M_1 (x_1, y_1) и M_2 (x_2, y_2), скаларните параметрични уравнения на g имат вида $\begin{vmatrix} x = x_1 + \ell .(x_2 - x_1) \\ y = y_1 + \ell .(y_2 - y_1) \end{vmatrix}$, тъй като g може да се зададе с точката M_1 (x_1, y_1) и ненулевият вектор $\overrightarrow{M_1M_2}$ ($x_2 - x_1$, $y_2 - y_1$).

Нека в пространството е дадена равнина $\mapsto$ Тя може да се определи еднозначно с точка M_0, която лежи в нея и два ненулеви, неколинеарни вектора $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$, компланарни с нея. Произволна точка M в пространството лежи в $\mapsto$точно когато векторът $\overrightarrow{M_0M}$ е компланарен с $\mapsto$ което е изпълнено тогава и само тогава, когато векторът $\overrightarrow{M_0M}$ е линейна комбинация на $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$, т.е. съществуват реални числа ℓ, $\mp$ такива, че $\overrightarrow{M_0M} = \ell . \overrightarrow{p_1} + \mp . \overrightarrow{p_2}$. Ако фиксираме афинна координатна система $K = O\overrightarrow{e_1}\overrightarrow{e_2}\overrightarrow{e_3}$ в пространството, то $\overrightarrow{M_0M} = \overrightarrow{OM} - \overrightarrow{OM_0}$ и горното равенство можем да запишем във вида $\overrightarrow{OM} = \overrightarrow{OM_0} + \ell . \overrightarrow{p_1} + \mp . \overrightarrow{p_2}$ или $\vec{r} = \vec{r_0} + \ell . \overrightarrow{p_1} + \mp . \overrightarrow{p_2}$. Последното равенство се нарича **векторно параметрично уравнение** на равнината $\mapsto$ зададена с точката M_0 и векторите $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$.

Ако спрямо K имаме M_0 (x_0, y_0, z_0), M (x, y, x), $\overrightarrow{p_1}$ (a_1, b_1, c_1) и

$\overrightarrow{p_2}$ (a_2, b_2, c_2) горното равенство можем да запишем във вида

$$\begin{cases} x = x_0 + \ell.a_1 + \mp.a_2 \\ y = y_0 + \ell.b_1 + \mp.b_2 \\ z = z_0 + \ell.c_1 + \mp.c_2 \end{cases}$$ - **скаларни параметрични уравнения** на

равнината $\mapsto$ зададена с точката M_0 и векторите $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$.

Ако равнината $\mapsto$е зададена с три неколинеарни точки M_1 (x_1, y_1, z_1), M_2 (x_2, y_2, z_2) и M_3 (x_3, y_3, z_3), скаларните параметрични уравнения на $\mapsto$имат вида

$$\begin{cases} x = x_1 + \ell.(x_2 - x_1) + \mp.(x_3 - x_1) \\ y = y_1 + \ell.(y_2 - y_1) + \mp.(y_3 - y_1) \\ z = z_1 + \ell.(z_2 - z_1) + \mp.(z_3 - z_1) \end{cases}$$, тъй като $\mapsto$може да се зададе с точката M_1 (x_1, y_1, z_1) и ненулевите, неколинеарни вектори $\overrightarrow{M_1M_2}$ ($x_2 - x_1$, $y_2 - y_1$, $z_2 - z_1$) и $\overrightarrow{M_1M_3}$ ($x_3 - x_1$, $y_3 - y_1$, $z_3 - z_1$).

Отново разглеждаме равнината и фиксирана в нея афинна координатна система $K = O\overrightarrow{e_1}\overrightarrow{e_2}$.

Теорема: Нека е дадена правата g. Съществуват реални числа A, B, C, такива че (A, B) $\neq$ (0, 0) и точката M (x, y) лежи на g $\Leftrightarrow$ $A.x + B.y + C = 0$.

Доказателство: Върху правата g избираме точка M_0 (x_0, y_0) и ненулев вектор $\vec{p}$ (a, b), колинеарен с g. Точката M (x, y) лежи на g $\Leftrightarrow$ векторите $\vec{p}$ и $\overrightarrow{M_0M}$ са колинеарни, т.е. $\overrightarrow{M_0M} = \ell.\vec{p}$ за някое реално число ℓ. Последното равенство скаларно се записва така: $x = x_0 + \ell.a$, $y = y_0 + \ell.b$ $\Rightarrow$ $b.x = b.x_0 + \ell.a.b$, $a.y = a.y_0 + \ell.a.b$ $\Rightarrow$ $\Rightarrow$ $b.x - a.y - b.x_0 + a.y_0 = 0$. Сега полагаме A = b, B = -a, $C = -b.x_0 + a.y_0$. Естествено, (A, B) $\neq$ (0, 0), тъй като $\vec{p}$ е ненулев вектор. Дотук, ако M (x, y) лежи на g, то $A.x + B.y + C = 0$.

Нека M (x, y) е такава, че $A.x + B.y + C = 0$, т.е. $b.x - a.y - b.x_0 + a.y_0 = 0$. Нека за определеност $b \neq 0$ $\Rightarrow$ $x - x_0 = \frac{a}{b}.(y - y_0)$. Тогава имаме $\overrightarrow{M_0M}(x - x_0, y - y_0) =$

$= (\frac{a}{b}.(y - y_0), y - y_0) = \frac{y - y_0}{b}.(a, b) = \ell.\vec{p}$ за $\ell = \frac{y - y_0}{b}$.

Така M лежи на g.

Теорема: Нека A, B, C са три реални числа, такива че (A, B) $\neq$ (0, 0). Съществува единствена права g, такава че точката M (x, y) лежи на g $\Leftrightarrow$ $A.x + B.y + C = 0$.

Доказателство: Тъй като (A, B) $\neq$ (0, 0), то уравнението $A.x + B.y + C = 0$ има решения, нека (x_0, y_0) е едно такова решение, т.е. $A.x_0 + B.y_0 + C = 0$. Да означим точката M_0 (x_0, y_0) и векторът

$\vec{p}$ (- B, A), който е ненулев. Те определят единствена права g, за която от предишната теорема имаме M (x, y) лежи на g $\Leftrightarrow$ A.x + B.y – A.x_0 – B.y_0 = 0 $\Leftrightarrow$ A.x + B.y + C = 0. Да допуснем, че правата h е такава, че M (x, y) лежи на h $\Leftrightarrow$ A.x + B.y + C = 0. Тогава M (x, y) лежи на h $\Leftrightarrow$ M (x, y) лежи на g $\Leftrightarrow$ h $\Leftrightarrow$ g.

Така от двете теореми получаваме, че в равнината, спрямо фиксирана координатна система, всяка права g се задава еднозначно с линейно уравнение от вида A.x + B.y + C = 0, където (A, B) $\neq$ (0, 0) и векторът $\vec{p}$ (- B, A) е колинеарен с g. Това уравнение се нарича **общо уравнение** на правата g в равнината.

Нека сега правата g пресича ординатната ос, т.е. векторът $\overrightarrow{e_2}$ не е колинеарен с g. Нека $\vec{p}$ (a, b) е ненулев вектор, колинеарен с g. Тъй като $\vec{p}$ и $\overrightarrow{e_2}$ не са колинеарни, то a $\neq$ 0. Следователно, можем да образуваме k = $\frac{b}{a}$. Това число k не зависи от избора на вектора $\vec{p}$. Действително, ако $\overrightarrow{p_1}$ (a_1, b_1) е ненулев и колинеарен с g, то $\vec{p}$ е колинеарен с $\overrightarrow{p_1}$ $\Leftrightarrow$ $\overrightarrow{p_1}$ = ℓ.$\vec{p}$, при това ℓ $\neq$ 0, тъй като $\overrightarrow{p_1}$ е ненулев. От последното равенство получаваме a_1 = ℓ.a, b_1 = ℓ.b $\Rightarrow$ $\frac{b_1}{a_1} = \frac{\ell.b}{\ell.a} = \frac{b}{a}$. Числото k се нарича **ъглов коефициент** на правата g. Когато координатната система е ортонормирана, k = tg φ, където φ е мярката на ориентирания ъгъл между абсцисната ос и правата g. Нека сега g пресича ординатната ос в точката P (0, n). Произволна точка M (x, y) лежи на g $\Leftrightarrow$ векторът $\overrightarrow{PM}$ (x, y – n) е колинеарен с g $\Leftrightarrow$ $\frac{y-n}{x}$ = k $\Leftrightarrow$ y = k.x + n.
Последното уравнение се нарича **декартово уравнение** на правата g с ъглов коефициент k и вертикален отрез n.
Ако g е зададена с общото си уравнение A.x + B.y + C = 0, то B $\neq$ 0, тъй като g пресича ординатната ос $\Rightarrow$ уравнението може да се запише във вида y = $-\frac{A}{B}$.x $-\frac{C}{B}$, т.е. g има ъглов коефициент k = $-\frac{A}{B}$ и вертикален отрез n = $-\frac{C}{B}$.

Теорема: Нека двете прави g_1 и g_2 са зададени с общите си уравнения g_1 : A_1.x + B_1.y + C_1 = 0 и g_2 : A_2.x + B_2.y + C_2 = 0.
Да означим A = $\begin{pmatrix} A_1 & B_1 \\ A_2 & B_2 \end{pmatrix}$, B = $\begin{pmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{pmatrix}$. Тогава

1. g_1 и g_2 се пресичат $\Leftrightarrow$ r (A) = r (B) = 2.
2. g_1 и g_2 са успоредни $\Leftrightarrow$ r (A) = 1 и r (B) = 2.

3. g_1 и g_2 съвпадат $\Leftrightarrow$ r (A) = r (B) = 1.

Доказателство: Определянето на общите точки g_1 на g_2 е еквивалентно с решаването на системата $\begin{vmatrix} A_1.x + B_1.y + C_1 = 0 \\ A_2.x + B_2.y + C_2 = 0 \end{vmatrix}$.

Правите g_1 и g_2 се пресичат $\Leftrightarrow$ системата има единствено решение $\Leftrightarrow$ r (A) = r (B) = 2 $\Leftrightarrow$ detA = $A_1.B_2 - A_2.B_1 \neq 0$.

Правите g_1 и g_2 са успоредни $\Leftrightarrow$ системата няма решение $\Leftrightarrow$ $\Leftrightarrow$ r (A) = 1 и r (B) = 2.

Правите g_1 и g_2 съвпадат $\Leftrightarrow$ системата има безброй много решения $\Leftrightarrow$ r (A) = r (B) = 1 $\Leftrightarrow$ съществува реално число $\ell \neq 0$, такова че $A_1 = \ell.A_2$, $B_1 = \ell.B_2$, $C_1 = \ell.C_2$.

Следствие: Ако правата g има общо уравнение A.x + B.y + C = 0, (A, B) $\neq$ (0, 0), то всичките нейни общи уравнения имат вида ℓ.A.x + ℓ.B.y + ℓ.C = 0, където ℓ е реално ненулево число.

Нека в равнината е фиксирана ортонормирана координатна система K = $O\vec{e_1}\vec{e_2}$. Нека g е права в равнината с общо уравнение A.x + B.y + C = 0, (A, B) $\neq$ (0, 0). Векторът $\vec{p}$(- B, A) е колинеарен с g. Разглеждаме векторът $\vec{n}$(A, B). Имаме $\vec{p}.\vec{n}$ = -B.A + A.B = 0 и тъй като $\vec{p}$ и $\vec{n}$ са ненулеви, $\vec{p} \perp \vec{n}$ $\Leftrightarrow$ $\vec{n} \perp$ g. Ненулевите вектори, които са перпендикулярни на g наричаме **нормални вектори** на правата g. Така коефициентите пред x и y във всяко общо уравнение на правата g са координати на нормален вектор на g. Онова общо уравнение на правата g, в което коефициентите пред x и y са координати на единичен нормален вектор на g, се нарича **нормално уравнение** на правата g. С други думи, уравнението A.x + B.y + C = 0 на правата g е нормално $\Leftrightarrow$ $A^2 + B^2 = 1$.

Нека ℓ.A.x + ℓ.B.y + ℓ.C = 0, $\ell \neq 0$ е произволно уравнение на правата g. Тогава то е нормално $\Leftrightarrow$ $(\ell.A)^2 + (\ell.B)^2 = 1 \Leftrightarrow \ell^2 = \frac{1}{A^2+B^2} \Leftrightarrow$ $\ell = \pm\frac{1}{\sqrt{A^2+B^2}}$. Така правата g има точно две нормални уравнения:

$\frac{A}{\sqrt{A^2+B^2}}.x + \frac{B}{\sqrt{A^2+B^2}}.y + \frac{C}{\sqrt{A^2+B^2}} = 0$ и

$-\frac{A}{\sqrt{A^2+B^2}}.x - \frac{B}{\sqrt{A^2+B^2}}.y - \frac{C}{\sqrt{A^2+B^2}} = 0$.

Нека правата g е зададена с нормалното си уравнение $\frac{A}{\sqrt{A^2+B^2}}.x + \frac{B}{\sqrt{A^2+B^2}}.y + \frac{C}{\sqrt{A^2+B^2}} = 0$, векторът $\vec{n}\left(\frac{A}{\sqrt{A^2+B^2}}, \frac{B}{\sqrt{A^2+B^2}}\right)$ е единичен нормален вектор на g.

Нека P (x_0, y_0) е произволна точка. Нека P_1 (x_1, y_1) е ортогоналната проекция на P върху правата g. Тогава разстоянието δ от P до g е точно числото $|\overrightarrow{P_1P}|$.

Имаме $\vec{n}.\overrightarrow{P_1P} = |\vec{n}|.|\overrightarrow{P_1P}|.\cos\sphericalangle(\vec{n}, \overrightarrow{P_1P}) = \delta|\overrightarrow{P_1P}|$.

От друга страна, $\vec{n}.\overrightarrow{P_1P} = \frac{A}{\sqrt{A^2+B^2}}.(x_0 - x_1) + \frac{B}{\sqrt{A^2+B^2}}.(y_0 - y_1)$.

При това, P_1 лежи на g $\Leftrightarrow -\frac{A}{\sqrt{A^2+B^2}}.x_1 - \frac{B}{\sqrt{A^2+B^2}}.y_1 = \frac{C}{\sqrt{A^2+B^2}}$ $\Leftrightarrow$

$\Leftrightarrow \vec{n}.\overrightarrow{P_1P} = \frac{A}{\sqrt{A^2+B^2}}.x_0 + \frac{B}{\sqrt{A^2+B^2}}.y_0 + \frac{C}{\sqrt{A^2+B^2}}$.

Така разстоянието от точката P до правата g е

$\delta = |\vec{n}.\overrightarrow{P_1P}| = \frac{|A.x_0 + B.y_0 + C|}{\sqrt{A^2+B^2}}$.

Разглеждаме произволни прави g_1 и g_2, зададени чрез произволни техни общи уравнения $g_1 : A_1.x + B_1.y + C_1 = 0$ и $g_2 : A_2.x + B_2.y + C_2 = 0$. Векторът $\overrightarrow{n_1}(\frac{A_1}{\sqrt{A_1^2+B_1^2}}, \frac{B_1}{\sqrt{A_1^2+B_1^2}})$ е единичен нормален вектор на g_1. Векторът $\overrightarrow{n_2}(\frac{A_2}{\sqrt{A_2^2+B_2^2}}, \frac{B_2}{\sqrt{A_2^2+B_2^2}})$ е единичен нормален вектор на g_2.

Ясно е, че $\sphericalangle(g_1, g_2) = \sphericalangle(\overrightarrow{n_1}, \overrightarrow{n_2}) \Rightarrow \cos\sphericalangle(g_1, g_2) =$

$= \cos\sphericalangle(\overrightarrow{n_1}, \overrightarrow{n_2}) = \frac{\overrightarrow{n_1}.\overrightarrow{n_2}}{|\overrightarrow{n_1}|.|\overrightarrow{n_2}|} = \overrightarrow{n_1}.\overrightarrow{n_2} = \frac{A_1.A_2 + B_1.B_2}{\sqrt{A_1^2+B_1^2}.\sqrt{A_2^2+B_2^2}}$.

Пренасяме се в пространството. Нека е фиксирана афинна координатна система $K = O\overrightarrow{e_1}\overrightarrow{e_2}\overrightarrow{e_3}$.

Теорема: Нека $\mapsto$е произволна равнина в пространството. Съществуват реални числа A, B, C, D, такива че (A, B, C) $\neq$ (0, 0, 0) и една точка M (x, y, z) лежи в $\mapsto \Leftrightarrow$ A.x + B.y + C.z + D = 0.

Доказателство: Нека да изберем една точка M_0 (x_0, y_0, z_0) в $\mapsto$и два ненулеви, неколинеарни вектора $\overrightarrow{p_1}(a_1, b_1, c_1)$ и $\overrightarrow{p_2}(a_2, b_2, c_2)$, компланарни с $\mapsto$ Точката M (x, y, z) лежи в $\mapsto \Leftrightarrow$ векторите $\overrightarrow{M_0M}$, $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$ са компланарни $\Leftrightarrow$ смесеното произведение $\overrightarrow{M_0M}\,\overrightarrow{p_1}\,\overrightarrow{p_2} = 0$

$\Leftrightarrow \begin{vmatrix} x-x_0 & y-y_0 & z-z_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{vmatrix} = 0 \Leftrightarrow$ A.x + B.y + C.z + D = 0, където

$A = b_1.c_2 - b_2.c_1$, $B = a_2.c_1 - a_1.c_2$, $C = a_1.b_2 - a_2.b_1$ и

$D = -\begin{vmatrix} x_0 & y_0 & z_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{vmatrix}$. При това, (A, B, C) $\neq$ (0, 0, 0), тъй като $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$ не са колинеарни, т.е. координатите им не са пропорционални.

Теорема: Нека A, B, C, D са четири реални числа, такива че (A, B, C) $\neq$ (0, 0, 0). Съществува единствена равнина $\mapsto$ такава че точката M (x, y, z) лежи в $\mapsto \Leftrightarrow$ A.x + B.y + C.z + D = 0.
Доказателство: Тъй като (A, B, C) $\neq$ (0, 0, 0), уравнението A.x + B.y + C.z + D = 0 има поне едно решение (x_0, y_0, z_0).
Да вземем точката M_0 (x_0, y_0, z_0). Без ограничение на общността можем да считаме, че B $\neq$ 0. Разглеждаме векторите $\overrightarrow{p_1}$ (-B, A, 0) и $\overrightarrow{p_2}$ (0, $-\frac{C}{B}$, 1), които очевидно са ненулеви и неколинеарни.
Съществува единствена равнина $\mapsto$ в която лежи точката M_0 и която е компланарна с $\overrightarrow{p_1}$ и $\overrightarrow{p_2}$. От предната теорема, точката

M (x, y, z) лежи в $\mapsto \Leftrightarrow \begin{vmatrix} x-x_0 & y-y_0 & z-z_0 \\ -B & A & 0 \\ 0 & -\frac{C}{B} & 1 \end{vmatrix} = 0 \Leftrightarrow$

$\Leftrightarrow$ A.(x – x_0) + B.(y – y_0) + C.(z – z_0) = 0 $\Leftrightarrow$
$\Leftrightarrow$ A.x + B.y + C.z – A.x_0 – B.y_0 – C.z_0 = 0 $\Leftrightarrow$ A.x + B.y + C.z + D = 0, тъй като (x_0, y_0, z_0) е решение на горното уравнение. Да допуснем, че $\updownarrow$ е равнина, такава че M (x, y, z) лежи в $\updownarrow \Leftrightarrow$
$\Leftrightarrow$ A.x + B.y + C.z + D = 0. Тогава M (x, y, z) лежи в $\updownarrow \Leftrightarrow$ M (x, y, z) лежи в $\mapsto \Leftrightarrow \mapsto \Leftrightarrow \updownarrow$.

Така от двете теореми получаваме, че в пространството, спрямо фиксирана координатна система, всяка равнина $\mapsto$ се задава еднозначно с линейно уравнение от вида A.x + B.y + C.z + D = 0, където (A, B, C) $\neq$ (0, 0, 0). Това уравнение се нарича **общо уравнение** на равнината $\mapsto$ в пространството.

Теорема: Нека $\mapsto_1$ и $\mapsto_2$ са две равнини, зададени с общите си уравнения $\mapsto_1$: A_1.x + B_1.y + C_1.z + D_1 = 0 и
$\mapsto_2$: A_2.x + B_2.y + C_2.z + D_2 = 0. Да означим A = $\begin{pmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{pmatrix}$,
B = $\begin{pmatrix} A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \end{pmatrix}$. Тогава

1. $\mapsto_1$ и $\mapsto_2$ се пресичат $\Leftrightarrow$ r (A) = r (B) = 2.
2. $\mapsto_1$ и $\mapsto_2$ са успоредни $\Leftrightarrow$ r (A) = 1 и r (B) = 2.
3. $\mapsto_1$ и $\mapsto_2$ съвпадат $\Leftrightarrow$ r (A) = r (B) = 1.

Доказателство: Определянето на общите точки на ↦₁ и ↦₂ е еквивалентно на решаването на системата $\begin{vmatrix} A_1.x + B_1.y + C_1.z + D_1 = 0 \\ A_2.x + B_2.y + C_2.z + D_2 = 0 \end{vmatrix}$.

Равнините ↦₁ и ↦₂ се пресичат (имат единствена обща права) ⇔ пространството на решенията на системата е едномерно ⇔ r (A) = r (B) = 2.

Равнините ↦₁ и ↦₂ са успоредни ⇔ системата няма решения ⇔ r (A) = 1 и r (B) = 2.

Равнините ↦₁ и ↦₂ съвпадат ⇔ пространството от решенията на системата е двумерно ⇔ r (A) = r (B) = 1.

Следствие: Ако равнината ↦има общо уравнение A.x + B.y + C.z + D = 0, то всичките нейни общи уравнения имат вида ℓ.A.x + ℓ.B.y + ℓ.C.z + ℓ.D = 0, където ℓ е реално, ненулево число.

Твърдение: Нека е дадена равнината ↦с общо уравнение A.x + B.y + C.z + D = 0. Векторът $\vec{p}$ (a, b, c) е компланарен с ↦⇔ ⇔ A.a + B.b + C.c = 0.

Доказателство: Да фиксираме точка M_0 (x_0, y_0, z_0) върху ↦ Съществува единствен представител на вектора $\vec{p}$ с начало M_0 – нека той е $\overrightarrow{M_0M_1}$, M_1 (x_1, y_1, z_1). Тъй като $\vec{p} = \overrightarrow{M_0M_1}$ ⇔ $x_1 - x_0 = a$, $y_1 - y_0 = b$, $z_1 - z_0 = c$ ⇔ $x_0 = x_1 - a$, $y_0 = y_1 - b$, $z_0 = z_1 - c$.

Тъй като M_0 лежи в ↦ то $A.x_0 + B.y_0 + C.z_0 + D = 0$ ⇔ ⇔ $A.(x_1 - a) + B.(y_1 - b) + C.(z_1 - c) + D = 0$ ⇔ $A.x_1 + B.y_1 + C.z_1 + D =$ $= A.a + B.b + C.c$. Така $\vec{p}$ е компланарен с ↦⇔ M_1 лежи в ↦⇔ ⇔ $A.x_1 + B.y_1 + C.z_1 + D = 0$ ⇔ $A.a + B.b + C.c = 0$.

Нека в пространството е фиксирана ортонормирана координатна система K = O$\vec{e_1}\vec{e_2}\vec{e_3}$. Нека ↦е равнина в пространството с общо уравнение A.x + B.y + C.z + D = 0, (A, B, C) ⇔ (0, 0, 0). Разглеждаме векторът $\vec{n}$ (A, B, C). От горното твърдение, за всеки вектор $\vec{p}$ (a, b, c), който е компланарен с ↦имаме A.a + B.b + C.c = 0, т.е. $\vec{n}.\vec{p} = 0$ ⇔ $\vec{n}$ ⇔ $\vec{p}$ за всеки вектор $\vec{p}$, компланарен с ↦и тъй като $\vec{n}$ е ненулев ⇔ $\vec{n}$ ⇔ ↦ Ненулевите вектори, които са перпендикулярни на ↦наричаме **нормални вектори** на равнината ↦ Така коефициентите пред x, y и z във всяко общо уравнение на равнината ↦са координати на нормален вектор на ↦ Онова общо уравнение на равнината ↦ в което коефициентите пред x, y и z са координати на единичен нормален вектор на ↦ се нарича **нормално уравнение** на равнината ↦ С други думи, уравнението A.x + B.y + C.z + D = 0 на равнината ↦е нормално ⇔ $A^2 + B^2 + C^2 = 1$.

Нека $\ell.A.x + \ell.B.y + \ell.C.z + \ell.D = 0$, $\ell \neq 0$ е произволно уравнение на равнината $\mapsto$ Тогава то е нормално $\Leftrightarrow$ $(\ell.A)^2 + (\ell.B)^2 + (\ell.C)^2 = 1$ $\Leftrightarrow$ $\ell^2 = \frac{1}{A^2+B^2+C^2}$ $\Leftrightarrow$ $\ell = \pm\frac{1}{\sqrt{A^2+B^2+C^2}}$. Така равнината $\mapsto$има точно две нормални уравнения:

$$\frac{A}{\sqrt{A^2+B^2+C^2}}.x + \frac{B}{\sqrt{A^2+B^2+C^2}}.y + \frac{C}{\sqrt{A^2+B^2+C^2}}.z + \frac{D}{\sqrt{A^2+B^2+C^2}} = 0 \text{ и}$$

$$-\frac{A}{\sqrt{A^2+B^2+C^2}}.x - \frac{B}{\sqrt{A^2+B^2+C^2}}.y - \frac{C}{\sqrt{A^2+B^2+C^2}}.z - \frac{D}{\sqrt{A^2+B^2+C^2}} = 0.$$

Нека равнината $\mapsto$е зададена с нормалното си уравнение

$$\frac{A}{\sqrt{A^2+B^2+C^2}}.x + \frac{B}{\sqrt{A^2+B^2+C^2}}.y + \frac{C}{\sqrt{A^2+B^2+C^2}}.z + \frac{D}{\sqrt{A^2+B^2+C^2}} = 0,$$

векторът $\vec{n}\left(\frac{A}{\sqrt{A^2+B^2+C^2}}, \frac{B}{\sqrt{A^2+B^2+C^2}}, \frac{C}{\sqrt{A^2+B^2+C^2}}\right)$ е единичен нормален вектор на $\mapsto$

Нека $P(x_0, y_0, z_0)$ е произволна точка. Нека $P_1(x_1, y_1, z_1)$ е ортогоналната проекция на P върху равнината $\mapsto$ Тогава разстоянието δ от P до $\mapsto$е точно числото $|\overrightarrow{P_1P}|$.

Имаме $\vec{n}.\overrightarrow{P_1P} = |\vec{n}|.|\overrightarrow{P_1P}|.\cos\sphericalangle(\vec{n}, \overrightarrow{P_1P}) = \pm|\overrightarrow{P_1P}|$. От друга страна,

$$\vec{n}.\overrightarrow{P_1P} = \frac{A}{\sqrt{A^2+B^2+C^2}}.(x_0 - x_1) + \frac{B}{\sqrt{A^2+B^2+C^2}}.(y_0 - y_1) +$$

$$+ \frac{C}{\sqrt{A^2+B^2+C^2}}.(z_0 - z_1).$$ При това, P_1 лежи в $\mapsto$

$$-\frac{A}{\sqrt{A^2+B^2+C^2}}.x_1 - \frac{B}{\sqrt{A^2+B^2+C^2}}.y_1 - \frac{C}{\sqrt{A^2+B^2+C^2}}.z_1 = \frac{D}{\sqrt{A^2+B^2+C^2}} \Leftrightarrow$$

$$\Leftrightarrow \vec{n}.\overrightarrow{P_1P} = \frac{A}{\sqrt{A^2+B^2+C^2}}.x_0 + \frac{B}{\sqrt{A^2+B^2+C^2}}.y_0 + \frac{C}{\sqrt{A^2+B^2+C^2}}.z_0 +$$

$$+ \frac{D}{\sqrt{A^2+B^2+C^2}}.$$

Така разстоянието от точката P до равнината $\mapsto$е

$$\delta = |\vec{n}.\overrightarrow{P_1P}| = \frac{|A.x_0 + B.y_0 + C.z_0 + D|}{\sqrt{A^2+B^2+C^2}}.$$

24. Линейни обикновени диференциални уравнения. Уравнения с постоянни коефициенти.

Разглеждаме уравнение от вида

$$y^{(n)} + a_1(x).y^{(n-1)} + \ldots + a_{n-1}(x).y' + a_n(x).y = f(x),$$

където функциите $a_1(x), \ldots, a_n(x), f(x)$ са дефинирани и непрекъснати в интервал $I \subset \mathbb{R}$ и приемат комплексни стойности.

Това уравнение наричаме **линейно обикновено диференциално уравнение от n-ти ред**.
Под **решение** на уравнението разбираме комплекснозначна функция $\varphi(x)$, дефинирана в целия интервал I и притежаваща непрекъснати производни до n-ти ред включително и такава, че $\varphi^{(n)}(x) + a_1(x).\varphi^{(n-1)}(x) + \ldots + a_{n-1}(x).\varphi'(x) + a_n(x).\varphi(x) = f(x)$ за всяко $x \in I$.
Уравнението се нарича **хомогенно**, ако $f(x) = 0$ за всяко $x \in I$ и **нехомогенно**, в противен случай.
Нека $x_0 \in I$. Ако освен уравнението са зададени и n начални условия (n е редът на уравнението): $\begin{matrix} y(x_0) = y_0 \\ y'(x_0) = y_1 \\ \ldots \\ y^{(n-1)}(x_0) = y_{n-1} \end{matrix}$, където y_i са комплексни числа, казваме че е зададена **задача на Коши**.
Под решение на задачата на Коши ще разбираме решение на уравнението, което удоволетворява началните условия.

Теорема (за съществуване и единственост): При направените предположения съществува единствено решение на задачата на Коши, което е дефинирано в целия интервал I.

Теорема: Ако $y_1(x)$ и $y_2(x)$ са две решения на хомогенното уравнение, то всяка тяхна линейна комбинация
$y(x) = \ell.y_1(x) + \mp.y_2(x)$, $\ell, \mp \in \mathbb{C}$ също е решение на това уравнение.
Доказателство: За всяко $x \in I$ имаме:

$$(\ell.y_1 + \mp.y_2)^{(n)}(x) + a_1(x).(\ell.y_1 + \mp.y_2)^{(n-1)}(x) + \ldots + a_{n-1}(x).(\ell.y_1 + \mp.y_2)'(x) +$$
$$+ a_n(x).(\ell.y_1 + \mp.y_2)(x) = (\ell.y_1)^{(n)}(x) + a_1(x).(\ell.y_1)^{(n-1)}(x) + \ldots + a_{n-1}(x).(\ell.y_1)'(x) +$$
$$+ a_n(x).(\ell.y_1)(x) + (\mp.y_2)^{(n)}(x) + a_1(x).(\mp.y_2)^{(n-1)}(x) + \ldots + a_{n-1}(x).(\mp.y_2)'(x) +$$
$$+ a_n(x).(\mp.y_2)(x) = \ell.\left(y_1^{(n)}(x) + a_1(x).y_1^{(n-1)}(x) + \ldots + a_{n-1}(x).y_1'(x) + \right.$$
$$\left. + a_n(x).y_1(x)\right) + \mp.\left(y_2^{(n)}(x) + a_1(x).y_2^{(n-1)}(x) + \ldots + a_{n-1}(x).y_2'(x) + a_n(x).y_2(x)\right) =$$

$= \ell.0 + \mp.0 = 0$. И така $\ell.y_1 + \mp.y_2$ е решение на хомогенното уравнение.

Тази теорема ни позволява да твърдим, че съвкупността от решенията на хомогенното уравнение е линейно пространство над $\mathbb{C}$.

Нека $y_1(x), y_2(x), \ldots, y_n(x)$ са произволни функции, дефинирани и притежаващи непрекъснати производни до ред (n-1), включително, в интервала I и приемащи комплексни стойности. Дефинираме **детерминанта на Вронски** за тези функции:

$$W(x) = \begin{vmatrix} y_1(x) & y_2(x) & \dots & y_n(x) \\ y_1'(x) & y_2'(x) & \dots & y_n'(x) \\ \dots & \dots & \dots & \dots \\ y_1^{(n-1)}(x) & y_2^{(n-1)}(x) & \dots & y_n^{(n-1)}(x) \end{vmatrix}.$$

Теорема: Нека $y_1(x), y_2(x), \dots, y_n(x)$ са произволни решения на хомогенното уравнение. Следните три условия са еквивалентни:

1. Функциите $y_1(x), y_2(x), \dots, y_n(x)$ са линейно независими в I.
2. $W(x_0) \neq 0$ за някое $x_0 \in I$.
3. $W(x) \neq 0$ за всяко $x \in I$.

Доказателство: Ще проведем доказателството по следната схема от импликации: $3 \Rightarrow 2 \Rightarrow 1 \Rightarrow 3$.

Импликацията $3 \Rightarrow 2$ е очевидна.

Ще покажем, че е в сила $2 \Rightarrow 1$.

Да допуснем, че функциите $y_1(x), y_2(x), \dots, y_n(x)$ са линейно зависими в I. Тогава съществува ненулева n-торка от комплексни константи $c_1, c_2, \dots, c_n$, такива че $\sum_{k=1}^{n} c_k.y_k(x) = 0$ за всяко $x \in I$.

Като диференцираме това тъждество n-1 пъти последователно получаваме:

$c_1.y_1(x) + c_2.y_2(x) + \dots + c_n.y_n(x) = 0$

$c_1.y_1'(x) + c_2.y_2'(x) + \dots + c_n.y_n'(x) = 0$

…

$c_1.y_1^{(n-1)}(x) + c_2.y_2^{(n-1)}(x) + \dots + c_1.y_n^{(n-1)}(x) = 0$

При $x = x_0$ получаваме линейна хомогенна система с n уравнения и n неизвестни и тя има ненулево решение $(c_1, \dots, c_n)$. При това положение детерминантата на системата, която е точно $W(x_0) = 0$, което е противоречие.

Сега ще покажем, че е в сила $1 \Rightarrow 3$.

Да допуснем противното, т.е. нека съществува $x_0 \in I$, такова че $W(x_0) = 0$. Тогава горната линейна хомогенна система има ненулево решение $c_1, c_2, \dots, c_n$. Да разгледаме функцията $y(x) = \sum_{k=1}^{n} c_k.y_k(x)$. Тя очевидно е решение на хомогенното уравнение, тъй като е линейна комбинация на решения.

Също така, от горните равенства получаваме

$y(x_0) = y'(x_0) = \dots = y^{(n-1)}(x_0) = 0$. Така $y(x)$ е решение на задачата на Коши за хомогенното уравнение с начални условия 0, 0, …, 0. От друга страна, константата 0 очевидно също е решение на тази задача на Коши. Сега от теоремата за единственост получаваме, че $y(x) = 0$ за всяко $x \in I$ и тогава функциите $y_1(x), \dots, y_n(x)$ са линейно зависими в I, което е противоречие.

Системата от решения $y_1(x), y_2(x), \dots, y_n(x)$ на хомогенното уравнение се нарича **фундаментална система решения**, ако функциите $y_1(x), y_2(x), \dots, y_n(x)$ са линейно независими в I.

Теорема: Съществуват безброй много фундаментални системи решения на хомогенното уравнение.
Доказателство:

Нека $\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$ е произволна ненулева детерминанта,

където a_{ij} са комплексни числа. Фиксираме $x_0 \in I$.
Нека функциите $y_j(x)$, $j = 1, 2, \dots, n$ са решения на задачата на Коши с начални условия: $y_j^{(k-1)}(x_0) = a_{kj}$, $k = 1, 2, \dots, n$. Такива решения има по теоремата за съществуване. При това детерминантата на Вронски $W(x)$ за тази система решения, изчислена в точката x_0, е точно дадената детерминантата, която е различна от 0. От горната теорема системата $\{y_j(x)\}$ е линейно независима в I, т.е. $y_1(x), y_2(x), \dots, y_n(x)$ образуват фундаментална система решения на хомогенното уравнение. Тъй като числата a_{ij} могат да се изберат по безброй много начини, съществуват безброй много фундаментални системи решения на хомогенното уравнение.

Теорема: Линейното пространство от решенията на хомогенното уравнение има размерност n и всяка фундаментална система решения представлява базис на това пространство.
Доказателство: Нека $y_1(x), y_2(x), \dots, y_n(x)$ е фундаментална система решения на хомогенното уравнение. Нека $y(x)$ е произволно решение на хомогенното уравнение. Фиксираме $x_0 \in I$. Разглеждаме следната линейна система от уравнения:

$c_1.y_1(x_0) + c_2.y_2(x_0) + \dots + c_n.y_n(x_0) = y(x_0)$
$c_1.y_1'(x_0) + c_2.y_2'(x_0) + \dots + c_n.y_n'(x_0) = y'(x_0)$
…
$c_1.y_1^{(n-1)}(x_0) + c_2.y_2^{(n-1)}(x_0) + \dots + c_1.y_n^{(n-1)}(x_0) = y^{(n-1)}(x_0)$.

Детерминантата на тази система е $W(x_0)$, при това $W(x_0) \neq 0$, тъй като $\{y_j(x)\}$ са линейно независими в I. Така системата има (единствено) решение $c_1, c_2, \dots, c_n$. Разглеждаме функцията $z(x) = \sum_{k=1}^{n} c_k.y_k(x)$, очевидно $z(x)$ е решение на хомогенното уравнение. Горните равенства ни дават, че $z(x_0) = y(x_0)$, $z'(x_0) = y'(x_0)$, …, $z^{(n-1)}(x_0) = y^{(n-1)}(x_0)$. Така $z(x)$ и $y(x)$ са решения на една и съща задача на Коши и от теоремата за единственост получаваме, че $z(x) = y(x)$ за всяко $x \in I$, т.е. $y(x) = \sum_{k=1}^{n} c_k.y_k(x)$ за всяко $x \in I$. Така функциите $\{y_j(x)\}$ пораждат пространството от решенията на хомогенното уравнение и освен това са линейно независими в I $\Rightarrow \{y_j(x)\}$ е базис на пространството от решенията на хомогенното уравнение $\Rightarrow$ това пространство има размерност n.

Следователно, ако $y_1(x), y_2(x), \ldots, y_n(x)$ е фундаментална система решения на хомогенното уравнение, то всички решения се дават с формулата $\sum_{k=1}^{n} c_k.y_k(x)$, където c_j са произволни комплексни константи.

Отново разглеждаме нехомогенното уравнение
$y^{(n)} + a_1(x).y^{(n-1)} + \ldots + a_{n-1}(x).y' + a_n(x).y = f(x)$,
със съответните предположения за $a_j(x)$, $f(x)$.

Теорема: Нека $y_0(x)$ е едно решение на нехомогенното уравнение. Нека $y_1(x), y_2(x), \ldots, y_n(x)$ е фундаментална система решения на хомогенното уравнение, получено от нехомогенното със замяна на $f(x)$ с 0. Тогава всички решения на нехомогенното уравнения се дават с формулата $y_0(x) + \sum_{k=1}^{n} c_k.y_k(x)$, където c_j са произволни комплексни константи.
Доказателство: Нека $y(x)$ е произволно решение на нехомогенното уравнение. Да разгледаме функцията $z(x) = y(x) - y_0(x)$, $x \in I$.
Имаме $z^{(n)}(x) + a_1(x).z^{(n-1)}(x) + \ldots + a_{n-1}(x).z'(x) + a_n(x).z(x) =$
$= y^{(n)}(x) + a_1(x).y^{(n-1)}(x) + \ldots + a_{n-1}(x).y'(x) + a_n(x).y(x) -$
$- \left(y_0^{(n)}(x) + a_1(x).y_0^{(n-1)}(x) + \ldots + a_{n-1}(x).y_0'(x) + a_n(x).y_0(x)\right) = f(x) - f(x) = 0.$

Така $z(x)$ е решение на хомогенното уравнение $\Rightarrow$ съществуват константи $c_1, c_2, \ldots, c_n$, такива че $z(x) = \sum_{k=1}^{n} c_k.y_k(x) \Rightarrow$

$\Rightarrow y(x) = y_0(x) + \sum_{k=1}^{n} c_k.y_k(x)$. Обратно, очевидно всяка функция от този вид е решение на нехомогенното уравнение.

Сега разглеждаме **хомогенното уравнение от n-ти ред с постоянни коефициенти**
$y^{(n)} + a_1.y^{(n-1)} + \ldots + a_{n-1}.y' + a_n.y = 0$, където в общия случай a_j са комплексни числа.
Полиномът от n-та степен $\ell^n + a_1.\ell^{n-1} + \ldots + a_{n-1}.\ell + a_n$ се нарича **характеристичен полином** на горното уравнение.
Нека ℓ_0 е негов корен. Да разгледаме функцията $e^{\ell_0.x}$.
Имаме $(e^{\ell_0.x})^{(n)} + a_1.(e^{\ell_0.x})^{(n-1)} + \ldots + a_{n-1}.(e^{\ell_0.x})' + a_n.(e^{\ell_0.x}) =$
$\Rightarrow \ell_0^n.(e^{\ell_0.x}) + a_1.\ell_0^{n-1}.(e^{\ell_0.x}) + \ldots + a_{n-1}.\ell_0.(e^{\ell_0.x}) + a_n.(e^{\ell_0.x}) =$
$\Rightarrow (e^{\ell_0.x}).\left(\ell_0^n + a_1.\ell_0^{n-1} + \ldots + a_{n-1}.\ell_0 + a_n\right) \Rightarrow 0$.

Така $e^{\ell_0.x}$ е решение на уравнението.

От основната теорема на алгебрата, характеристичният полином има n комплексни корена $\ell_1, \ell_2, \ldots, \ell_n$.

Да предположим, че всички тези корени са прости, т.е. $\ell_i \neq \ell_j$ при $i \neq j$. Тогава функциите $e^{\ell_1.x}, e^{\ell_2.x}, \ldots, e^{\ell_n.x}$ образуват фундаментална система решения на уравнението. Действително, вече показхме, че те са решения. Остава да покажем, че са линейно независими в $\mathbb{R}$. Образуваме детерминантата на Вронски

W (x) за тези решения, W (x) = $\begin{vmatrix} e^{\ell_1.x} & e^{\ell_2.x} & \ldots & e^{\ell_n.x} \\ \ell_1.e^{\ell_1.x} & \ell_2.e^{\ell_2.x} & \ldots & \ell_n.e^{\ell_n.x} \\ \ldots & \ldots & \ldots & \ldots \\ \ell_1^{n-1}.e^{\ell_1.x} & \ell_2^{n-1}.e^{\ell_2.x} & \ldots & \ell_n^{n-1}.e^{\ell_n.x} \end{vmatrix} \neq$

$= e^{\ell_1.x}.e^{\ell_2.x}.\ldots.e^{\ell_n.x}.\begin{vmatrix} 1 & 1 & \ldots & 1 \\ \ell_1 & \ell_2 & \ldots & \ell_n \\ \ldots & \ldots & \ldots & \ldots \\ \ell_1^{n-1} & \ell_2^{n-1} & \ldots & \ell_n^{n-1} \end{vmatrix} \neq 0$, тъй като детерминантата

на Вандермонд е равна на 0 тогава и само тогава, когато $\ell_i = \ell_j$ за някои $i \neq j \in \{1, 2, \ldots, n\}$. Така общото решение на хомогенното уравнение с постоянни коефициенти се дава с формулата $C_1.\ e^{\ell_1.x} + C_2.\ e^{\ell_2.x} + \ldots + C_n.\ e^{\ell_n.x}$, където $C_1, C_2, \ldots, C_n$ са произволни комплексни константи.

Сега да разгледаме общият случай, при който е възможно някои от корените на характеристичния полином да са кратни.

Нека $\ell^n + a_1.\ell^{n-1} + \ldots + a_{n-1}.\ell + a_n = \prod_{j=1}^{m} (\ell - \ell_j)^{r_j}$, т.е.

различните корени на характеристичния полином са $\ell_1, \ell_2, \ldots, \ell_m$ и техните кратности са съответно $r_1, r_2, \ldots, r_m$, $r_1 + r_2 + \ldots + r_m = n$.

Теорема: Нека $P_1 (x), P_2 (x), \ldots, P_k (x)$ са полиноми, $\ell_1, \ell_2, \ldots, \ell_k$ са различни комплексни числа, $I \subset \mathbb{R}$ е интервал. Тогава равенството $P_1(x).e^{\ell_1.x} + P_2(x).e^{\ell_2.x} + \ldots + P_k(x).e^{\ell_k.x} = 0$ за всяко $x \in I$ е възможно само тогава, когато $P_1 (x) = P_2 (x) = \ldots = P_k (x) = 0$ за всяко $x \in I$.

Доказателство: Провеждаме индукция по k.

База: При k = 1 имаме: $P_1(x).e^{\ell_1.x} = 0$ и тъй като $e^{\ell_1.x} \neq 0$ за всяко $x \in \mathbb{R}$, то $P_1 (x) = 0$ за всяко $x \in I$.

Предположение: Нека твърдението е изпълнено за числото k, $k \geq 1$.

Стъпка: Нека за всяко $x \in I$ имаме равенството $P_1(x).e^{\ell_1.x} + P_2(x).e^{\ell_2.x} + \ldots + P_k(x).e^{\ell_k.x} + P_{k+1}(x).e^{\ell_{k+1}.x} = 0$. Тогава за всяко $x \in I$, $P_1(x).e^{-\ell_1.x} + P_2(x).e^{-\ell_2.x} + \ldots + P_k(x).e^{-\ell_k.x} + P_{k+1}(x) = 0$.

За произволен полином P (x) и $\ell \in \mathbb{R}$, $\ell \neq 0$, имаме:

$(P (x).e^{\ell.x})' = P'(x) + \ell.P (x).e^{\ell.x} = Q (x).e^{\ell.x}$, където

deg Q (x) = deg P (x), тъй като deg P′ (x) < deg P (x) и $\ell \neq 0$.

Диференцираме равенството от по-горе $\deg P_{k+1} (x)+1$ пъти.

Ясно е, че получаваме:

$Q_1(x).e^{(\ell_1-\ell_{k+1}).x} + Q_2(x).e^{(\ell_2-\ell_{k+1}).x} + \ldots + Q_k(x).e^{(\ell_k-\ell_{k+1}).x} = 0$, където
$\deg Q_j(x) = \deg P_j(x)$ за $j = 1, 2, \ldots, k$, тъй като $\ell_j \neq \ell_{k+1}$.
Използваме индукционното предположение и получаваме, че
$Q_1(x) = Q_2(x) = \ldots = Q_k(x) = 0$ за всяко $x \in I$, но $\deg Q_j(x) = \deg P_j(x)$ за $j = 1, 2, \ldots, k \Rightarrow P_1(x) = P_2(x) = \ldots = P_k(x) = 0$ за всяко $x \in I$.
След заместване в първоначалното равенство получаваме $P_{k+1}(x).e^{\ell_{k+1}.x} = 0$ за всяко $x \in I$ и тъй като $e^{\ell_{k+1}.x} \neq 0$ за всяко $x \in \mathbb{R}$, то $P_{k+1}(x) = 0$ за всяко $x \in I$.

Теорема: Нека ℓ_1 е корен с кратност $r_1 \geq 1$ на характеристичния полином на уравнението. Тогава $e^{\ell_1.x}$, $x.e^{\ell_1.x}$, ..., $x^{r_1-1}.e^{\ell_1.x}$ са решения на уравнението.
Доказателство: Да означим $L(y) = y^{(n)} + a_1.y^{(n-1)} + \ldots + a_{n-1}.y' + a_n.y$, $P(\ell) = \ell^n + a_1.\ell^{n-1} + \ldots + a_{n-1}.\ell + a_n$. Вече знаем, че $L(e^{\ell.x}) = P(\ell).e^{\ell.x}$ за всяко $\ell \in \mathbb{R}$. Диференцираме последното равенство по ℓ и получаваме: $L(e^{\ell.x})'_\ell = P'(\ell).e^{\ell.x} + P(\ell).e^{\ell.x}.x = L(x.e^{\ell.x}) + P'(\ell).e^{\ell.x}.x$.
Използвали сме, че

$$L(e^{\ell.x})'_\ell = (e^{\ell.x})^{(n)} + a_1.(e^{\ell.x})^{(n-1)} + \ldots + a_{n-1}.(e^{\ell.x})' + a_n.(e^{\ell.x})'_\ell =$$

$$= (e^{\ell.x})'^{(n)}_\ell + a_1.(e^{\ell.x})'^{(n-1)}_\ell + \ldots + a_{n-1}.(e^{\ell.x})'^{'}_\ell + a_n.(e^{\ell.x})'_\ell =$$

$$= (e^{\ell.x})'^{(n)}_\ell + a_1.(e^{\ell.x})'^{(n-1)}_\ell + \ldots + a_{n-1}.(e^{\ell.x})'_\ell + a_n.(e^{\ell.x})'_\ell =$$

$= L((e^{\ell.x})'_\ell) = L(x.e^{\ell.x})$. Във втората стъпка използвахме, че коефициентите на уравнението не зависят от ℓ, а в третата стъпка използвахме, че е в сила теоремата за смесените производни (независимост от последователността, в която се диференцира).
След k диференцирания по ℓ получаваме:
$L(x^k.e^{\ell.x}) = P'(\ell).e^{\ell.x}.x^k + \ldots + P^{(k)}(\ell).e^{\ell.x}$. Сега използваме формулата на Лайбниц за диференциране на произведение и получаваме:

$$L(x^k.e^{\ell.x}) = \sum_{m=0}^{k} \binom{k}{m}.P^{(m)}(\ell).e^{\ell.x}.x^{k-m} + \ldots + \sum_{m=0}^{k} \binom{k}{m}.P^{(m)}(\ell).x^{k-m}.e^{\ell.x}$$

Имаме $P^{(m)}(\ell_1) = 0$ за $m = 0, 1, \ldots, r_1 - 1$, тъй като ℓ_1 е r_1-кратен корен на $P(\ell)$. Така в горното равенство при $k = 0, 1, \ldots, r_1 - 1$ и $\ell = \ell_1$ получаваме $L(x^k.e^{\ell_1.x}) = 0$, т.е. $e^{\ell_1.x}$, $x.e^{\ell_1.x}$, ..., $x^{r_1-1}.e^{\ell_1.x}$ са решения на уравнението.

И така в случая на кратни корени можем да построим n решения:
за корена ℓ_1 - $e^{\ell_1.x}$, $x.e^{\ell_1.x}$, ..., $x^{r_1-1}.e^{\ell_1.x}$,
за корена ℓ_2 - $e^{\ell_2.x}$, $x.e^{\ell_2.x}$, ..., $x^{r_2-1}.e^{\ell_2.x}$,
...
за корена ℓ_m - $e^{\ell_m.x}$, $x.e^{\ell_m.x}$, ..., $x^{r_m-1}.e^{\ell_m.x}$.

Ще покажем, че тези решения образуват фундаментална система от решения на уравнението.
Да допуснем, че съществуват константи $C_0^1, C_1^1, \ldots, C_{r_1-1}^1, C_0^2, C_1^2, \ldots, C_{r_2-1}^2, \ldots, C_0^m, C_1^m, \ldots, C_{r_m-1}^m \in \mathbb{C}$, такива че

$\sum_{j=1}^{m} C_0^j.e^{\ell_j.x} + C_1^j.x.e^{\ell_j.x} + \ldots + C_{r_j-1}^j.x^{r_j-1}.e^{\ell_j.x} = 0$ за всяко $x \in \mathbb{R}$.

Тогава от по-предната теорема директно получаваме, че за всяко $j = 1, 2, \ldots, m$ имаме: $C_0^j + C_1^j.x + \ldots + C_{r_j-1}^j.x^{r_j-1} = 0$ за всяко $x \in \mathbb{R}$, т.е. всички константи се анулират и така решенията действително са линейно независими. Накрая общото решение на хомогенното уравнение с постоянни коефициенти се записва във вида

$\sum_{j=1}^{m} C_0^j.e^{\ell_j.x} + C_1^j.x.e^{\ell_j.x} + \ldots + C_{r_j-1}^j.x^{r_j-1}.e^{\ell_j.x}$, където C_k^j са произволни комплексни константи.

Сега да предположим, че коефициентите в уравнението са реални числа. Ще покажем, че в този случай можем да изберем фундаментална система решения от реалнозначни функции.
Тъй като коефициентите на характеристичния полином са реални, ако ℓ е корен на характеристичния полином, то $\overline{\ell}$ също е негов корен, при това със същата кратност.
Оттук директно получаваме, че за всяко $j = 1, 2, \ldots, m$ ако $e^{\ell_j.x}$, $x.e^{\ell_j.x}$, ..., $x^{r_j-1}.e^{\ell_j.x}$ са решения, които присъстват във фундаменталната система от по-горе, то в нея задължително присъстват и решенията $e^{\overline{\ell}_j.x}$, $x.e^{\overline{\ell}_j.x}$, ..., $x^{r_j-1}.e^{\overline{\ell}_j.x}$.
Тъй като решенията образуват линейно пространство, то за всеки $j = 1, 2, \ldots, m$, $k = 0, 1, \ldots, r_j - 1$, $\frac{1}{2}(x^k.e^{\ell_j.x} + x^k.e^{\overline{\ell}_j.x}) = x^k.e^{a_j.x}.\cos b_j x$ и $\frac{1}{2.i}(x^k.e^{\ell_j.x} - x^k.e^{\overline{\ell}_j.x}) = x^k.e^{a_j.x}.\sin b_j x$, където $\ell_j = a_j + i.b_j$, са решения на уравнението, но те вече са реалнозначни.
Сега за всеки $j = 1, 2, \ldots, m$, $k = 0, 1, \ldots, r_j - 1$ заменяме решенията $x^k.e^{\ell_j.x}$, $x^k.e^{\overline{\ell}_j.x}$ с решенията $x^k.e^{a_j.x}.\cos b_j x$, $x^k.e^{a_j.x}.\sin b_j x$. (Тук се обхваща и случаят когато $\ell_j \in \mathbb{R}$, т.е. $\ell_j = a_j$ и $b_j = 0$ – тогава просто нищо не заменяме). Ясно е, че отново получаваме фундаментална система решения, тъй като получената система решения е линейно еквивалентна на първоначалната фундаментална система и се състои от n на брой решения, колкото е размерността на пространството от решенията. Така общото реално решение на хомогенното линейно уравнение с постоянни реални коефициенти има вида $\sum_{j=1}^{s} C_0^j.e^{\ell_j.x} + C_1^j.x.e^{\ell_j.x} + \ldots + C_{r_j-1}^j.x^{r_j-1}.e^{\ell_j.x}$

$$\sum_{j=s+1}^{\frac{m+s}{2}} C_0^j.e^{a_j.x}.\cos b_j.x + C_0^j.x.e^{a_j.x}.\cos b_j.x + \ldots + C_{r_j-1}^j.x^{r_j-1}.e^{a_j.x}.\cos b_j.x$$

$$\sum_{j=s+1}^{\frac{m+s}{2}} D_0^j.e^{a_j.x}.\sin b_j.x + D_0^j.x.e^{a_j.x}.\sin b_j.x + \ldots + D_{r_j-1}^j.x^{r_j-1}.e^{a_j.x}.\sin b_j.x,$$

където $\ell_1, \ell_2, \ldots, \ell_m$ са различните корени на характеристичния полином с кратности съответно $r_1, r_2, \ldots, r_m$, като първите s от тях $\ell_1, \ell_2, \ldots, \ell_s$ са реални, а останалите са комплексни и нереални

$\ell_{s+1}, \ell_{s+2}, \ldots, \ell_{\frac{m+s}{2}}, \ell_{\frac{m+s}{2}+1}, \ell_{\frac{m+s}{2}+2}, \ldots, \ell_m$, $\ell_{\frac{m+s}{2}+i} = \overline{\ell_{s+i}}$, $i = 1, 2, \ldots, \frac{m-s}{2}$,

$\ell_j = a_j + i.b_j$, $j = s+1, s+2, \ldots, \frac{m+s}{2}$ и $C_{\ldots}^j$, $D_{\ldots}^j$ са произволни реални константи.

25. Диференчни методи за задачата на Коши за обикновено диференциално уравнение от първи ред.

Отпада.

26. Итерационни методи за решаване на нелинейни уравнения.

Голяма част от методите за приближено пресмятане на корените на уравнение са **итерационни**. При тях се тръгва от някакво начално приближение x_0 и след това с извършването на определена процедура (итерация) се намира следващото приближение x_1. Въз основа на x_0 и x_1 се определя x_2 и т.н. По този начин се построява една редица $x_0, x_1, \ldots, x_n, \ldots$, която клони към корена α на уравнението $f(x) = 0$.
Нека $f(x)$ е функция, определена в интервала $[a, b]$. Изследваме уравнението $f(x) = 0$. Удобно е да запишем това уравнение във вида $x = \varphi(x)$. Това може да стане, например, като добавим x към двете страни на $f(x) = 0$ или направим друго еквивалентно преобразуване. Така α е корен на уравнението $f(x) = 0$, т.е. $f(\alpha) = 0$ тогава и само тогава, когато $\alpha = \varphi(\alpha)$, т.е. α е **неподвижна точка** на φ. Да изберем точка $x_0 \in [a, b]$ и да построим редицата $x_0, x_1, \ldots, x_n, \ldots$ по правилото $x_{n+1} = \varphi(x_n)$, $n = 0, 1, \ldots$.
Целта е редицата $\{x_n\}$ да клони към корена α на уравнението $x = \varphi(x)$. Ясно е, че построената редица няма такова свойство за произволна функция φ. Сега ще видим какви условия за φ биха гарантирали такава сходимост.

Твърдение: Ако φ е изображение на интервала $[a, b]$ в себе си, то при произволно начално приближение $x_0 \in [a, b]$, всички точки от редицата $\{x_n\}$ принадлежат също на $[a, b]$.

Доказателство: Тривиална индукция по n. При n = 0, $x_0 \in [a, b]$. Ако $x_n \in [a, b]$, то $x_{n+1} = \varphi(x_n) \in [a, b]$.

Ние търсим корена на уравнението x = φ (x), т.е. неподвижна точка $\xi \in [a, b]$ на φ. Следващото просто условие върху φ гарантира поне една неподвижна точка.

Твърдение: Ако φ е непрекъснато изображение на интервала [a, b] в себе си, то φ има поне една неподвижна точка.
Доказателство: Ако φ (a) = a или φ (b) = b, то φ има неподвижна точка. Нека сега φ (a) ≠ a и φ (b) ≠ b. Тъй като φ : [a, b] → [a, b], имаме a < φ (a) и φ (b) < b. Разглеждаме функцията r (x) = x - φ (x). Очевидно тя е дефинирана и непрекъсната в [a, b].
Имаме r (a) = a - φ (a) < 0 и r (b) = b - φ (b) > 0. От теоремата на Болцано-Коши съществува точка $\xi \in (a, b)$, такава че r (ξ) = 0 ⇔ ξ = φ (ξ), т.е. ξ е неподвижна точка на φ.

Остава да видим какви условия за φ ще гарантират сходимост на редицата $\{x_n\}$ към неподвижната точка ξ.
Казваме, че функцията g удоволетворява **условието на Липшиц** с константа q в интервала [a, b], ако $|g(x) - g(y)| \le q.|x - y|$ за всеки $x, y \in [a, b]$.

Теорема: Нека φ е непрекъснато изображение на [a, b] в себе си, което удоволетворява условието на Липшиц с константа q < 1. Тогава уравнението x = φ (x) има единствен корен ξ в [a, b] и редицата $\{x_n\}$ ($x_0 \in [a, b]$ е произволно, $x_{n+1} = \varphi(x_n)$, n = 0, 1, …) клони към ξ при $n \to \infty$. При това, $|x_n - \xi| \le (b - a).q^n$ за всяко n = 0, 1, ….
Доказателство: От предното твърдение, φ има поне една неподвижна точка в [a, b]. Да допуснем, че ξ_1 и ξ_2 са две различни неподвижни точки на φ в [a, b], т.е. $\xi_1 = \varphi(\xi_1)$, $\xi_2 = \varphi(\xi_2)$ и $\xi_1, \xi_2 \in [a, b]$, $\xi_1 \ne \xi_2$. Имаме $|\xi_1 - \xi_2| = |\varphi(\xi_1) - \varphi(\xi_2)| \le q.|\xi_1 - \xi_2| < |\xi_1 - \xi_2|$ - противоречие. Използвали сме, че φ удоволетворява условието на Липшиц с константата q < 1 и, че $\xi_1 \ne \xi_2$. Така φ има единствена неподвижна точка ξ в [a, b]. С индукция по n ще покажем горната оценка. При n = 0, $|x_0 - \xi| \le (b - a).q^0 = b - a$, тъй като $x_0 \in [a, b]$ и $\xi \in [a, b]$. Нека $|x_n - \xi| \le (b - a).q^n$. Тогава $|x_{n+1} - \xi| = |\varphi(x_n) - \varphi(\xi)| \le q.|x_n - \xi| \le q.(b - a).q^n = (b - a).q^{n+1}$.
Така $|x_n - \xi| \le (b - a).q^n$ за всяко n = 0, 1, … $x_n \to \xi$ при $n \to \infty$.

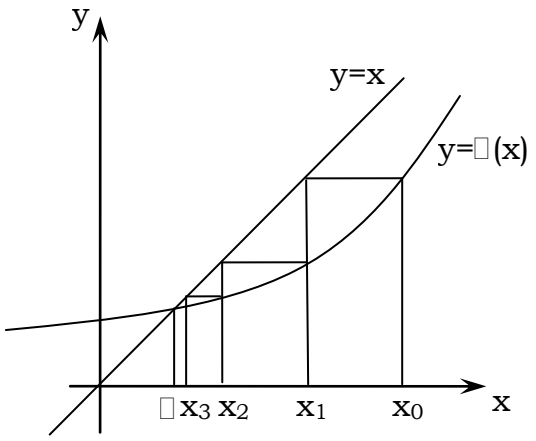
Изображение φ, което удоволетворява условието на Липшиц с константа q < 1 се нарича **свиващо** изображение. При него разстоянието между образите φ (x) и φ (y) е строго по-малко от разстоянието между праобразите x и y, т.е. φ "свива" разстоянията.

Твърдение: Нека φ е диференцируема функция в [a, b] и нека $|\varphi'(x)| \leq q < 1$ за всяко $x \in [a, b]$. Тогава φ е свиващо изображение.
Доказателство: Нека x, y $\in$ [a, b]. Тогава от теоремата за крайните нараствания на Лагранж, $\varphi(x) - \varphi(y) = \varphi'(\hbar).(x - y)$, $\hbar \in [a, b]$ $\Rightarrow$ $|\varphi(x) - \varphi(y)| = |\varphi'(\hbar)|.|x - y| \leq q.|x - y|$ и q < 1.

Нека уравнението x = φ(x) има корен ξ в интервала [a, b]. Казваме, че **итерационният процес**, породен от φ **е сходящ** в [a, b], ако за всяко начално приближение $x_0 \in [a, b]$, редицата $\{x_n\}$, построена по формулата $x_{n+1} = \varphi(x_n)$, n = 0, 1, …е сходяща към корена ξ. Горната теорема, показва сходимостта на итерационните процеси, породени от свиващи изображения. Сега ще приведем една по-слаба форма на теоремата.

Следствие: Нека ξ е корен на уравнението x = φ(x). Да предположим, че φ има непрекъсната производна в околност U на ξ и $|\varphi'(\xi)| < 1$. Тогава при достатъчно добро начално приближение x_0, итерационният процес, породен от φ, е сходящ. Нещо повече, съществуват константи C > 0 и 0 < q < 1, такива че $|x_n - \xi| \leq C.q^n$ за всяко n = 0, 1, ….
Доказателство: Тъй като $\varphi'(x)$ е непрекъсната функция в U и $|\varphi'(\xi)| < 1$, то съществуват q < 1 и $\delta > 0$, такива че $|\varphi'(x)| \leq q$ за всяко $x \in [\xi - \delta, \xi + \delta]$. Освен това, за всяко $x \in [\xi - \delta, \xi + \delta]$ имаме $|\varphi(x) - \xi| = |\varphi(x) - \varphi(\xi)| = |\varphi'(\hbar)|.|x - \xi| \leq q.\delta < \delta$, $\hbar$ е между x и ξ, така че $\varphi(x) \in [\xi - \delta, \xi + \delta]$. Следователно, φ е свиващо изображение на интервала $[\xi - \delta, \xi + \delta]$ в себе си и всички твърдения на следствието следват от теоремата.

Ето и геометрична илюстрация на итерационният процес, породен от свиващо изображение φ.



Скоростта на сходимост в разгледания итерационен процес се определя от общия член q^n на една геометрична прогресия. Затова

казваме, че итерационният процес е сходящ **със скоростта на геометрична прогресия**. Има и по-бързо сходящи итерационни процеси. За да характеризираме тяхната скорост въвеждаме понятието **ред на сходимост**.
Казваме, че итерационният процес $\{x_n\}$ има ред на сходимост p, ако съществуват константи C > 0 и 0 < q < 1, такива че
$|x_n - \alpha| \leq C.q^{p^n}$ за всяко n = 0, 1, ….
Ще посочим без доказателство една теорема, която ни позволява да определяме реда на сходимост на итерационен процес, породен от изображението φ.

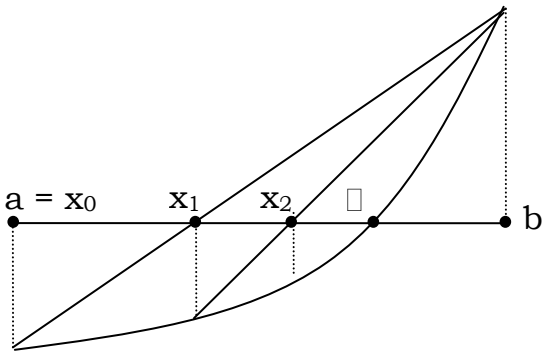
Теорема: Нека изображението φ има непрекъснати производни до p-ти ред включително в околност на точката α. Нека $\varphi(\alpha) = \alpha$, $\varphi'(\alpha) = \varphi''(\alpha) = \ldots = \varphi^{(p-1)}(\alpha) = 0$, $\varphi^{(p)}(\alpha) \neq 0$. Тогава при достатъчно добро начално приближение x_0, итерационният процес, породен от φ има ред на сходимост p.

Сега ще разгледаме три класически итерационни метода за приближено решаване на уравнения. При всички тях предполагаме, че се търси корен на уравнението f (x) = 0, където f е функция, която удоволетворява следните условия:

1. f е дефинирана в интервала [a, b].
2. f притежава непрекъсната втора производна в [a, b].
3. f (a).f (b) < 0.
4. $f'(x) \neq 0$ за всяко $x \in [a, b]$.
5. $f''(x) \neq 0$ за всяко $x \in [a, b]$.

Тези условия гарантират съществуването на единствен корен α на уравнението f (x) = 0 в интервала [a, b]. Действително, условието 2. осигурява непрекъснатост на f, условието 3. показва, че f приема противоположни знаци в краищата на интервала [a, b] $\Rightarrow$ f има поне един корен. От условие 4. следва, че първата производна на f има постоянен знак $\Rightarrow$ f е строго монотонна функция $\Rightarrow$ графиката на f може да пресича най-много един път оста Ox, т.е. f (x) = 0 може да има най-много едно решение. Също, от условие 5. и непрекъснатостта на $f'' \Rightarrow f''$ има постоянен знак в [a, b] $\Rightarrow$ f е или изпъкнала (при $f''(x) > 0$ за всяко $x \in [a, b]$) или вдлъбната (при $f''(x) < 0$ за всяко $x \in [a, b]$).

Методът на хордите е итерационен метод, при който се построява редица от последователни приближения $x_0, x_1, \ldots, x_n, \ldots$ на корена α на уравнението f (x) = 0 по следния начин: избираме $x_0 = a$ или $x_0 = b$, така че да имаме $f(x_0).f''(x_0) < 0$. Нека за определеност сме избрали $x_0 = a$. Тогава за n = 0, 1, …, ако сме избрали x_n, то x_{n+1} избираме по следния начин: прекарваме права през $(x_n, f(x_n))$ и (b, f (b)) (през $(x_n, f(x_n))$ и (a, f (a)), ако $x_0 = b$) и x_{n+1} е пресечната точка на тази права с оста Ox. За по-голяма нагледност привеждаме геометрична илюстрация:



Тя отговаря на случая $x_0 = a$, $f(a) < 0$ и $f''(x) > 0$ за всяко $x \in [a, b]$. Сега ще изведем формула за x_{n+1} чрез x_n. Правата през $(x_n, f(x_n))$ и $(b, f(b))$ има уравнение $y = f(x_n) + (x - x_n).f[x_n; b]$, така че x_{n+1} е решението на уравнението $0 = f(x_n) + (x - x_n).f[x_n; b]$, т.е.

$x_{n+1} = x_n - \frac{f(x_n)}{f[x_n; b]} = x_n - \frac{f(x_n)}{f(b) - f(x_n)}.(b - x_n)$. Като се използва, че $f(a) < 0$ и f е изпъкнала в $[a, b]$ (или $f(a) > 0$ и f е вдлъбната в $[a, b]$) може да се покаже, че редицата $\{x_n\}$ е монотонно растяща и ограничена отгоре $\Rightarrow \{x_n\}$ е сходяща. Нека $x_n \to \alpha$ при $n \to \infty$.

В равенството $x_{n+1} = x_n - \frac{f(x_n)}{f(b) - f(x_n)}.(b - x_n)$ извършваме граничен преход при $n \to \infty$, като използваме непрекъснатостта на f и получаваме $\alpha = \alpha - \frac{f(\alpha)}{f(b) - f(\alpha)}.(b - \alpha) \Rightarrow f(\alpha) = 0 \Rightarrow x = \alpha$ и сходимостта на редицата $\{x_n\}$ към α е доказана. Ясно е, че при методът на хордите итерационният процес е породен от следната функция φ:

$\varphi(x) = x - \frac{f(x)}{f(b) - f(x)}.(b - x)$. Уравнението $f(x) = 0$ е еквивалентно на уравнението $x = \varphi(x)$. Ще приложим следствието от по-горе за да дадем оценка за сходимостта на итерационния процес.
За целта изчисляваме $\varphi'(\alpha)$.

Имаме $\varphi'(\alpha) = 1 - \frac{f'(\alpha)}{f(b) - f(\alpha)}.(b - \alpha) - f(\alpha).\left[\frac{(b - x)}{f(b) - f(x)}\right]'_{x = \alpha} =$

$= 1 - \frac{f'(\alpha)}{f(b)}.(b - \alpha) = \frac{f(b) - f'(\alpha).(b - \alpha)}{f(b)}$. Използвали сме, че $f(\alpha) = 0$.

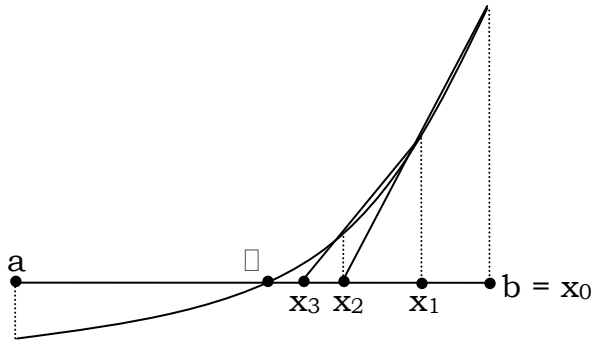
Сега използваме формулата на Тейлър: съществуват $\eta_1, \eta_2 \in [a, b]$, такива че $f(b) = f(\alpha) + f'(\alpha).(b - \alpha) + \frac{f''(\eta_1)}{2}.(b - \alpha)^2$ и

$f(b) = f(\alpha) + f'(\eta_2).(b - \alpha)$. Така $\varphi'(\alpha) = \frac{f''(\eta_1).(b - \alpha)}{2.f'(\eta_2)}$.

Нека $M = \max_{\eta \in [a, b]} |f''(\eta)|$, $m = \min_{\eta \in [a, b]} |f'(\eta)|$, $m > 0$.

Тогава $|\varphi'(\xi)| \le \frac{M}{2.m}.(b - \xi)$. Така стига интервалът [a, b] да е достатъчно малък, $|\varphi'(\xi)|$ може да стане по-малко от произволно отнапред избрано q < 1. Оттук по следствието от по-горе получаваме, че при достатъчно малък интервал, съдържащ корена, методът на хордите е сходящ със скоростта на геометрична прогресия.

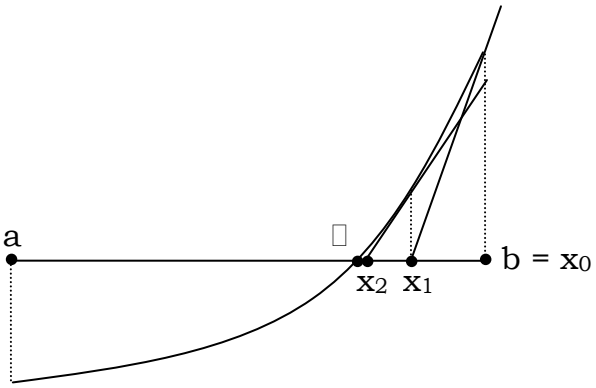
Методът на секущите е итерационен метод, при който се построява редица от последователни приближения $x_0, x_1, \ldots, x_n, \ldots$ на корена ξ на уравнението f (x) = 0 по следния начин: избираме $x_0 = a$ или $x_0 = b$, така че да имаме $f(x_0).f''(x_0) > 0$. След това избираме точка $x_1 \neq x_0$, $x_1 \in [a, b]$, такава че $f(x_1).f''(x_1) > 0$. По-нататък, ако за n = 1, 2, … сме избрали x_{n-1} и x_n, то x_{n+1} избираме по следния начин: прекарваме права през $(x_{n-1}, f(x_{n-1}))$ и $(x_n, f(x_n))$ и x_{n+1} е пресечната точка на тази права с оста Ox. За по-голяма нагледност привеждаме геометрична илюстрация:



Тя отговаря на случая $x_0 = b$, f (b) > 0 и $f''(x) > 0$ за всяко $x \in [a, b]$. Сега ще изведем формула за x_{n+1} чрез x_n и x_{n-1}. Правата през $(x_n, f(x_n))$ и $(x_{n-1}, f(x_{n-1}))$ има уравнение $y = f(x_n) + (x - x_n).f[x_{n-1}; x_n]$, така x_{n+1} е решението на уравнението $0 = f(x_n) + (x - x_n).f[x_{n-1}; x_n]$, т.е. $x_{n+1} = x_n - \frac{f(x_n)}{f[x_{n-1}; x_n]} = x_n - \frac{f(x_n)}{f(x_{n-1}) - f(x_n)}.(x_{n-1} - x_n)$.

Може да се покаже, че ако интервалът [a, b] е достатъчно малък, методът на секущите е сходящ и има ред на сходимост $\frac{1+\sqrt{5}}{2}$.

Методът на допирателните (още известен като **метод на Нютон**) е итерационен метод, при който се построява редица от последователни приближения $x_0, x_1, \ldots, x_n, \ldots$ на корена ξ на уравнението f (x) = 0 по следния начин: избираме $x_0 = a$ или $x_0 = b$, така че да имаме $f(x_0).f''(x_0) > 0$. Тогава за n = 0, 1, …, ако сме избрали x_n, то x_{n+1} избираме по следния начин: прекарваме допирателната към графиката на f в точката $(x_n, f(x_n))$ и x_{n+1} е пресечната точка на тази допирателна с оста Ox. За по-голяма яснота привежда геометрична илюстрация:



Тя отговаря на случая $x_0 = b$, $f(b) > 0$ и $f''(x) > 0$ за всяко $x \in [a, b]$. Сега ще изведем формула за x_{n+1} чрез x_n. Допирателната към графиката на f в точката $(x_n, f(x_n))$ има уравнение $y = f(x_n) + f'(x_n).(x - x_n)$, така x_{n+1} е решението на уравнението $0 = f(x_n) + f'(x_n).(x - x_n)$, т.е. $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Ясно е, че при методът на допирателните, итерационният процес е породен от следната функция φ: $\varphi(x) = x - \frac{f(x)}{f'(x)}$. Уравнението $f(x) = 0$ е еквивалентно на уравнението $x = \varphi(x)$. Лесно се вижда, че $\varphi'(\xi) = 0$. Също може да се покаже, че в общия случай $\varphi''(\xi) \neq 0$. Така от теоремата по-горе, ако интервалът [a, b] е достатъчно малък, методът на допирателните е сходящ и има ред на сходимост 2.

27. Дискретни разпределения.

В теория на вероятностите първично понятие е **елементарно събитие**. Множеството от всички елементарни събития наричаме **сигурно събитие** и бележим с Ω. Подмножествата на Ω се наричат **събития**. Празното подмножество $\emptyset$ наричаме **невъзможно събитие**. Можем да си мислим, че елементарните събития са всевъзможните изходи от някакъв експеримент.
Ако елементарното събитие $w \in \Omega$ се сбъдне и $w \in A \subset \Omega$, ще казваме че се е сбъднало събитието A. Тъй като за всяко w имаме $w \in \Omega$, то Ω се сбъдва при всеки експеримент. Тъй като за всяко w имаме $w \notin \emptyset$, то събитието $\emptyset$ не може да се сбъдне при никой експеримент.
Нека $\mathcal{F}$ е фамилия от подмножества на Ω, т.е. $\mathcal{F}$ е множество от събития. Ще казваме, че $\mathcal{F}$ е **булова алгебра**, ако са в сила следните свойства:

1. $\Omega \in \mathcal{F}$;
2. ако $A \in \mathcal{F}$, то $\overline{A} \in \mathcal{F}$;
3. ако $A, B \in \mathcal{F}$, то $A \cup B \in \mathcal{F}$.

Ще казваме, че буловата алгебра $\mathcal{A}$ е **булова σ-алгебра**, ако тя е затворена относно изброимите сечения и обединения, т.е.

$A_i \in \mathcal{A}$, $i = 1, 2, \ldots \Rightarrow \bigcup_{i=1}^{\infty} A_i \in \mathcal{A}$ и $A_i \in \mathcal{A}$, $i = 1, 2, \ldots \Rightarrow \bigcap_{i=1}^{\infty} A_i \in \mathcal{A}$.

За краткост буловите σ-алгебри ще наричаме просто σ-алгебри.

Двойката $(\Omega, \mathcal{A})$, където $\mathcal{A}$ е булова алгебра от подмножества на Ω се нарича **измеримо пространство**. Елементите на $\mathcal{A}$ се наричат **случайни събития**.

Нека $(\Omega, \mathcal{A})$ е измеримо пространство. **Вероятност P** в измеримото пространство наричаме всяка реална функция, дефинирана върху случайните събитията от $\mathcal{A}$, която удоволетворява следните аксиоми:

1. За всяко $A \in \mathcal{A}$, $\mathbf{P}(A) \geq 0$. (**неотрицателност**)
2. $\mathbf{P}(\Omega) = 1$. (**нормираност**)
3. $\mathbf{P}(A + B) = \mathbf{P}(A) + \mathbf{P}(B)$, ако $A \cap B = \emptyset$. (**адитивност**)
4. За всяка редица $\{A_n\}$, такава че $A_n \downarrow \emptyset$, т.е. $A_{n+1} \subset A_n$ и $\bigcap_{n=1}^{\infty} A_n = \emptyset$, имаме $\lim_{n \to \infty} \mathbf{P}(A_n) = 0$. (**непрекъснатост в $\emptyset$**)

При това положение тройката $(\Omega, \mathcal{A}, \mathbf{P})$ наричаме **вероятностно пространство**.

От аксиомите получаваме следните свойства:

1. $\mathbf{P}(\emptyset) = 0$.
2. Ако $A \subset B$, то $\mathbf{P}(A) \leq \mathbf{P}(B)$. (**монотонност**)
3. За всяко $A \in \mathcal{A}$, $0 \leq \mathbf{P}(A) \leq 1$.
4. За всяко $A \in \mathcal{A}$, $\mathbf{P}(\overline{A}) = 1 - \mathbf{P}(A)$.
5. За всеки $A, B \in \mathcal{A}$, $\mathbf{P}(A \cup B) = \mathbf{P}(A) + \mathbf{P}(B) - \mathbf{P}(AB)$.
6. За всеки $A_1, A_2, \ldots, A_n \in \mathcal{A}$, такива че $A_i \cap A_j = \emptyset$ при $i \neq j$, $\mathbf{P}(\sum_{i=1}^{n} A_i) = \sum_{i=1}^{n} \mathbf{P}(A_i)$. (**крайна адитивност**)
7. За всеки $A_n \in \mathcal{A}$, $n = 1, 2, \ldots$, такива че $A_i \cap A_j = \emptyset$ при $i \neq j$ и $\sum_{n=1}^{\infty} A_n \in \mathcal{A}$ имаме $\mathbf{P}(\sum_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mathbf{P}(A_n)$. (**$\sigma$-адитивност**)

Теорема (Каратеодори): Всяка вероятност, дефинирана върху буловата алгебра $\mathcal{A}$ може да се продължи по единствен начин върху най-малката σ-алгебра, която съдържа $\mathcal{A}$.

По-точно теоремата казва, че ако $(\Omega, \mathcal{A}, \mathbf{P})$ е вероятностно пространство, където $\mathcal{A}$ е булова алгебра, то може да се построи по единствен начин вероятностно пространство $(\Omega, \mathcal{A}', \mathbf{P}')$, където $\mathcal{A}'$ е най-малката σ-алгебра, която съдържа $\mathcal{A}$, така че $\mathbf{P}(A) = \mathbf{P}'(A)$ за всяко $A \in \mathcal{A}$. Така оттук нататък можем да считаме, че във всяко вероятностно пространство $(\Omega, \mathcal{A}, \mathbf{P})$ имаме, че $\mathcal{A}$ е σ-алгебра.

Нека (Ω, $\mathcal{A}$, **P**) е вероятностно пространство.
Ще казваме, че функцията $\xi : \Omega \to \mathbb{R}$ е **случайна величина**, ако за всяко $x \in \mathbb{R}$ имаме, че $L_\xi(x) = \{ w \mid \xi(w) < x \} \in \mathcal{A}$, т.е. $L_\xi(x) \in \mathcal{A}$.
За всяка случайна величина ξ, дефинираме функцията $\mathbf{F}_\xi(x) = \mathbf{P}(\{ w \mid \xi(w) < x\})$, която наричаме **функция на разпределение** на случайната величина ξ.
С други думи $\mathbf{F}_\xi(x)$ е вероятността случайната величина ξ да приеме стойност по-малка от x. Бележим още $\mathbf{F}_\xi(x) = \mathbf{P}(\xi < x)$.
По-нататък ще изпускаме индекса ξ и ще считаме, че той се подразбира.
Свойства на функцията на разпределение:

1. За всяко $x \in \mathbb{R}$, $0 \le \mathbf{F}(x) \le 1$.
2. За всеки $x_1, x_2 \in \mathbb{R}$, $x_1 < x_2 \Rightarrow \mathbf{F}(x_1) \le \mathbf{F}(x_2)$. (**монотонност**)
3. За всяка монотонно растяща редица $\{x_n\}$, $x_n \in \mathbb{R}$, $x_n \to x_0$ при $n \to \infty$ имаме, че $\mathbf{F}(x_n) \to \mathbf{F}(x_0)$. (**непрекъснатост отляво**)
4. $\lim_{x \to -\infty} \mathbf{F}(x) = 0$, $\lim_{x \to +\infty} \mathbf{F}(x) = 1$.

Не е задължително, функцията на разпределение **F** (x) да бъде непрекъсната, въпреки че тя е непрекъсната отляво. В общия случай, **F** (x) има точки на прекъсване със скок. Лесно се вижда, че те са не повече от изброимо много.
На всяка случайна величина се съпоставя функция на разпределение. Лесно се вижда, че това съпоставяне не е еднозначно. В теорията на вероятностите, обаче, често случайните величини с еднакви разпределения се считат за неразличими, т.е. пренебрегват се вероятностните пространства, в които случайните величини са реализирани.

Казваме, че случайните величини ξ и η са **независими**, ако за всеки x, y $\in \mathbb{R}$ имаме $\mathbf{P}(\xi < x, \eta < y) = \mathbf{P}(\xi < x).\mathbf{P}(\eta < y)$, бележим $\xi \perp \eta$.
Казваме, че случайните величини $\xi_1, \xi_2, \ldots, \xi_n$ са **независими в съвкупност**, ако за всеки $x_1, x_2, \ldots, x_n \in \mathbb{R}$ имаме, че
$\mathbf{P}(\xi_1 < x_1, \xi_2 < x_2, \ldots, \xi_n < x_n) =$
$= \mathbf{P}(\xi_1 < x_1).\mathbf{P}(\xi_2 < x_2). \ldots .\mathbf{P}(\xi_n < x_n)$.
Ако $\xi_1, \xi_2, \ldots, \xi_n$ са независими в съвкупност, то $\xi_i \perp \xi_j$ при $i \ne j$, т.е. те са независими по двойки. Обратното не е вярно, както лесно се показва с пример.

Казваме, че случайната величина ξ е **дискретна**, ако тя приема изброимо много стойности $x_1, x_2, \ldots, x_n, \ldots$върху случайни събития от $\mathcal{A}$. Лесно се проверява, че дефиницията е коректна, т.е. дефинирата по този начин функция действително е случайна величина.

Нека $\mathbf{P}(\xi = x_i) = p_i$, $i = 1, 2, \ldots$. Естествено, $p_i \geq 0$ и $\sum_{i=1}^{\infty} p_i = 1$.

За функцията на разпределение получаваме

$\mathbf{F}(x) = \mathbf{P}(\xi < x) = \mathbf{P}\left(\bigcup_{x_k<x} \xi = x_k\right) = \sum_{x_k<x} \mathbf{P}(\xi = x_k) = \sum_{x_k<x} p_k$.

В случая когато $x_1 < x_2 < \ldots < x_n < \ldots$ за $x \in (x_m, x_{m+1}]$ имаме $\mathbf{F}(x) = p_1 + p_2 + \ldots + p_m$, $m = 1, 2, \ldots$и така $\mathbf{F}(x)$ има скокове в точките $x_1, x_2, \ldots$ с височина съответно $p_1, p_2, \ldots$.

Нека ξ е дискретна случайна величина, която приема стойности $x_1, x_2, \ldots$. Да означим $p_i = \mathbf{P}(\xi = x_i)$, $i = 1, 2, \ldots$, $p_i \geq 0$ и $\sum_{i=1}^{\infty} p_i = 1$.

Функционалът $\mathbf{E}\xi = \sum_{k=1}^{\infty} x_k p_k$ наричаме **математическо очакване** на дискретната случайна величина ξ. Естествено, възможно е редът $\sum_{k=1}^{\infty} x_k p_k$ да не е сходящ и тогава ξ няма математическо очакване. Функционалът $\mathbf{D}\xi = \mathbf{E}(\xi - \mathbf{E}\xi)^2$ наричаме **дисперсия** на дискретната случайна величина ξ. Естествено, не всички дискретни случайни величини имат дисперсии.

Свойства на математическото очакване и дисперсията:

1. $\mathbf{E}C = C$. (**запазване на константите**)
2. $\mathbf{E}(\alpha\xi + \beta\eta) = \alpha\mathbf{E}\xi + \beta\mathbf{E}\eta$ (**линейност**)
3. Ако $\xi \geq \eta$, то $\mathbf{E}\xi \geq \mathbf{E}\eta$ (**монотонност**)
4. Ако ξ и η, то $\mathbf{E}\xi.\eta = \mathbf{E}\xi.\mathbf{E}\eta$. По-общо, ако $\xi_1, \xi_2, \ldots, \xi_n$ са независими в съвкупност, то $\mathbf{E}\xi_1.\xi_2. \ldots.\xi_n = \mathbf{E}\xi_1.\mathbf{E}\xi_2.\ldots.\mathbf{E}\xi_n$.
5. $\mathbf{D}\xi \geq 0$.
6. $\mathbf{D}\xi = \mathbf{E}\xi^2 - (\mathbf{E}\xi)^2$.
7. $\mathbf{D}\alpha\xi = \alpha^2\mathbf{D}\xi$.
8. $\mathbf{D}(\xi + C) = \mathbf{D}\xi$.
9. Ако ξ и η, то $\mathbf{D}(\xi+\eta) = \mathbf{D}\xi + \mathbf{D}\eta$. По-общо, ако $\xi_1, \xi_2, \ldots, \xi_n$ са независими в съвкупност, то $\mathbf{D}(\xi_1 + \xi_2 + \ldots + \xi_n) = \mathbf{D}\xi_1 + \mathbf{D}\xi_2 + \ldots + \mathbf{D}\xi_n$.

Една случайна величина ξ наричаме **целочислена**, ако тя приема за стойности естествени числа, т.е. елементи на $\mathbb{N} = \{0, 1, 2, \ldots\}$. Естествено, всяка целочислена случайна величина е дискретна.

Нека n е произволно естествено число, $n \geq 1$. Разглеждаме целочислена случайна величина ξ с област от стойности множеството $\{1, 2, \ldots, n\}$, като ξ приема всяка от тези стойности с равна вероятност. Така в случая, $x_i = i$, $p_i = \frac{1}{n}$, $i = 1, 2, \ldots, n$.

За математическото очакване на ξ имаме

$\mathbf{E}\xi = \sum_{k=1}^{n} k.\frac{1}{n} = \frac{1}{n}.\sum_{k=1}^{n} k = \frac{1}{n}.\frac{n.(n+1)}{2} = \frac{n+1}{2}$.

За дисперсията на ξ имаме

$\mathbf{D}\xi = \mathbf{E}\xi^2 - (\mathbf{E}\xi)^2 = \sum_{k=1}^{n} k^2.\frac{1}{n} - \left(\frac{n+1}{2}\right)^2 = \frac{1}{n}.\frac{n.(n+1).(2.n+1)}{6} - \left(\frac{n+1}{2}\right)^2 =$

$= \frac{n+1}{2}.\left(\frac{2.n+1}{3} - \frac{n+1}{2}\right) = \frac{n+1}{2}.\frac{4.n+2-3.n-3}{6} = \frac{n+1}{2}.\frac{n-1}{6} = \frac{n^2-1}{12}$.

В случая казваме, че случайната величина ξ има **равномерно разпределение**.

Пример за задача, в която възниква равномерно разпределение при n = 6 е задачата за хвърляне на правилен зар. Величината "числото, което се е паднало" е равномерно разпределена – тя приема стойностите 1, 2, 3, 4, 5, 6 с една и съща вероятност $\frac{1}{6}$.

Нека p е реално число, $0 \le p \le 1$, q = 1 – p.

Разглеждаме безкрайна последователност $\xi_1, \xi_2, \ldots, \xi_n, \ldots$ от случайни величини, като всеки краен брой от тях са независими в съвкупност и всяка от които приема две стойности 1 и 0 с вероятности съответно p и q. Такава последователност наричаме **схема на Бернули**. Всяка от тези случайни величини представя някакъв опит, като случайната величина приема стойност 1, ако опитът е успешен и стойност 0, ако опитът е неуспешен. Вероятността за успех при всеки от опитите е p, а за неуспех е q. Освен това, опитите са независими, т.е. резултатът от кой да е опит не влияе на резултата от останалите опити.

Нека сега n е естествено число, $n \ge 1$.

Разглеждаме случайната величина $\hbar_n = \xi_1 + \xi_2 + \ldots + \xi_n$.

Ясно е, че $\hbar_n$ е целочислена случайна величина, която приема стойности от 0 до n. Интерпретацията на $\hbar_n$ е следната: броят на успешните опити от първите n опита в схемата на Бернули. Казваме, че случайната величина $\hbar_n$ има **биномно разпределение**.

Теорема: $\mathbf{P}(\hbar_n = k) = \binom{n}{k} p^k.q^{n-k}$ за всяко k = 0, 1, …, n.

Доказателство: Да означим с $W(\eta_1, \eta_2, \ldots, \eta_n)$ събитието $(\xi_1 = \eta_1, \xi_2 = \eta_2, \ldots, \xi_n = \eta_n)$. Тъй като случайните величини $\xi_1, \xi_2, \ldots, \xi_n$ са независими в съвкупност, то

$\mathbf{P}(W(\eta_1, \eta_2, \ldots, \eta_n)) = \mathbf{P}(\xi_1 = \eta_1).\ \mathbf{P}(\xi_2 = \eta_2).\ \ldots.\ \mathbf{P}(\xi_n = \eta_n) =$

$= p^{\sum_{i=0}^{n} \eta_i}.q^{n-\sum_{i=0}^{n} \eta_i}$. Тогава $\mathbf{P}(\hbar_n = k) = \mathbf{P}\left(\bigcup_{\eta_1+\eta_2+\ldots+\eta_n = k} W(\eta_1, \eta_2, \ldots, \eta_n)\right) =$

$= \sum_{\eta_1+\eta_2+\ldots+\eta_n = k} \mathbf{P}(W(\eta_1, \eta_2, \ldots, \eta_n)) = \sum_{\eta_1+\eta_2+\ldots+\eta_n = k} p^{\sum_{i=0}^{n} \eta_i}.q^{n-\sum_{i=0}^{n} \eta_i} = \sum_{\eta_1+\eta_2+\ldots+\eta_n = k} p^k.q^{n-k} =$

$= p^k.q^{n-k}. \sum_{\xi_1+\xi_2+\ldots+\xi_n=k} 1 = \binom{n}{k}.p^k.q^{n-k}.$

Сега ще пресметнем математическото очакване и дисперсията на биномно разпределената случайна величина $\hbar_n$.

Имаме $\mathbf{E}\hbar_n = \mathbf{E}(\xi_1 + \xi_2 + \ldots + \xi_n) = \mathbf{E}\xi_1 + \mathbf{E}\xi_2 + \ldots + \mathbf{E}\xi_n =$

$= (1.p + 0.q) + (1.p + 0.q) + \ldots + (1.p + 0.q) = n.p.$

Също $\mathbf{D}\hbar_n = \mathbf{D}(\xi_1 + \xi_2 + \ldots + \xi_n) = \mathbf{D}\xi_1 + \mathbf{D}\xi_2 + \ldots + \mathbf{D}\xi_n =$

$= \mathbf{E}\xi_1^2 - (\mathbf{E}\xi_1)^2 + \mathbf{E}\xi_2^2 - (\mathbf{E}\xi_2)^2 + \ldots + \mathbf{E}\xi_n^2 - (\mathbf{E}\xi_n)^2 =$

$= (1^2.p + 0^2.q) - (1.p + 0.q)^2 + (1^2.p + 0^2.q) - (1.p + 0.q)^2 + \ldots$

$+ (1^2.p + 0^2.q) - (1.p + 0.q)^2 = n.(p - p^2) = n.p.(1 - p) = n.p.q.$

При изчисляването на дисперсията използвахме, че случайните величини $\xi_1, \xi_2, \ldots, \xi_n$ са независими в съвкупност.

Пример за задача, в която възниква биномно разпределение е задачата за хвърляне на правилна монета. Величината "броят на хвърлените ези от n хвърляния" е биномно разпределена – при нея $p = q = \frac{1}{2}$.

Нека p е реално число, $0 \leq p < 1$, $q = 1 - p$, $0 < q \leq 1$.

Казваме, че една целочислена случайна величина ξ има **геометрично разпределение**, ако $\mathbf{P}(\xi = m) = p^m.q$, $m = 0, 1, 2, \ldots$.

Случайната величина ξ може да се интепретира като брой успехи до първи неуспех в схемата на Бернули. Посоченото разпределение е коректно: $\mathbf{P}(\xi = m) > 0$ за всяко $m \in \mathbb{N}$ и

$\sum_{m=0}^{\infty} \mathbf{P}(\xi = m) = \sum_{m=0}^{\infty} p^m.q = \frac{q}{1-p} = 1$. За да пресметнем математическо очакване и дисперсията на ξ разглеждаме функцията

$f(s) = \sum_{m=0}^{\infty} (p.s)^m.q$, дефинирана при $|s| < \frac{1}{p}$ или за всяко $s \in \mathbb{R}$, ако $p = 0$. $f(s)$ се нарича **пораждаща функция** на целочислената случайна величина ξ. Имаме $f(s) = \sum_{m=0}^{\infty} (p.s)^m.q = q.\sum_{m=0}^{\infty} (s.p)^m = \frac{q}{1-s.p}$.

Също така, $f'(s) = \left(\sum_{m=0}^{\infty} (p.s)^m.q\right)' = \sum_{m=0}^{\infty} m.s^{m-1}.p^m.q$ при $|s| < \frac{1}{p}$.

Използвахме, че във вътрешността на областта на сходимост на реда $f(s)$ е в сила теоремата за почленно диференциране. От друга страна, $f'(s) = \left(\frac{q}{1-s.p}\right)' = \frac{q.(-p)}{(1-s.p)^2} = \frac{p.q}{(1-s.p)^2}$ при $|s| < \frac{1}{p}$.

Също така, $f''(s) = \left(\sum_{m=0}^{\infty} (p.s)^m.q\right)' = \sum_{m=0}^{\infty} m.(m-1).s^{m-2}.p^m.q$ при $|s| < \frac{1}{p}$.

Отново използвахме теоремата за почленно диференциране.

От друга страна, $f''(s) = \left(\frac{q}{(1-s.p)^2}\right)' = \frac{p.q}{(1-s.p)^3} + \frac{2.p.q.(-p)}{(1-s.p)^3} = \frac{2.p^2.q}{(1-s.p)^3}$

при $|s| < \frac{1}{p}$. Сега пресмятаме $\mathbf{E}\xi = \sum_{m=0}^{\infty} m.p^m.q = f'(1) =$

$= \frac{p.q}{(1-1.p)^2} = \frac{p.q}{q^2} = \frac{p}{q}$. Също, $\mathbf{D}\xi = \mathbf{E}\xi.(\xi - 1) + \mathbf{E}\xi - (\mathbf{E}\xi)^2 =$

$= \sum_{m=0}^{\infty} m.(m-1).p^m.q = \frac{p}{q} + \frac{p^2}{q^2} = f''(1) + \frac{p}{q} + \frac{p^2}{q^2} =$

$= \frac{2.p^2.q}{(1-1.p)^3} + \frac{p}{q} + \frac{p^2}{q^2} = \frac{2.p^2.q}{q^3} + \frac{p}{q} + \frac{p^2}{q^2} + \frac{p}{q} + \frac{p^2}{q^2} = \frac{p.q+p^2}{q^2} + \frac{p}{q^2}$.

Пример за задача, в която възниква геометрично разпределение е задачата за стрелба по мишена. Величината “брой улучвания преди първи пропуск” има геометрично разпределение.

Нека ℓ е реално число, $\ell > 0$. Казваме, че една целочислена случайна величина ξ има **поасоново разпределение**, ако $\mathbf{P}(\xi = k) = e^{-\ell}.\frac{\ell^k}{k!}$, $k = 0, 1, 2, \ldots$. Посоченото разпределение е коректно: $\mathbf{P}(\xi = k) > 0$ за всяко $k \in \mathbb{N}$ и освен това $\sum_{k=0}^{\infty} e^{-\ell}.\frac{\ell^k}{k!} = e^{-\ell}.\sum_{k=0}^{\infty} \frac{\ell^k}{k!} = e^{-\ell}.e^{\ell} = 1$. За да пресметнем математическо очакване и дисперсията на ξ разглеждаме нейната пораждаща функция $f(s) = \sum_{k=0}^{\infty} e^{-\ell}.\frac{\ell^k}{k!}.s^k = e^{-\ell}.\sum_{k=0}^{\infty} \frac{(\ell.s)^k}{k!} = e^{\ell.(s-1)}$, дефинирана за всяко $s \in \mathbb{R}$. От една страна, като използваме теоремата за почленно диференциране, получаваме $f'(s) = \sum_{k=0}^{\infty} e^{-\ell}.\frac{\ell^k}{k!}.k.s^{k-1}$, от друга страна $f'(s) = e^{\ell.(s-1)}.\ell$. Също с теоремата за почленно диференциране, $f''(s) = \sum_{k=0}^{\infty} e^{-\ell}.\frac{\ell^k}{k!}.k.(k-1).s^{k-2}$, а от друга страна, $f''(s) = (e^{\ell.(s-1)}.\ell)' = e^{\ell.(s-1)}.\ell^2$.

Сега вече можем да пресметнем математическото очакване и дисперсията на ξ.

$\mathbf{E}\xi = \sum_{k=0}^{\infty} k.e^{-\ell}.\frac{\ell^k}{k!} = f'(1) = e^0.\ell = \ell$.

$\mathbf{D}\xi = \mathbf{E}\xi.(\xi - 1) + \mathbf{E}\xi - (\mathbf{E}\xi)^2 = \sum_{k=0}^{\infty} k.(k-1).e^{-\ell}.\frac{\ell^k}{k!} + \ell - \ell^2 = f''(1) + \ell - \ell^2 =$

$= e^0.\ell^2 + \ell - \ell^2 = \ell$.

Може да се покаже, че поасоновото разпределение е граница на биномни разпределения при $n \to \infty$, така че $np \to \ell > 0$.
Разпределението на Поасон е особено подходящо за моделиране на броя на случайни редки събития – например, брой засечени радиоактивни частици на единица обем, брой на радиоактивните разпадания за единица време и т.н.

Нека са фиксирани N, M, $n \in \mathbb{N}$, при това $n \leq N$, $M \leq N$.
Казваме, че една целочислена случайна величина ξ има **хипергеометрично разпределение**,

ако $\mathbf{P}(\xi = m) = \dfrac{\binom{M}{m}\binom{N-M}{n-m}}{\binom{N}{n}}$, $m = 0, 1, \ldots, n$.

Считаме, че $\binom{s}{k} = 0$ при $s < k$. Посоченото разпределение е коректно: $\mathbf{P}(\xi = m) \geq 0$ за всяко $m \in \{0, 1, \ldots, n\}$ и

$$\sum_{m=0}^{n} \mathbf{P}(\xi = m) = \sum_{m=0}^{n} \frac{\binom{M}{m}\binom{N-M}{n-m}}{\binom{N}{n}} = \frac{1}{\binom{N}{n}} \sum_{m=0}^{n} \binom{M}{m}\binom{N-M}{n-m} = \frac{1}{\binom{N}{n}} \cdot \binom{N}{n} = 1.$$

Използвали сме известното тъждество на Коши.
Сега ще пресметнем математическото очакване и дисперсията на хипергеометрично разпределената целочислена случайна величина ξ.

$$\mathbf{E}\xi = \sum_{m=0}^{n} m.\frac{\binom{M}{m}\binom{N-M}{n-m}}{\binom{N}{n}} = \frac{1}{\binom{N}{n}} \sum_{m=1}^{n} m.\binom{M}{m}\binom{N-M}{n-m} = \frac{1}{\binom{N}{n}} \sum_{m=1}^{n} M.\binom{M-1}{m-1}\binom{N-M}{n-m} =$$

$$= \frac{M}{\binom{N}{n}} \sum_{i=0}^{n-1} \binom{M-1}{i}\binom{N-M}{n-1-i} = \frac{M}{\binom{N}{n}} \cdot \underbrace{\binom{N-M-(M-1)}{n-1}}_{} = \frac{M}{\frac{N}{n} \cdot \binom{N-1}{n-1}} \cdot \underbrace{\binom{N-M-(M-1)}{n-1}}_{} = \frac{M.n}{N}.$$

Използвахме известното равенство $k.\binom{s}{k} = s.\binom{s-1}{k-1}$, а на четвъртата стъпка в сумата направихме смяната $i = m-1$.

$$\mathbf{E}\xi.(\xi - 1) = \sum_{m=0}^{n} m.(m-1).\frac{\binom{M}{m}\binom{N-M}{n-m}}{\binom{N}{n}} = \frac{1}{\binom{N}{n}} \sum_{m=2}^{n} (m-1).M.\binom{M-1}{m-1}\binom{N-M}{n-m} =$$

$$= \frac{M}{\binom{N}{n}} \sum_{m=2}^{n} (M-1).\binom{M-2}{m-2}\binom{N-M}{n-m} = \frac{M.(M-1)}{\binom{N}{n}} \sum_{i=0}^{n-2} \binom{M-2}{i}\binom{N-M}{n-2-i}$$

$$= \frac{M.(M-1)}{\binom{N}{n}} \binom{N-M-(M-2)}{n-2} = \frac{M.(M-1)}{\frac{N.(N-1)}{n.(n-1)}\binom{N-2}{n-2}} = \frac{M.(M-1).n.(n-1)}{N.(N-1)}.$$

На четвъртата стъпка в сумата направихме смяната i = m-2.

$\mathbf{D}\xi = \mathbf{E}\xi.(\xi - 1) + \mathbf{E}\xi - (\mathbf{E}\xi)^2 =$

$$= \frac{M.(M-1).n.(n-1)}{N.(N-1)} - \frac{M.n}{N}\left(\frac{M.n}{N}\right)^2 = \frac{M.n}{N}.\left[\frac{(M-1).(n-1)}{(N-1)} - 1 - \frac{M.n}{N}\right]$$

$$= \frac{M.(M-1).n.(n-1)}{N.(N-1)} - \frac{M.n}{N}\left(\frac{M.n}{N}\right)^2 = \frac{M.n}{N}.\left[\frac{(M-1).(n-1)}{(N-1)} - 1 - \frac{M.n}{N}\right]$$

$$= \frac{M.n}{N^2.(N-1)}.\left[N.(M-1).(n-1) - N.(N-1) - M.n.(N-1)\right] = \frac{M.n}{N^2.(N-1)}.\left[N.M.n - N.M - \right.$$

$$\left. -N.n + N - N^2 - N - M.n.N. + M.n\right] = \frac{M.n}{N^2.(N-1)}.\left[-N.M - N.n + N^2 + M.n\right]$$

$$= \frac{M.n}{N^2.(N-1)}.\left[N.(N-n) - M.(N-n)\right] = \frac{M.n.(N-M).(N-n)}{N^2.(N-1)} = \frac{M.n}{N}.\left[1 - \frac{M}{N}\right].\frac{N-n}{N-1}.$$

Ако означим $p = \frac{M}{N}$, $q = 1 - p$, имаме $\mathbf{E}\xi = n.p$ и $\mathbf{D}\xi = n.p.q.\frac{N-n}{N-1}$.

Оттук се вижда, че при $N \to \infty$ математическото очакване и дисперсията на хипергеометричното разпределение клонят към тези на биномното разпределение.

За да дадем интерпретация на хипергеометрично разпределената случайна величина, нека разгледаме партида от N изделия, от които M са дефектни. С цел да се реализира статистически качествен контрол се прави случайна извадка без връщане с обем n от партидата. Тогава величината “брой извадени дефектни изделия” има хипергеометрично разпределение. Действително, броят на всевъзможните извадки с обем n от партидата с N изделия е $\binom{N}{n}$. “Благоприятните”, т.е. тези които съдържат точно m дефектни изделия могат да се получат чрез комбиниране на извадка от M на m дефектни изделия и извадка от N-M на n-m изправни. Тъй като тези извадки се комбинират независимо и без ограничения, общият брой на благоприятните извадки е $\binom{M}{m}\binom{N-M}{n-m}$ $\Rightarrow$ вероятността в извадката от n изделия да има точно

m дефектни е точно $\frac{\binom{M}{m}\binom{N-M}{n-m}}{\binom{N}{n}}$.