1.Теорема Поста о полноте систем функций в алгебре логики.

Определение. Множество функций алгебры логики A называется *полной системой* (в P_2), если любую функцию алгебры логики можно выразить формулой над A.

Определение 1. Пусть A $\subset$ P^2. Тогда замыканием A называется множество всех функций алгебры логики, которые можно выразить формулами над A. Замыкание обозначается как [A].
Свойства замыкания:

1) [A] $\supseteq$ A;
2) A $\supseteq$ B ==> [A] $\supseteq$ [B], причём, если в левой части импликации строгое вложение, то из него вовсе не следует строгое вложение в правой части — верно лишь A $\supset$ B ==> [A] $\supseteq$ [B];
3) [[A]] = [A].

Определение 2. Система функций алгебры логики A называется полной, если [A] = P^2.

Определение 3. Пусть A $\subset$ P^2. Тогда система A называется замкнутым классом, если замыкание A совпадает с самим A: [A] = A.

Утверждение. Пусть A — замкнутый класс, A $\neq$ P^2 и B $\subset$ A. Тогда B — неполная система (подмножество неполной системы будет также неполной системой).
Доказательство. B $\subset$ A ==> [B] $\subset$ [A] = A $\neq$ P2 ==> [B] $\neq$ P2. Следовательно, B — неполная система. Утверждение доказано.

Теорема. Класс $T_0 = \{f(x_1, \ldots, x_n) \mid f(0, \ldots, 0) = 0\}$ —замкнутый.
Доказательство. Пусть $\{f(x_1,..,x_n), g_1(y_{11},...y_{1,m1}),\ldots,g_n(y_{n1},\ldots,y_{n,mn})\} \subset T_0$. Рассмотрим функцию $h(y_1,\ldots,y_r) = f(g_1(y_{11},\ldots,y_{1,m1}),\ldots,g_n(y_{n1},\ldots,y_{m,nn}))$. Среди переменных функций g_i могут встречаться и одинаковые, поэтому в качестве переменных функции h возьмём все различные из них. Тогда $h(0, \ldots, 0) = f(g_1(0, \ldots, 0), \ldots, g_n(0, \ldots, 0)) = f(0, \ldots, 0) = 0$, следовательно, функция h также сохраняет ноль. Рассмотрен только частный случай (без переменных в качестве аргументов). Однако, поскольку тождественная функция сохраняет ноль, подстановка простых переменных эквивалентна подстановке тождественной функции, теорема доказана.

Теорема. Класс $T_1 = \{f(x_1, \ldots, x_n) \mid f(1, 1, \ldots, 1) = 1\}$ замкнут.
Доказательство повторяет доказательство аналогичной теоремы для класса T_0.

Определение. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется линейной, если $f(x_1, \ldots, x_n) = a_0 \oplus a_1x_1 \oplus \ldots \oplus a_nx_n$, где $a_i \in \{0, 1\}$. Иными словами, в полиноме линейной функции нет слагаемых, содержащих конъюнкцию.

Теорема. Класс L замкнут.
Доказательство. Поскольку тождественная функция — линейная, достаточно рассмотреть только случай подстановки в формулы функций: пусть $f(x_1, \ldots, x_n) \in L$ и $g_i \in L$. Достаточно доказать, что $f(g_1, \ldots, g_n) \in L$
Действительно, если не учитывать слагаемых с коэффициентами $a_i = 0$, то всякую линейную функцию можно представить в виде $x_{i1} \oplus x_{i2} \oplus x_{ik} \oplus a_0$. Если теперь вместо каждого x_{ij} подставить линейное выражение,
то получится снова линейное выражение (или константа единица или нуль).

Определение. Функцией, двойственной к функции алгебры логики $f(x_1, \ldots, x_n)$, называется функция $f^*(x_1, \ldots, x_n) = \overline{f}(\overline{x_1}, \ldots, \overline{x_n})$

Теорема (принцип двойственности). Пусть $\Phi(y_1,\ldots,y_m) = f(g_1(y_{11},\ldots,y_{1k1}),\ldots,(y_{n1},\ldots,y_{nkn}))$. Тогда $\Phi^*(y_1,\ldots,y_m) = f^*(g_1^*(y_{11},\ldots,y_{1k1}),\ldots,g_n^*(y_{n1},\ldots,y_{nkn}))$.

Доказательство. Рассмотрим $\Phi^*(y_1,\ldots,y_m) = \overline{f}(g_1(\overline{y_{11}},\ldots,\overline{y_{1k1}}),\ldots,g_n(\overline{y_{n1}},\ldots,\overline{y_{nkn}})) = \overline{f}(\overline{\overline{g}_{11}},\ldots,\overline{y_{1k1}}),\ldots,\overline{\overline{g}_n}(\overline{y_{n1}},\ldots,\overline{y_{nkn}}) = \overline{f}(\overline{g_1^*}(y_{11},\ldots,y_{1k1}),\ldots,\overline{g_n^*}(y_{n1},\ldots,y_{nkn}) = f^*(g_1^*(y_{11},\ldots,y_{1k1}),\ldots,g_n^*(y_{n1},\ldots,y_{nkn}))$.

Следствие. Пусть функция $\Phi(y_1, \ldots, y_m)$ реализуется формулой над $A = \{f1, f2, \ldots\}$. Тогда если в этой формуле всюду заменить вхождения f_i на f_i^*, то получится формула, реализующая $\Phi^*(y_1, \ldots, y_m)$.
Утверждение. Для любой функции алгебры логики $f(x_1, \ldots, x_n)$ справедливо равенство $f(x_1, \ldots, x_n)=f^{**}(x_1, \ldots, x_n)$.

Определение. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется самодвойственной, если

$f(x_1, \ldots, x_n) = f^*(x_1, \ldots, x_n)$. Иначе говоря, $S = \{f \mid f = f^*\}$.
Теорема. Класс S замкнут.
Доказательство. Пусть $f(x_1, \ldots, x_n) \in S$, для любого i $g_i(y_{i1},\ldots,y_{iki}) \in S$, $i = 1, 2, \ldots, n$ и $\Phi = f(g_1(y_{11},\ldots,y_{1k1}),\ldots,g_n(y_{n1},\ldots,y_{nkn}))$. Тогда из принципа двойственности следует, что $\Phi^* = f^*(g_1^*(y_{11},\ldots,y_{1k1}),\ldots,g_n^*(y_{n1},\ldots,y_{nk}))$. Таким образом, мы получили, что $\Phi = \Phi^*$ и $\Phi \in S$.

Определение. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется монотонной, если для любых двух сравнимых наборов α и β выполняется импликация $\alpha <= \beta \Longrightarrow f(\alpha) <= f(\beta)$.

Теорема. Класс M монотонных функций замкнут.

Доказательство. Поскольку тождественная функция монотонна, достаточно проверить лишь случай суперпозиции функций. Пусть $f(x_1, \ldots, x_n) \in M$, для любого j $g_j(y_1, \ldots, y_m) \in M$ и $\Phi(y_1, \ldots, y_m) = f(g_1(y_1, \ldots, y_m), \ldots, g_n(y_1, \ldots, y_m))$. Рассмотрим произвольные наборы $\alpha = (\alpha_1, \ldots, \alpha_m)$ $\beta = (\beta_1, \ldots \beta_m)$ такие, что $\alpha <= \beta$. Обозначим $g_i(\alpha) = \gamma_i$, $g_i(\beta) = \delta_i$. Тогда для любого i имеем $\gamma_i <= \delta_i$. Тогда по определению $\gamma = (\gamma_1, \ldots, \gamma_m) <= \delta = (\delta_1, \ldots, \delta_m)$ и, в силу монотонности функции f, $f(\gamma) <= f(\delta)$. Но $\Phi(\alpha) = f(\gamma_1, \ldots, \gamma_m) = f(\gamma)$, $\Phi(\beta) = f(\delta_1, \ldots, \delta_m) = f(\delta)$ и неравенство $f(\gamma) <= f(\delta) <\Longrightarrow \Phi(\alpha) <= \Phi(\beta)$, следовательно, $\Phi \in M$.

Лемма (о несамодвойственной функции). Из любой несамодвойственной функции алгебры логики $f(x_1, \ldots, x_n)$, подставляя вместо всех переменных функции $\overline{x}$ и x, можно получить $\varphi(x) \equiv const$.

Доказательство. Пусть $f \notin S$. Тогда $\overline{f}(\overline{x_1}, \ldots, \overline{x_n}) \neq f(x_1, \ldots x_n) \Longrightarrow \exists\, \sigma = (\sigma_1, \ldots, \sigma_n)$: $\overline{f}(\overline{\sigma_1}, \ldots, \overline{\sigma_n}) \neq f(\sigma_1, \ldots \sigma_n) <\Longrightarrow f(\overline{\sigma_1}, \ldots, \overline{\sigma_n}) = f(\sigma_1, \ldots, \sigma_n)$. Построим функцию $\varphi(x)$ так: $\varphi(x) = f(x \oplus \sigma_1, \ldots, x \oplus \sigma_n)$. Действительно, $\varphi(0) = f(\sigma_1, \ldots, \sigma_n)$, $\varphi(1) = f(\overline{\sigma_1}, \ldots, \overline{\sigma_n})$ и $\varphi(0) = \varphi(1) \Longrightarrow \varphi(x) = const$. Заметим, что подстановка удовлетворяет условию теоремы, так как

$$x \oplus \sigma = \begin{cases} x, \sigma = 0 \\ \overline{x}, \sigma = 1 \end{cases}$$

Лемма (о немонотонной функции). Из любой немонотонной функции алгебры логики $f(x_1, \ldots, x_n)$, подставляя вместо всех переменных функции x, 0, 1, можно получить функцию $\varphi(x) \equiv \overline{x}$.

Доказательство. Пусть $f \notin M$. Тогда существуют такие наборы $\alpha = (\alpha_1, \ldots, \alpha_n)$ и $\beta = (\beta_1, \ldots, \beta_n)$, что $\alpha < \beta$ и $f(\alpha) > f(\beta)$. Выделим те разряды $i_1, \ldots, i_k$ наборов α и β, в которых они различаются. Очевидно, в наборе α эти разряды равны 0, а в наборе β — 1. Рассмотрим последовательность наборов $\alpha^0, \alpha^1, \ldots, \alpha^k$, таких, что $\alpha = \alpha^0 < \alpha^1 < \alpha^2 < \ldots < \alpha^k = \beta$, где α^{i+1} получается из α^i заменой одного из нулей на 1. Поскольку $f(\alpha) = 1$, а $f(\beta) = 0$, то найдутся соседние α^i и α^{i+1}, что $f(\alpha^i) = 1$ и $f(\alpha^{i+1}) = 0$. Пусть они различаются в r-ом разряде: $\alpha^i = (\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, 0, \alpha_{r+1}, \ldots, \alpha_n)$, $\alpha^{i+1} = (\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, 1, \alpha_{r+1}, \ldots, \alpha_n)$.
Тогда определим функцию $\varphi(x)$ так: $\varphi(x) = f(\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, x, \alpha_{r+1}, \ldots, \alpha_n)$. Действительно, тогда $\varphi(0) = f(\alpha^i) = 1$, $\varphi(1) = f(\alpha^{i+1}) = 0$, $\varphi(x) \equiv \overline{x}$.

Определение. Полиномом Жегалкина над $x_1, \ldots, x_n$ называется выражение вида $K_1 \oplus K_2 \oplus K_3 \oplus \ldots \oplus K_l$, либо 0, где $l >= 1$ и все K_j есть выражения вида $x_{i1} x_{i2} \ldots x_{ik}$, где все переменные различны, либо 1.

Лемма (о нелинейной функции). Из любой нелинейной функции алгебры логики $f(x_1, \ldots, x_n)$, подставляя вместо всех переменных x, $\bar{x}$, y, $\bar{y}$, 0, 1, можно получить $\varphi(x, y) = x \cdot y$ или $\varphi(x, y) = \overline{x \cdot y}$.

Доказательство. Пусть $f(x_1, \ldots, x_n) \notin L$. Рассмотрим полином Жегалкина этой функции. Из её нелинейности следует, что в нём присутствуют слагаемые вида $x_{i_1} \cdot x_{i_2} \cdot \ldots$. Не ограничивая общности рассуждений, будем считать, что присутствует произведение $x_1 \cdot x_2 \cdot \ldots$. Таким образом, полином Жегалкина этой функции выглядит так:

$$f(x_1, \ldots, x_n) = x_1 \cdot x_2 \cdot P_1(x_3, \ldots, x_n) \oplus x_1 \cdot P_2(x_3, \ldots, x_n) \oplus x_2 \cdot P_3(x_3, \ldots, x_n) \oplus P_4(x_3, \ldots, x_n),$$

причем $P_1(x_3, \ldots, x_n) \neq 0$.

Иначе говоря, $\exists\, a_3, a_4, \ldots, a_n \in E_2 = \{0, 1\}$ такие, что $P_1(a_3, a_4, \ldots, a_n) = 1$. Рассмотрим вспомогательную функцию $f(x_1, x_2, a_3, a_4, \ldots, a_n) = x_1 x_2 \cdot 1 \oplus x_1 \cdot b \oplus x_2 \cdot c \oplus d$. Тогда функция

$$f(x \oplus c, y \oplus b, a_3, a_4, \ldots, a_n) = (x \oplus c)(y \oplus b) \oplus (x \oplus c)b \oplus (y \oplus b)c \oplus d =$$

$$= xy \oplus x \cdot b \oplus y \cdot c \oplus b \cdot c \oplus x \cdot b \oplus b \cdot c \oplus y \cdot c \oplus b \cdot c \oplus d = xy \oplus (bc \oplus d) = \begin{cases} xy, & bc \oplus d = 0 \\ \overline{xy}, & bc \oplus d = 1 \end{cases}.$$

Лемма доказана.

Теорема 12 (теорема Поста). Система функций алгебры логики $A = \{f_1, f_2, \ldots\}$ является полной в P_2 тогда и только тогда, когда она не содержится целиком ни в одном из следующих классов: T_0, T_1, S, L, M.

Доказательство. Необходимость. Пусть A — полная система, N — любой из классов T_0, T_1, S, L, M и пусть $A \subseteq N$. Тогда $[A] \subseteq [N] = N \neq P_2$ и $[A] \neq P_2$. Полученное противоречие завершает обоснование необходимости.

Достаточность. Пусть $A \not\subset T_0, A \not\subset T_1, A \not\subset M, A \not\subset L, A \not\subset S$. Тогда в A существуют функции $f_0 \notin T_0, f_1 \notin T_1, f_M \notin M, f_L \notin L, f_S \notin S$. Достаточно показать, что $[A] \supseteq [f_0, f_1, f_M, f_L, f_S] = P_2$. Разобьём доказательство на три части: получение отрицания, констант и конъюнкции.

a) Получение $\bar{x}$. Рассмотрим функцию $f_0(x_1, \ldots, x_n) \notin T_0$ и введём функцию $\varphi_0(x) = f_0(x, x, \ldots, x)$. Так как функция f_0 не сохраняет нуль, $\varphi_0(0) = f(0, 0, \ldots, 0) = 1$. Возможны два случая: либо $\varphi_0(x) = \bar{x}$, либо $\varphi_0(x) \equiv 1$. Рассмотрим функцию $f_1(x_1, \ldots, x_n) \notin T_1$ и аналогичным образом введём функцию $\varphi_1(x) = f_1(x, x, \ldots, x)$. Так как функция f_1 не сохраняет единицу, $\varphi_1(1) = f(1, 1, \ldots, 1) = 0$. Возможны также два случая: либо $\varphi_1(x) = \bar{x}$, либо $\varphi_1(x) \equiv 0$. Если хотя бы в одном случае получилось искомое отрицание, пункт завершён. Если же в обоих случаях получились константы, то согласно лемме о немонотонной функции, подставляя в функцию $f_M \notin M$ вместо всех переменных константы и тождественные функции, можно получить отрицание. Отрицание получено.

b) Получение констант 0 и 1. Имеем $f_S \notin S$. Согласно лемме о несамодвойственной функции, подставляя вместо всех переменных функции f_S отрицание (которое получено в пункте *a*) и тождественную функцию, можно получить константы: $[f_S, \bar{x}\,] \ni [0, 1]$. Константы получены.

c) Получение конъюнкции $x \cdot y$. Имеем функцию $f_L \notin L$. Согласно лемме о нелинейной функции, подставляя в функцию f_L вместо всех переменных константы и отрицания (которые были получены на предыдущих шагах доказательства), можно получить либо конъюнкцию, либо отрицание конъюнкции. Однако на первом этапе отрицание уже получено, следовательно, всегда можно получить конъюнкцию: $[f_L, 0, 1, \bar{x}\,] \ni [xy, \overline{xy}\,]$. Конъюнкция получена.

В результате получено, что $\left[f_0, f_1, f_M, f_L, f_S\right] \supseteq \left[\bar{x}, xy\right] = P_2$. Последнее равенство следует из пункта 2 теоремы 4. В силу леммы 2 достаточность доказана.

2. Графы, деревья, планарные графы; их свойства. Оценка числа деревьев.

Определение 1. *Графом* называется произвольное множество элементов V и произвольное семейство E пар из V. Обозначение: $G = (V, E)$.

В дальнейшем будем рассматривать конечные графы, то есть графы с конечным множеством элементов и конечным семейством пар.

Определение 2. Если элементы из E рассматривать как неупорядоченные пары, то граф называется *неориентированным*, а пары называются *рёбрами*. Если же элементы из E рассматривать как упорядоченные, то граф *ориентированный*, а пары — *дуги*.

Определение 3. Пара вида (a, a) называется *петлёй*, если пара (a, b) встречается в семействе E несколько раз, то она называется *кратным ребром* (*кратной дугой*).

Определение 4. В дальнейшем условимся граф без петель и кратных рёбер называть *неориентированным графом* (или просто *графом*), граф без петель — *мультиграфом*, а мультиграф, в котором разрешены петли — *псевдографом*.

Определение 5. Две вершины графа называются *смежными*, если они соединены ребром.

Определение 6. Говорят, что вершина и ребро *инцидентны*, если ребро содержит вершину.

Определение 7. *Степенью вершины* ($\deg v$) называется количество рёбер, инцидентных данной вершине. Для псевдографа полагают учитывать петлю дважды.

Утверждение 1. В любом графе (псевдографе) справедливо следующее соотношение: $\sum_{i=1}^{p} \deg v_i = 2q$, где p — число вершин, а q — число рёбер.

Доказательство. Когда мы считаем степень одной вершины, мы считаем все рёбра, выходящие из неё. Вычисляя сумму всех степеней, мы получаем, что каждое ребро считается дважды, так как оно инцидентно двум вершинам (петли по определению степени также посчитаются дважды). Поэтому общая сумма будет равна удвоенному числу рёбер. Утверждение доказано.

Определение 8. Пусть множество вершин графа $V = \{v_1, v_2, \ldots, v_p\}$. Тогда *матрицей смежности* этого графа назовём матрицу $A = \|a_{ij}\|$, где $a_{ij} = 1$, если вершины v_i и v_j смежны (2, 3, … для мультиграфа или псевдографа) и 0 в противном случае, a_{ii} при этом равно числу петель в вершине v_i.

Определение 9. Два графа (или псевдографа) $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ называются *изоморфными*, если существуют два взаимно однозначных отображения $\varphi_1: V_1 \to V_2$ и $\varphi_2: E_1 \to E_2$ такие, что для любых двух вершин u и v графа G_1 справедливо $\varphi_2(u, v) = (\varphi_1(u), \varphi_1(v))$.

Определение 10 (изоморфизм графов без петель и кратных рёбер). Два графа $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ называются *изоморфными*, если существует взаимно однозначное отображение $\varphi_1: V_1 \to V_2$ такое, что $(u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$.

Определение 11. Граф $G_1 = (V_1, E_1)$ называется *подграфом* графа $G = (V, E)$, если

$$V_1 \subseteq V, E_1 \subseteq E.$$

Определение 12. *Путём* в графе $G = (V, E)$ называется любая последовательность вида

$$v_0, (v_0, v_1), v_1, (v_1, v_2), \ldots, v_{n-1}, (v_{n-1}, v_n), v_n.$$

Число n в данных обозначениях называется *длиной пути*.

Определение 13. *Цепью* называется путь, в котором нет повторяющихся рёбер.

Определение 14. *Простой цепью* называется путь без повторения вершин.

Утверждение 2. Пусть в $G = (V, E)$ $v_1 \neq v_2$ и пусть P — путь из v_1 в v_2. Тогда в P можно выделить подпуть из v_1 в v_2, являющийся простой цепью.

Доказательство. Пусть данный путь — не простая цепь. Тогда в нём повторяется некоторая вершина v, то есть он имеет вид: $P_1 = v_0C_1vC_2vC_3v_2$. Тогда он содержит подпуть $P_2 = v_0C_1vC_3v_2$. Если в P_2 повторяется некоторая вершина, то аналогично удалим ещё кусок и так далее. Процесс должен закончиться, так как P_1 — конечный путь. Утверждение доказано.

Определение 15. Путь называется *замкнутым*, если $v_0 = v_n$.

Определение 16. Путь называется *циклом*, если он замкнут, и рёбра в нём не повторяются.

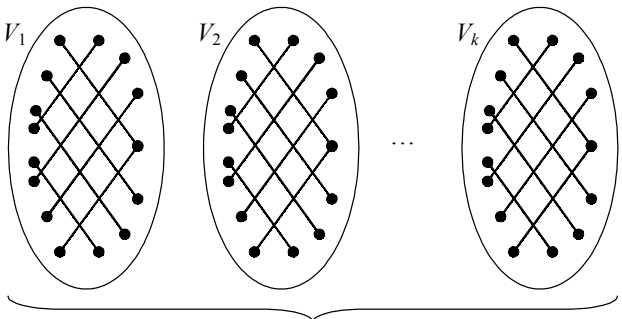
Определение 17. Путь называется *простым циклом*, если $v_0 = v_n$ и вершины не повторяются.

Определение 18. Граф $G = (V, E)$ называется *связным*, если для любых вершин $v_i, v_j \in V$ $(v_i \neq v_j)$ существует путь из v_i в v_j.

Рассмотрим отношение $v_i \to v_j$ существования пути из v_i в v_j. Оно

1) симметрично, так как $(v_i \to v_j) \Rightarrow (v_j \to v_i)$,
2) транзитивно, так как $(v_i \to v_j)\ \&\ (v_j \to v_k) \Rightarrow (v_i \to v_k)$,
3) рефлексивно, так как $\forall i\ (v_i \to v_i)$.

Таким образом, получено, что $v_i \to v_j$ — отношение эквивалентности и множество вершин разбивается на конечное число классов эквивалентности: $V \to V_1 \cup V_2 \cup \ldots \cup V_k$, $V_i \cap V_j = \varnothing \Leftarrow$ $\Leftarrow i \neq j$. При этом граф G разбивается на связные подграфы, которые называются *компонентами связности*.



Связные компоненты графа G

Определение 1. *Деревом* называется связный граф без циклов.

Определение 2. Подграф $G_1 = (V_1, E_1)$ графа $G = (V, E)$, называется *остовным деревом* в графе $G = (V, E)$, если $G_1 = (V_1, E_1)$ — дерево и $V_1 = V$.

Лемма 1. Если граф $G = (V, E)$ связный и ребро (a, b) содержится в некотором цикле в графе G, то при выбрасывании из графа G ребра (a, b) снова получится связный граф.

Доказательство. Это утверждение следует из того, что при выбрасывании из графа G ребра (a, b) вершины a и b всё равно остаются в одной связной компоненте, поскольку из a в b можно пройти по оставшейся части цикла. Лемма доказана.

Теорема 1. Любой связный граф содержит хотя бы одно остовное дерево.

Доказательство. Если в G нет циклов, то G является искомым остовным деревом. Если в G есть циклы, то удалим из G какое-нибудь ребро, входящее в цикл. Получится некоторый подграф G_1. По лемме 1 G_1 — связный граф. Если в G_1 нет циклов, то G_1 и есть искомое остовное дерево, иначе продолжим этот процесс. Процесс должен завершиться, так как E — конечное множество. Теорема доказана.

Лемма 2. Если к связному графу добавить новое ребро на тех же вершинах, то появится цикл.

Доказательство. Рассмотрим произвольный связный граф $G = (V, E)$. Пусть $u \in V$, $v \in V$, $(u, v) \notin E$. Так как G — связный граф, то в нём есть путь из v в u. Тогда в G есть и простая цепь C из v в u. Поэтому в полученном графе есть цикл C, (u, v), v. Лемма доказана.

Лемма 3. Пусть в графе $G = (V, E)$ p вершин и q рёбер. Тогда в G не менее $p - q$ связных компонент. Если при этом в G нет циклов, то G состоит ровно из $p - q$ связных компонент.

Доказательство. Пусть к некоторому графу H, содержащему вершины u и v, добавляется ребро (u, v). Тогда если u и v лежат в разных связных компонентах графа H, то число связных компонент уменьшится на 1. Если u, v лежат в одной связной компоненте графа H, то число связных компонент не изменится. В любом случае, число связных компонент уменьшается не более чем на 1. Значит, при добавлении q рёбер число связных компонент уменьшается не более чем на q. Так как граф G получается из графа $G_1 = (V, \varnothing)$ добавлением q рёбер, то в G не менее $p - q$ связных компонент. Пусть теперь в G нет циклов, и пусть в процессе получения G из G_1 добавляется ребро (u, v). Если бы u, v лежали уже в одной связной компоненте, то в G, согласно лемме 2, возникал бы цикл. Следовательно, u, v лежат в разных связных компонентах и при добавлении ребра (u, v) число связных компонент уменьшается ровно на 1. Тогда G состоит ровно из $p - q$ связных компонент. Лемма доказана.

Теорема 2 (о различных определениях дерева). Следующие пять определений эквивалентны (p — число вершин, q — число рёбер):

1) G — дерево;
2) G — без циклов и $q = p - 1$;
3) G — связный граф и $q = p - 1$;
4) G — связный граф, но при удалении любого ребра становится несвязным;
5) G — без циклов, но при добавлении любого ребра на тех же вершинах появляется цикл.

Доказательство. Докажем следующие переходы: 1) $\Rightarrow$ 2) $\Rightarrow$ 3) $\Rightarrow$ 4) $\Rightarrow$ 5) $\Rightarrow$ 1), откуда будет следовать, что из любого условия вытекает любое другое.

1) $\Rightarrow$ 2): так как G — связный граф и G не содержит циклов, то $p - q = 1$ по лемме 3. Отсюда $q = p - 1$.

2) $\Rightarrow$ 3): по лемме 3 в G число связных компонент равно $p - q = 1$, то есть G — связный граф.

3) $\Rightarrow$ 4): при удалении одного ребра $p - q = 2$. Тогда по лемме 3 число связных компонент не менее чем $p - q = 2$.

4) $\Rightarrow$ 5): если G имеет цикл, то согласно лемме 1 можно выбросить одно ребро так, что граф останется связным. Согласно лемме 2, если добавить любое новое ребро к связному графу G на тех же вершинах, то появится цикл.

5) $\Rightarrow$ 1): если G не связный граф и вершины u, v лежат в разных связных компонентах графа G, то добавление к G ребра (u, v), очевидно, не порождает циклов, что противоречит 5). Отсюда следует, что G — связный граф. Теорема доказана.

Определение 1. Любое дерево, в котором выделена одна вершина, называемая *корнем*, называется *корневым деревом*.

Определение 2. 1) Граф, состоящий из одной вершины, которая выделена, называется *корневым деревом*.

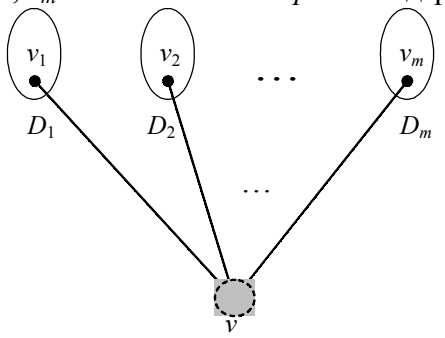
2) Пусть имеются корневые деревья D_1, D_2, …, D_m с корнями v_1, v_2, …, v_m, $D_i = (V_i, E_i)$, $V_i \cap V_j = \varnothing$ $(i \neq j)$. Тогда граф $D = (V, E)$, полученный следующим образом:

$$V = V_1 \cup V_2 \cup \ldots \cup V_m \cup \{v\}\ (v \notin V_i, \forall i\ ),\ E = E_1 \cup E_2 \cup \ldots \cup E_m \cup \{(v, v_1), (v, v_2), \ldots, (v, v_m)\}$$

и в котором выделена вершина v, называется *корневым деревом*.

3) Только те объекты являются *корневыми деревьями*, которые можно построить согласно пунктам 1) и 2).

При таком определении $D_1, D_2, \ldots, D_m$ называются *поддеревьями* дерева D.



Утверждение. Определения 1 и 2 эквивалентны.

Определение 3. *Упорядоченным корневым деревом* называется корневое дерево, в котором

1) задан порядок поддеревьев и
2) каждое поддерево D_i является упорядоченным поддеревом.

Дерево с одной вершиной также является упорядоченным поддеревом.

Теорема 3. Число упорядоченных корневых деревьев с q рёбрами не превосходит 4^q.

Доказательство. Рассмотрим алгоритм обхода упорядоченного дерева, называемого «поиском в глубину». Этот обход описывается рекурсивно следующим образом:

1) Начать с корня. Пока есть поддеревья выполнять:
2) перейти в корень очередного поддерева, обойти это поддерево «в глубину».
3) Вернуться в корень исходного поддерева.

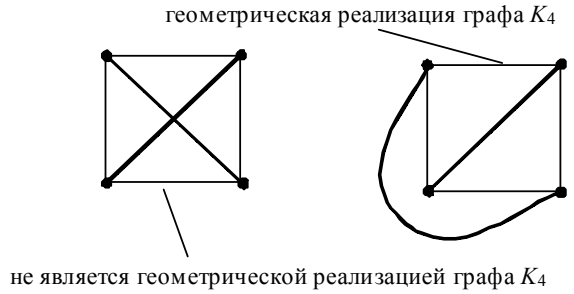
В результате обход «в глубину» проходит по каждому ребру дерева ровно 2 раза: один раз при переходе в очередное поддерево, второй раз при возвращении из этого поддерева. В соответствии с обходом «в глубину» будем строить последовательность из нулей и единиц, записывая на каждом шаге нуль или единицу, причём нуль будем записывать, если происходит переход в очередное поддерево, а единицу, если мы возвращаемся из поддерева. Получим последовательность из 0 и 1 длины $2q$, которую назовём кодом дерева. По этому коду однозначно восстанавливается дерево, поскольку каждый очередной разряд однозначно указывает, начинать ли строить новое очередное поддерево или возвращаться на ярус ближе к корню. Таким образом, упорядоченных корневых деревьев с q рёбрами не больше, чем последовательностей из 0 и 1 длины $2q$, а их число равно $2^{2q} = 4^q$. Теорема доказана.

Изоморфизм корневых деревьев определяется так же, как и изоморфизм графов, но с дополнительным требованием: корень должен отображаться в корень. Для упорядоченных корневых деревьев также требуется сохранение порядка поддеревьев.

Следствие. Число неизоморфных корневых деревьев с q рёбрами и число неизоморфных деревьев с q рёбрами не превосходит 4^q.

Доказательство. Выделяя в неизоморфных деревьях по одной вершине, мы получим неизоморфные корневые деревья. Упорядочивая поддеревья в неизоморфных корневых деревьях, мы получим различные упорядоченные корневые деревья. Поэтому число неизоморфных деревьев с q рёбрами не превосходит числа неизоморфных корневых деревьев с q рёбрами, которое, в свою очередь, не превосходит числа различных упорядоченных корневых деревьев с q рёбрами. Отсюда и из теоремы следует утверждение следствия. Следствие доказано.

Определение. Пусть задан некоторый неориентированный граф $G = (V, E)$. Пусть любой вершине v_i графа G сопоставлена некоторая точка a_i: $v_i \to a_i$, $a_i \neq a_j$ $(i \neq j)$, а любому ребру $e = (a, b)$ сопоставлена некоторая непрерывная кривая L, соединяющая точки a_i и a_j и не проходящая через другие точки a_k $(k \neq i, j)$. Тогда если все кривые, сопоставленные рёбрам, не имеют общих точек, кроме концевых, то говорят, что задана *геометрическая реализация графа G*.



Теорема 4. Для любого графа существует его реализация в трёхмерном пространстве.

Доказательство. Возьмём в пространстве любую прямую l и разместим на ней все вершины графа G. Пусть в G имеется q рёбер. Проведём связку из q различных полуплоскостей через l. После этого каждое ребро графа G можно изобразить линией в своей полуплоскости и они, очевидно, не будут пересекаться. Теорема доказана.

Определение 1. Граф называется *планарным*, если существует его геометрическая реализация на плоскости.

Определение 2. Если имеется планарная реализация графа и мы «разрежем» плоскость по всем линиям этой планарной реализации, то плоскость распадётся на части, которые называются *гранями* этой планарной реализации (одна из граней бесконечна, она называется *внешней гранью*).

Теорема 5 (формула Эйлера). Для любой планарной реализации связного планарного графа $G = (V, E)$ с p вершинами, q рёбрами и r гранями выполняется равенство: $p - q + r = 2$.

Доказательство. Докажем теорему при фиксированном p индукцией по q. Так как G — связный граф, то $q \geq p - 1$.

a) Базис индукции: $q = p - 1$. Так как G — связный и $q = p - 1$, то согласно пункту 3 теоремы 2 G — дерево, то есть, в G нет циклов. Тогда $r = 1$. Отсюда $p - q + r = p - (p - 1) + 1 = 2$.

b) Пусть для q: $p - 1 \leq q < q_0$ теорема справедлива. Докажем, что для $q = q_0$ она также справедлива. Пусть G — связный граф с p вершинами и q_0 рёбрами и пусть в его планарной реализации r граней. Так как $q_0 > p - 1$, то G — не дерево. Следовательно, в G есть цикл. Пусть ребро e входит в цикл. Тогда к нему с двух сторон примыкают разные грани. Удалим ребро e из G. Тогда две грани сольются в одну, а полученный граф G_1 останется связным. При этом получится планарная реализация графа G_1 с p вершинами и $q_0 - 1$ рёбрами и $r - 1$ гранями. Так как $q_0 - 1 < q_0$, то, по предположению индукции, для G_1 справедлива формула Эйлера, то есть $p - (q_0 - 1) + (r - 1) = 2$, откуда $p - q_0 + r = 2$. Что и требовалось доказать.

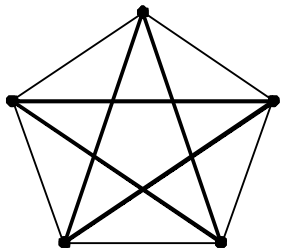
Следствие 1. Формула Эйлера справедлива и для геометрической реализации связных графов на сфере.

Доказательство. Пусть связный граф G с p вершинами и q рёбрами реализован на сфере S так, что число граней равно r. Пусть точка A на сфере не лежит на линиях этой геометрической реализации. Пусть P — некоторая плоскость. Поставим сферу S на плоскость P так, чтобы точка A была самой удалённой от плоскости. Спроектируем S на P центральным проектированием с центром в точке A. Тогда на плоскости P мы получим геометрическую реализацию связного графа с p вершинами и q рёбрами, причём число граней будет равно r (грань на сфере, содержащая A, отображается на внешнюю грань на плоскости). По теореме получаем $p - q + r = 2$. Следствие доказано.

Следствие 2. Для любого выпуклого многогранника справедливо равенство $p - q + r = 2$, где p — число вершин, q — число рёбер, r — число граней.

Доказательство. Пусть выпуклый многогранник M имеет p вершин, q рёбер и r граней. Пусть O — внутренняя точка многогранника. Разместим сферу S с центром в точке O настолько большого радиуса, чтобы M целиком содержался в S. Рассмотрим центральное проектирование с центром в точке O, и спроектируем вершины и рёбра M на S. Тогда на S мы получим геометрическую реализацию некоторого связного графа с p вершинами, q рёбрами и r гранями. Отсюда согласно следствию 1 $p - q + r = 2$. Следствие 2 доказано.

Определение 1. Графом K_5 называется граф с пятью вершинами, в котором каждая пара вершин соединена ребром.

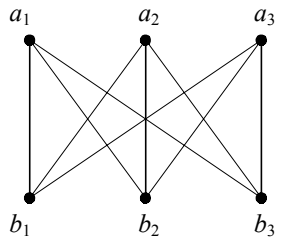


K_5

Теорема 6. Граф K_5 не планарен.

Доказательство. Допустим, что для графа K_5 существует планарная реализация. Так как граф K_5 связен, то для этой планарной реализации справедлива формула Эйлера $p - q + r = 2$. Поскольку в графе K_5 имеем $p = 5$ и $q = 10$, то число всех граней должно равняться $r = 2 - p + q = 7$. Пусть грани занумерованы 1, 2, …, r и пусть при обходе i-ой грани по периметру (по её краю) проходится q_i рёбер. Так как при этом каждое ребро обходится дважды (оно является стороной для двух граней), то $\sum_{i=1}^{r} q_i = 2q = 20$. Но в каждой грани не менее трёх сторон. Поэтому $q_i \geq 3$ для всех i. Отсюда $\sum_{i=1}^{r} q_i \geq 3r = 21$. Получаем $20 \geq 21$ — противоречие. Значит, для графа K_5 не существует планарной реализации.

Определение 2. Графом $K_{3,3}$ называется граф с шестью вершинами a_1, a_2, a_3, b_1, b_2, b_3, в котором каждая вершина a_i соединена ребром с каждой вершиной b_j и других рёбер нет.



$K_{3,3}$

Теорема 7. Граф $K_{3,3}$ не планарен.

Доказательство. Допустим, что для графа $K_{3,3}$ существует планарная реализация. Так как граф $K_{3,3}$ связен, то для этой планарной реализации справедлива формула Эйлера $p - q + r = 2$. Поскольку в графе $K_{3,3}$ имеем $p = 6$ и $q = 9$, то число всех граней должно равняться $r = 2 - p + q = 5$. Так же, как в доказательстве предыдущей теоремы, получаем, что $\sum_{i=1}^{r} q_i = 2q = 18$, где q_i — число сторон в i-ой грани. Но в графе $K_{3,3}$ нет циклов длины 3. Поэтому в каждой грани не менее 4 сторон. Следовательно, $q_i \geq 4$ для всех i. Отсюда $\sum_{i=1}^{r} q_i \geq 4r = 20$. Получаем $18 \geq 20$ — противоречие. Значит, для графа $K_{3,3}$ не существует планарной реализации.

Определение 3. *Подразделением ребра* (a, b) называется операция, состоящая в следующих действиях:

1) удаление (a, b),
2) добавление новой вершины c,
3) добавление рёбер (a, c) и (c, b).

Определение 4. Граф H называется *подразделением графа* G, если H можно получить из G путём конечного числа подразделений своих рёбер.

Определение 5. Два графа называются *гомеоморфными*, если существуют их подразделения, которые изоморфны.

Теорема 8 (Понтрягина-Куратовского). Граф является планарным тогда и только тогда, когда он не содержит ни одного подграфа, гомеоморфного графам K_5 или $K_{3,3}$.

Доказательство. Необходимость. Пусть G — планарный. Допустим, что он содержит подграф G_1, гомеоморфный графу K_5 или $K_{3,3}$. Рассмотрим планарную реализацию графа G. Удалив лишние вершины и рёбра, мы получим планарную реализацию подграфа G_1. Но G_1 геометрически — это граф K_5 или $K_{3,3}$ с точками на рёбрах. Если проигнорировать эти точки, то мы получим планарную реализацию графа K_5 или $K_{3,3}$. Но это невозможно в силу теорем 1 и 2. Необходимость доказана.

Достаточность без доказательства.

3. Логика 1-го порядка. Выполнимость и общезначимость. Общая схема метода резолюций.

Базовые символы. Предметные переменные

Предметные константы Функциональные символы Предикатные символы

$\mathrm{Var} = \{x_1, x_2, \ldots, x_k, \ldots\}$; $\mathrm{Const} = \{c_1, c_2, \ldots, c_l, \ldots\}$;

$\mathrm{Func} = \{f^{(n1)}, f^{(n2)}, \ldots, f^{(nr)}, \ldots\}$; 12^r

$\mathrm{Pred} = \{P^{(m1)}, P^{(m2)}, \ldots, P^{(ms)}, \ldots\}$.

Тройка $\langle \mathrm{Const}, \mathrm{Pred}, \mathrm{Func} \rangle$ называется сигнатурой алфавита.

Логические связки и кванторы.

Конъюнкция	(логическое И)	$\&$
Дизъюнкция	(логическое ИЛИ)	$\vee$
Отрицание	(логическое НЕ)	$\neg$
Импликация	(логическое ЕСЛИ-ТО)	$\rightarrow$.

Квантор всеобщности	(«для каждого»)	$\forall$
Квантор существования	(«хотя бы один»)	$\exists$

Знаки препинания.

Разделитель	,
Скобки	( )

Терм — это

x	, если $x \in Var$	x — переменная;
c	, если $c \in Const$	c — константа;
$f^{(n)}(t_1, t_2, \ldots, t_n)$	, если $f^{(n)} \in Func$, $t_1, t_2, \ldots, t_n$ — термы	составной терм.

$Term$ — множество термов заданного алфавита.

Var_t — множество переменных, входящих в состав терма t.

$t(x_1, x_2, \ldots, x_n)$ — запись обозначающая терм t, у которого $Var_t \subseteq \{x_1, x_2, \ldots, x_n\}$.

Формула — это

атомарная формула

$P^{(m)}(t_1, t_2, \ldots, t_m)$, если $P^{(m)} \in Pred$, $\{t_1, t_2, \ldots, t_m\} \subseteq Term$;

составная формула

$(\varphi \& \psi)$, если φ, ψ — формулы;

$(\varphi \vee \psi)$

$(\varphi \rightarrow \psi)$

$(\neg\varphi)$

$(\forall x\varphi)$
$(\exists x\varphi)$, если $x \in Var$, φ — формула.

$Form$ — множество всех формул заданного алфавита.

Квантор связывает ту переменную, которая следует за ним.

Вхождение переменной в области действия квантора, связывающего эту переменную, называется **связанным** .

Вхождение переменной в формулу, не являющееся связанным, называется **свободным** .

Переменная называется **свободной** , если она имеет свободное вхождение в формулу.

Var_φ — множество свободных переменных формулы φ.

$$\varphi = P^{(m)}(t_1, t_2, \ldots, t_m) \quad Var_\varphi = \bigcup_{i=1}^{m} Var_{t_i};$$

$$\varphi = (\psi_1 \& \psi_2) \qquad Var_\varphi = Var_{\psi_1} \cup Var_{\psi_2};$$
$$\varphi = (\psi_1 \vee \psi_2)$$
$$\varphi = (\psi_1 \to \psi_2)$$
$$\varphi = (\neg\psi) \qquad Var_\varphi = Var_\psi;$$

$$\varphi = (\forall x\psi) \qquad Var_\varphi = Var_\psi \setminus \{x\}.$$
$$\varphi = (\exists x\psi)$$

$\varphi(x_1, x_2, \ldots, x_n)$ — запись, обозначающая формулу φ, у которой $Var_\varphi \subseteq \{x_1, x_2, \ldots, x_n\}$.

Если $Var_\varphi = \emptyset$, то формула φ называется **замкнутой формулой** , или **предложением** .

$CForm$ — множество всех замкнутых формул.

Соглашение о приоритете логических операций

В порядке убывания приоритета связки и кванторы располагаются так:

$\neg$, $\forall$, $\exists$

$\&$

$\vee$

$\to$

Интерпретация сигнатуры $\langle Const, Func, Pred\rangle$ — это алгебраическая система $I = \langle D_I, \overline{Const}, \overline{Func}, \overline{Pred}\rangle$, где

- D_I — непустое множество, которое называется областью интерпретации , предметной областью , или универсумом ;
- $\overline{Const} : Const \to D_I$ — оценка констант , сопоставляющая каждой константе c элемент (предмет) $\bar{\mathbf{c}}$ из области интерпретации;
- $\overline{Func} : Func^{(n)} \to (D_I^n \to D_I)$ — оценка функциональных символов , сопоставляющая каждому функциональному символу $f^{(n)}$ местности n всюду определенную n-местную функцию $\bar{\mathbf{f}}^{(n)}$ на области интерпретации;

- $\overline{Pred} : Pred^{(m)} \to (D_I^m \to \{\mathbf{true}, \mathbf{false}\})$ — оценка предикатных символов, сопоставляющая каждому предикатному символу $P^{(m)}$ местности m всюду определенное m-местное отношение $\bar{\mathbf{P}}^{(m)}$ на области интерпретации.

Пусть заданы интерпретация $I = \langle D_I, \overline{Const}, \overline{Func}, \overline{Pred} \rangle$, терм $t(x_1, x_2, \ldots, x_n)$ и набор $d_1, d_2, \ldots, d_n$ элементов (предметов) из области интерпретации D_I.
Значение $t(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$ терма $t(x_1, x_2, \ldots, x_n)$ на наборе $d_1, d_2, \ldots, d_n$ определяется рекурсивно.

- Если $t(x_1, x_2, \ldots, x_n) = x_i$, то
  $t(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n] = d_i$;
- Если $t(x_1, x_2, \ldots, x_n) = c$, то
  $t(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n] = \bar{\mathbf{c}}$;
- Если $t(x_1, x_2, \ldots, x_n) = f(t_1, \ldots, t_k)$, то
  $t(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n] =$
  $\bar{\mathbf{f}}(t_1[d_1, d_2, \ldots, d_n], \ldots, t_k[d_1, d_2, \ldots, d_n])$.

Значение формул в интерпретации определяется при помощи отношения выполнимости $\models$.

Пусть заданы интерпретация $I = \langle D_I, \overline{Const}, \overline{Func}, \overline{Pred} \rangle$, формула $\varphi(x_1, x_2, \ldots, x_n)$ и набор $d_1, d_2, \ldots, d_n$ элементов (предметов) из области интерпретации D_I.
Отношение выполнимости $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$ формулы φ в интерпретации I на наборе $d_1, d_2, \ldots, d_n$ определяется рекурсивно.

- Если $\varphi(x_1, x_2, \ldots, x_n) = P(t_1, \ldots, t_m)$, то
  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
  $\Longleftrightarrow$
  $\bar{\mathbf{P}}(t_1[d_1, d_2, \ldots, d_n], \ldots, t_m[d_1, d_2, \ldots, d_n]) = \mathbf{true}$;
- Если $\varphi(x_1, x_2, \ldots, x_n) = \psi_1 \& \psi_2$, то
  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
  $\Longleftrightarrow$
  $\begin{cases} I \models \psi_1(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n] \\ I \models \psi_2(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n] \end{cases}$
- Если $\varphi(x_1, x_2, \ldots, x_n) = \psi_1 \vee \psi_2$, то
  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
  $\Longleftrightarrow$
  $I \models \psi_1(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
  или
  $I \models \psi_2(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
- Если $\varphi(x_1, x_2, \ldots, x_n) = \psi_1 \to \psi_2$, то
  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
  $\Longleftrightarrow$
  $I \not\models \psi_1(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$
  или
  $I \models \psi_2(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$

- Если $\varphi(x_1, x_2, \ldots, x_n) = \neg\psi$, то

  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$

  $\Longleftrightarrow$

  $I \not\models \psi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$

- Если $\varphi(x_1, x_2, \ldots, x_n) = \forall x_0\ \psi(x_0, x_1, x_2, \ldots, x_n)$, то

  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$

  $\Longleftrightarrow$

  для **любого** элемента d_0, $d_0 \in D_I$, имеет место

  $I \models \psi(x_0, x_1, x_2, \ldots, x_n)[d_0, d_1, d_2, \ldots, d_n]$

- Если $\varphi(x_1, x_2, \ldots, x_n) = \exists x_0\ \psi(x_0, x_1, x_2, \ldots, x_n)$, то

  $I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$

  $\Longleftrightarrow$

  для **некоторого** элемента d_0, $d_0 \in D_I$, имеет место

  $I \models \psi(x_0, x_1, x_2, \ldots, x_n)[d_0, d_1, d_2, \ldots, d_n]$

Формула $\varphi(x_1, \ldots, x_n)$ называется выполнимой в интерпретации I, если существует такой набор элементов $d_1, \ldots, d_n \in D_I$, для которого имеет место $I \models \varphi(x_1, \ldots, x_n)[d_1, \ldots, d_n]$.

Формула $\varphi(x_1, \ldots, x_n)$ называется истинной в интерпретации I, если для любого набора элементов $d_1, \ldots, d_n \in D_I$ имеет место $I \models \varphi(x_1, \ldots, x_n)[d_1, \ldots, d_n]$.

Формула $\varphi(x_1, \ldots, x_n)$ называется выполнимой , если есть интерпретация I, в которой эта формула выполнима.

Формула $\varphi(x_1, \ldots, x_n)$ называется общезначимой (или тождественно истинной), если эта формула истинна в любой интерпретации.

Формула $\varphi(x_1, \ldots, x_n)$ называется противоречивой (или невыполнимой), если она не является выполнимой.

Пусть Γ — некоторое множество замкнутых формул, $\Gamma \subseteq CForm$. Тогда каждая интерпретация I, в которой выполняются все формулы множества Γ, называется моделью для множества Γ.

Пусть Γ — некоторое множество замкнутых формул, и φ — замкнутая формула. Формула φ называется логическим следствием множества предложений (базы знаний) Γ, если каждая модель для множества формул Γ является моделью для формулы φ, т. е. для любой интерпретации I верно

$$I \models \Gamma \iff I \models \varphi$$

Запись $\Gamma \models \varphi$ обозначает, что φ — логическое следствие Γ .

Для обозначения общезначимости формулы φ будем использовать запись $\models \varphi$.

Теорема о логическом следствии

Пусть $\Gamma = \{\psi_1, \ldots, \psi_n\} \subseteq CForm$, $\varphi \in CForm$. **Тогда**

$$\Gamma \models \varphi \iff \models \psi_1 \& \ldots \& \psi_n \to \varphi.$$

Доказательство. $\Rightarrow$ Пусть I — произвольная интерпретация.

Если $I \not\models \psi_1 \& \ldots \& \psi_n$, то $I \models \psi_1 \& \ldots \& \psi_n \to \varphi$.

Если $I \models \psi_1 \& \ldots \& \psi_n$, то $I \models \psi_i$, $1 \le i \le n$, т. е. I — модель для Γ. Поскольку $\Gamma \models \varphi$, получаем $I \models \varphi$.
Значит, $I \models \psi_1 \& \ldots \& \psi_n \to \varphi$.

Таким образом, для любой интерпретации I имеет место $I \models \psi_1 \& \ldots \& \psi_n \to \varphi$.

Значит, $\psi_1 \& \ldots \& \psi_n \to \varphi$ — общезначимая формула.

Доказательство. $\Leftarrow$ Пусть I — модель для множества предложений Γ, т. е. $I \models \psi_i$, $1 \le i \le n$.

Тогда $I \models \psi_1 \& \ldots \& \psi_n$.

Так как $\psi_1 \& \ldots \& \psi_n \to \varphi$ — общезначимая формула, имеет место $I \models \psi_1 \& \ldots \& \psi_n \to \varphi$.

Значит, $I \models \varphi$.

Так как I — произвольная модель для Γ, приходим к заключению $\Gamma \models \varphi$.

Общезначимые формулы — это каналы причинно-следственной связи, по которым передаются знания, представленные в виде логических формул, преобразуясь при этом из одной формы в другую.
Практически важно уметь определять эти каналы и настраивать их на извлечение нужных знаний.

- База знаний — множество предложений Γ;
- Запрос к базе знаний — предложение φ;
- Получение ответа на запрос — проверка логического следствия $\Gamma \models \varphi$.

Если Γ — конечное множество, то проверка логического следствия сводится к проверке общезначимости формулы

$$\psi_1 \& \ldots \& \psi_n \to \varphi \ .$$

ОБЩАЯ СХЕМА МЕТОДА РЕЗОЛЮЦИЙ

Задача проверки общезначимости формул логики предикатов.

$$\models \varphi \, ?$$

Этап 1. Сведение проблемы общезначимости к проблеме противоречивости.

$$\varphi \ \rightsquigarrow \ \varphi_0 = \neg\varphi$$

φ общезначима $\iff$ φ_0 противоречива.

Этап 2. Построение предваренной нормальной формы (ПНФ).

$$\varphi_0 \ \rightsquigarrow \ \varphi_1 = \ Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \ (D_1 \& D_2 \& \ldots \& D_N)$$

φ_0 равносильна φ_1, т. е. $I \models \varphi_0 \ \Leftrightarrow \ I \models \varphi_1$.

Этап 3. Построение сколемовской стандартной формы (ССФ).

$$\varphi_1 \rightsquigarrow \varphi_2 = \forall x_{i_1} \forall x_{i_2} \ldots \forall x_{i_k} \, (D_1 \& D_2 \& \ldots \& D_N)$$

φ_1 противоречива $\Longleftrightarrow$ φ_2 противоречива.

Этап 4. Построение системы дизъюнктов.

$$\varphi_2 \rightsquigarrow S_\varphi = \{D_1, \ D_2, \ \ldots, D_N\},$$

где $D_i = L_{i1} \vee L_{i2} \vee \cdots \vee L_{im_i}$.

φ_2 противоречива $\Longleftrightarrow$ система дизъюнктов S_φ противоречива.

Этап 5. Резолютивный вывод тождественно ложного (противоречивого) дизъюнкта $\square$ из системы S_φ.

Правило резолюции *Res* : $\dfrac{D_1 = D_1' \vee L, \ D_2 = D_2' \vee \neg L}{D_0 = D_1' \vee D_2'}$.

Дизъюнкт D_0 называется резольвентой дизъюнктов D_1 и D_2.

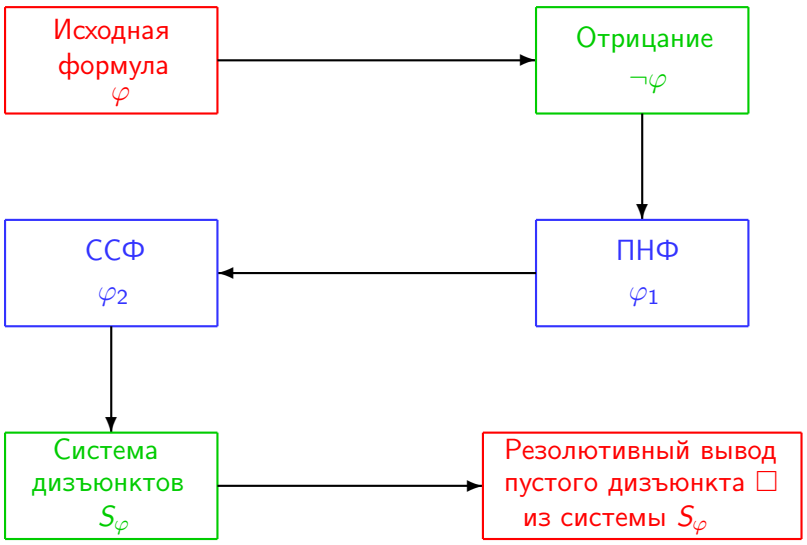
Резольвенты строят, пока не будет получен пустой дизъюнкт $\square$.

Это возможно в случае $D_1 = L$, $D_2 = \neg L$:

$$\frac{D_1 = L, \ D_2 = \neg L}{D_0 = \square}$$

Система дизъюнктов S_φ противоречива $\Leftrightarrow$ из S_φ резолютивно выводим пустой дизъюнкт $\square$.

ИТОГ. Формула φ общезначима $\Leftrightarrow$ из системы дизъюнктов S_φ резолютивно выводим пустой дизъюнкт $\square$.



Введем вспомогательную логическую связку эквиваленции $\equiv$.

Выражение $\varphi \equiv \psi$ — это сокращенная запись формулы $(\varphi \to \psi)\&(\psi \to \varphi)$.

Формулы φ и ψ будем называть равносильными , если формула $\varphi \equiv \psi$ общезначима, т. е. $\models (\varphi \to \psi)\&(\psi \to \varphi)$.

Запись $\varphi[\psi]$ означает, что формула φ содержит подформулу ψ. Запись $\varphi[\psi/\chi]$ обозначает формулу, которая образуется из формулы φ заменой некоторых (не обязательно всех) вхождений подформулы ψ на формулу χ.

Теорема

$$\models \psi \equiv \chi \quad \Longrightarrow \quad \models \varphi[\psi] \equiv \varphi[\psi/\chi]$$

Замкнутая формула φ называется предваренной нормальной формой (ПНФ) , если

$$\varphi \;=\; Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \; M(x_1, x_2, \ldots, x_n),$$

где

- $Q_1 x_1 \; Q_2 x_2 \; \ldots \; Q_n x_n$ — кванторная приставка , состоящая из кванторов $Q_1, Q_2, \ldots, Q_n$,
- $M(x_1, x_2, \ldots, x_n)$ — матрица — бескванторная конъюнктивная нормальная форма (КНФ), т. е.
  $$M(x_1, x_2, \ldots, x_n) = D_1 \;\&\; D_2 \;\&\ldots\&\; D_N,$$
  где $D_i = L_{i1} \vee L_{i2} \vee \cdots \vee L_{ik_i}$ — дизъюнкты , состоящие из литер $L_{ij} = A_{ij}$ или $L_{ij} = \neg A_{ij}$, где A_{ij} — атомарная формула.

Теорема о ПНФ

Для любой замкнутой формулы φ существует равносильная предваренная нормальная форма ψ.

Предваренная нормальная форма вида

$$\varphi \;=\; \forall x_{i_1} \forall x_{i_2} \ldots \forall x_{i_m} \; M(x_{i_1}, x_{i_2}, \ldots, x_{i_m}),$$

в которой кванторная приставка не содержит кванторов $\exists$, называется сколемовской стандартной формой (ССФ) .

Теорема о ССФ

Для любой замкнутой формулы φ существует такая сколемовская стандартная форма ψ, что

$$\varphi \text{ выполнима} \quad \Longleftrightarrow \quad \psi \text{ выполнима}.$$

Утверждение

$$\models \forall x \; (\varphi \;\&\; \psi) \;\equiv\; \forall x \; \varphi \;\&\; \forall x \psi$$

Иначе говоря, кванторы $\forall$ можно равномерно распределить по сомножителям (дизъюнктам) КНФ.

Теорема

Сколемовская стандартная форма

$$\varphi \;=\; \forall x_1 \forall x_2 \ldots \forall x_m \; (D_1 \;\&\; D_2 \;\&\ldots\&\; D_N)$$

невыполнима тогда и только тогда, когда множество формул

$$S_\varphi = \{\forall x_1 \forall x_2 \ldots \forall x_m D_1, \forall x_1 \forall x_2 \ldots \forall x_m D_2, \ldots, \forall x_1 \forall x_2 \ldots \forall x_m D_N\}$$

не имеет модели.

Каждая формула множества S_φ имеет вид

$$\forall x_1 \forall x_2 \ldots \forall x_m (L_1 \vee L_2 \vee \cdots \vee L_k)$$

и называется дизъюнктом .
В дальнейшем (по умолчанию) будем полагать, что все переменные дизъюнкта связаны кванторами $\forall$, и кванторную приставку выписывать **не будем**.

Каждый дизъюнкт состоит из литер $L_1, L_2, \ldots, L_k$. Литера — это либо атом, либо отрицание атома.

Особо выделен дизъюнкт, в котором **нет ни одной литеры**. Такой дизъюнкт называется пустым дизъюнктом и обозначается $\square$. Пустой дизъюнкт $\square$ тождественно ложен.

Систему дизъюнктов, не имеющую моделей, будем называть невыполнимой , или противоречивой системой дизъюнктов.

Задача проверки общезначимости формул логики предикатов.

$$\models \varphi \,?$$

φ общезначима $\iff$ $\varphi_0 = \neg\varphi$ невыполнима.

φ_0 невыполнима $\iff$ ПНФ φ_1 невыполнима.

φ_1 невыполнима $\iff$ ССФ φ_2 невыполнима.

φ_2 невыполнима $\iff$ система дизъюнктов S_φ невыполнима.

Пусть задано выражение E и подстановка θ.

Подстановка θ : **Var** $\to$ **Var** называется переименованием , если θ — биекция.

Если θ — переименование, то пример $E\theta$ называется вариантом выражения E.

Подстановка θ называется унификатором выражений E_1 и E_2, если $E_1\theta = E_2\theta$.

Подстановка θ называется наиболее общим унификатором (НОУ) выражений E_1 и E_2, если

1. θ — унификатор выражений E_1 и E_2;
2. для любого унификатора η выражений E_1 и E_2 существует такая подстановка ρ, для которой верно равенство $\eta = \theta\rho$

Правило резолюции.

Пусть $D_1 = D_1' \vee L_1$ и $D_2 = D_2' \vee \neg L_2$ — два дизъюнкта.

Пусть $\theta \in HOV(L_1, L_2)$.

Тогда дизъюнкт $D_0 = (D_1' \vee D_2')\theta$ называется резольвентой дизъюнктов D_1 и D_2.

Пара литер L_1 и $\neg L_2$ называется контрарной парой .

Правило резолюции

$$\frac{D_1' \vee L_1,\ D_2' \vee \neg L_2}{(D_1' \vee D_2')\theta}, \quad \theta \in HOV(L_1, L_2)$$

Правило склейки.

Пусть $D_1 = D_1' \vee L_1 \vee L_2$ — дизъюнкт.

Пусть $\eta \in HOV(L_1, L_2)$.

Тогда дизъюнкт $D_0 = (D_1' \vee L_1)\eta$ называется склейкой дизъюнкта D_1.

Пара литер L_1 и L_2 называется склеиваемой парой .

Правило склейки

$$\frac{D_1' \vee L_1 \vee L_2}{(D_1' \vee L_1)\eta}, \quad \eta \in HOV(L_1, L_2)$$

Определение резолютивного вывода.

Пусть $S = \{D_1, D_2, \ldots, D_N\}$ — система дизъюнктов.

Резолютивным выводом из системы дизъюнктов S называется конечная последовательность дизъюнктов

$$D_1', D_2', \ldots, D_i', D_{i+1}', \ldots, D_n',$$

в которой для любого i, $1 \le i \le n$, выполняется одно из трех условий:

1. либо D_i' — **вариант** некоторого дизъюнкта из S;
2. либо D_i' — резольвента дизъюнктов D_j' и D_k', где $j, k < i$;
3. либо D_i' — склейка дизъюнкта D_j', где $j < i$.

Дизъюнкты $D_1', D_2', \ldots, D_n'$ считаются резолютивно выводимыми из системы S.

Резолютивный вывод называется успешным (или, по другому, резолютивным опровержением), если этот вывод оканчивается пустым дизъюнктом $\square$.

Успешный вывод — это свидетельство того, что система дизъюнктов S **противоречива** и опровергнуто предположение о ее выполнимости!

Теорема корректности резолютивного вывода

Если из системы дизъюнктов S резолютивно выводим пустой дизъюнкт $\square$, то S — противоречивая система дизъюнктов.

Теорема о полноте резолютивного вывода

Если S — противоречивая система дизъюнктов, то из S резолютивно выводим пустой дизъюнкт $\square$.

Рассмотрим формулу φ

$$\forall x \quad \Big(\forall y \exists v \forall u \quad \Big((A(u,v) \to B(y,u)) \& \\ (\neg \exists w A(w,u) \to \forall z A(z,v))\Big) \quad \to \exists y B(x,y)\Big)$$

Этап 1. Покажем, что формула $\varphi_1 = \neg\varphi$ противоречивая.

$$\varphi_1 \;=\; \neg \forall x \quad \Big(\forall y \exists v \forall u \quad \Big((A(u,v) \to B(y,u)) \; \& \\ (\neg \exists w A(w,u) \to \forall z A(z,v))\Big) \quad \to \exists y B(x,y)\Big)$$

Этап 2. Приведем φ_1 к ПНФ φ_2.

Переименование переменных

$$\neg \forall x \quad \Big(\forall y' \exists v \forall u \quad \Big((A(u,v) \to B(y',u)) \; \& \\ (\neg \exists w A(w,u) \to \forall z A(z,v))\Big) \quad \to \exists y'' B(x,y'')\Big)$$

Удаление импликаций

$$\neg \forall x \quad \Big(\neg \forall y' \exists v \forall u \quad \Big((\neg A(u,v) \vee B(y',u)) \; \& \\ (\neg\neg \exists w A(w,u) \vee \forall z A(z,v))\Big) \quad \vee \exists y'' B(x,y'')\Big)$$

Продвижение отрицаний

$$\exists x \quad \Big(\forall y' \exists v \forall u \quad \Big((\neg A(u,v) \vee B(y',u)) \; \& \\ (\exists w A(w,u) \vee \forall z A(z,v))\Big) \quad \& \; \forall y'' \neg B(x,y'')\Big)$$

Вынесение кванторов

$$\varphi_2 = \exists x \forall y' \exists v \forall u \exists w \forall z \forall y'' \quad \Big(\begin{array}{l} (\neg A(u,v) \vee B(y',u)) \; \& \\ (A(w,u) \vee A(z,v)) \; \& \\ \neg B(x,y'') \end{array} \Big)$$

Этап 3. Приведем φ_2 к ССФ φ_3.

$$\varphi_3 = \quad \forall y' \quad \forall u \quad \forall z \forall y'' \quad \Big(\begin{array}{l} (\neg A(u, f(y')) \vee B(y', u)) \ \& \\ (A(g(y', u), u) \vee A(z, f(y'))) \ \& \\ \neg B(c, y'') \end{array} \Big)$$

Этап 4. Формирование системы дизъюнктов S_φ.

$$S_\varphi \ = \ \left\{ \begin{array}{rcl} D_1 & = & \neg A(u, f(y')) \vee B(y', u), \\ D_2 & = & A(g(y', u), u) \vee A(z, f(y')), \\ D_3 & = & \neg B(c, y'') \end{array} \right\}$$

Этап 5. Резолютивный вывод из S_φ.

$$S_\varphi \ = \ \left\{ \begin{array}{rcl} D_1 & = & \neg A(u, f(y')) \vee B(y', u), \\ D_2 & = & A(g(y', u), u) \vee A(z, f(y')), \\ D_3 & = & \neg B(c, y'') \end{array} \right\}$$

1. $D'_1 = \neg A(u_1, f(y'_1)) \vee B(y'_1, u_1)$, (вариант D_1)
2. $D'_2 = A(g(y'_2, u_2), u_2) \vee A(z_2, f(y'_2))$, (вариант D_2)
3. $D'_3 = A(g(y'_3, f(y'_3)), f(y'_3))$, (склейка D'_2)
4. $D'_4 = B(y'_4, g(y'_4, f(y'_4)))$, (резольвента D'_1 и D'_3)
5. $D'_5 = \neg B(c, y''_5)$, (вариант D_3)
6. $D'_6 = \square$. (резольвента D'_4 и D'_5)

Заключение. Успешный резолютивный вывод из S_φ означает, что S_φ — противоречивая система дизъюнктов.

Значит, $\varphi_1 = \neg\varphi$ — невыполнимая формула.

Значит, φ — общезначимая формула,

$$\models \varphi.$$

4. Логическое программирование. Декларативная семантика и операционная семантика; соотношение между ними. Стандартная стратегия выполнения логических программ.

Синтаксис логических программ

Пусть $\sigma = \langle Const, Func, Pred \rangle$ — некоторая сигнатура, в которой определяются термы и атомы.

«заголовок» ::= «атом»

«тело» ::= «атом» | «тело», «атом»

«правило» ::= «заголовок» $\leftarrow$ «тело»;

«факт» ::= «заголовок»;

«утверждение» ::= «правило» | «факт»

«программа» ::= «пусто» | «утверждение» «программа»

«запрос» ::= $\square$ | ? «тело»

Терминология

Пусть $G = ?C_1, C_2, \ldots, C_m$ — запрос. Тогда

- атомы $C_1, C_2, \ldots, C_m$ называются подцелями запроса G,
- переменные множества $\bigcup_{i=1}^{m} Var_{C_i}$ называются целевыми переменными ,
- запрос $\square$ называется пустым запросом ,
- запросы будем также называть целевыми утверждениями .

Для удобства обозначения условимся в дальнейшем факты A; рассматривать как правила $A \leftarrow$; с заголовком A и пустым телом.

Главная особенность логического программирования — полисемантичность : одна и та же логическая программа имеет две равноправные семантики, два смысла.
Человек–программист и компьютер–вычислитель имеют две разные точки зрения на программу.

Программисту важно понимать, **ЧТО** вычисляет программа. Такое понимание программы называется декларативной семантикой программы.

Компьютеру важно «знать», **КАК** проводить вычисление программы. Такое понимание программы называется операционной семантикой программы.

Как нужно понимать логические программы?

Декларативная семантика	Операционная семантика
Правило $A_0 \leftarrow A_1, A_2, \ldots, A_n$;	
Если выполнены условия $A_1, A_2, \ldots, A_n$, то справедливо и утверждение A_0.	Чтобы решить задачу A_0, достаточно решить задачи $A_1, A_2, \ldots, A_n$.
Факт A_0;	
Утверждение A_0 считается верным.	Задача A_0 объявляется решенной.
Запрос $?C_1, C_2, \ldots, C_m$	
При каких значениях целевых переменных будут верны все отношения $C_1, C_2, \ldots, C_m$?	Решить список задач $C_1, C_2, \ldots, C_m$.

Логические программы и логические формулы

Каждому утверждению логической программы сопоставим логическую формулу:

Правило: $D' = A_0 \leftarrow A_1, A_2, \ldots, A_n$

$$D' = \forall X_1 \ldots \forall X_k (A_1 \& A_2 \& \ldots \& A_n \rightarrow A_0), \text{ где } \{X_1, \ldots, X_k\} = \bigcup_{i=0}^{n} Var_{A_i}$$

Факт: $D'' = A$

$$D'' = \forall X_1 \ldots \forall X_k A, \text{ где } \{X_1, \ldots, X_k\} = Var_A$$

Запрос: $G = ?\ C_1, C_2, \ldots, C_m$

$$G = C_1 \& C_2 \& \ldots \& C_m$$

С точки зрения декларативной семантики,

- программные утверждения D и запросы G — это логические формулы,
- программа $\mathcal{P}$ — это множество формул (база знаний),
- а правильный ответ на запрос — это такие значения переменных (подстановка), при которой запрос оказывается логическим следствием базы знаний.

Определение (правильного ответа)

Пусть $\mathcal{P}$ — логическая программа, G — запрос к $\mathcal{P}$ с множеством целевых переменных $Y_1, \ldots, Y_k$.

Тогда всякая подстановка $\theta = \{Y_1/t_1, \ldots, Y_k/t_k\}$ называется ответом на запрос G к программе $\mathcal{P}$.

Ответ $\theta = \{Y_1/t_1, \ldots, Y_k/t_k\}$ называется правильным ответом на запрос G к программе $\mathcal{P}$, если

$$\mathcal{P} \models \forall Z_1 \ldots \forall Z_N G\theta, \qquad \text{где } \{Z_1, \ldots, Z_N\} = \bigcup_{i=1}^{k} Var_{t_i}.$$

Теорема (об основном правильном ответе)

Пусть $G = ?C_1, C_2, \ldots, C_m$ — запрос к хорновской логической программе $\mathcal{P}$. Пусть $Y_1, \ldots, Y_k$ — целевые переменные, $t_1, \ldots, t_k$ — основные термы.
Тогда подстановка $\theta = \{Y_1/t_1, \ldots, Y_k/t_k\}$ является правильным ответом на запрос G к программе $\mathcal{P}$ тогда и только тогда, когда $\mathcal{P} \models (C_1 \& \ldots \& C_m)\theta$.

Концепция операционной семантики

Под операционной семантикой понимают правила построения **вычислений программы**. Операционная семантика описывает, КАК достигается результат работы программы.

Результат работы логической программы — это **правильный ответ** на запрос к программе. Значит, операционная семантика должна описывать метод вычисления правильных ответов.

Запрос к логической программе порождает задачу о логическом следствии. Значит, вычисление ответа на запрос должно приводить к решению этой задачи.

Таким методом вычисления может быть разновидность **метода резолюций**, учитывающая особенности устройства программных утверждений

Определение (SLD-резолюции)

Пусть

- $G = ?\ C_1, \ldots, C_i, \ldots, C_m$ — целевое утверждение, в котором выделена подцель C_i,
- $D' = A'_0 \leftarrow \mathbf{A'_1}, \mathbf{A'_2}, \ldots, \mathbf{A'_n}$ — **вариант** некоторого программного утверждения, в котором $Var_G \cap Var_{D'} = \emptyset$,
- $\theta \in HOV(C_i, A'_0)$ — наиб. общ. унификатор подцели C_i и заголовка программного утверждения A_0.

Тогда запрос

$$G' = ?(C_1, \ldots, C_{i-1}, \mathbf{A'_1}, \mathbf{A'_2}, \ldots, \mathbf{A'_n}, C_{i+1}, \ldots, C_m)\theta$$

называется SLD-резольвентой программного утверждения D' и запроса G с выделенной подцелью C_i и унификатором θ.

Пусть

- $G_0 = ?\ C_1, C_2, \ldots, C_m$ — целевое утверждение,
- $\mathcal{P} = \{D_1, D_2, \ldots, D_N\}$ — хорновская логическая программа.

Тогда (частичным) SLD-резолютивным вычислением , порожденным запросом G_0 к логической программе $\mathcal{P}$ называется последовательность троек (конечная или бесконечная)

$$(D_{j_1}, \theta_1, G_1), (D_{j_2}, \theta_2, G_2), \ldots, (D_{j_n}, \theta_n, G_n), \ldots,$$

в которой для любого i, $i \geq 1$,

- $D_{j_i} \in \mathcal{P}$, $\theta_i \in Subst$, G_i — целевое утверждение (запрос);
- запрос G_i является SLD-резольвентой программного утверждения D_{j_i} и запроса G_{i-1} с унификатором θ_i.

Частичное SLD-резолютивное вычисление

$$comp = (D_{j_1}, \theta_1, G_1), (D_{j_2}, \theta_2, G_2), \ldots, (D_{j_k}, \theta_n, G_n)$$

называется

- успешным вычислением (SLD-резолютивным опровержением), если $G_n = \square$;

- бесконечным вычислением , если *comp* — это **бесконечная** последовательность;
- тупиковым вычислением , если *comp* — это **конечная** последовательность, и при этом для запроса G_n невозможно построить ни одной SLD-резольвенты.

Пусть

- $G_0 = ?\ C_1, C_2, \ldots, C_m$ — целевое утверждение с целевыми переменными $Y_1, Y_2, \ldots, Y_k$,
- $\mathcal{P} = \{D_1, D_2, \ldots, D_N\}$ — хорновская логическая программа,
- $comp = (D_{j_1}, \theta_1, G_1), (D_{j_2}, \theta_2, G_2), \ldots, (D_{j_n}, \theta_n, \square)$ — успешное SLD-резолютивное вычисление, порожденное запросом G к программе $\mathcal{P}$.

Тогда подстановка $\theta = (\theta_1\theta_2\ldots\theta_n)|_{Y_1,Y_2,\ldots,Y_k}$,
представляющая собой композицию всех вычисленных унификаторов $\theta_1,\ \theta_2,\ldots,\theta_n$, ограниченную целевыми переменными $Y_1, Y_2, \ldots, Y_k$,
называется вычисленным ответом на запрос G_0 к программе $\mathcal{P}$.

Теперь у нас есть два типа ответов на запросы к логическим программам:

- **правильные ответы**, которые логически следуют из программы;
- **вычисленные ответы**, которые конструируются по ходу SLD-резолютивных вычислений.

Правильные ответы — это то, что мы хотим получить, обращаясь с вопросами к программе.

Вычисленные ответы — это то, что нам в действительности выдает компьютер (интерпретатор программы).

Теорема (корректности операционной семантики относительно декларативной семантики)

Пусть

- $G_0 = ?\ C_1, C_2, \ldots, C_m$ — целевое утверждение,
- $\mathcal{P} = \{D_1, D_2, \ldots, D_N\}$ — хорновская логическая программа,
- θ — вычисленный ответ на запрос G_0 к программе $\mathcal{P}$.

Тогда θ — правильный ответ на запрос G_0 к программе $\mathcal{P}$.

Теорема полноты (главная).

Пусть θ — правильный ответ на запрос $?G$ к хорновской логической программе $\mathcal{P}$.

Тогда существует такой вычисленный ответ η на запрос $?G$ к программе $\mathcal{P}$, что $\theta = \eta\rho$ для некоторой подстановки ρ.

Отображение R, которое сопоставляет каждому непустому запросу $G : ?C_1, C_2, \ldots, C_m$ одну из подцелей $C_i = R(G)$ в этом запросе, называется **правилом выбора подцелей**.

Для заданного правила выбора подцелей R вычисление запроса G к логической программе $\mathcal{P}$ называется **R-вычислением**, если на каждом шаге вычисления очередная подцель в запросе выбирается по правилу R.

Ответ, полученный в результате успешного R-вычисления, называется **R-вычисленным**.

Теорема сильной полноты

Каково бы ни было правило выбора подцелей R, если θ — правильный ответ на запрос G_0 к хорновской логической программе $\mathcal{P}$, то существует такой R-**вычисленный ответ** η, что равенство

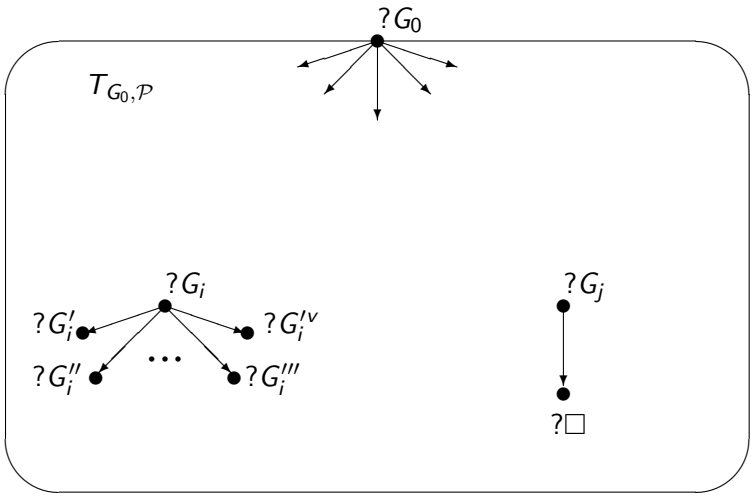
$$\theta = \eta\rho$$

выполняется для некоторой подстановки ρ.

Деревом SLD-резолютивных вычислений запроса G_0 к логической программе $\mathcal{P}$ называется помеченное корневое дерево $T_{G_0,\mathcal{P}}$, удовлетворяющее следующим требованиям:

1. Корнем дерева является исходный запрос G_0;
2. Потомками каждой вершины G являются всевозможные SLD-резольвенты запроса G (при фиксированном стандартном правиле выбора подцелей);
3. Листовыми вершинами являются пустые запросы (завершающие успешные вычисления) и запросы, не имеющие SLD-резольвент (завершающие тупиковые вычисления).

Иллюстрация



Определение

Стратегией вычисления запросов к логическим программам называется алгоритм построения (обхода) дерева SLD-резолютивных вычислений $T_{G_0,\mathcal{P}}$ всякого запроса G_0 к произвольной логической программе $\mathcal{P}$

Стратегия вычислений называется **вычислительно полной**, если для любого запроса G_0 и любой логической программы $\mathcal{P}$ эта стратегия строит (обнаруживает) **все** успешные вычисления запроса G_0 к программы $\mathcal{P}$

\

Фактически, стратегия вычисления — это одна стратегий обхода корневого дерева. Как известно, таких стратегий существует много, но среди них выделяются две наиболее характерные:

- **стратегия обхода в ширину**, при которой дерево строится (обходится) поярусно — вершина i-го не строится, до тех пор пока не будут построены все вершины $(i-1)$-го яруса;
- **стратегия обхода в глубину с возвратом**, при которой ветви дерева обходятся поочередно — очередная ветвь дерева не обохдится, до тех пор пока не будут пройдены все вершины текущей ветви.

Стратегия обхода в ширину является вычислительно полной, поскольку

- каждый запрос имеет конечное число SLD-резольвент, и поэтому в каждом ярусе дерева SLD-резолютивных вычислений имеется конечное число вершин;
- каждое успешное вычисление завершается на некотором ярусе;
- и поэтому каждое успешное вычисление будет рано или поздно полностью построено.

Но строить интерпретатор логических программ на основе стратегии обхода в ширину **нецелесообразно**. При обходе дерева в ширину нужно обязательно хранить в памяти **все** вершины очередного яруса. Это требует большого расхода памяти. Например, в 100-м ярусе двоичного дерева содержится 2^{99} вершин. Вычислительных ресурсов всего земного шара не хватит, чтобы хранить информацию обо всех этих вершинах.

Стратегия обхода в глубину с возвратом основана на следующих принципах:

1. все программные утверждения упорядочиваются;
2. на каждом шаге обхода из текущей вершины G осуществляется переход
   - либо в новую вершину-потомок G', которая является SLD-резольвентой запроса G и первого по порядку программного утверждения D, ранее не использованного для этой цели;
   - либо в ранее построенную родительскую вершину G'' (откат), если все программные утверждения уже были опробованы для построения SLD-резольвент запроса G.

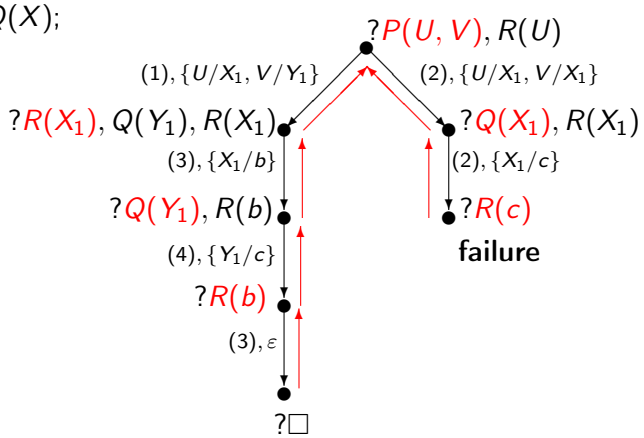
Стратегия обхода в глубину с возвратом

- имеет эффективную реализацию: в памяти нужно хранить лишь запросы той ветви, по которой идет обход, и каждый запрос должен вести учет использованных программных утверждений;
- является, к сожалению, вычислительно неполной.

Стратегия обхода в глубину чувствительна к порядку расположения программных утверждений в логических программах. Результат вычисления запроса может измениться при перестановке программных утверждений. Поскольку соображения эффективности превалируют над требованиями вычислительной полноты, в качестве **стандартной стратегии** вычисления логических программ выбрана стратегия обхода в глубину. Программист должен сам выбрать нужный порядок расположения программных утверждений, чтобы стандартная стратегия вычисления отыскала все вычисленные ответы.

Пример обхода в глубину с возвратом.

$\mathcal{P}$: $P(X,Y) \leftarrow R(X), Q(Y)$;
$P(X,X) \leftarrow Q(X)$;
$R(b) \leftarrow$;
$Q(c) \leftarrow$;



5. Транзакционное управление в СУБД. Методы сериализации транзакций.

Поддержание механизма транзакций - показатель уровня развитости СУБД. Корректное поддержание транзакций одновременно является основой обеспечения целостности баз данных (и поэтому транзакции вполне уместны и в однопользовательских персональных СУБД), а также составляют базис изолированности пользователей во многопользовательских системах. Часто эти два аспекта рассматриваются по отдельности, но на самом деле они взаимосвязаны, что и будет показано в этой лекции.

Под **транзакцией** понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации) такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в БД, либо воздействие всех этих операторов полностью отсутствует. Лозунг транзакции - "Все или ничего": при завершении транзакции оператором COMMIT результаты гарантированно фиксируются во внешней памяти (смысл слова commit - "зафиксировать" результаты транзакции); при завершении транзакции оператором ROLLBACK результаты гарантированно отсутствуют во внешней памяти (смысл слова rollback - ликвидировать результаты транзакции).

Понятие транзакции имеет непосредственную связь с понятием целостности БД. Очень часто БД может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения БД. Например, в базе данных СОТРУДНИКИ-ОТДЕЛЫ естественным ограничением целостности является совпадения значения атрибута ОТД_РАЗМЕР в кортеже отношения ОТДЕЛЫ, описывающем данный отдел (например, отдел 320), с числом кортежей отношения СОТРУДНИКИ таких, что значение атрибута СОТР_ОТД_НОМЕР равно 320. Как в этом случае принять на работу в отдел 320 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в отношение СОТРУДНИКИ или модификация существующего кортежа в отношении ОТДЕЛЫ, после выполнения операции база данных окажется в нецелостном состоянии.

Поэтому для поддержания подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах с развитыми средствами ограничения и контроля целостности каждая транзакция начинается при целостном состоянии БД и должна оставить это состояние целостными после своего завершения. Несоблюдение этого условия приводит к тому, что вместо фиксации результатов транзакции происходит ее откат (т.е. вместо оператора COMMIT выполняется оператор ROLLBACK), и БД остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Если быть немного более точным, различаются два вида ограничений целостности: немедленно проверяемые и откладываемые. К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать. Примером ограничения, проверку которого откладывать бессмысленно, являются ограничения домена (возраст сотрудника не может превышать 150 лет). Более сложным ограничением, проверку которого невозможно отложить, является следующее: зарплата сотрудника не может быть увеличена за одну операцию более, чем на 100,000 рублей. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности - это ограничения на базу данных, а не на какие-

либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора COMMIT на оператор ROLLBACK. Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор ROLLBACK или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).
И еще одно замечание. С точки зрения внешнего представления в момент завершения транзакции проверяются все откладываемые ограничения целостности, определенные в этой базе данных. Однако при реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных СОТРУДНИКИ-ОТДЕЛЫ в ней не выполнялись операторы вставки или удаления кортежей из отношения СОТРУДНИКИ, то проверять упоминавшееся выше ограничение целостности не требуется (а проверка подобных ограничений вызывает достаточно большую работу).

Во многопользовательских системах с одной базой данных одновременно могут работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т.е**. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с БД в одиночку**.
В связи со свойством сохранения целостности БД транзакции являются подходящими единицами изолированности пользователей. Действительно, если с каждым сеансом работы с базой данных ассоциируется транзакция, то каждый пользователь начинает работу с согласованным состоянием базы данных, т.е. с таким состоянием, в котором база данных могла бы находиться, даже если бы пользователь работал с ней в одиночку.
При соблюдении обязательного требования поддержания целостности базы данных возможны следующие **уровни изолированности транзакций**:

1. Первый уровень - **отсутствие потерянных изменений**. Рассмотрим следующий сценарий совместного выполнения двух транзакций. Транзакция 1 изменяет объект базы данных A. До завершения транзакции 1 транзакция 2 также изменяет объект A. Транзакция 2 завершается оператором ROLLBACK (например, по причине нарушения ограничений целостности). Тогда при повторном чтении объекта A транзакция 1 не видит изменений этого объекта, произведенных ранее. Такая ситуация называется ситуацией потерянных изменений. Естественно, она противоречит требованию изолированности пользователей. Чтобы избежать такой ситуации в транзакции 1 требуется, чтобы до завершения транзакции 1 никакая другая транзакция не могла изменять объект A. Отсутствие потерянных изменений является минимальным требованием к СУБД по части синхронизации параллельно выполняемых транзакций.
2. Второй уровень - **отсутствие чтения "грязных данных**". Рассмотрим следующий сценарий совместного выполнения транзакций 1 и 2. Транзакция 1 изменяет объект базы данных A. Параллельно с этим транзакция 2 читает объект A. Поскольку операция изменения еще не завершена, транзакция 2 видит несогласованные "грязные" данные (в частности, операция транзакции 1 может быть отвернута при проверке немедленно проверяемого ограничения целостности). Это тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и в праве ожидать видеть согласованные данные). Чтобы избежать ситуации чтения "грязных" данных, до завершения транзакции 1, изменившей объект A, никакая другая транзакция не должна читать объект A (минимальным

требованием является блокировка чтения объекта A до завершения операции его изменения в транзакции 1).

3. Третий уровень - **отсутствие неповторяющихся чтений**. Рассмотрим следующий сценарий. Транзакция 1 читает объект базы данных A. До завершения транзакции 1 транзакция 2 изменяет объект A и успешно завершается оператором COMMIT. Транзакция 1 повторно читает объект A и видит его измененное состояние. Чтобы избежать неповторяющихся чтений, до завершения транзакции 1 никакая другая транзакция не должна изменять объект A. В большинстве систем это является максимальным требованием к синхронизации транзакций, хотя, как мы увидим немного позже, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных. Как мы уже отмечали, для поддержания целостности достаточен первый уровень. Существует ряд приложений, для которых первого уровня достаточно (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.
К более тонким проблемам изолированности транзакций относится так называемая проблема **кортежей-"фантомов",** вызывающая ситуации, которые также противоречат изолированности пользователей. Рассмотрим следующий сценарий. Транзакция 1 выполняет оператор A выборки кортежей отношения R с условием выборки S (т.е. выбирается часть кортежей отношения R, удовлетворяющих условию S). До завершения транзакции 1 транзакция 2 вставляет в отношение R новый кортеж r, удовлетворяющий условию S, и успешно завершается. Транзакция 1 повторно выполняет оператор A, и в результате появляется кортеж, который отсутствовал при первом выполнении оператора. Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-фантомов, требуется более высокий "логический" уровень синхронизации транзакций. Идеи такой синхронизации (предикатные синхронизационные захваты) известны давно, но в большинстве систем не реализованы.

Понятно, что для того, чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций.
План (**способ) выполнения набора транзакций называется сериальным**, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций.
Сериализация транзакций - это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций обеспечивает реальную изолированность пользователей.
Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограничивал бы их параллельность. Приходящим на ум тривиальным решением является действительно последовательное выполнение транзакций. Но существуют ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.
Между транзакциями могут существовать следующие виды конфликтов:
W-W - транзакция 2 пытается изменять объект, измененный не закончившейся транзакцией 1;
R-W - транзакция 2 пытается изменять объект, прочитанный не закончившейся транзакцией 1;

W-R - транзакция 2 пытается читать объект, измененный не закончившейся транзакцией 1. Практические методы сериализации транзакций основывается на учете этих конфликтов.

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре «клиент-сервер») является подход, основанный на **соблюдении двухфазного протокола синхронизационных захватов объектов баз данных** (Two-Phase Locking Protocol, 2PL). В общих чертах подход состоит в том, что перед выполнением любой операции в транзакции `T` над объектом базы данных *`o`* от имени транзакции T **запрашивается синхронизационная блокировка объекта *`o`* в соответствующем режиме (в зависимости от вида операции).**

Основными режимами синхронизационных блокировок являются следующие:

> **совместный режим** – S (Shared), означающий совместную (по чтению) блокировку объекта и требуемый для выполнения операции чтения объекта;
>
> **монопольный режим** – X (eXclusive), означающий монопольную (по записи) блокировку объекта и требуемый для выполнения операций вставки, удаления и модификации объекта.

Блокировки одних и тех же объектов по чтению несколькими транзакциями **совместимы**, т.е. **нескольким транзакциям допускается одновременно читать один и тот же объект**. **Блокировка объекта одной транзакцией по чтению не совместима с блокировкой другой транзакцией того же объекта по записи**, т.е. **никакой транзакции нельзя изменять объект, читаемый некоторой транзакцией** (кроме самой этой транзакции), и **никакой транзакции нельзя читать объект, изменяемый некоторой транзакцией** (кроме самой этой транзакции). **Блокировки одного и того же объекта по записи разными транзакциями не совместимы**, т.е. никакой транзакции нельзя изменять объект, изменяемый некоторой транзакцией (кроме самой этой транзакции). Правила совместимости захватов одного объекта разными транзакциями приведены в таблице 9.1. В первом столбце приведены возможные состояния объекта с точки зрения синхронизационных захватов. При этом "-" соответствует состоянию объекта, для которого не установлен никакой захват. **Транзакция, запросившая синхронизационный захват объекта БД, уже захваченный другой транзакцией в несовместимом режиме, блокируется до тех пор, пока захват с этого объекта не будет снят**.

Таблица 9.1. Совместимость блокировок S и X

	X	S
-	да	да
X	нет	нет

S	нет	да

Для обеспечения сериализации транзакций (третьего уровня изолированности) синхронизационные захваты объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении. Это требование порождает двухфазный протокол синхронизационных захватов - 2PL. В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

первая фаза транзакции - накопление захватов;

вторая фаза (фиксация или откат) - освобождение захватов.

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных захватов действительно обеспечивается сериализация транзакций на третьем уровне изолированности. Основная проблема состоит в том, что следует считать объектом для синхронизационного захвата?

В контексте реляционных баз данных возможны следующие альтернативы:

- **файл** - физический (с точки зрения базы данных) объект, область хранения нескольких отношений и, возможно, индексов;
- **отношение** - логический объект, соответствующий множеству кортежей данного отношения;
- **страница данных** - физический объект, хранящий кортежи одного или нескольких отношений, индексную или служебную информацию;
- **кортеж** - элементарный физический объект базы данных.

На самом деле, когда мы говорим про операции над объектами базы данных, то любая операция над кортежем, фактически, является и операцией над страницей, в которой этот кортеж хранится, и над соответствующим отношением, и над файлом, содержащем отношение. Поэтому действительно имеется выбор уровня объекта захвата.

Понятно, что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект - логический или физический), тем меньше синхронизационных захватов будет поддерживаться в системе, и на это, соответственно, будут тратиться меньшие накладные расходы. Более того, если выбрать в качестве уровня объектов для захватов файл или отношение, то будет решена даже проблема фантомов.

Но вся беда в том, что при использовании для захватов крупных объектов возрастает вероятность конфликтов транзакций и тем самым уменьшается допускаемая степень их параллельного выполнения. Фактически, при укрупнении объекта синхронизационного захвата мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, когда на самом деле конфликтов нет.

Гранулированные синхронизационные захваты

При применении этого подхода синхронизационные захваты могут запрашиваться по отношению к объектам разного уровня: файлам, отношениям и кортежам. Требуемый уровень объекта определяется тем, какая операция выполняется (например, для выполнения операции уничтожения отношения объектом синхронизационного захвата должно быть все отношение, а для выполнения операции удаления кортежа - этот кортеж). Объект любого уровня может быть захвачен в режиме S или X.

Теперь наиболее важное отличие, на котором, собственно, держится соответствие захватов разного уровня. Вводится специальные протокол гранулированных захватов и новые типы захватов: перед захватом объекта в режиме S или X соответствующий объект более верхнего

уровня должен быть захвачен в режиме IS, IX или SIX. Что же из себя представляют эти режимы захватов?

IS (Intented for Shared lock) по отношению к некоторому составному объекту O означает намерение захватить некоторый входящий в O объект в совместном режиме. Например, при намерении читать кортежи из отношения R это отношение должно быть захвачено в режиме IS (а до этого в таком же режиме должен быть захвачен файл).

IX (Intented for eXclusive lock) по отношению к некоторому составному объекту O означает намерение захватить некоторый входящий в O объект в монопольном режиме. Например, при намерении удалять кортежи из отношения R это отношение должно быть захвачено в режиме IX (а до этого в таком же режиме должен быть захвачен файл).

SIX (Shared, Intented for eXclusive lock) по отношению к некоторому составному объекту O означает совместный захват всего этого объекта с намерением впоследствии захватывать какие-либо входящие в него объекты в монопольном режиме. Например, если выполняется длинная операция просмотра отношения с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего захватить это отношение в режиме SIX (а до этого захватить файл в режиме IS).

Довольно трудно описать словами все возможные ситуации. Мы ограничимся приведением полной таблицы совместимости захватов, анализируя которую можно выявить все случаи:

Совместимость блокировок S, X, IS, IX и SIX

	X	S	IX	IS	SIX
-	да	да	да	да	да
X	нет	нет	нет	нет	нет
S	нет	да	нет	да	нет
IX	нет	нет	да	да	нет
IS	нет	да	да	да	да
SIX	нет	нет	нет	да	нет

Несмотря на привлекательность метода гранулированных синхронизационных захватов, следует отметить что он не решает проблему фантомов (если, конечно, не ограничиться использованием захватов отношений в режимах S и X). Давно известно, что для решения этой проблемы необходимо перейти от захватов индивидуальных объектов базы данных, к захвату условий (предикатов), которым удовлетворяют эти объекты. Проблема фантомов не возникает при использовании для синхронизации уровня отношений именно потому, что отношение как логический объект представляет собой неявное условие для входящих в него кортежей. Захват отношения - это простой и частный случай предикатного захвата.

Поскольку любая операция над реляционной базой данных задается некоторым условием (т.е. в ней указывается не конкретный набор объектов базы данных, над которыми нужно выполнить операцию, а условие, которому должны удовлетворять объекты этого набора), идеальным выбором было бы требовать синхронизационный захват в режиме S или X именно этого условия. Но если посмотреть на общий вид условий, допускаемых, например, в языке SQL, то становится абсолютно непонятно, как определить совместимость двух предикатных захватов. Ясно, что без этого использовать предикатные захваты для синхронизации транзакций невозможно, а в общей форме проблема неразрешима.

К счастью, эта проблема сравнительно легко решается для случая простых условий. Будем называть простым условием конъюнкцию простых предикатов, имеющих вид

имя-атрибута { = > < } значение

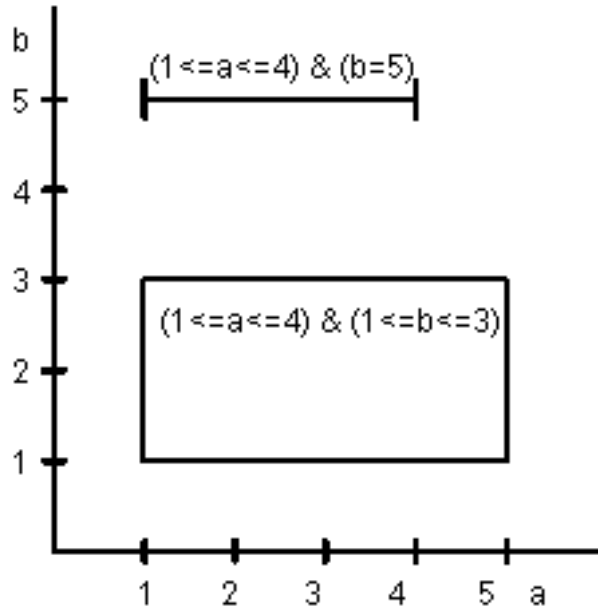
В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистем управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию исходного оператора со сложным условием в последовательность обращений к ядру СУБД, в каждом из которых содержатся только простые условия. Следовательно, в случае типовой организации реляционной СУБД простые условия можно использовать как основу предикатных захватов.

Для простых условий совместимость предикатных захватов легко определяется на основе следующей геометрической интерпретации. Пусть R отношение с атрибутами a_1, a_2, ..., a_n, а m_1, m_2, ..., m_n - множества допустимых значений a_1, a_2, ..., a_n соответственно (все эти множества - конечные). Тогда можно сопоставить R конечное n-мерное пространство возможных значений кортежей R. Любое простое условие "вырезает" m-мерный прямоугольник в этом пространстве (m <= n).

Тогда S-X, X-S, X-X предикатные захваты от разных транзакций совместимы, если соответствующие прямоугольники не пересекаются.

Это иллюстрируется следующим примером, показывающим, что в каких бы режимах не требовала транзакция 1 захвата условия (1<=a<=4) & (b=5), а транзакция 2 - условия (1<=a<=5) & (1<=b<=3), эти захваты всегда совместимы.

Пример: (n = 2)



Заметим, что предикатные захваты простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов.

Тупики, распознавание и разрушение

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных захватов является возможность возникновение тупиков (deadlocks) между транзакциями. Тупики возможны при применении любого из рассмотренных нами вариантов.

Вот простой пример возникновения тупика между транзакциями T1 и T2:

транзакции T1 и T2 установили монопольные захваты объектов r1 и r2 соответственно;

после этого T1 требуется совместный захват r2, а T2 - совместный захват r1;

ни одна из транзакций не может продолжаться, следовательно, монопольные захваты не будут сняты, а совместные - не будут удовлетворены.

Поскольку тупики возможны, и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций - это ориентированный двудольный граф, в котором существует два типа вершин - вершины, соответствующие транзакциям, и вершины, соответствующие объектам захвата. В этом графе существует дуга, ведущая из вершины-транзакции к вершине-объекту, если для этой транзакции существует удовлетворенный захват объекта. В графе существует дуга из вершины-объекта к вершине-транзакции, если транзакция ожидает удовлетворения захвата объекта.

Легко показать, что в системе существует ситуация тупика, если в графе ожидания транзакций имеется хотя бы один цикл.

Для распознавание тупика периодически производится построение графа ожидания транзакций (как уже отмечалось, иногда граф ожидания поддерживается постоянно), и в этом графе ищутся циклы. Традиционной техникой (для которой существует множество разновидностей) нахождения циклов в ориентированном графе является редукция графа.

Не вдаваясь в детали, редукция состоит в том, что прежде всего из графа ожидания удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов. (Это как бы соответствует той ситуации, что транзакции, не ожидающие удовлетворения

захватов, успешно завершились и освободили захваты). Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация исходящих дуг изменяется на противоположную (это моделирует удовлетворение захватов). После этого снова срабатывает первый шаг и так до тех пор, пока на первом шаге не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл.

Предположим, что нам удалось найти цикл в графе ожидания транзакций. Что делать теперь? Нужно каким-то образом обеспечить возможность продолжения работы хотя бы для части транзакций, попавших в тупик. Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, т.е. транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Грубо говоря, критерием выбора является стоимость транзакции; жертвой выбирается самая дешевая транзакция. Стоимость транзакции определяется на основе многофакторная оценка, в которую с разными весами входят время выполнения, число накопленных захватов, приоритет.

После выбора транзакции-жертвы выполняется откат этой транзакции, который может носить полный или частичный характер. При этом, естественно, освобождаются захваты и может быть продолжено выполнение других транзакций.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей, которого невозможно избежать.

Заметим, что в централизованных системах стоимость построения графа ожидания сравнительно невелика, но она становится слишком большой в по-настоящему распределенных СУБД, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

Еще одно замечание. Чтобы минимизировать число конфликтов между транзакциями, в некоторых СУБД (например, в Oracle) используется следующее развитие подхода.

Монопольный захват объекта блокирует только изменяющие транзакции. После выполнении операции модификации предыдущая версия объекта остается доступной для чтения в других транзакциях. Кратковременная блокировка чтения требуется только на период фиксации изменяющей транзакции, когда обновленные объекты становятся текущими.

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редких конфликтов транзакций и не требующий построения графа ожидания транзакций. основан на **использовании временных меток**.

Основная идея метода (у которого существует множество разновидностей) состоит в следующем: если транзакция T1 началась раньше транзакции T2, то система обеспечивает такой режим выполнения, как если бы T1 была целиком выполнена до начала T2.

Для этого каждой транзакции T предписывается временная метка t, соответствующая времени начала T. При выполнении операции над объектом r транзакция T помечает его своей временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом r транзакция T1 выполняет следующие действия:

1. Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T1 помечает объект r и выполняет свою операцию.
2. Если транзакция T не завершилась, то T1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T1 выполняет свою операцию.
3. Если операции T1 и T конфликтуют, то если t(T) > t(T1) (т.е. транзакция T является более "молодой", чем T), производится откат T и T1 продолжает работу.
4. Если же t(T) < t(T1) (T "старше" T1), то T1 получает новую временную метку и начинается заново.

К недостаткам метода временных меток относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в распределенных системах стоит очень дорого.

6. Аппаратно-программные средства поддержки мультипрограммного режима – система прерываний, защита памяти, привилегированный режим.

Как уже упоминалось на первой лекции, *мультизадачность* или режим мультипрограммирования – это такой режим работы вычислительной системы, при котором несколько программ могут выполняться в системе одновременно.

Две задачи, запущенные на одной вычислительной системе, называются выполняемыми *одновременно*, если периоды их выполнения (временнóй отрезок с момента запуска до момента завершения каждой из задач) полностью или частично перекрываются.

Итак, если процессор, работая в каждый момент времени с одной задачей, при этом переключается между несколькими задачами, уделяя внимание то одной из них, то другой, эти задачи в соответствии с нашим определением будут считаться выполняемыми одновременно (см. рис. 2).

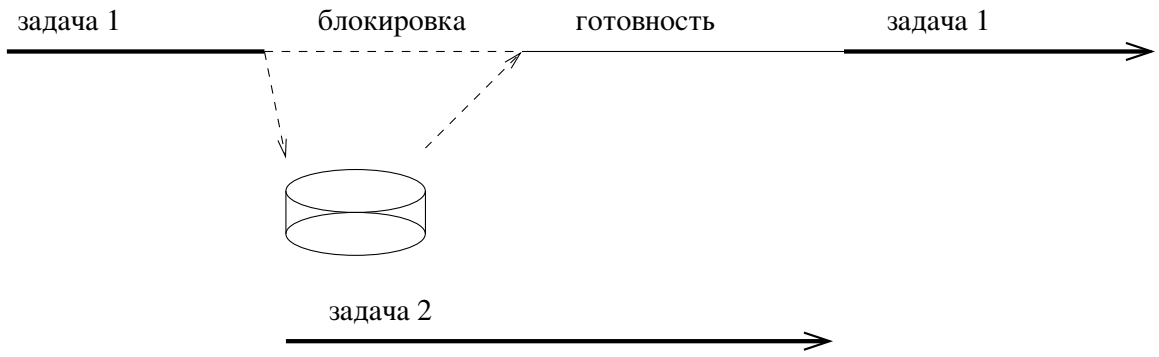


Рис. 4: Пакетная ОС

1. Необходимо наличие аппарата прерываний. Как минимум в машине должно быть прерывание по таймеру, что позволит избежать “зависания“ всей системы при зацикливании одной из программ (обеспечивает саму возможность реакции на события и автоматического переключения с одной программы на другую, сигнализировать о том, что процесс полез не в свою память или попытался выполнить запрещенную команду)

Проблема состоит в том, каким образом операционная система узнает о завершении операции ввода-вывода, если процессор при этом занят выполнением другой задачи и непрерывного опроса контроллера не производит.

Решить проблему позволяет аппарат *прерываний*. В данном конкретном случае в момент завершения операции контроллер подает центральному процессору определенный сигнал (электрический импульс), называемый *запросом прерывания*. Центральный процессор, получив этот сигнал, прерывает выполнение активной задачи и передает управление процедуре операционной системы, которая выполняет все необходимые по окончании операции ввода-вывода действия. После этого управление возвращается активной задаче.

Аппарат прерываний ЭВМ - возможность аппаратуры ЭВМ стандартным образом обрабатывать возникающие в вычислительной системе события. Данные события будем называть прерываниями.

Итак, ***прерывание*** - одно из событий в вычислительной системе, на возникновение которого предусмотрена стандартная реакция аппаратуры ЭВМ. Количество различных типов прерываний ограничено и определяется при разработке аппаратуры ЭВМ.

4.5.2 Защита памяти

Рассмотрим другие проблемы, возникающие при одновременном нахождении в памяти машины нескольких программ. Если не предпринять специальных мер, одна из программ может модифицировать данные или код других программ или самой операционной системы. Даже если допустить отсутствие злого умысла у разработчиков всех запускаемых программ, от случайных ошибок в программах нас это допущение не спасет.

Ясно, что необходимы средства ограничения возможностей работающей программы по доступу к областям памяти, занятым другими программами. Программно такую защиту можно реализовать разве что путем интерпретации всего машинного кода исполняющейся программы, что категорически недопустимо из соображений эффективности. Таким образом, необходима аппаратная поддержка защиты памяти.

Аппаратная возможность ассоциирования некоторых областей ОЗУ с одним из выполняющихся процессов/программ. Настройка аппарата защиты памяти происходит аппаратно, то есть назначение программе/процессу области памяти происходит программно (т.е., в общем случае операционная система устанавливает соответствующую информацию в специальных регистрах), а контроль за доступом – автоматически. При этом при попытке другим процессом/программой обратиться к этим областям ОЗУ происходит прерывание “Защита памяти”

4.5.3 Привилегированный и ограниченный режимы

Коль скоро существует защита памяти, процессор должен иметь набор команд для управления этой защитой. Если, опять таки, не предпринять специальных мер, то такие команды сможет исполнить любая из выполняющихся программ, сняв защиту памяти или модифицировав ее конфигурацию. Такая возможность делает саму защиту памяти практически бессмысленной.

Рассматриваемая проблема касается не только защиты памяти, но и работы с внешними устройствами. Чтобы обеспечить нормальное взаимодействие всех программ с устройствами ввода-вывода, операционная система должна взять непосредственную работу с устройствами на себя, а пользовательским программам предоставлять интерфейс для обращения к операционной системе за услугами по работе с устройствами. Иначе говоря, пользовательские программы должны иметь возможность работы с внешними устройствами только через операционную систему. Соответственно, необходимо запретить пользовательским программам выполнение команд процессора, осуществляющих чтение/запись портов ввода-вывода.

(В привилегированном режиме центральному процессору разрешается выполнять *все* команды языка машины, а в режиме пользователя – только обычные (не привилегированные) команды. При попытке выполнить привилегированную команду в пользовательском режиме центральным процессором вырабатывается сигнал прерывания, а сама команда, естественно, не выполняется. Привилегированными должны быть команды задания адресного пространства для программы. Привилегированными должны быть и все команды, которые обращаются к внешним (периферийным) устройствам. Например, нельзя разрешать запись на диск в режиме пользователя, так как диск – это тоже *общая* память для всех программ, только внешняя, и одна программа может испортить на диске данные (файлы), принадлежащие другим программам. То же самое относится и к печатающему устройству: если разрешить всем программам бесконтрольно выводить свои данные на печать, то, конечно, разобраться в том, что же получится на бумаге, будет чаще всего невозможно. Как же тогда быть, если программе необходимо, например, считать данные из *своего* файла на диске в оперативную память? Выход один – программа пользователя должна обратиться к определённым служебным процедурам, с просьбой выполнить для неё ту работу, которую сама программа пользователя сделать не в состоянии. Эти служебные процедуры, естественно, должны работать в привилегированном режиме. Перед выполнением запроса из программы пользователя, такая служебная процедура проверяет, имеет ли эта программа пользователя *право* на запрашиваемое действие, например, что эта программа имеет необходимые *полномочия* на чтение из указанного файла.□Переключение в привилегированный режим производится центральным процессором при вызове специальных системных процедур, которые имеют *полномочия* для работы в привилегированном режиме.

4.5.4 Таймер

Для реализации пакетного мультизадачного режима перечисленных аппаратных средств уже достаточно. Если же необходимо реализовать систему разделения времени или реального времени, в аппаратуре вычислительной системы требуется наличие еще одного компонента — *таймера*.

Действительно, планировщику операционной системы разделения времени нужна возможность отслеживания истечения квантов времени, выделенных пользовательским программам; в системе реального времени такая возможность также необходима, причем требования к ней даже более жесткие: не сняв вовремя с процессора активное на тот момент приложение, планировщик рискует попросту не успеть выделить более важным программам необходимое им процессорное время, в результате чего могут наступить неприятные последствия (вспомните пример с автопилотом самолета).

Таймер представляет собой сравнительно простое устройство, вся функциональность которого сводится в простейшем случае к генерации прерываний через равные промежутки времени. Эти прерывания дают возможность операционной системе получить управление, проанализировать текущее состояние имеющихся задач и при необходимости сменить активную задачу.

Прерывания в изначальном смысле уже знакомы нам по предыдущему параграфу. Те или иные устройства вычислительной системы могут осуществлять свои функции независимо от центрального процессора; в этом случае им может время от времени требоваться внимание операционной системы, но единственный центральный процессор (или, что ничуть не лучше, все имеющиеся в системе центральные процессоры) может быть именно в такой момент занят обработкой пользовательской программы.

Аппаратные (или *внешние*) прерывания были призваны решить эту проблему. Для поддержки аппаратных прерываний процессор обычно имеет специально предназначенные для этого контакты; электрический импульс, представляющий обычно логическую единицу, поданный на такой контакт, воспринимается процессором как сигнал о том, что некоторому устройству требуется внимание операционной системы.

В современных архитектурах, основанных на общей шине, для запроса прерывания используется одна из дорожек шины.

Последовательность событий при возникновении и обработке прерывания выглядит приблизительно следующим образом[6]:

1. Устройство, которому требуется внимание процессора, устанавливает на шине сигнал «запрос прерывания».

2. Процессор доводит выполнение текущей программы до такой точки, в которой выполнение можно прервать так, чтобы потом восстановить его с того же места; после этого процессор выставляет на шине сигнал «подтверждение прерывания». При этом другие прерывания блокируются.

3. Получив подтверждение прерывания, устройство передает по шине некоторое число, идентифицирующее данное устройство; это число называют *номером прерывания*.

4. Процессор сохраняет где-то (обычно на стеке) текущие значения счетчика команд и регистра слова состояния процесса; это называется *малым упрятыванием*. Счетчик команд и слово состояния должны быть сохранены по той причине, что выполнение первой же инструкции обработчика прерывания изменит (испортит) и то, и другое, сделав прозрачный возврат из прерывания невозможным; остальные регистры обработчик прерывания может при необходимости сохранить самостоятельно.

5. Устанавливается привилегированный режим работы центрального процессора, после чего управление передается на точку входа процедуры в операционной системе, называемой *обработчиком прерывания*. Адрес обработчика может быть предварительно считан из специальных областей памяти, либо вычислен иным способом.

Обработчик прерывания может сразу же вернуть управление активной задаче, выполнив команду `IRET` (interrupt return). Это называется *коротким прерыванием*. При этом процессор выполнит восстановление слова состояния и счетчика задач, то есть прерванный процесс продолжится в точности с того места и состояния, на котором его прервали. Короткие прерывания система выполняет в случае, если (с точки зрения системы) пришедшее прерывание никаких действий не требует; например, коротким может быть прерывание от системного таймера в случае, если никто кроме активной задачи не претендует на процессор.

Если операционной системе требуется выполнение каких-либо действий в ответ на поступившее прерывание, действия обработчика прерывания (называемые *длинным прерыванием*) будут более сложными. Поскольку для выполнения любых действий требуются регистры общего назначения, обработчик прежде всего сохраняет на стеке значения регистров. Это называется *полным упрятыванием*. Затем необходимо покинуть критическую область ядра, отвечающую за начальную стадию обработки прерывания, и перейти к исполнению процедур, прерывание которых не вызывает ошибок. При этом прерывания следует разблокировать, дав возможность работать другим устройствам.

После выполнения системных действий, которые повлекло за собой прерывание, следует вызвать планировщик, чтобы выяснить, не пришло ли время заменить активную задачу на другую. Возможно, пришедшее прерывание перевело из режима блокировки в режим готовности[7] процесс, имеющий бóльший приоритет, нежели прерванная задача; в этом случае все значения

регистров, сохраненные на стеке, а равно и сам указатель стека переписываются во внутреннюю структуру данных операционной системы, отвечающую за прерванный процесс, после чего восстанавливается стек (и, при необходимости, его содержимое) процесса, который необходимо сделать активным вместо прерванного.

Следует обратить внимание на то, что переключение из привилегированного режима работы центрального процессора в ограниченный можно осуществить простой командой (поскольку в привилегированном режиме доступны все возможности процессора); в то же время, переход из ограниченного (пользовательского) режима обратно в привилегированный произвести с помощью обычной команды нельзя, поскольку это лишило бы смысла само существование привилегированного и ограниченного режимов.

Таким образом, **прерывание является единственным (из известных нам на текущий момент) способом переключения процессора в привилегированный режим**.

5.2 Внутренние прерывания (ловушки)

Чтобы понять, о чем пойдет речь в этом параграфе, рассмотрим следующий вопрос: что следует делать центральному процессору, если активная задача выполнила целочисленное деление на ноль?

Завершить текущую задачу процессор самостоятельно не может. Это слишком сложное действие, зависящее от реализации операционной системы.

Остается только один вариант: передать управление операционной системе с сообщением о происшедшем. Что делать с аварийной задачей, операционная система решит самостоятельно.

Отметим, что требуется, вообще говоря, переключиться в привилегированный режим и передать управление на некоторый обработчик; перед этим желательно сохранить регистры (хотя бы счетчик команд и слово состояния); даже если задача ни при каких условиях не будет продолжена с того же места (а предполагать это, вообще говоря, процессор не вправе), значения регистров в любом случае пригодятся операционной системе для анализа происшествия. Более того, каким-то образом следует сообщить операционной системе о причине того, что управление передано ей; кроме деления на ноль, такими причинами могут быть нарушение защиты памяти, попытка выполнить запрещенную или несуществующую инструкцию, попытка прочитать слово по нечетному адресу и т.п.

Легко заметить, что действия, которые должен выполнить процессор, оказываются очень похожи на рассмотренный ранее случай аппаратного прерывания. Основное отличие состоит в отсутствии обмена по шине (запроса и подтверждения прерывания): действительно, информация о перечисленных событиях возникает внутри процессора, а не вне его[8]. Остальные шаги по обработке деления на ноль и других подобных ситуаций повторяют шаги по обработке аппаратного прерывания практически дословно.

Поэтому обработку ситуаций, в которых дальнейшее выполнение активной задачи оказывается невозможной по причине выполненных ею некорректных действий, называют так же, как и действия по запросу внешних устройств — прерываниями. Чтобы не путать разные по своей природе прерывания, их делят на внешние (аппаратные) и внутренние; такая терминология оправдана тем, что причина внешнего прерывания находится вне центрального процессора, тогда как причина внутреннего — у ЦП внутри. Иногда внутренние прерывания называют иначе, например ловушками (traps) или как-то еще.

Как уже говорилось, пользовательской задаче не позволяется делать ничего, кроме преобразования данных в отведенной ей памяти. Все действия, затрагивающие внешний по отношению к задаче мир, выполняются через операционную систему. Соответственно, необходим механизм, позволяющий пользовательской задаче обратиться к ядру операционной системы за теми или иными услугами.

Обращение пользовательской задачи (процесса) к ядру операционной системы за услугами называется *системным вызовом*.

Ясно, что по своей сути системный вызов — это передача управления от пользовательской задачи ядру операционной системы. Однако здесь есть две проблемы. Во-первых, ядро работает в привилегированном режиме, а пользовательский процесс — в ограниченном. Во-вторых, пространство адресов ядра для пользовательского процесса недоступно (как мы увидим на одной из следующих лекций, в адресном пространстве процесса этих адресов может вообще не быть). Впрочем, даже если бы оно было доступно, позволить процессу передавать управление в произвольную точку ядра было бы несколько странно.

Таким образом, для осуществления системного вызова необходимо сменить режим выполнения с пользовательского на привилегированный и передать управление в некоторую точку входа в операционной системе.

Нам уже известны два случая, в которых такие действия выполняются — это аппаратные и внутренние прерывания. Изобретать дополнительный механизм для системного вызова ни к чему; для его реализации можно использовать частный случай внутреннего прерывания, инициируемый специально предназначенной для этого машинной инструкцией (на разных архитектурах соответствующая инструкция может называется TRAP, SVC, INT и т.д.). Отличие этого вида прерывания от остальных состоит в том, что оно происходит по инициативе пользовательской задачи, тогда как другие прерывания случаются без ее ведома (внешние — по требованию внешних устройств, внутренние — в случае непредвиденных обстоятельств, которые вряд ли были выполняемой программой предусмотрены).

Прерывание, возникающее по инициативе выполняющейся задачи, называется *программным прерыванием* [9].

В основе операционной системы всегда находится программа, осуществляющая работу с аппаратурой, обрабатывающая прерывания и обслуживающая системные вызовы. Эта программа названа *ядром* операционной системы.

Единственной программой, выполняющейся в привилегированном режиме во время работы операционной системы, является ядро операционной системы. Все остальные программы, вне зависимости