§5. Понятие замкнутого класса. Замкнутость классов T_0, T_1 и L.

1°. Понятие замкнутого класса.

Определение 1. Пусть $A \subseteq P_2$. Тогда *замыканием* A называется множество всех функций алгебры логики, которые можно выразить формулами над A.

Обозначение: $[A]$.

Имеют место следующие свойства:

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

Определение 2. Система функций алгебры логики A называется полной, если $[A] = P_2$.

Определение 3. Пусть $A \subseteq P_2$. 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 $B \subseteq A$. Тогда B — неполная система (подмножество неполной системы будет также неполной системой).

Доказательство. $B \subseteq A \Rightarrow [B] \subseteq [A] = A \neq P_2 \Rightarrow [B] \neq P_2$. Следовательно, B — неполная система. Утверждение доказано.

2°. Примеры замкнутых классов.

Класс T_0 $= \{f(x_1, \ldots, x_n) \mid f(0, \ldots, 0) = 0\}$.

Классу T_0 принадлежат, например, функции 0, x, xy, $x \vee y$, $x \oplus y$.

Классу T_0 не принадлежат функции 1, $\bar{x}$, $x \to y$, $x \mid y$, $x \downarrow y$, $x \sim y$.

Подсчитаем число функций в классе T_0. Для этого построим следующую таблицу:

x_1	…	x_n	
0	…	0	0
…	…	…	} $2^n - 1$

.

Все функции, принадлежащие указанному классу, принимают на нулевом наборе нулевое значение. Таким образом, всего функций в классе T_0 столько, сколько существует булевых векторов длины $2^n - 1$ (первый разряд вектора длины 2^n необходимо равен нулю), то есть $|T_0| = 2^{2^n - 1} = \frac{1}{2} 2^{2^n}$.

Теорема 6. Класс T_0 —замкнутый.

Доказательство. Пусть $\{f(x_1,\ldots,x_n), g_1(y_{11},\ldots,y_{1m_1}),\ldots,g_n(y_{n1},\ldots,y_{nm_n})\} \subseteq T_0$. Рассмотрим функцию $h(y_1,\ldots,y_r) = f(g_1(y_{11},\ldots,y_{1m_1}),\ldots,g_n(y_{n1},\ldots,y_{nm_n}))$. 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.

Класс T_1 $= \{f(x_1, \ldots, x_n) \mid f(1, 1, \ldots, 1) = 1\}$.

Классу T_1 принадлежат функции 1, x, xy, $x \vee y$, $x \to y$, $x \sim y$.

Классу T_1 не принадлежат функции 0, $\bar{x}$, $x \oplus y$, $x \mid y$, $x \downarrow y$.

Теорема 7. Класс T_1 замкнут.

Доказательство повторяет доказательство аналогичной теоремы для класса T_0.

Класс *L* линейных функций.

Определение 4. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется линейной, если

$$f(x_1, \ldots, x_n) = a_0 \oplus a_1 x_1 \oplus \ldots \oplus a_n x_n, \text{ где } a_i \in \{0, 1\}.$$

Иными словами, в полиноме линейной функции нет слагаемых, содержащих конъюнкцию.

Классу L принадлежат функции 0, 1, $\bar{x} = x \oplus 1$, $x \sim y$, $x \oplus y$.

Классу L не принадлежат функции xy, $x \vee y$, $x \to y$, $x \mid y$, $x \downarrow y$.

Теорема 8. Класс L замкнут.

Доказательство. Поскольку тождественная функция — линейная, достаточно (как и в теоремах 6 и 7) 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 $x_{i_1} \oplus x_{i_2} \oplus \ldots \oplus x_{i_k} \oplus a_0$. Если теперь вместо каждого x_{i_j} подставить линейное выражение, то получится снова линейное выражение (или константа единица или нуль).

§6. Двойственность. Класс самодвойственных функций, его замкнутость.

1°. Двойственность.

Определение 1. Функцией, *двойственной* к функции алгебры логики $f(x_1, \ldots, x_n)$, называется функция $f^*(x_1, \ldots, x_n) = \bar{f}(\overline{x_1}, \ldots, \overline{x_n})$.

Теорема 9 (принцип двойственности). Пусть

$$\Phi(y_1, \ldots, y_m) = f\left(g_1(y_{11}, \ldots, y_{1k_1}), \ldots, g_n(y_{n1}, \ldots, y_{nk_n})\right).$$

Тогда $\Phi^*(y_1, \ldots, y_m) = f^*\left(g_1^*(y_{11}, \ldots, y_{1k_1}), \ldots, g_n^*(y_{n1}, \ldots, y_{nk_n})\right)$.

Доказательство. Рассмотрим

$$\Phi^*(y_1, \ldots y_m) = \bar{f}\left(g_1(\overline{y_{11}}, \ldots, \overline{y_{1k_1}}), \ldots, g_n(\overline{y_{n1}}, \ldots, \overline{y_{nk_n}})\right) =$$
$$= \bar{f}\left(\overline{\overline{g_1}}(\overline{y_{11}}, \ldots, \overline{y_{1k_1}}), \ldots, \overline{\overline{g_n}}(\overline{y_{n1}}, \ldots, \overline{y_{nk_n}})\right) = \bar{f}\left(\overline{g_1^*}(y_{11}, \ldots, y_{1k_1}), \ldots, \overline{g_n^*}(y_{n1}, \ldots, y_{nk_n})\right) =$$
$$= f^*\left(g_1^*(y_{11}, \ldots, y_{1k_1}), \ldots, g_n^*(y_{n1}, \ldots, y_{nk_n})\right).$$

Теорема доказана.

Следствие. Пусть функция $\Phi(y_1, \ldots, y_m)$ реализуется формулой над $A = \{f_1, f_2, \ldots\}$. 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

$$f(x_1, \ldots, x_n) = f^{**}(x_1, \ldots, x_n).$$

Доказательство. $f^{**} = \left[\bar{f}(\overline{x_1}, \ldots, \overline{x_n})\right]^* = \bar{\bar{f}}(\bar{\overline{x_1}}, \ldots, \bar{\overline{x_n}}) = f(x_1, \ldots, x_n)$, и утверждение доказано.

2°. Класс *S* самодвойственных функций.

Определение 2. 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, $S = \{f \mid f = f^*\}$.

Классу S принадлежат функции

$$x,\ \bar{x},\ x \oplus y \oplus z \oplus a,\ m(x, y, z) = xy \vee yz \vee zx = \begin{cases} 1, x + y + z \geq 2 \\ 0, x + y + z \leq 1 \end{cases}.$$

Классу S не принадлежат функции

0 ($f(x) \equiv 0 \Rightarrow f^*(x) = \bar{f}(\bar{x}) \equiv 1$), 1, $x \vee y$ (поскольку $(x \vee y)^* = \overline{\bar{x} \vee \bar{y}} = x \cdot y \neq x \vee y$), xy.

Теорема 10. Класс S замкнут.

Доказательство. Пусть $f(x_1, \ldots, x_n) \in S$, $\forall i \; g_i(y_{i1}, \ldots, y_{ik_i}) \in S$, $i = 1, 2, \ldots, n$ и

$$\Phi = f\left(g_1(y_{11}, \ldots, y_{1k_1}), \ldots, g_n(y_{n1}, \ldots, y_{nk_n})\right).$$

Тогда из принципа двойственности следует, что

$$\Phi^* = f^*\left(g_1^*(y_{11}, \ldots, y_{1k_1}), \ldots, g_n^*(y_{n1}, \ldots, y_{nk_n})\right) = f(g_1(\ldots), \ldots, g_n(\ldots)).$$

Таким образом, мы получили, что $\Phi = \Phi^*$ и $\Phi \in S$. Теорема доказана.

§7. Класс монотонных функций, его замкнутость.

Определение 1. Пусть $\tilde{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ и $\tilde{\beta} = (\beta_1, \beta_2, \ldots, \beta_n)$. Тогда

$$\tilde{\alpha} \leq \tilde{\beta} \Leftrightarrow \forall i (\alpha_i \leq \beta_i) .$$

Замечание. Существуют наборы, для которых неприменимо отношение упорядоченности, определённое выше. Так, например, наборы (0, 0, 1) и (0, 1, 0) несравнимы.

Определение 2. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется *монотонной*, если для любых двух сравнимых наборов $\tilde{\alpha}$ и $\tilde{\beta}$ выполняется импликация

$$\tilde{\alpha} \leq \tilde{\beta} \Rightarrow f(\tilde{\alpha}) \leq f(\tilde{\beta}).$$

Класс M всех монотонных функций.

Классу M принадлежат функции 0 , 1 , x , xy , $x \vee y$, $m(x, y, z) = xy \vee yz \vee zx$.

Классу M не принадлежат функции $\bar{x}$, $x \mid y$, $x \downarrow y$, $x \oplus y$, $x \sim y$, $x \to y$.

Теорема 11. Класс M 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 $\tilde{\alpha} = (\alpha_1, \ldots, \alpha_m)$, $\tilde{\beta} = (\beta_1, \ldots, \beta_m)$ такие, что $\tilde{\alpha} \leq \tilde{\beta}$. Обозначим

$$g_i(\tilde{\alpha}) = \gamma_i, g_i(\tilde{\beta}) = \delta_i .$$

Тогда для любого i имеем $g_i \in M$ и $g_i(\tilde{\alpha}) \leq g_i(\tilde{\beta})$, то есть $\forall i (\gamma_i \leq \delta_i)$. Обозначим

$$\tilde{\gamma} = (\gamma_1, \gamma_2, \ldots, \gamma_n), \tilde{\delta} = (\delta_1, \delta_2, \ldots, \delta_n).$$

Тогда по определению $\tilde{\gamma} \leq \tilde{\delta}$ и, в силу монотонности функции f, $f(\tilde{\gamma}) \leq f(\tilde{\delta})$. Но

$$\Phi(\tilde{\alpha}) = f(\gamma_1, \ldots, \gamma_n) = f(\tilde{\gamma}), \; \Phi(\tilde{\beta}) = f(\delta_1, \ldots, \delta_n) = f(\tilde{\delta}),$$

и неравенство $f(\tilde{\gamma}) \leq f(\tilde{\delta}) \Leftrightarrow \Phi(\tilde{\alpha}) \leq \Phi(\tilde{\beta})$, следовательно, $\Phi \in M$. Теорема доказана.

§8. Лемма о несамодвойственной функции.

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

Доказательство. Пусть $f \notin S$. Тогда $\bar{f}(\overline{x_1},\ldots,\overline{x_n}) \neq f(x_1,\ldots,x_n) \Rightarrow \exists \tilde{\sigma} = (\sigma_1,\ldots,\sigma_n)$:

$$\bar{f}(\overline{\sigma_1},\ldots,\overline{\sigma_n}) \neq f(\sigma_1,\ldots,\sigma_n) \Leftrightarrow f(\overline{\sigma_1},\ldots,\overline{\sigma_n}) = f(\sigma_1,\ldots,\sigma_n).$$

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 $\varphi(0) = \varphi(1) \Rightarrow \varphi(x) = \text{const}$. Заметим, что подстановка удовлетворяет условию теоремы, так как $x \oplus \sigma = \begin{cases} x, \sigma = 0 \\ \bar{x}, \sigma = 1 \end{cases}$. Лемма доказана.

§9. Лемма о немонотонной функции.

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

Доказательство. Пусть $f \notin M$. Тогда существуют такие наборы $\tilde{\alpha} = (\alpha_1,\ldots,\alpha_n)$ и $\tilde{\beta} = (\beta_1,\ldots,\beta_n)$, что $\tilde{\alpha} < \tilde{\beta}$ (то есть $\forall j$ ($\alpha_j \le \beta_j$) и $\tilde{\alpha} \neq \tilde{\beta}$) и $f(\tilde{\alpha}) > f(\tilde{\beta})$. Выделим те разряды $i_1, \ldots, i_k$ наборов $\tilde{\alpha}$ и $\tilde{\beta}$, в которых они различаются. Очевидно, в наборе $\tilde{\alpha}$ эти разряды равны 0, а в наборе $\tilde{\beta}$ — 1. Рассмотрим последовательность наборов

$$\tilde{\alpha}_0, \tilde{\alpha}_1, \tilde{\alpha}_2, \ldots, \tilde{\alpha}_k$$

таких, что $\tilde{\alpha} = \tilde{\alpha}_0 < \tilde{\alpha}_1 < \tilde{\alpha}_2 < \ldots < \tilde{\alpha}_k = \tilde{\beta}$, где $\tilde{\alpha}_{i+1}$ получается из $\tilde{\alpha}_i$ заменой одного из нулей, расположенного в одной из позиций $i_1, \ldots, i_k$, на единицу (при этом наборы $\tilde{\alpha}_i$ и $\tilde{\alpha}_{i+1}$ — соседние). Поскольку $f(\tilde{\alpha}) = 1$, а $f(\tilde{\beta}) = 0$, среди наборов $\tilde{\alpha}_0, \tilde{\alpha}_1, \tilde{\alpha}_2, \ldots, \tilde{\alpha}_k$ найдутся два соседних $\tilde{\alpha}_i$ и $\tilde{\alpha}_{i+1}$, такие что $f(\tilde{\alpha}_i) = 1$ и $f(\tilde{\alpha}_{i+1}) = 0$. Пусть они различаются в r-ом разряде: $\tilde{\alpha}_i = (\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, 0, \alpha_{r+1}, \ldots, \alpha_n)$, $\tilde{\alpha}_{i+1} = (\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, 1, \alpha_{r+1}, \ldots, \alpha_n)$. 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 $\varphi(0) = f(\tilde{\alpha}_i) = 1$, $\varphi(1) = f(\tilde{\alpha}_{i+1}) = 0$ и $\varphi(x) \equiv \bar{x}$. Лемма доказана.

§10. Лемма о нелинейной функции.

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.

§11. Теорема Поста о полноте системы функций алгебры логики.

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

§12. Теорема о максимальном числе функций в базисе алгебры логики.

Определение. Система функций алгебры логики $A \subseteq P_2$ называется *базисом* (в P_2), если

1) $[A] = P_2$;
2) $\forall f \in A\ ([A \setminus \{f\}] \neq P_2)$.

Теорема 13. Максимальное число функций в базисе алгебры логики равно 4.

Доказательство. 1) Докажем, что из любой полной системы можно выделить полную подсистему, содержащую не более четырёх функций. Действительно, если A — полная система ($[A] = P_2$), то согласно теореме Поста в ней существуют пять функций $f_0 \notin T_0, f_1 \notin T_1, f_S \notin S, f_M \notin M, f_L \notin L$. По теореме Поста система функций $\{f_0, f_1, f_S, f_M, f_L\}$ полна. Рассмотрим функцию $f_0(x_1, \ldots, x_n) \notin T_0$ ($f_0(0, 0, \ldots, 0) = 1$). Возможны два случая:

a) $f_0(1, 1, \ldots, 1) = 1 \Rightarrow f_0 \notin S \Rightarrow [f_0, f_1, f_L, f_M] = P_2$ и система $\{f_0, f_1, f_L, f_M\}$ полна.
b) $f_0(1, 1, \ldots, 1) = 0 \Rightarrow f_0 \notin M, T_1 \Rightarrow [f_0, f_L, f_S] = P_2$ и система $\{f_0, f_L, f_S\}$ полна.

Глава II. Основы теории графов.

§15. Основные понятия теории графов. Изоморфизм графов. Связность.

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*.

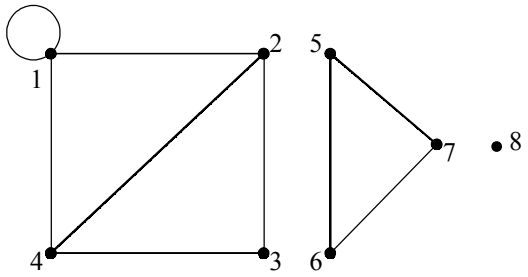
Определение 3. Пара вида (a, a) называется *петлёй*, если пара (a, b) встречается в семействе E несколько раз, то она называется *кратным ребром* (*кратной дугой*).

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.

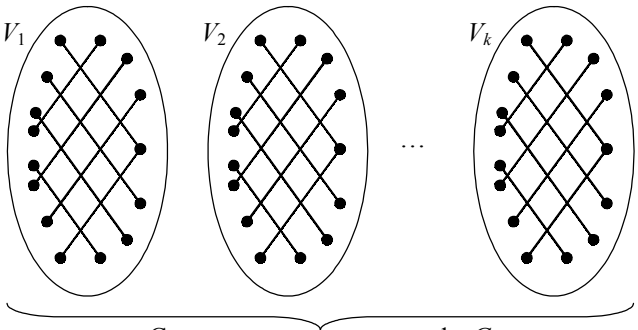
Определение 17. Путь называется *простым циклом*, если $v_0 = v_n$ и вершины не повторяются.

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

§16. Деревья. Свойства деревьев.

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.

§17. Корневые деревья. Верхняя оценка их числа.

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*.

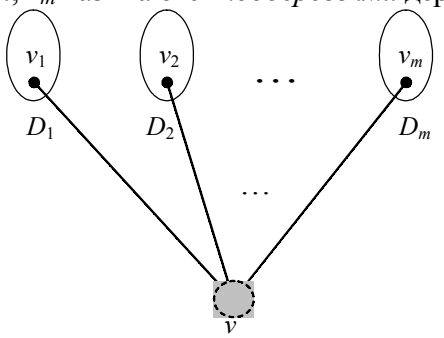
2) Пусть имеются корневые деревья $D_1, D_2, \ldots, D_m$ с корнями $v_1, v_2, \ldots, v_m$, $D_i = (V_i, E_i)$, $V_i \cap V_j = \varnothing$ $(i \neq j)$. Тогда граф $D = (V, E)$, полученный следующим образом:

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

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.

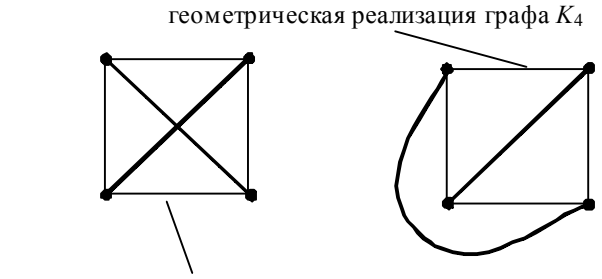
Следствие. Число неизоморфных корневых деревьев с q рёбрами и число неизоморфных деревьев с q рёбрами не превосходит 4^q.

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.

§18. Геометрическая реализация графов. Теорема о реализации графов в трёхмерном пространстве.

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.

§19. Планарные (плоские) графы. Формула Эйлера.

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.

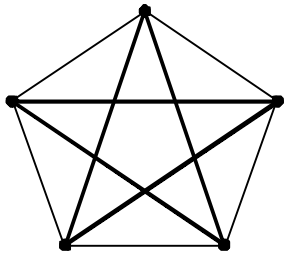
Доказательство. Пусть связный граф G с p вершинами и q рёбрами реализован на сфере S так, что число граней равно r. Пусть точка A на сфере не лежит на линиях этой геометрической реализации. Пусть P — некоторая плоскость. Поставим сферу S на плоскость P так, чтобы точка A была самой удалённой от плоскости. 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.

§20. Доказательство непланарности графов K_5 и $K_{3,3}$. Теорема Понтрягина-Куратовского (доказательство в одну сторону).

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

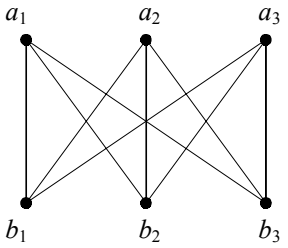


K_5

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. По-

Основы математической логики и логического программирования

Владимир Анатольевич Захаров

zakh@cs.msu.su

http://mathcyb.cs.msu.su/courses/logprog.html

Билет 1.

Классическая логика предикатов первого порядка. Выполнимые и общезначимые формулы. Метод резолюции проверки общезначимости формул.

АЛФАВИТ

Базовые символы.

Предметные переменные $Var = \{x_1, x_2, \ldots, x_k, \ldots\}$;

Предметные константы $Const = \{c_1, c_2, \ldots, c_l, \ldots\}$;

Функциональные символы $Func = \{f_1^{(n_1)}, f_2^{(n_2)}, \ldots, f_r^{(n_r)}, \ldots\}$;

Предикатные символы $Pred = \{P_1^{(m_1)}, P_2^{(m_2)}, \ldots, P_s^{(m_s)}, \ldots\}$.

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

АЛФАВИТ

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	( )

СИНТАКСИС: ТЕРМЫ И ФОРМУЛЫ

Определение терма.

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 $Var_t \subseteq \{x_1, x_2, \ldots, x_n\}$.

СИНТАКСИС: ТЕРМЫ И ФОРМУЛЫ

Определение формулы.

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;
$(\varphi \vee \psi)$
$(\varphi \to \psi)$
$(\neg\varphi)$

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

Form — множество всех формул заданного алфавита.

СИНТАКСИС: ТЕРМЫ И ФОРМУЛЫ

Свободные и связанные переменные.

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.

СИНТАКСИС: ТЕРМЫ И ФОРМУЛЫ

Свободные и связанные переменные.

Var_φ — множество свободных переменных формулы φ.

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

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

$$\varphi = (\neg\psi) \quad Var_\varphi = Var_\psi;$$

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

СИНТАКСИС: ТЕРМЫ И ФОРМУЛЫ

$\varphi(x_1, x_2, \ldots, x_n)$ — 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:

$$\neg,\ \forall,\ \exists$$
$$\&$$
$$\vee$$
$$\rightarrow$$

СЕМАНТИКА: ИНТЕРПРЕТАЦИИ

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.

СЕМАНТИКА: ИНТЕРПРЕТАЦИИ

Значение терма

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
  $$t(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n] = \bar{\mathbf{f}}(t_1[d_1, d_2, \ldots, d_n], \ldots, t_k[d_1, d_2, \ldots, d_n]).$$

СЕМАНТИКА: ВЫПОЛНИМОСТЬ ФОРМУЛ

Отношение выполнимости формул

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

$$I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$$
$$\Longleftrightarrow$$
$$\bar{\mathbf{P}}(t_1[d_1, d_2, \ldots, d_n], \ldots, t_m[d_1, d_2, \ldots, d_n]) = \mathbf{true};$$

СЕМАНТИКА: ВЫПОЛНИМОСТЬ ФОРМУЛ

Отношение выполнимости формул

- 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

$$I \models \psi_2(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$$

СЕМАНТИКА: ВЫПОЛНИМОСТЬ ФОРМУЛ

Отношение выполнимости формул

- Если $\varphi(x_1, x_2, \ldots, x_n) = \psi_1 \to \psi_2$, то

$$I \models \varphi(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$$
$$\Longleftrightarrow$$
$$I \not\models \psi_1(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$$
$$\text{или}$$
$$I \models \psi_2(x_1, x_2, \ldots, x_n)[d_1, d_2, \ldots, d_n]$$

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

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

СЕМАНТИКА: ВЫПОЛНИМОСТЬ ФОРМУЛ

Отношение выполнимости формул

- 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

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

ВЫПОЛНИМЫЕ И ОБЩЕЗНАЧИМЫЕ ФОРМУЛЫ

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.

МОДЕЛИ. ЛОГИЧЕСКОЕ СЛЕДСТВИЕ

Определение

Пусть Γ — некоторое множество замкнутых формул, $\Gamma \subseteq \mathit{CForm}$. 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 $\models \varphi$.

МОДЕЛИ. ЛОГИЧЕСКОЕ СЛЕДСТВИЕ

Теорема о логическом следствии

Пусть $\Gamma = \{\psi_1, \ldots, \psi_n\} \subseteq \mathit{CForm}$, $\varphi \in \mathit{CForm}$. **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 $I \models \psi_i$, $1 \leq i \leq n$, 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.

МОДЕЛИ. ЛОГИЧЕСКОЕ СЛЕДСТВИЕ

Теорема о логическом следствии

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

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

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

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 $\Gamma \models \varphi$.

ПРОБЛЕМА ОБЩЕЗНАЧИМОСТИ ФОРМУЛ

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

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

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

ПРОБЛЕМА ОБЩЕЗНАЧИМОСТИ ФОРМУЛ

Таким образом, возникает проблема общезначимости формул:

Для заданной формулы φ проверить ее общезначимость:

$$\models \varphi?$$

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. $I \models \varphi_0 \Leftrightarrow I \models \varphi_1$.

ОБЩАЯ СХЕМА МЕТОДА РЕЗОЛЮЦИЙ

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 $\iff$ 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 $\iff$ система дизъюнктов S_φ противоречива.

ОБЩАЯ СХЕМА МЕТОДА РЕЗОЛЮЦИЙ

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.

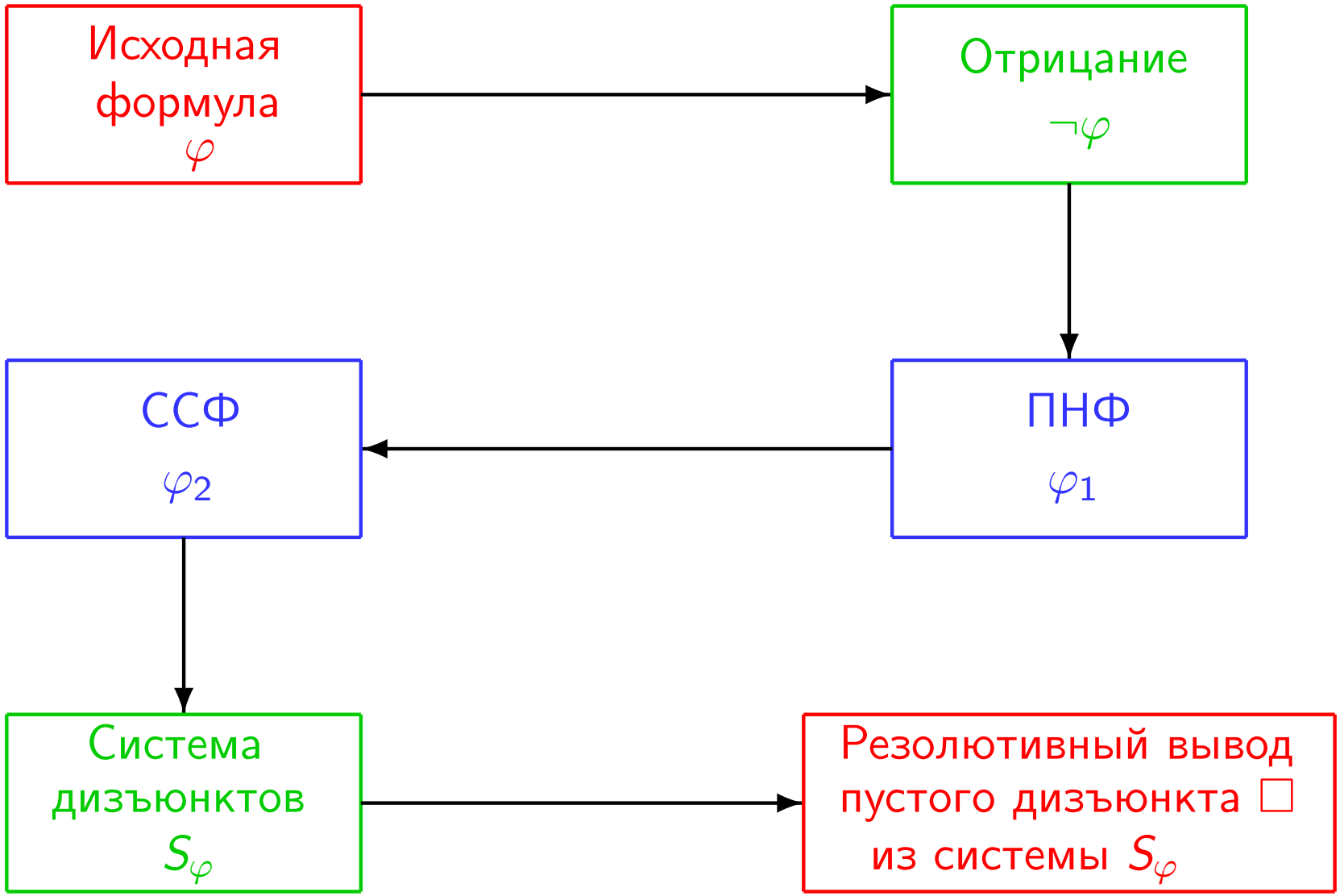
Резольвенты строят, пока не будет получен пустой дизъюнкт $\square$.
Это возможно в случае $D_1 = L$, $D_2 = \neg L$:

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

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 $\square$.

ОБЩАЯ СХЕМА МЕТОДА РЕЗОЛЮЦИЙ



РАВНОСИЛЬНЫЕ ФОРМУЛЫ

Введем вспомогательную логическую связку эквиваленции $\equiv$.
Выражение $\varphi \equiv \psi$ — это сокращенная запись формулы $(\varphi \to \psi)\&(\psi \to \varphi)$.

Определение

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

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

ПРЕДВАРЕННЫЕ НОРМАЛЬНЫЕ ФОРМЫ

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

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

где

- $Q_1 x_1 \; Q_2 x_2 \; \dots \; Q_n x_n$ — кванторная приставка , состоящая из кванторов $Q_1, Q_2, \dots, Q_n$,
- $M(x_1, x_2, \dots, x_n)$ — матрица — бескванторная конъюнктивная нормальная форма (КНФ), т. е.
$$M(x_1, x_2, \dots, x_n) = D_1 \;\&\; D_2 \;\&\; \dots \;\&\; D_N,$$
где $D_i = L_{i1} \vee L_{i2} \vee \dots \vee L_{ik_i}$ — 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.

СКОЛЕМОВСКИЕ СТАНДАРТНЫЕ ФОРМЫ

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

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

СИСТЕМЫ ДИЗЪЮНКТОВ

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.

СИСТЕМЫ ДИЗЪЮНКТОВ

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.

СИСТЕМЫ ДИЗЪЮНКТОВ

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.

Итак, проверка общезначимости $\models \varphi\,?$ сводится к проверке противоречивости системы дизъюнктов S_φ.

РЕЗОЛЮТИВНЫЙ ВЫВОД

О терминологии.

Пусть задано выражение E и подстановка θ.

Подстановка θ : **Var** $\rightarrow$ **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

$$\eta = \theta\rho$$

РЕЗОЛЮТИВНЫЙ ВЫВОД

Правило резолюции.

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

Пусть $\theta \in НОУ(L_1, L_2)$.

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

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

РЕЗОЛЮТИВНЫЙ ВЫВОД

Правило склейки.

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

Пусть $\eta \in НОУ(L_1, L_2)$.

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

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

РЕЗОЛЮТИВНЫЙ ВЫВОД

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, $1 \leq i \leq n$, 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

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 $\square$.

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Пример.

Рассмотрим формулу φ

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

Задача

Проверить, верно ли, что $\models \varphi$.

Решение

Методом резолюций.

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

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

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 2. Приведем φ_1 к ПНФ φ_2.

Исходная формула

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 2. Приведем φ_1 к ПНФ φ_2.

Переименование переменных

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 2. Приведем φ_1 к ПНФ φ_2.

Удаление импликаций

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 2. Приведем φ_1 к ПНФ φ_2.

Продвижение отрицаний

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 2. Приведем φ_1 к ПНФ φ_2.

Вынесение кванторов

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 3. Приведем φ_2 к ССФ φ_3.

$$\varphi_3 = \quad \forall y' \quad \forall u \quad \forall z \forall y'' \quad \Big(\quad (\neg A(u, f(y')) \vee B(y', u)) \;\&\; (A(g(y', u), u) \vee A(z, f(y'))) \;\&\; \neg B(c, y'') \quad \Big)$$

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

Этап 4. Формирование системы дизъюнктов S_φ.

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

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

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')

ПРИМЕНЕНИЕ МЕТОДА РЕЗОЛЮЦИЙ

Решение

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,

$$\models \varphi.$$

КОНЕЦ ОТВЕТА НА БИЛЕТ 1.

Основы математической логики и логического программирования

В.А. Захаров

Билет 2.

Хорновские логические программы: синтаксис. Декларативная семантика логических программ. Операционная семантика логических программ. Стратегии вычисления логических программ.

ХОРНОВСКИЕ ЛОГИЧЕСКИЕ ПРОГРАММЫ

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»

ХОРНОВСКИЕ ЛОГИЧЕСКИЕ ПРОГРАММЫ

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.

ХОРНОВСКИЕ ЛОГИЧЕСКИЕ ПРОГРАММЫ

Как нужно понимать логические программы?

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.

ХОРНОВСКИЕ ЛОГИЧЕСКИЕ 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 $C_1, C_2, \ldots, C_m$.

ДЕКЛАРАТИВНАЯ СЕМАНТИКА

Более строгое описание семантик требует привлечения аппарата математической логики.

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

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

ДЕКЛАРАТИВНАЯ СЕМАНТИКА

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.

ДЕКЛАРАТИВНАЯ СЕМАНТИКА

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

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

ДЕКЛАРАТИВНАЯ СЕМАНТИКА

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 $\mathcal{P} \models (C_1 \& \ldots \& C_m)\theta$.

ОПЕРАЦИОННАЯ СЕМАНТИКА ЛОГИЧЕСКИХ ПРОГРАММ

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

SLD-РЕЗОЛЮТИВНЫЕ ВЫЧИСЛЕНИЯ

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 $Var_G \cap Var_{D'} = \emptyset$,
- $\theta \in HOУ(C_i, A'_0)$ — 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.

SLD-РЕЗОЛЮТИВНЫЕ ВЫЧИСЛЕНИЯ

Определение (SLD-резолютивного вычисления)

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.

SLD-РЕЗОЛЮТИВНЫЕ ВЫЧИСЛЕНИЯ

Определение (SLD-резолютивного вычисления)

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.

SLD-РЕЗОЛЮТИВНЫЕ ВЫЧИСЛЕНИЯ

Определение (SLD-резолютивного вычисления)

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 $\mathcal{P}$.

SLD-РЕЗОЛЮТИВНЫЕ ВЫЧИСЛЕНИЯ

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).

Какова связь между правильными и вычисленными ответами?

КОРРЕКТНОСТЬ ОПЕРАЦИОННОЙ СЕМАНТИКИ

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 $\mathcal{P}$.

ПОЛНОТА ОПЕРАЦИОННОЙ СЕМАНТИКИ

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.

ПРАВИЛА ВЫБОРА ПОДЦЕЛЕЙ

Определение.

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

$$\theta = \eta\rho$$

ДЕРЕВЬЯ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

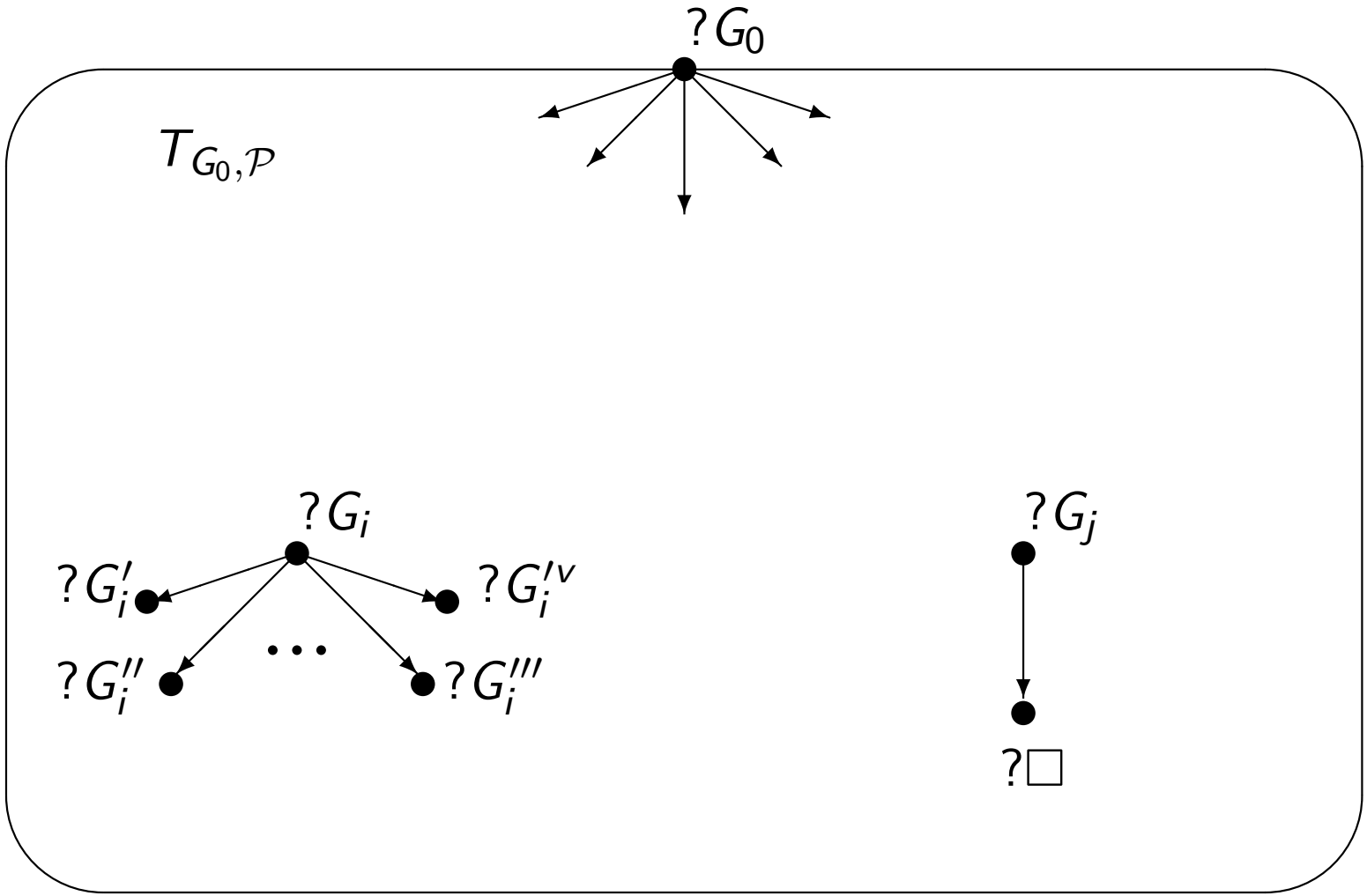
Определение

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).

ДЕРЕВЬЯ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

Иллюстрация



СТРАТЕГИИ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

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 $\mathcal{P}$

СТРАТЕГИИ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

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.

СТРАТЕГИИ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

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.

СТРАТЕГИИ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

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.

СТРАТЕГИИ ВЫЧИСЛЕНИЙ ЛОГИЧЕСКИХ ПРОГРАММ

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.

КОНЕЦ ОТВЕТА НА БИЛЕТ 2.

5.Транзакционное управление в СУБД. Методы сериализации транзакций.

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. 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.

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.

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.

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.

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.

При соблюдении обязательного требования поддержания целостности базы данных возможны следующие уровни изолированности транзакций:

1) Первый уровень - отсутствие потерянных изменений.
2) Второй уровень - отсутствие чтения "грязных данных".

3) Третий уровень - отсутствие неповторяющихся чтений.

К более тонким проблемам изолированности транзакций относится так называемая проблема кортежей-"фантомов".

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:

1) W-W - транзакция 2 пытается изменять объект, измененный не закончившейся транзакцией 1;
2) R-W - транзакция 2 пытается изменять объект, прочитанный не закончившейся транзакцией 1;
3) W-R - транзакция 2 пытается читать объект, измененный не закончившейся транзакцией 1.

Существуют два базовых подхода к сериализации транзакций - основанный на синхронизационных захватах объектов базы данных и на использовании временных меток.

Синхронизационные захваты

В общих чертах протокол состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных r от имени транзакции T запрашивается синхронизационный захват объекта r в соответствующем режиме (в зависимости от вида операции).

Основными режимами синхронизационных захватов являются:

1) совместный режим - S (Shared), означающий разделяемый захват объекта и требуемый для выполнения операции чтения объекта;
2) монопольный режим - X (eXclusive), означающий монопольный захват объекта и требуемый для выполнения операций занесения, удаления и модификации.

	S	X
-	+	+
S	+	-
X	-	-

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:

1) первая фаза транзакции - накопление захватов;
2) вторая фаза (фиксация или откат) - освобождение захватов.

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.

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.

IS (IntentedforSharedlock) 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 (IntentedforeXclusivelock) 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, IntentedforeXclusivelock) 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).

	X	S	IS	IX	SIX
-	+	+	+	+	+
X	-	-	-	-	-
S	-	+	+	-	-
IS	-	+	+	+	+
IX	-	-	+	+	-
SIX	-	-	+	-	-

Предикатные синхронизационные захваты

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.

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

имя-атрибута { = >< } значение

Для простых условий совместимость предикатных захватов легко определяется на основе следующей геометрической интерпретации. Пусть R отношение с атрибутами a1, a2, ..., an, а m1, m2, ..., mn - множества допустимых значений a1, a2, ..., an 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.

Тупики, распознавание и разрушение

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных захватов является возможность возникновение тупиков (deadlocks) между транзакциями.

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.

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.

Методвременныхметок

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.

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

Прерыванием называется событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий. Состав прерываний — множество разновидностей событий, на возникновение которых предусмотрена стандартная реакция центрального процессора, — фиксирован и определяется конструктивно при разработке компьютера. Аппарат прерываний компьютера позволяет организовывать стандартную обработку всех прерываний, возникающих при функционировании вычислительной системы.

Традиционно прерывания разделяются на две группы: внутренние прерывания и внешние прерывания.

Внутренние прерывания инициируются схемами контроля работы процессора. К примеру, внутреннее прерывание может возникнуть в процессоре при попытке выполнения команды деления, операнд-делитель которой равен нулю. Также внутреннее прерывание возникнет в ситуации, когда при обработке очередной команды адрес одного из операндов выходит за пределы адресного пространства оперативной памяти.

Внешние прерывания — события, возникающие в компьютере в результате взаимодействия центрального процессора с внешними устройствами. Примером внешнего прерывания может служить событие, связанное с вводом символа с клавиатуры персонального компьютера.

Обработка прерывания предполагает две стадии: аппаратную, которая включает реакцию процессора на возникновение прерывания, и программную, которая предполагает выполнение специальной программы обработки прерывания, являющейся частью операционной системы.

Этап аппаратной обработки прерывания.

1) Завершается выполнение текущей команды (за исключением случаев, когда прерываниевозникает по причине некорректного выполнения команды).
2) Обработка прерывания предполагает остановку выполнения текущей программы, запускспециальной программы обработки прерывания, а затем, возможно, продолжение выполнениянрерванной программы. Поэтому аппаратный этап обработки прерываний регламентируетперечень регистров, которые автоматически будут сохранены процессором. Это специальныерегистры, содержимое которых описывает состояние процессора в точке прерываниявыполнения программы (счетчик команд, регистр результатов, регистры, содержащие режимыработы процессора), а также несколько регистров общего назначения, которые могут бытьиспользованы программой обработки

прерываний в начальный момент времени. Процедурааппаратного сохранения регистров в различных компьютерах может происходить по-разному.Простейшая модель следующая. Включается режим блокировки прерываний. При этомрежиме в системе запрещается инициализация новых прерываний: возникающие в это времяпрерывания могут либо игнорироваться, либо откладываться.

3) Аппаратное копирование содержимого сохраняемых регистров. Включенный режимблокировки прерывания гарантирует сохранность этих данных до момента завершенияпредварительной обработки прерывания и выключения блокировки прерываний.
4) Переход на программный этап обработки прерываний.

Модели перехода на программный этап:

1) Использование регистра прерываний, по которому определяется програмный обработчик прерывания
2) Использование регистра слова состояния процессора. Часть разрядов используется для хранения номера прерывания.
3) Использование вектора прерываний. Предполагается, что поколичеству возможных прерываний в ОЗУ выделена группа машинных слов — векторпрерываний. Каждое слово вектора прерываний содержит адрес программы,обрабатывающей данное прерывание.

Этап программной обработки прерывания:
1) Анализ и предварительная обработка прерывания. Происходит идентификация типа прерывания, определяются причины.

- Если прерывание «короткое», т.е. обработка не требует дополнительных ресурсов ЦП ивремени, то прерывание обрабатывается, выключается режим блокировки прерываний, восстанавливается состояние процессора, соответствующее точке прерывания исходной программы, и передается управление на прерванную точку.

- Если прерывание является «фатальным» для программы, т.е. после этого прерыванияпродолжить выполнение программы невозможно (например, в программе произошло обращение к несуществующему в ОЗУ адресу), то выключается режим блокировкипрерываний, и управление передается в ту часть ОС, которая прекратит выполнение прерванной программы.

2) Сохранение всех регистров ЦП, использовавшихся прерванной программой, в специальную программную таблицу. В данную таблицу копируется содержимое регистровой или КЭШ-памяти, содержащей сохраненные значения ресурсов ЦП, а также копируются все оставшиеся регистры ЦП, используемые программно, но не сохраненные аппаратно.

3) Снятие режима блокировки прерываний

4) Операционная система завершает обработку прерывания.

Мультипрограммный режим - в обработке могут находиться две и более программы пользователей, и каждая из этих программ может находиться в одном из трех состояний: во-первых, программа может выполняться на процессоре, во-вторых, программа может ожидать завершения запрошенного ею обменаив-третьих, программы могут находиться в ожидании освобождения центрального процессора.

Первая проблема, которая может возникнуть, — это влияние программ друг на друга.

Очень нежелательна ситуация, когда одна программа может обратиться в адресное пространство другой программы и считать оттуда данные (поскольку все-таки необходимо обеспечивать конфиденциальность информации), и уж совсем плоха ситуация, когда другая программа может что-то записать в чужое адресное пространство. Соответственно, для корректного мультипрограммирования система должна обеспечивать эксклюзивное владение программ выделенными им участками памяти. Если возникает задача обеспечения множественного доступа к памяти, то это должно осуществляться с согласия владельца этой памятью. Итак, первое требование к системе — это наличие т.н. аппарата защиты памяти.

Реализация аппарата защиты памяти может быть достаточно простой: в процессоре могут быть специальные регистры (регистры границ), в которых устанавливаются границы диапазона доступных для исполняемой задачи адресов оперативной памяти. Соответственно, когда устройство управления в центральном процессоре вычисляет очередной исполнительный адрес (это может быть адрес следующей команды или же адрес необходимого операнда), автоматически проверяется, принадлежит ли полученный адрес заданному диапазону.

Система должна каким-то способом ранжировать и в соответствии с этим ранжированием ограничивать доступ пользователей различных категорий к машинным командам. Решением стала аппаратная возможность работы центрального процессора в двух режимах: в режиме работы операционнойсистемы (или привилегированном режиме, или режиме супервизора) и в пользовательском режиме.

Если программа, выполняющаяся в пользовательском режиме хочет получить доступ к внешним устройствам, то ей необходимо обратиться к ОС с соответствующим запросом.

ОС накапливает запросы на обращение к периферийным устройствам, а затем осуществляет запросы по одному из трех событий:

1) Программа успешно завершилась
2) Программа упала с фатальной ошибкой
3) Получена команда от планировщика

Еще одна проблема – программа может зациклиться, что привет к остановке работы всей системы. Для решения этой проблемы операционная система должна контролировать время использования центральногопроцессора программами пользователя. Для этих целей компьютеру требуется прерывание потаймеру. Резюмируя, можно сказать, что для

реализации мультипрограммного режима необходимо наличие аппарата прерываний, и этот аппарат, как минимум, должен включать в себя аппарат прерывания по таймеру.

Включение/выключение режима супервизора зависит от архитектуры. Например, при входе в определенную область памяти режим супервизора включается. Выключение режима супервизора, как правило, реализуется программно, т.е. ОС при запуске процесса понижает его права.

7. Организация взаимодействия процессов и средства их синхронизации. Классические задачи синхронизации.

Будем говорить, что процессы называются параллельными, если время их выполнения хотя бы частично перекрываются.

Параллельные процессы могут быть независимыми и взаимодействующими.

Независимые процессы используют множество независимых ресурсов, т.е. те ресурсы, которые принадлежат независимым процессам, в пересечении дают пустое множество. Альтернативой независимым процессам являются взаимодействующие процессы — те процессы, пересечение множеств ресурсов которых непустое.

Совместное использование ресурсов двумя и более процессами, когда каждый из них некоторое время владеет этими ресурсами, называется разделением ресурсов (как аппаратных, так и программных, или виртуальных). Разделяемый ресурс, использование которого организовано таким образом, что он может быть доступен в каждый момент времени только одному из взаимодействующих процессов, называется критическим ресурсом. Соответственно, часть программы, в рамках которой осуществляется работа с критическим ресурсом, называется критической секцией.

Ситуация, когда процессы конкурируют за разделяемый ресурс, называются гонкой процессов (raceconditions).

Для минимизации проблем, возникающих при гонках, используется взаимное исключение — такой способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

Проблемы, которые могут возникать при организации взаимного исключения — это тупики и блокировки.

Блокировка — это ситуация, когда доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается за счет активности более приоритетных процессов.

Тупик, или deadlock, — это ситуация «клинчевая», когда из-за некорректной организации доступа к разделяемым ресурсам происходит взаимоблокировка.

Рассмотрим три способа организации взаимодействия между процессорами:

1) Семафоры Дейкстры

   Имеется специальный тип данных — семафор. Переменная типа семафор может иметь целочисленные значения. Над этими переменными определены следующие**атомарные** операции: down(S) (или P(S)) и up(S) (или V(S)).

Операция down(S) проверяет значение семафора S, и если оно больше нуля, то уменьшает его на 1. Если же это не так, процесс блокируется, причем связанная с заблокированным процессом операция down считается незавершенной.

Операция up(S) увеличивает значение семафора на 1. При этом, если в системе присутствуют процессы, блокированные ранее при выполнениидown на этом семафоре, один из них разблокировывается и завершает выполнение операции down, т.е. вновь уменьшает значение семафора. Выбор процесса никак не оговаривается.

Перед входом в критическую секцию процесс вызывает down(S), где S - семафор, отвечающий за соответствующую критическую секцию. После выходы – вызывает up(S).

2) Мониторы Хоара
   Монитор — это специализированный модуль, включающий в себя некие процедуры и функции, а также данные, с которыми работают эти процедуры и функции. При этом данный модуль обладает следующими свойствами:
   - данные монитора доступны только через процедуры и функции этого монитора;
   - считается, что процесс занимает (или входит) монитор тогда, когда он начинает использоватьодну из процедур или функций монитора;
   - в любой момент времени внутри монитора может находиться не более одного процесса,остальные процессы в зависимости от используемой стратегии поведения либо получает отказ,либо блокируется, становясь в очередь.
3) Аппарат передачи сообщений
   Механизм основан на двух функциональных примитивах: send(отправить сообщение) и receive (принять сообщение).
   Операции посылки/приема сообщений могут быть блокирующими и неблокирующими.
   Адресация может быть прямой, когда указывается конкретный адрес получателя и/или отправителя (например, когда получатель ожидает сообщения от конкретного отправителя,игнорируя сообщения других отправителей), или косвенной. В случае косвенной адресации не указывается адрес получателя при отправке или отправителя при получении; сообщение «бросается» в некоторый общий пул, в котором могут быть реализованы различные стратегии доступа (FIFO, LIFO и т.д.).

Классические задачи синхронизации

Обедающие философы. Пускай существует круглый стол, за которым сидит группа философов: они пришли пообщаться и покушать. Кушают они спагетти, которое находится вобщей миске, стоящей в центре стола. Для приема пищи они пользуются двумя вилками: одна в левой руке, другая — в правой. Вилки располагаются по одной между каждыми

двумя философами. Любой философ может взять обе вилки, покушать, затем положить вилки на стол, после этого вилки может взять его сосед и повторить эти действия.Если мы организуем работу таким образом, что любой философ, желающий поесть, берет сначала левую вилку, затем правую, после чего начинает кушать, то в какой-то момент может возникнуть ситуация тупика (когда каждый возьмет по одной левой вилке, а правая будет захвачена соседом).

Чтобы решить ситуацию с тупиком, рассмотрим следующий алгоритм.

Пусть ученый может находиться в одном из трех состояний : думает, желает поесть, ест.

Каждый философ живет по аналогичному циклическому распорядку: размышляет некоторое время, затем берет вилки, кушает, кладет вилки.

Рассмотрим процедуру получения вилок. Опускается семафор mutex, который используется для синхронизации входа в критическую секцию. Внутри критической секции меняем состоянии философа (помечаем его состоянием «голоден»). Затем предпринимается попытка начать есть(вызывается функция Test). Функция Test проверяет, что если i-ый философ голоден, а его соседи в данный момент не едят (т.е. правая и левая вилки свободны), то этот философ начинает прием пищи (состояние EATING), а его семафор поднимается (заметим, что изначально этот семафор инициализирован нулем). После этого мы возвращаемся обратно в функцию TakeForks, в которой далее происходит выход из критической секции (подымаем семафор mutex), а затем опускаем семафор этого философа. Если внутри функции Test философу удалось начать прием пищи, то семафор поднят, и операция down обнулит его, не блокируясь. Если же функция Test не изменит состояние философа, а также не поднимет его семафор, то операция down в этой точке заблокируется до тех пор, пока оба соседа не освободят вилки.Внутри функции освобождения вилок PutForks первым делом происходит опускание семафора mutex, происходит вход в критическую секцию. Затем меняется статус философа (на статус THINKING), после чего проверяем его соседей: если любой из них был заблокирован лишь из-за того, что наш i-ый философ забрал его вилку, то мы его разблокируем, и он начинает прием пищи. После этого происходит выход из критической секции путем подъема семафора mutex.

Задача «читателей и писателей»

Одни процессы могут читать информацию, а другие — ее изменять, корректировать. Соответственно, возникает все тот же вопрос, как организовать корректную совместную работу этих процессов. Это означает, что в любой момент времени читать данные могут любое количество процессов-читателей, но если процесс-писатель начал свою работу, то все остальные процессы (и читатели, и писатели) будут блокированы на входе в систему.

В приведенном решении процесс-читатель в каждом цикле своей работы входит в критическую секцию (за счет опускания семафора mutex), увеличивает счетчик читателей,находящихся в хранилище, на 1. Затем проверяет, что если он является первым читателем (т.е. вданный момент он единственный клиент в хранилище), то опускает

семафор db, тем самым, препятствуя писателем войти в систему, если они того пожелают. Если же семафор db уже был опущен, то это означает, что в данный момент в хранилище присутствует писатель, и этот первый читатель заблокируется на этой операции, ожидая выхода писателя из системы. (Заметим, что это блокировка происходит внутри критической секции, поэтому остальные читатели будут блокироваться на опускании семафора mutex.) После этого происходит выход из критической секции (подымаем семафор mutex), чтение информации из хранилища.на выходе мы уменьшаем число читателей в хранилище, и если этот читатель является последним клиентом в библиотеке, то происходит поднятие семафора db, разрешая работу писателям (которые к этому моменту могли быть заблокированы на входе). В конце цикла работы читатель обрабатывает полученные данные из хранилища, после чего цикл повторяется.

Писатель в начале каждого цикла своей работы подготавливает данные для сохранения, затем пытается войти в хранилище, опуская семафор db. Если в хранилище кто-то есть, то он будет ожидать, пока последний клиент (независимо, читатель это или писатель) не покинет его. После этого он производит корректировку данных в хранилище и покидает его, поднимая семафор db.

Задача о «спящем парикмахере»

Представим себе парикмахерскую, в которой имеется единственно рабочее кресло и единственный цирюльник. В парикмахерской есть комната ожидания, в которой стоят N кресел, на которых могут сидеть клиенты, ожидающие своей очереди. Если свободных кресел нет, то вновь приходящие клиенты сразу же покидают заведение. Когда посетителей нет, парикмахер может сидеть в своем кресле и дремать. Данная задача является иллюстрацией модели клиент-сервер с ограничением на длину очереди клиентов.

Процесс-парикмахер первым делом опускает семафор customers, уменьшив тем самым количество ожидающих посетителей на 1. Если в комнате ожидания никого нет, то он «засыпает» в своем кресле, пока не появится клиент, который его разбудит. Затем парикмахер входит в критическую секцию, уменьшает счетчик ожидающих клиентов, поднимает семафорbarbers, сигнализируя клиенту о своей готовности его обслужить, а потом выходит из критической секции. После описанных действий он начинает стричь волосы посетителю.

Посетитель парикмахерской входит в критическую секцию. Находясь в ней, он первым делом проверяет, есть ли свободные места в зале ожидания. Если нет, то он просто уходит (покидает критическую секцию, поднимая семафор mutex). Иначе он увеличивает счетчик ожидающих процессов и поднимает семафор customers. Если же этот посетитель является единственным в данный момент клиентом брадобрея, то он этим действием разбудит брадобрея. После этого он выходит из критической секции и «захватывает» брадобрея (опуская семафор barbers). Если же этот семафор опущен, то клиент будет дожидаться, когда брадобрей егоподнимет, известив тем самым, что готов к работе. В конце клиент обслуживается (GetHaircut).

8. Виртуальная память. Модели организации оперативной памяти.

Аппарат виртуальной памяти — это аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе, адресам физической памяти, в которой размещена программа при выполнении. И реализацией одной из моделей аппарата виртуальной памяти является аппарат базирования адресов.

Механизм базирования адресов основан на двоякой интерпретации получаемых в ходе выполнения программы исполнительных адресов. С одной стороны, его можно интерпретировать как абсолютный исполнительный адрес, когда физический адрес в некотором смысле соответствует исполнительному адресу программы. С другой стороны, исполнительный адрес программы можно проинтерпретировать как относительный адрес, т.е. адрес, зависящий от места дислокации программы в ОЗУ.

Во втором случае реальный физический адрес будет вычисляться как сумма регистра базы и относительного адреса.программа представляется в виде непрерывной области виртуальной памяти, которая загружается в непрерывный фрагмент физической памяти.

Модели организации памяти

1) Одиночное непрерывное распределение

   Вся память делится на две области - В одной части памяти располагается и функционирует операционная система, а другая часть выделяется для выполнения прикладных процессов.

   Граница между этими областями обеспечивается с помощью регистра границ, если получаемый исполнительный адрес оказывается меньше значения этого регистра, то это адрес в пространстве операционной системы, иначе в пространстве процесса.

2) Распределение неперемещаемыми разделами
   Все адресноепространство оперативной памяти делится на две части. Одна часть отводится под операционнуюсистему, все оставшееся пространство отводится под работу прикладных процессов, причем этопространство заблаговременно делится на N частей (назовем их разделами), каждая из которых вобщем случае имеет произвольный фиксированный размер. Эта настройка происходит на уровнеоперационной системы. Соответственно, очередь прикладных процессов разделяется по этимразделам.

   Существуют концептуально два варианта организации этой очереди. Первый вариант предполагает наличие единственной сквозной очереди, которая по каким-то соображениям распределяется между этими разделами. Второй вариант организовантак, что с каждым разделом ассоциируется своя очередь, и поступающий процесс сразу попадает водну из этих очередей.

Контроль может осуществляться с помощью граничных регистров.

3) Распределение перемещаемыми разделами
   Данная модель распределения разрешает загрузку произвольного (нефиксированного) числа процессов в оперативную память, и под каждый процесс отводится раздел необходимогоразмера. Соответственно, система допускает перемещение раздела, а, следовательно, и процесса.
   Такой подход позволяет избавиться от фрагментации.
   Что касается аппаратной поддержки, то здесь она аналогична предыдущей модели: требуются аппаратные средства защиты памяти (регистры границ или же ключи защиты) иаппаратные средства, позволяющая осуществлять перемещение процессов (в большинстве случаевдля этих целей используется регистр базы, который в некоторых случаях может совпадать с однимиз регистров границ).
4) Страничная организация
   Данная модель представляет все адресное пространство оперативной памяти в виде последовательности страниц. Страница — это область адресного пространства фиксированногоразмера: обычно размер страницы кратен степени двойки, будем считать, что размер страницы 2^k.
   Структура адреса представима в виде двух полей :правые k разрядов представляют адрес внутри страницы, а оставшиеся разряды отвечают за номерстраницы.
   В центральном процессоре имеется аппаратная таблица, называемая таблицей страниц, предназначенная для следующих целей. Количество строк в этой таблице определяетсямаксимальным числом виртуальных страниц, ограниченное схемами работы процессора имаксимальной адресной разрядностью процессора. Каждой виртуальной странице ставится всоответствие строка таблицы страниц с тем же номером (нулевой странице соответствует нулеваястрока, и т.п.). Внутри каждой записи таблицы страниц находится номер физической страницы, вкоторой размещается соответствующая виртуальная страница программы. Аппарат виртуальной страничной памяти позволяет автоматически (т.е. аппаратно) преобразовывать номер виртуальной страницы на номер физической страницы посредствомобращения к таблице страниц.

   К аппаратуре предъявляются следующие требования: должен быть регистр, ссылающийся на начало таблицы в ОЗУ, а также должно аппаратноподдерживаться обращение в оперативную память по адресу, хранящемуся в указанном регистре, извлечение данных изтаблицы и осуществление преобразования.
   В качестве одного из первых решений оптимизации работы с памятью стало использование т.н. TLB-таблиц.
   Данный метод подразумевает наличие аппаратной таблицы относительно небольшого размера (порядка 8 – 128 записей). Данная таблицы концептуально содержит три столбца: первыйстолбец — это номер виртуальной страницы, второй — это номер физической страницы, вкоторой находится указанная виртуальная страница, а третий столбец содержитатрибут присутствия/отсутствия страницы,

атрибут режима защиты страницы (чтение, запись, выполнение), флаг модификации содержимого страницы, атрибут, характеризующий обращения к данной странице, чтобы иметь возможностьопределения «старения» страницы, атрибут блокировки кэширования и т.д.
Теперь, имея виртуальный адрес, состоящий из номера виртуальной страницы (VP) и смещения в ней (offset). Страница изымает из этого адреса номер виртуальной страницы иосуществляет оптимизированный поиск (т.е. поиск не последовательный, а параллельный) этогономера по TLB-таблице. Если искомый номер найден, то система автоматически на уровнеаппаратуры осуществляет проверку соответствия атрибутов, и если проверка успешна, топроисходит подмена номера виртуальной страницы номером физической страницы, и, таким образом, получается физический адрес. Если же при поиске происходит промах (номер виртуальной странице не найден), то в этомслучае система обращается в программную таблицу, выкидывает самую старую запись из TLB,загружает в нее найденную запись из программной таблицы, и затем вычисляется физический адрес. Таким образом, получается, что TLB-таблица является некоторым КЭШем.
Одним из решений, позволяющих снизить размер таблицы страниц, является модель иерархической организации таблицы страниц . В этом случае информация о страницепредставляется не в виде одного номера страницы, а в виде совокупности номеров, используякоторые посредством обращения к соответствующим таблицам, участвующим в иерархии (этоможет быть 2-х-, 3-х- или даже 4-хуровневая иерархия), можно получить номер соответствующей физической страницы.

5) Сегментная организация
Данная модель представляет каждый процесс в виде совокупности сегментов, каждый из которых может иметь свой размер. Каждый из сегментовможет иметь собственную функциональность: существуют сегменты кода, сегменты статическихданных, сегмент стека и т.д. Для организации работы с сегментами может использоватьсянекоторая таблица, в которой хранится информация о каждом сегменте (его номер, размер и пр.).Тогда виртуальный адрес может быть проинтерпретирован, как номер сегмента и величина
смещения в нем.
Существует аппаратная таблица сегментов с фиксированным числом
записей. Каждая запись этой таблицы соответствует своему сегменту и хранит информацию о размере сегмента и адрес начала сегмента (т.е. адрес базы), а также тут могут присутствоватьразличные атрибуты, которые будут оговаривать права и режимы доступа к содержимому
сегмента.
Итак, имея виртуальный адрес, система аппаратным способом извлекает из него номерсегмента i, обращается к i-ой строке таблицы, из которой извлекается информация о сегменте.После чего происходит проверка, не превосходит ли

величина сегмента размера самого сегмента.Если превосходит, то происходит прерывание, иначе, складывая базу со смещением, вычисляетсяфизический адрес.

6) Сегментно-страничная организация

   Эта модель рассматривает виртуальный адрес, как номер сегмента и смещение в нем. Имеется также аппаратная таблица сегментов, посредствомкоторой из виртуального адреса получается т.н. линейный адрес, который, в свою очередь, представляется в виде номера страницы и величины смещения в ней. А затем, используя таблицу страниц, получается непосредственно физический адрес.
   Итак, данный механизм подразумевает, что в процессе имеется ряд виртуальных сегментов, которые дробятся на страницы. Поэтому данная модель сочетает в себе, с одной стороны,логическое сегментирование, а с другой стороны, преимущества страничной организации (когдаможно работать с отдельными страницами памяти, не требуя при этом полного размещениясегмента в ОЗУ).

9. Алгоритм Сети-Ульмана оптимального распределения регистров и его обоснование.

Пусть система команд машины имеет неограниченное число универсальных регистров, в которых выполняются арифметические команды. Рассмотрим, как можно сгенерировать код, используя для данного арифметического выражения минимальное число регистров.

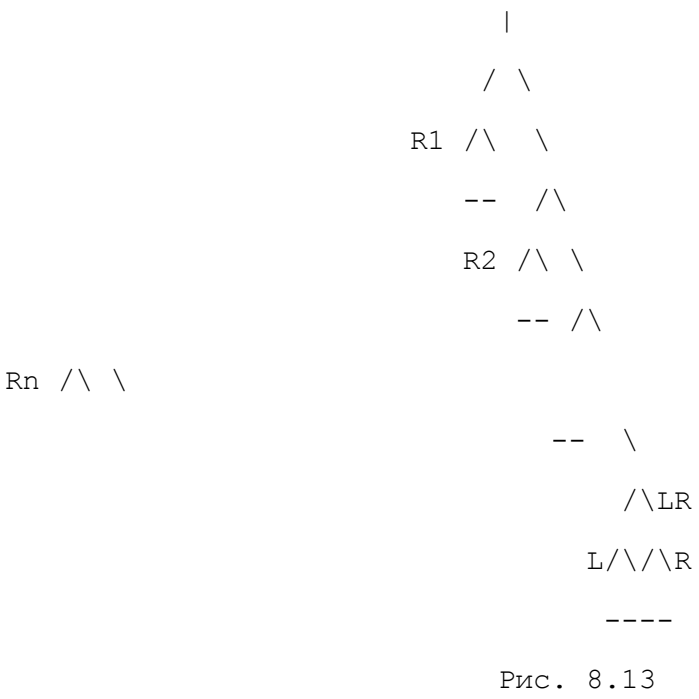


Рис. 8.13

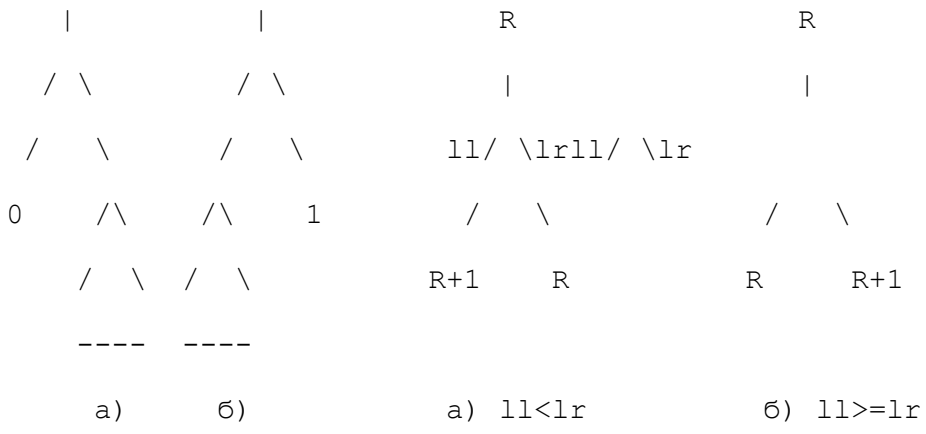
Предположим сначала, что распределение регистров осуществляется по простейшей схеме слева-направо, как изображено на рис. 8.13. Тогда к моменту генерации кода для поддерева LR занято n регистров. Пусть поддерево L требует nl регистров, а поддерево R - nr регистров. Если nl=nr, то при вычислении L будет использовано nl регистров и под результат будет занят n+1-й регистр. Еще nr (=nl) регистров будет использовано при вычислении R. Таким образом, общее число использованных регистров будет равно n+nl+1. Если nl>nr, то при вычислении L будет использовано nl регистров. При вычислении R будет использовано nr<nl регистров, и всего будет использовано не более чем n+nl регистров.

Если nl<nr, то после вычисления L под результат будет занят один регистр (предположим n+1-й) и nr регистров будет использовано для вычисления R. Всего будет использовано n+nr+1 регистров. Видно, что для деревьев, совпадающих с точностью до порядка потомков каждой вершины, минимальное число регистров при распределении их слева-направо достигается на дереве, у которого в каждой вершине слева расположено более "сложное" поддерево, требующее большего числа

регистров. Таким образом, если дерево таково, что в каждой внутренней вершине правое поддерево требует меньшего числа регистров, чем левое, то, обходя дерево слева направо, можно оптимально распределить регистры. Без перестройки дерева это означает, что если в некоторой вершине дерева справа расположено более сложное поддерево, то сначала сгенерируем код для него, а затем уже для левого поддерева. Алгоритм работает следующим образом. Сначала осуществляется разметка синтаксического дерева по следующим правилам.

Правила разметки:

1) если вершина - правый лист или дерево состоит из единственной вершины, помечаем эту вершину числом 1, если вершина - левый лист, помечаем ее 0 (рис. 8.14).



а) б) а) ll<lr б) ll>=lr

Рис. 8.14 Рис. 8.15

2) если вершина имеет прямых потомков с метками l1 и l2, то в качестве метки этой вершины выбираем большее из чисел l1 или l2 либо число l1+1, если l1=l2. Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления. Затем осуществляется распределение регистров для результатов операций.

Правила распределения регистров:

1) Корню назначается первый регистр.

2) Если метка левого потомка меньше метки правого, то левому потомку назначается регистр на единицу больший, чем предку, а правому - с тем же номером (сначала вычисляется правое поддерево и его результат помещается в регистр R). Если же метка

левого потомка больше или равна метке правого потомка, то наоборот, сначала вычисляется левое поддерево и его результат помещается в регистр R (рис. 8.15). После этого формируется код по следующим правилам.

Правила генерации кода:

1) если вершина - правый лист с меткой 1, то ей соответствует код LOAD X,R, где R - регистр, назначенный этой вершине, а X - адрес переменной, связанной с вершиной (рис. 8.16.б);

2) если вершина внутренняя и ее левый потомок - лист с меткой 0, то ей соответствует код

Код правого поддерева

Op X,R

где снова R - регистр, назначенный этой вершине, X - адрес переменной, связанной с вершиной, а Op - операция, примененная в вершине (рис. 8.16.а);

3) если непосредственные потомки вершины не листья и метка правой вершины больше метки левой, то вершине соответствует код

Код правого поддерева

Код левого поддерева

Op R+1,R

где R - регистр, назначенный внутренней вершине, и операция Op, вообще говоря, не коммутативная (рис. 8.17 б)).

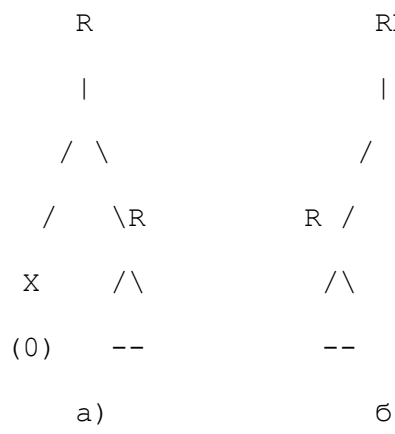


Рис. 8.16

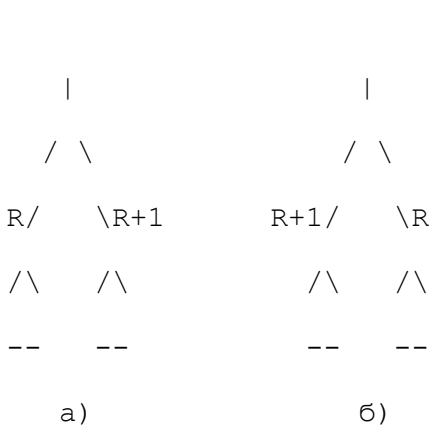


Рис. 8.17

Если метка правой вершины меньше или равна метке левой вершины, то вершине соответствует код

Код левого поддерева

Код правого поддерева

Op R,R+1

MOVE R+1,R

Последняя команда генерируется для того, чтобы получить результат в нужном регистре (в случае коммутативной операции операнды операции можно поменять местами и избежать дополнительной пересылки)(рис. 8.17 а)).

10. Основные принципы объектно-ориентированного программирования.

Объектно-Ориентированное Программирование - это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследования.

В этом определении можно выделить три части

1. ООП использует в качестве элементов конструкции объекты, а не алгоритмы [как структурное программирование]

2. Каждый объект является реализацией какого-либо определенного типа (класса)

3. Классы организованы иерархически

При несоблюдении хотя бы 1 из указанных требований программа перестает быть ОО. (В частности при нарушении 3 имеем программирование на основе Абстрактных Типов Данных)

Язык программирования называется ОО тогда и только тогда, когда выполнены следующие условия:

1. Имеется поддержка объектов в виде абстракции данных имеющих интерфейсную часть в виде поименованных операций и защищенную область локальных данных

2. Объекты относятся к соответствующим типам (классам)

3. Классы могут наследовать атрибуты и методы от суперклассов (базовых классов)

4. Имеется поддержка полиморфных функций

ОО подходу соответствуют 4 главных элемента:

Абстрагирование

Инкапсуляция (Ограничение доступа)

Иерархия (в частности наследование)

Полиморфизм

1. Абстракция - это такие существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов, и, таким образом четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от деталей их осуществления.

ОО стиль программирования связан с воздействием на объекты (напримерпередача ему сообщения). Путем воздействия на объект вызывается определенная реакция этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия составляют характер поведения объекта.

2. Ограничение доступа

Ограничение доступа - это процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта как целого.Ограничение доступа позволяет вносить в программу изменения, сохраняя ее надежность и позволяя минимизировать затраты на этот процесс.

Абстрагирование и ограничение доступа дополняют друг друга: абстрагирование фокусирует внимание на внешних особенностях объекта, а ограничение доступа не позволяет объектам-пользователям различать внутреннее устройство объекта.

В описании класса можно выделить две части: интерфейс и реализацию. Интерфейс отражает внешнее проявление объекта, создавая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает механизмы достижения желаемого поведения объекта. В интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношение к процессу взаимодействия объектов.

Изменение реализации, вообще говоря, не влечет за собой изменение интерфейса.

3. Иерархия

Иерархия - это ранжированная или упорядоченная иерархия абстракций.

Основными видами иерархических структур применительно к сложным системам являются структура классов(иерархия по номенклатуре) и структура объектов (иерархия по составу)

Примеры иерархий:

Наследование означает такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких других классов (соответственно простое или множественное наследование). Иными словами, наследование - такая иерархия абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. (Иерархия обобщение-специализция)

Агрегирование (отношение по составу). Объект состоит из подобъектов.

Принципы абстрагирования, ограничения доступа и иерархии конкурируют между собой. Абстрагирование данных состоит в установлении жестких границ, защищающих состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных классов. В связи с этим интерфейсная часть класса может быть разделена на три части:

Обособленную (private) - видимая только для самого класса

Защищенную (protected) - видимую также и для подклассов

Общедоступную (public) - видимую для всех

Наиболее полно контроль видимости реализован в C++.

4. Полиморфизм - способность операции (функции) с одним и тем же именем выполнять различные действия в зависимости от типа своих операндов.

Пример: В обычных языках : операции + и -.

В ОО языках - виртуальные функции

Полиморфизм бывает статический и динамический.

Статический - перекрытие операций (Ада, C++,ObjectPascal).

Динамический - механизм виртуальных функций

Виртуальные функции являются примером полиморфных функций. Виртуальная функция может быть переопределена в производном классе, следовательно ее реализация зависит от всей последовательности методических описаний и наследственной иерархии. Какая именно из виртуальных функций будет вызванозависит от динамического типа объекта и определяется в момент обращения к виртуальной функции. Для этого используется таблица виртуальных функций, определенная для каждого класса.

Дополнительные возможности ОО языков:

Некоторые языки позволяют определять несколько специальных методов класса:

Конструктор - специальная процедура класса для создания и/или инициализации начального состояния объекта. В частности, конструктор может инициализировать таблицу виртуальных функций.

Деструктор - специальная процедура класа, которая делает состояние объекта неопределенным и (или) ликвидирует сам объект.

Некоторые преимущества ОО подхода:

1. Использование объектного подхода существенно повышает качество разработки в целом и ее фрагментов. ОО Системы часто получаются более компактными чем их не-ОО аналоги

2. Использование объектного подхода приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к ее полной переработке в случае существенных изменений исходных требований

3. Ориентирован на человеческое восприятие мира

11.Основные этапы компиляции (лексический анализ, синтаксический анализ, семантический анализ, генерация кода и т.д.)

Лексический анализ

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, то есть выделить эти слова из непрерывнойпоследовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы (например, символ пробела в Фортране). ВСи разделительное значение символов-разделителей может блокироваться ("\" в конце строки внутри"...").

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатиричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

Для осуществления двух дальнейших фаз анализа лексический анализатор выдает информацию двух типов: для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контекстного анализатора, работающего вслед за синтаксическим, существенна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.).

Таким образом, общая схема работы лексического анализатора такова. Сначала выделяется отдельнаялексема (при этом, возможно, используются символы-разделители). Ключевые слова распознаются явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов.

Если выделенная лексема является ограничителем, то этот ограничитель (точнее, некоторый его признак) выдается как результат лексического анализа. Если выделенная лексема является ключевым словом, то выдается признак соответствующего ключевого слова. Если выделенная лексема является идентификатором - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Наконец, если выделенная лексема принадлежит какому-либо из других классов лексем (например, лексема

представляет собой число, строку и т.д.), то выдается признак соответствующего класса, а значение лексемы сохраняется отдельно.

Лексический анализатор может быть как самостоятельной фазой трансляции, так и подпрограммой, работающей по принципу "дай лексему". В первом случае (рис. 3.1, а) выходом анализатора является файл лексем, во втором - (рис. 3.1., б) лексема выдается при каждом обращении к анализатору (при этом, какправило, признак класса лексемы возвращается как результат функции "лексический анализатор", а значение лексемы передается через глобальную переменную). С точки зрения обработки значений лексем, анализатор может либо просто выдавать значение каждой лексемы, при этом построение таблиц объектов (идентификаторов, строк, чисел и т.д.) переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов. В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

Работа лексического анализатора задается некоторым конечным автоматом. Однако, непосредственное описание конечного автомата неудобно с практической точки зрения. Поэтому для задания лексического анализатора, как правило, используется либо регулярное выражение, либо праволинейная грамматика.

Все три формализма (конечных автоматов, регулярных выражений иправолинейных грамматик) имеют одинаковую выразительную мощность. В частности, по регулярному выражению или праволинейнойграмматике можно сконструировать конечный автомат, распознающий тот же язык.

Синтаксический анализ

Синтаксический анализ — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево).

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

Нисходящийпарсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.

Метод рекурсивного спуска

LL-анализатор

Восходящийпарсер (англ. bottom-upparser) — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.

LR-анализатор

GLR-парсер

Семантический анализ

Задачей контекстного анализа является установление свойств объектов и их использования. Наиболее часто решаемой задачей является определение существования объекта и соответствия его использования контексту, что осуществляется с помощью анализа типа объекта. Под контекстом здесь понимается вся совокупность свойств текущей точки программы, например множество доступных объектов, тип выражения и т.д.

Таким образом, необходимо хранить объекты и их типы, уметь находить эти объекты и определять их типы, определять характеристики контекста. Совокупность доступных в данной точке объектов будем называть средой.

В процессе работы компилятор хранит информацию об объектах программы в специальных таблицах символов. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и описания объекта. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрее, а требуемая память по возможности меньше. Кроме того, со стороны языка программирования могут быть дополнительные требования к организации информации. Имена могут иметь определенную область видимости. Например, поле записи должно бытьуникально в пределах структуры (или уровня структуры), но может совпадать с именем объекта вне записи (или другого уровня записи). В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть конфликт имен (или неоднозначность в трактовке имени). Если язык имеет блочную структуру, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых - эффективно освобождать память при выходе из блока. В некоторых языках (например, Аде) одновременно (в одном блоке) могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима.

Некоторые основные способы организации таблиц символов в компиляторе: таблицы идентификаторов, таблицы расстановки, двоичные деревья и реализация блочной структуры.

Оптимизация

Среди принципов оптимизации, применяемых в различных методах оптимизации в компиляторах (некоторые из них могут противоречить друг другу или быть неприменимыми при разных целях оптимизации):

- уменьшение избыточности: повторное использование результатов вычислений, сокращение числа перевычислений;
- компактификация кода: удаление ненужных вычислений и промежуточных значений;
- сокращение числа переходов в коде: например, использование встраивания функций (англ. *Inlineexpansion*) или размотки цикла позволяет во многих случаях ускорить выполнение программы ценой увеличения размера кода;
- локальность: код и данные, доступ к которым необходим в ближайшее время, должны быть размещены рядом друг с другом в памяти, чтобы следовать принципу локальности ссылок (англ. *Locality of reference*);
- использование иерархии памяти: размещать наиболее часто используемые данные в регистрах общего назначения, менее используемые — в кэш, ещё менее используемые — в оперативную память, наименее используемые — размещать на диске.
- распараллеливание: изменение порядка операций может позволить выполнить несколько вычислений параллельно, что ускоряет исполнение программы.

Генерация кода

В дополнение к основной задаче — преобразованию кода из промежуточного представления в машинные инструкции — генератор кода обычно пытается оптимизировать создаваемый код теми или иными способами. Например, он может использовать более быстрые инструкции, использовать меньше инструкций, использовать имеющиеся регистры и предотвращать избыточные вычисления.

Некоторые задачи, которые обычно решают сложные генераторы кода:

- Выбор инструкций: какие инструкции использовать
- Планирование инструкций: в каком порядке размещать эти инструкции. Планирование — это оптимизация, которая может значительно влиять на скорость выполнения программы на конвейерных процессорах
- Размещение в регистрах: размещение переменных программы в регистрах процессора.

Выбор инструкций обычно выполняется рекурсивным обходом абстрактного синтаксического дерева, в этом случае сравниваются части конфигураций дерева с шаблонами; например, дерево `W:=ADD(X,MUL(Y,Z))` может быть преобразовано в линейную последовательность инструкций рекурсивной генерации последовательностей `t1:=X` и `t2:=MUL(Y,Z)`, а затем инструкцией `ADD W,t1,t2`.

В компиляторах, использующих промежуточный язык, может быть две стадии выбора инструкций — одна для конвертирования дерева разбора в промежуточный код, а вторая (следует намного позже) — для преобразования промежуточного кода в инструкции целевой системы команд. Вторая стадия не требует обхода дерева: она может выполняться последовательно и обычно состоит из простой замены операций промежуточного языка соответствующими им кодами операций. На самом деле, если компилятор фактически является транслятором (например, один переводит Eiffel в C), то вторая стадия генерации кода может включать построение дерева из линейного промежуточного кода.

12. Построение детерминированного конечного автомата по регулярному выражению.

Пусть T - конечный алфавит. ***Регулярное множество*** в алфавите T определяется рекурсивно следующим образом (знаком '<-' будем обозначать принадлежность множеству, знаком '<=' включение):

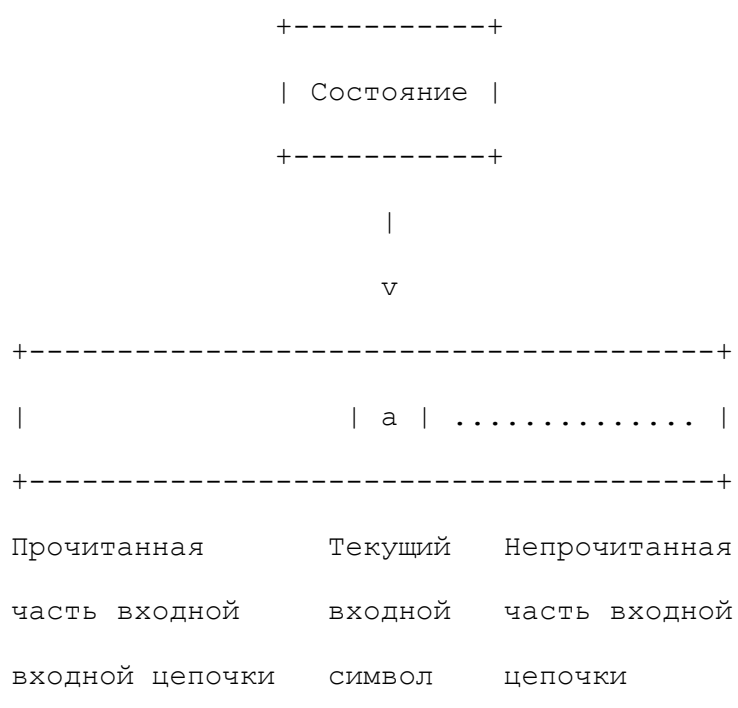
1. {} (пустое множество) - регулярное множество в алфавите T;
2. {a} - регулярное множество в алфавите T для каждого a<- T;
3. {e} - регулярное множество в алфавите T (e - пустая цепочка);
4. если P и Q - регулярные множества в алфавите T, то таковы же и множества
    - P U Q (объединение),
    - PQ (конкатенация, т.е. множество pq, p<-P, q<-Q),
    - P* (итерация: P*={e} U P U PP U...;
5. ничто другое не является регулярным множеством в алфавите T.

Итак, множество в алфавите T регулярно тогда и только тогда, когда оно либо {}, либо {e}, либо {a} для некоторого a<-T, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации. Приведенное выше определение регулярного множества одновременно определяет и форму его записи, которую будем называть регулярным выражением. Для сокращенного обозначения выражения PP* будем пользоваться записью P+ и там, где это необходимо, будем использовать скобки. В этой записи наивысшим приоритетом обладает операция *, затем конкатенация и, наконец, операция U, для записи которой иногда будем использовать значок '|'.

Детерминированный конечный автомат (ДКА) - это пятерка $M=(Q,T,D,q_0,F)$, где

1. Q - конечное множество состояний;
2. T - конечное множество допустимых входных символов;
3. D - функция переходов, отображающая множества QxTв множество Q и определяющая поведение управляющего устройства;
4. q_0<-Q - начальное состояние управляющего устройства;
5. F<=Q - множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и сдвига входной головки на одну ячейку вправо.



Текущее состояние управляющего устройства, символ под головкой и цепочка символов вправо от головки называются конфигурацией автомата. Конфигурация (q0,w) называется начальной, а пара (q,e), где q<-F, называется заключительной (или допускающей).

Такт автомата M представляется бинарным отношением |-, определенным на конфигурациях: отношение имеет место, если есть переход из конфигурации (q1,w1) в конфигурацию (q2,w2). Отношения |-+ и |-* - это, соответственно, транзитивное и рефлексивно-транзитивное замыкание отношения |-. Говорят, что автомат M допускает цепочку w, если (q0,w)|-*(q,e) для некоторого q<-F. Языком, допускаемым (распознаваемым, определяемым) автоматом M, (обозначается L(M)), называется множество входных цепочек, допускаемых автоматом M. Т.е.

L(M)={w | w<-T* и (q0,w)|-*(q,e) для некоторого q<-F}

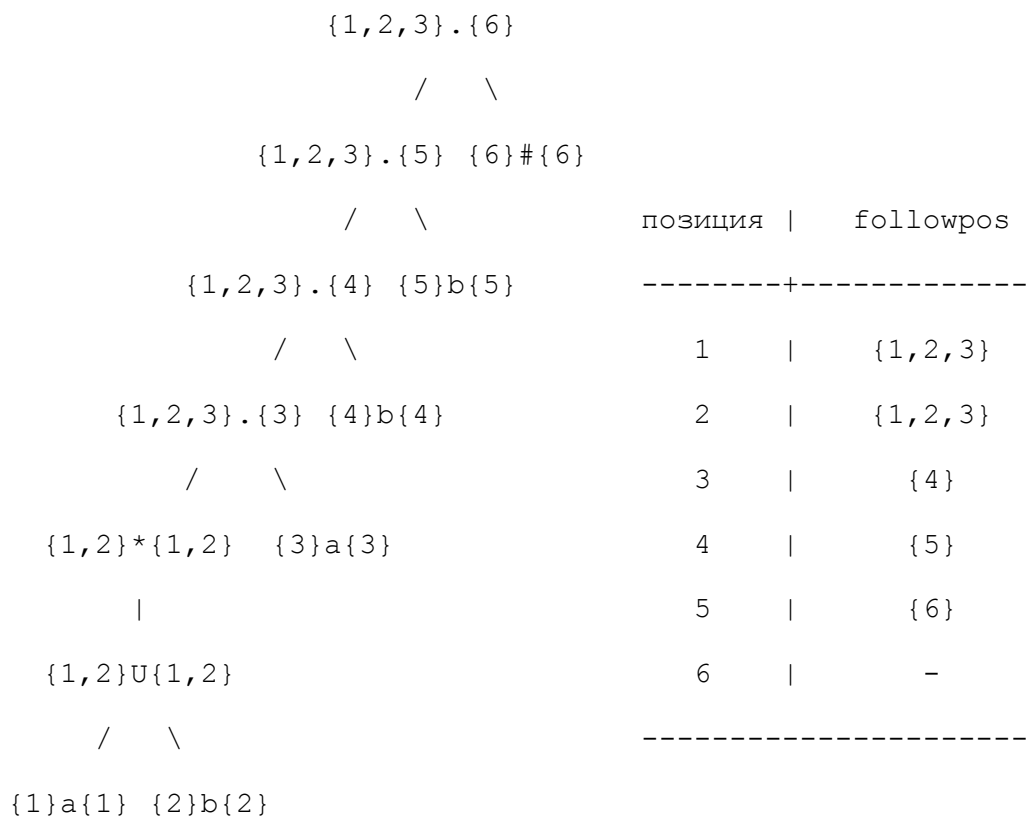
Построение детерминированного конечного автомата по регулярному выражению.

К регулярному выражению (сокращенно РВ) r добавим маркер конца: (r)#. После построения ДКА для расширенного РВ легко построить ДКА для исходного РВ: все состояния ДКА из которых есть переход в конечное с чтением символа "#", можно считать конечными, а символ "#" и соответствующие переходы удалить.

Представим РВ в виде дерева, листья которого - терминальные символы, а внутренние вершины - операции "." (конкатенации), "U" (объединение), "*" (итерация). Каждому листу дерева (кроме е-листьев) припишем уникальный номер и ссылаться на него будем, с одной стороны, как на позицию в дереве и, с другой стороны, как на позицию символа, соответствующего листу.

Теперь, обходя дерево T сверху-вниз слева-направо, вычислим четыре функции: nullable, firstpos, lastpos и followpos. Функции nullable, firstpos и lastpos определены на узлах дерева, а followpos - на множестве позиций. Значением всех функций, кроме nullable, является множество позиций. Функция followpos вычисляется через три остальные функции. Функция firstpos(n) для каждого узла n синтаксического дерева регулярного выражения дает множество позиций, которые соответствуют первым символам в подцепочках, генерируемых подвыражением с вершиной в n. Аналогично, lastpos(n) дает множество позиций, которым соответствуют последние символы в подцепочках, генерируемых подвыражениями с вершиной n. Для узлов n, поддеревья которых (т.е. дерево, у которого узел n является корнем) могут породить пустое слово, определим nullable(n)=true, а для остальных узлов false.

узел n	nullable(n)	firstpos(n)	lastpos(n)
листе	true	0	0
листі	false	{i}	{i}
U / \ a b	nullable(a) or nullable(b)	firstpos(a) U firstpos(b)	lastpos(a) U lastpos(b)
. / \ a b	nullable(a) and nullable(b)	if nullable(a) then firstpos(a) U firstpos(b) else firstpos(a)	if nullable(b) then lastpos(a) U lastpos(b) else lastpos(b)
* | a	true	firstpos(a)	lastpos(a)



Пример 2.3. Функции firstpos и lastpos для выражения (a+b)abb#

Слева от каждой вершины значение firstpos, справа - lastpos.

Заметим, что эти функции могут быть вычислены за один обход дерева. Если i - позиция, то followpos(i) есть множество позиций j таких, что существует некоторая строка ...cd..., входящая в язык, описываемый РВ, такая, что i - соответствует этому вхождению c, а j - вхождению d. Функция followpos может быть вычислена также за один обход дерева по следующим двум правилам

1. Пусть n - внутренний узел с операцией "." (конкатенация), a,b - его потомки. Тогда для каждой позиции i, входящей в lastpos(a), добавляем к множеству значений followpos(i) множество firstpos(b).
2. Пусть n - внутренний узел с операцией "*" (итерация), a - его потомок. Тогда для каждой позиции i, входящей в lastpos(a), добавляем к множеству значений followpos(i) множество firstpos(a).

Прямое построение ДКА по регулярному выражению.

Будем строить множество состояний автомата Dstates и помечать их. Состояния ДКА соответствуют множествам позиций. Начальным состоянием будет состояние firstpos(root), где root - вершина синтаксического дерева регулярного выражения, конечными - все состояния, содержащие позиции, связанные с символом "#". Сначала в Dstates имеется только одно непомеченное состояние firstpos(root).

while есть непомеченное состояние T в Dstatesdo

пометить T;

for каждого входного символа a<-T do

пусть символу a в T соответствуют позиции

p1,...,pi, ипусть S=U followpos(pi)

i

Если S не пусто и S не принадлежит Dstates, то

добавить непомеченное состояние S в Dstates

(рис. 2.8)

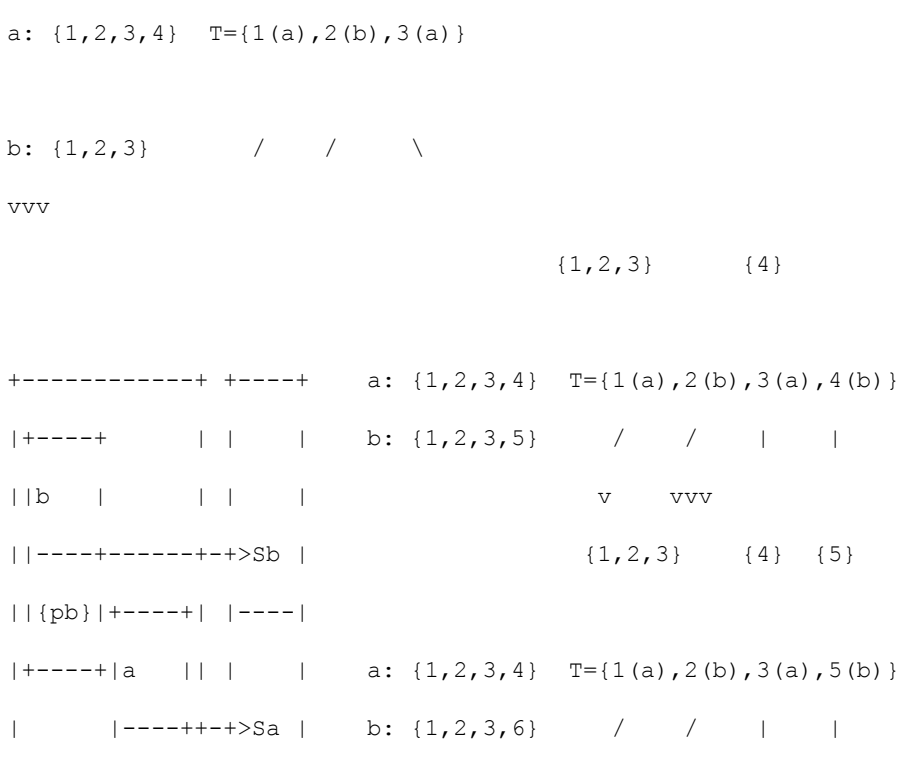
Функцию перехода Dtran для T и a определить как

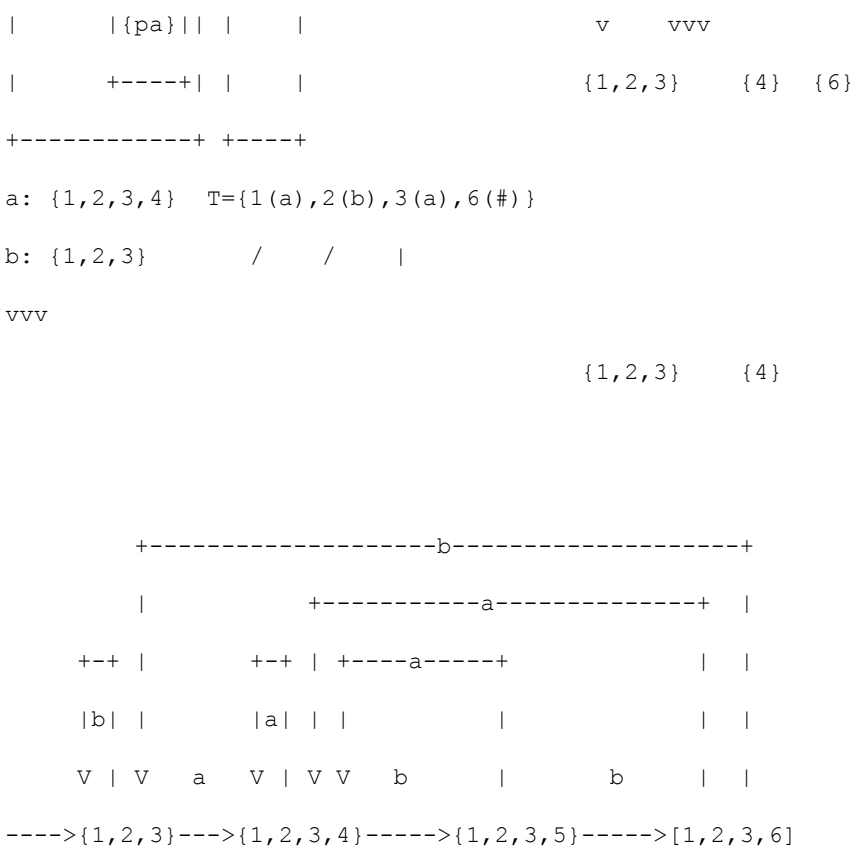
Dtran(T,a)=S.

end;

end;

Для примера 2.3 вначале T={1(a),2(b),3(a)}. Последовательность шагов алгоритма приведена на рис. 2.9. В результате будет построен детерминированный конечный автомат, изображенный на рис. 2.10. Состояния автомата обозначаются как множества позиций, например {1,2,3}, конечное состояние заключено в квадратные скобки [1,2,3,6].




|{pa}||
+-----+|
+------------+ +-----+
a: {1,2,3,4}
b: {1,2,3}
vvv
v vvv
{1,2,3} {4} {6}
T={1(a),2(b),3(a),6(#)}
/ / |
vvv
{1,2,3} {4}

b
a
b
a
b
V V a V V V b b
----->{1,2,3}--->{1,2,3,4}----->{1,2,3,5}----->[1,2,3,6]

13. Построение канонической системы множеств LR(1) ситуаций и таблиц действий и переходов для LR(1) грамматик.

Основные понятия и определения

Пусть G=<N,T,P,S> - контекстно-свободная грамматика, где N - множество нетерминальных символов, T - множество терминальных символов, P - множество правил вывода и S - аксиома. Будем говорить, что uxvвыводится за один шаг из uAv (и записывать это как uAv=>uxv), если A->x - правило вывода и u и v - произвольные строки из (N U T)*. Если u1=>u2=>...=>un, будем говорить, что из u1 выводится un, и записывать это как u1=>*un. Т.е.:

1) u=>*u для любой строки u,

2) если u=>*v и v=>*w, то u=>*w.

Аналогично, "=>+" означает выводится за один или более шагов.

Если дана грамматика G с начальным символом S, отношение =>+ можно использовать для определения L(G) - языка, порожденного G. Строки L(G) могут содержать только терминальные символы G. Строка терминалов w принадлежит L(G) тогда и только тогда, когда S=>+w. Строка w называется предложением в G.

Если S=>*u, где u может содержать нетерминалы, то u называется сентенциальной формой в G. Предложение - это сентенциальная форма, не содержащая нетерминалов.

Рассмотрим выводы, в которых в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала. Такой вывод называется левосторонним. Если S=>*u в процессе левостороннего вывода, то u - левая сентенциальная форма. Аналогично определяется правосторонний вывод.

Упорядоченным графом называется пара (V,E), где V обозначает множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид ((v,e1),(v,e2),...,(v,en)). Этот элемент указывает, что из вершины a выходят n дуг, причем первой из них считается дуга, входящая в вершину e1, второй - дуга, входящая в вершину e2, и т.д.

Дерево вывода в грамматике G=<N,T,P,S> - это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества N U T U {e}. Если внутренняя вершина помечена символом A, а ее прямые потомки - символами X1,..., Xn, то A->X1X2...Xn - правило этой грамматики.

Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике G(S)=(N,T,P,S), если выполнены следующие условия:

(1) корень дерева D помечен S;

(2) каждый лист помечен либо a<-T, либо e;

(3) каждая внутренняя вершина помечена нетерминалом;

(4) если N - нетерминал, которым помечена внутренняя вершина и X1,...,Xn - метки ее прямых потомков в указанном порядке, то N->X1...Xk - правило из множества P.

Автомат с магазинной памятью (сокращенно МП-автомат) - это семеркаP=(Q,T,Г,D,q0,Z0,F), где

(1) Q - конечное множество символов состояний, представляющих всевозможные состояния управляющего устройства;

(2) T - конечный входной алфавит;

(3) Г - конечный алфавит магазинных символов;

(4) D - функция переходов - отображение множества Qx(T U {e})xГвмножество конечных подмножествQxГ*, т.е. d:Qx(T U {e})xГ ->{QxГ*};

(5) q0<-Q - начальное состояние управляющего устройства;

(6) Z0<-Г - символ, находящийся в магазине в начальный момент (начальный символ);

(7) F<=Q - множество заключительных состояний

Конфигурацией МП-автомата называется тройка (q,w,u)<-QxT*xГ*, где

(1) q - текущее состояние управляющего устройства;

(2) w - неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если w e, то считается, что вся входная лента прочитана;

(3) u - содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если u=e, то магазин считается пустым.

Такт работы МП-автомата P будем представлять в виде бинарного отношения |-, определенного на конфигурациях. Будем писать(q,aw,Zu)|-(q',w,vu)если множество d(q,a,Z) содержит (q',v), где q, q'<-Q, a<-TU {e}, w<-T*, Z<-Г, u,v<-Г*.

Начальной конфигурацией МП-автомата P называется конфигурация вида (q0,w,Z0), где w<-T*, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, а в магазине есть только начальный символ Z0. Заключительная конфигурация - это конфигурация вида (q,e,u), где q<-F, u<-Г*.

Говорят, что цепочка w допускается МП-автоматом P, если (q0,w,Z0)|-*(q,e,u) для некоторых q<-F и u<-Г*. Языком, определяемым (или допускаемым) автоматом P (обозначается L(P)), называют множество цепочек, допускаемых автоматом P.Иногда

допустимость определяют несколько иначе: цепочка w допускается МП-автоматом P, если (q0,w,Z0)|-*(q,e,e). Эти определения эквивалентны.

Он состоит из входа, выхода, магазина, управляющей программы и таблицыанализа, которая имеет две части - действий и переходов. Управляющая программа одна и та же для всех анализаторов, разные анализаторы различаются таблицами анализа. Программа анализатора читает символы из входного буфера по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида S0X1S1X2S2...XmSm (Sm - верхушка магазина). Каждый Xi - символ грамматики (терминальный или нетерминальный), а Si - символ, называемый состоянием. Каждый символ состояния выражает информацию, содержащуюся в магазине ниже него, а комбинация символа состояния на верхушке магазина и текущего входного символа используется для индексации таблицы анализа и определяет решение о сдвиге или свертке.

Таблица анализа состоит из двух частей: действия (action) и переходов (goto). Начальное состояние этого ДКА - это состояние, помещенное на верхушку магазина LR-анализатора в начале работы.

Конфигурация-LR анализатора - это пара, первая компонента которой - содержимое магазина, а вторая - непросмотренный вход:

(S0 X1 S1 X2 S2 ... XmSm, ai ai+1 ... an $)

Эта конфигурация соответствует правой сентенциальной форме

X1 X2 ... Xmai ai+1 ... an

Префиксы правых сентенциальных форм, которые могут появиться в магазине анализатора, называются активными префиксами. Основа сентенциальной формы всегда располагается на верхушке магазина.

Таким образом, активный префикс - это такой префикс правой сентенциальной формы, который не переходит правую границу основы этой формы.

Очередной шаг анализатора определяется текущим входным символом aiи символом состояния на верхушке магазина Sm. Элемент таблицы действий action[Sm,ai] для состояния Sm и входа ai, может иметь одно ихчетырех значений:

1) shift S, сдвиг, где S - состояние,

2) reduce A->w, свертка по правилу грамматики A -> w,

3) accept, допуск,

4) error, ошибка.

Конфигурации, получающиеся после каждого из четырех типов шагов, следующие

1. Если action[Sm,ai]=shift S, то анализатор выполняет шаг сдвига, переходя в конфигурацию

(S0 X1 S1 X2 S2 ... XmSmaiS, ai+1 ... an $)

В магазин помещаются как входной символai, так и следующее состояние S, определяемоеaction[Sm,ai]. Текущим входным символом становится ai+1.

2. Если action[Sm,ai]=reduce A->w, то анализатор выполняет свертку, переходя в конфигурацию

(S0 X1 S1 X2 S2 ... Xm-r Sm-r A S, ai ai+1 ... an $)

где S=goto[Sm-r,A] и r - длина w, правой части правила вывода. Функция goto таблицы анализа, построенная по грамматике G, - это функция переходов детерминированного конечного автомата, распознающего активные префиксы G. Анализатор сначала удаляет из магазина 2r символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние Sm-r. Затем анализатор помещает в магазин как A - левую часть правила вывода, так и S - содержимое таблицы goto[Sm-r,A]. На шаге свертки текущий входной символ не меняется. Для LR-анализаторов Xm-r+1 ... Xm - последовательность символов грамматики, удаляемых из магазина, всегда соответствует w - правой части правила вывода, по которому делается свертка.

После осуществления шага свертки генерируется выход LR-анализатора, т.е. исполняются семантические действия, связанные с правилом, по которому делается свертка, например печатаются номера правил, по которым делается свертка.

3. Если action[Sm,ai]=accept, то разбор завершен.

4. Если action[Sm,ai]=error, анализатор обнаружил ошибку, то выполняются действия по диагностике и восстановлению.Ниже приведен алгоритм LR-анализа. Все LR-анализаторы ведут себя одинаково. Разница между ними заключается в различном содержании таблиц действий и переходов.

Рассмотрим теперь конструирование таблиц LR-анализатора.LR(1) ситуацией называется пара [A->u.v,a], где A->uv - правило грамматики, а a - терминал или правый концевой маркер $. "1" указывает на длину второй компоненты ситуации, которая называется аванцепочкой ситуации. Аванцепочка не играет роли в ситуациях вида [A->u.v,a], где v не равно e, но ситуация вида [A->u.,a] ведет к свертке по правилу A->u только если следующим входным символом является a. Таким образом свертка по правилу A->u требуется только для тех входных символов a, для которых [A->u.,a] является LR(1) ситуацией в состоянии на верхушке магазина.

Будем говорить, что LR(1)-ситуация [A->u.v,a] допустима для активного префикса z, если существует вывод S=>*yAw=>yuvw, где z=yu и либо a - первый символ w, либо w равно e и a равно $.

Будем говорить, что ситуация допустима, если она допустима для какого-либо активного префикса.

Алгоритм 3.8. Конструирование множеств LR(1)-ситуаций.

Алгоритм заключается в выполнении главной программы items, которая вызывает функции closure и goto.

SetOfItemsclosure(SetOfItems I)/*I - множествоситуаций*/

{do

{for (каждой ситуации [A->u.Bv,a] из I, каждого правила вывода B->w из G',

каждого терминала b из FIRST(va),)

добавить [B->.w,b] к I;

}

while (к I можно добавить новые ситуации);

return I;

}

В анализаторах типа LR(0) при построении closure не учитываются терминалы из FIRST(va). Если I - множество ситуаций, допустимых для некоторого активного префикса z, то goto(I,X) - множество ситуаций, допустимых для активного префикса zX.

SetOfItemsgoto(SetOfItemsI,Symbol X)/*I - множествоситуаций;

X - символграмматики*/

{ Пусть [A->u.Xv,a] принадлежит I;

Рассмотрим J - множество всех ситуаций вида {[A-uX.v,a]};

returnclosure(J)

}

Работа алгоритма построения множества LR(1)-ситуаций начинается с того, что берется C - множество ситуаций {closure({[S'->.S,$]})}. Затем из имеющегося множества с помощью

операции goto() строятся новые множества ситуаций. По-существу, goto(I,X) - переход конечного автомата из состояния I по символу X.

```
void items(Grammar G')
{ do
        {C={closure({[S'->.S,$]})};
        for (каждого множества ситуаций I из C,
                каждого символа грамматики X такого, что goto(I,X) не пусто и не
                принадлежит C) {
                добавить goto(I,X) к C;
        }
        while (к C можно добавить множество ситуаций);
}
```

Построение канонических таблиц LR анализатора.

Шаг1. Строим набор множеств LR(1)- ситуаций C={I0,I1,...,In} для G'.

Шаг 2. Состояние i анализатора строится из Ii. Действия анализатора для состояния i определяются следующим образом:

а) если [A->u.av,b] принадлежит Ii и goto(Ii,a)=Ij, то полагаем

б) если [A->u.,a] принадлежит Ii, A#S', тополагаем action[i,a]="reduce

в) если [S'->S.,$] принадлежит Ii, полагаем action[i,$]="accept".

Шаг 3. Переходы для состояния i определяются следующим образом: если goto(Ii,A)=Ij, то goto[i,A]=j (здесь A - нетерминал).action[i,a]="shiftj". Здесь a - терминал;A->u";

Шаг 4. Все входы, не определенные шагами 2 и 3, полагаем равными

Шаг 5. Начальное состояние анализатора строится из множества, содержащего ситуацию [S'->.S,$].

Если при применении этих правил возникает конфликт, т.е. в одном и том же множестве может быть более одного варианта действий (либо сдвиг/свертка, либо свертка/свертка), говорят, что грамматика не является LR(1), и алгоритм завершается неуспешно.

14. Сложность алгоритма как функция одного или нескольких числовых аргументов. Сложность в худшем случае.

Пусть по входным данным (входу) x алгоритм A вычисляет результат (выход) y. Такое вычисление связано с затратами времени и памяти. Допустим, что достигнуто соглашение о том, как измеряются эти затраты, тогда можно рассмотреть функции затрат$C_A^T(x)$, $C_A^S(x)$,где верхний индекс T указывает на временные затраты, S — на пространственные затраты (затраты памяти и пространственные затраты — это синонимы), соответствующие вычислениям, связанным с применением A к входу x;

Можно ввести некоторую неотрицательную числовую характеристику ||x|| возможных входов алгоритма и оценивать функции затрат с помощью функций, зависящих не от x, а от ||x||:

$$C_A^T(x) \le \varphi(||x||), C_A^S(x) \le \omega(||x||)$$

Избираемую нами величину ||·|| принято называть размером входа. В соответствии с нашим замыслом размер входа должен характеризовать «громоздкость» исходных данных; то, что x является числом, позволяет говорить о росте φ,ω и исследовать этот рост.

Определение 1.1. Пусть на возможных входах x алгоритма A определена неотрицательная числовая функция ||x|| (размер входа). Пусть также определены целочисленные неотрицательные функции$C_A^T(x)$, $C_A^S(x)$временных и пространственных затрат алгоритма A. Тогда временной и пространственной сложностями A называются функции числового аргумента:

$$T_A(n) = \max_{||x||=n} C_A^T(x), S_A(n) = \max_{||x||=n} C_A^S(x)$$

Более полно каждая такая сложность именуется сложностью в худшем случае.

Определение 1.2. Пусть функция$C_A^T(x)$ в определении 1.1 отражает затраты на выполнение лишь какого-то одного типа операций в предположении, что все эти операции требуют одинакового времени выполнения, не зависящего от вида и размера операндов, и это время равно 1. Тогда соответствующая функция$T_A(n)$— это сложность по числу операций рассматриваемого типа. Привлечение в качестве$C_A^S(x)$функции, отражающей только количество хранимых дополнительных величин того или иного фиксированного типа, приводит к пространственной сложности $S_A(n)$ по числу величин рассматриваемого типа.

Такую сложность называют алгебраической сложностью по числу операций или числу величин рассматриваемого типа.

Достаточно очевидно, что если алгоритм A состоит в последовательном (друг за другом) выполнении алгоритмов U и V, то мы, вообще говоря, не можем утверждать, что $T_A(n)$= $T_U(n)$+ $T_V(n)$, так как, например, размер входа алгоритма U может существенно отличаться, как в большую, так и в меньшую сторону, от размера его выхода.

Определение 2.1. Функции f (n) и g(n) имеют одинаковый порядок (пишут f (n)=θ(g(n))) тогда и только тогда, когда найдутся положительные c_1, c_2, N такие, что неравенства

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$$

Выполнены для всех n>N.

Определение 2.2. Соотношение f (n)=Ω (g(n)) имеет место тогда и только тогда, когда найдутся положительныес, N такие, что для всех n>N выполнено | f (n)|≥c|g(n)|.

Предложение 2.1. Соотношение f (n)=Θ(g(n)) имеет место тогда и только тогда, когда одновременно f (n) = O(g(n)) и f (n) =Ω (g(n));помимо этого, f (n) = Ω(g(n)) тогда и только тогда, когда g(n) =O(f (n)).

Определение 2.3. Если имеет место оценка f (n)= O(g(n)), то она называется точной, коль скоро существует неограниченно возрастающая последовательность неотрицательных целых чисел $\{n_k\}$ такая, что для φ(k)= f (n_k), ψ(k)= g(n_k) имеет место φ(k)=Ω (ψ(k)).

Часто, хотя и не всегда, для алгоритмов целочисленной арифметики, входом которых является целое неотрицательное число n, размером входа выбирают не само n, а его битовую длину, или, иными словами, количество λ(n) цифр в его двоичной записи.

Наряду с его уже рассмотренной сложностью $T_A(n)$ введем еще одну сложность $T_A^*(m)$, m=λ(n).

Лемма 4.1. Пусть f (x)—неубывающая функция вещественной переменной. Тогда если $T_A^*(m) \leq f(m)$, то $T_A(n) \leq f(\log_2 n + 1)$
Доказательство. Следует из неубыванияf.

Лемма 4.2. Пусть g(x)—неубывающая функция веществennойпе-
ременной. Тогда
(i) если$T_A(n) \leq g(n)$,то $T_A^*(m) \leq g(2^m)$,
(ii) если$T_A(n) \geq g(n)$то $T_A^*(m) \geq g(2^{m-1})$.

Теорема 4.1. Пусть f (x)—неубывающая функция веществennойпеременной. Тогдаесли $T_A^*(m) = O(f(m))$, то$T_A(n) = O(f(\log_2 n + 1))$;
как следствие, при f (x +1)= O(f (x)) имеем $T_A(n) = O(f(\log_2 n))$.

Теорема 4.2. Пусть g(x)—неубывающая функция веществennойпеременной. Тогда
(i) если$T_A(n) = O(g(n))$, то $T_A^*(m) = O(g(2^m))$,
(ii) если$T_A(n) = \Omega(g(n))$, то $T_A^*(m) = \Omega(g(2^{m-1}))$; как следствие,
При $g\left(\frac{x}{2}\right) = \Omega\big(g(x)\big)$, $T_A^*(m) = \Omega(g(2^m))$;

15. Сложность в среднем. Сложность рандомизированного алгоритма.

При каждом фиксированном значении sразмера входа сами соответствующие входы образуют конечное множество X_s = {x : ||x|| = s}. Будем предполагать, что каждому $x \in X_s$ приписана некоторая вероятность $P_s(x)$:

$$\sum_{x \in X_s} P_s(x) = 1$$

По умолчанию считается, что вероятность распределена равномерно:
$P_s(x) = \frac{1}{N_s}$, где $N_s = |X_s|$

Определение 5.1. Пусть при любом допустимом значении s множество X_sвсех входов размера s является вероятностным пространством, в силу чего временные и пространственные затраты алгоритма A для входов x (т. е. $C_A^T(x)$ и $C_A^S(x)$) размера s являются случайными величинами наX_s. Сложностью в среднем называется математическое ожидание соответствующей случайной величины:
$\sum_{x \in X_s} C_A^T(x) P_s(x)$или $\sum_{x \in X_s} C_A^S(x) P_s(x)$.

Для временной и пространственной сложности в среднем алгоритма A мы будем использовать обозначения $\overline{T_A}(\cdot)$и $\overline{S_A}(\cdot)$.

Теорема 5.1. Для любого алгоритма A при любом распределении вероятностей на множестве X_s, где s—некоторое допустимое значение размера $|\cdot|$, сложность в среднем не превосходит сложности в худшем случае: $\overline{T_A}(s) \le T_A(s), \overline{S_A}(s) \le S_A(s)$
Доказательство.
По определению $\overline{T_A}(s)$, $C_A^T(x)$можно ограничить $T_A(s)$. Сумма вероятностей равна 1.

Теорема 6.1. Сложность в среднем некоторого алгоритма по суммарному числу операций нескольких различных типов равна сумме сложностей в среднем этого алгоритма по числу операций каждого типа в отдельности.

Определение 8.1. Алгоритмы с элементами случайности, реализуемыми обращениями к генераторам случайных чисел, называются рандомизированными.

Анализ сложности рандомизированного алгоритма сводится к нахождению математических ожиданий некоторых случайных величин. Но ситуация здесь отличается от той, когда множество входов фиксированного размера рассматривается как вероятностное пространство и затраты алгоритма, однозначно определенные для каждого конкретного входа, становятся случайными величинами на этом пространстве (мы шли этим путем в двух предыдущих параграфах). При исследовании рандомизированных алгоритмов вероятностное пространство, на котором рассматриваются случайные величины, состоит из сценариев выполнения алгоритма для фиксированного входа, и каждый сценарий определяется тем, какие случайные числа будут сгенерированы в соответствующие моменты выполнения алгоритма; за каждым таким сценарием закрепляется некоторая вероятность.

В нашем контексте генератор случайных чисел можно представлять себе как стандартную функцию random(N) целого положительного аргумента N, результатом выполнения которой является элемент множества {0, 1, ..., N − 1}, но невозможно предсказать

точно,каким именно будет значение этой функции,—любой из элементов указанного множества может появиться с вероятностью 1/N. Таким образом, затраты рандомизированного алгоритма при фиксированном входе, вообще говоря, не определяются однозначно, но зависят от сценария вычисления. При фиксированном входе мы можем рассмотреть множество всех сценариев и, приписав адекватным образом каждому из сценариев некоторую вероятность, ввести на полученном вероятностном пространстве случайную величину, значение которой для данного сценария равно соответствующим вычислительным затратам. Значение функции затрат на данном входе можно положить равным математическому ожиданию этой случайной величины (усредненным затратам для данного входа).

16. Основные принципы построения сети Интернет. Иерархическая модель компьютерной сети. Адресация в сети Интернет, протоколы ARP, DHCP. Модели основных протоколов IP, TCP, ICMP. Модель взаимодействия приложений в Интернет.

Модель OSI имеет уровневую организацию. Она включает в себя **семь уровней**: физический, канальный, сетевой, транспортный, сессии, представления и прикладной.

Модель TCP/IP включает в себя **3 основных уровня**:

- межсетевой уровень
- транспортный уровень
- уровень приложений.

В модели TCP/IP нет уровней сессии и представления, поскольку необходимость в них была неочевидна для ее создателей.

Internet – метасеть, состоящая из многих сетей, которые работают согласно протоколам семейства TCP/IP, объединены через шлюзы и используют единое адресное пространство и пространство имен.

Среду передачи данных в Internet нельзя рассматривать только как паутину проводов или оптоволоконных линий. Оцифрованные данные пересылаются через **маршрутизаторы**, которые соединяют сети и с помощью сложных алгоритмов выбирают наилучшие маршруты для информационных потоков.

Глобальная сеть включает подсеть связи, к которой подключаются локальные сети, отдельные компоненты и терминалы (средства ввода и отображения информации).

Компьютеры, за которыми работают пользователи-клиенты, называются **рабочими станциями**, а компьютеры, являющиеся источниками ресурсов сети, предоставляемых пользователям, называются **серверами**. Такая структура сети получила название **узловой**.

Инфраструктура Интернет:

- Магистральный уровень (система связанных высокоскоростных телекоммуникационных серверов).
- Уровень сетей и точек доступа (крупные телекоммуникационные сети), подключенных к магистрали.
- Уровень региональных и других сетей.
- ISP – интернет-провайдеры.
- Пользователи.

Адресация.

Есть две таблицы. Первая показывает как достичь интересующей сети. Вторая – как достичь узел внутри сети. Когда поступает IP пакет, маршрутизатор ищет его адрес доставки в таблице маршрутизации. Если это адрес другой сети, то пакет передают дальше тому маршрутизатору, который отвечает за связь с этой сетью. Если это адрес в локальной сети, то маршрутизатор направляет пакет прямо по месту назначения. Если адреса нет в таблице, то маршрутизатор направляет пакет специально выделенному по умолчанию маршрутизатору, который должен разобраться с этим случаем с помощью более подробной таблицы. Такая организация алгоритма позволяет существенно сократить размер таблиц в маршрутизаторах. С появлением подсети структура адресов меняется. Теперь записи в таблице имеют форму (эта_сеть, подсеть, 0) и (эта_сеть, эта_подсеть, машина). Таким образом, маршрутизатор подсети в данной локальной сети знает, как достичь любую подсеть в данной локальной сети, и как найти конкретную машину в своей подсети. Все что ему нужно – это знать маску подсети. С помощью логической операции И маршрутизатор выделяет адрес подсети с помощью маски. По своим таблицам он определяет как достичь нужной подсети или, если этого локальная подсеть данного маршрутизатора, как достичь конкретной машины.

Address Resolution Protocol – протокол определения адреса.

Ethernet адрес. Этот адрес имеет 48 разрядов. Сетевая карта знает только такие адреса и ничего об 32-разрядных IP. Как отобразить 32-разрядный IP адрес на адреса канального уровня, например, Ethernet адрес. Для отображения IP адреса на Ethernet адрес, в подсеть посылается запрос у кого такой IP адрес. Машина с указанным адресом шлет ответ. Протокол, который реализует рассылку запросов и сбор ответов – ARP протокол. Практически каждая машина в Internet имеет этот протокол. Для того чтобы узнать адрес в другой сети есть два решения – есть определенный маршрутизатор, который принимает все сообщения, адресованные определенной сети или группе адресов – proxy ARP. Этот маршрутизатор знает как найти адресуемую машину. Другое решение – выделенный маршрутизатор, который управляет маршрутизацией удаленного трафика. Машина определяет, что обращение идет в удаленную сеть и шлет сообщение на этот маршрутизатор.

DHCP (англ. Dynamic Host Configuration Protocol – протокол динамической настройки узла) – сетевой протокол, позволяющий компьютерам автоматически получать IP-адрес и другие параметры, необходимые для работы в сети TCP/IP. Данный протокол работает по модели «клиент-сервер». Для автоматической конфигурации компьютер-клиент на этапе конфигурации сетевого устройства обращается к так называемому серверу DHCP, и получает от него нужные параметры. Сетевой администратор может задать диапазон адресов, распределяемых сервером среди

компьютеров. Это позволяет избежать ручной настройки компьютеров сети и уменьшает количество ошибок. Протокол DHCP используется в большинстве сетей TCP/IP.
Протокол DHCP предоставляет три способа распределения IP-адресов:

- **Ручное распределение.** При этом способе сетевой администратор сопоставляет аппаратному адресу (для Ethernet сетей это MAC-адрес) каждого клиентского компьютера определённый IP-адрес. Фактически, данный способ распределения адресов отличается от ручной настройки каждого компьютера лишь тем, что сведения об адресах хранятся централизованно (на сервере DHCP), и потому их проще изменять при необходимости.
- **Автоматическое распределение.** При данном способе каждому компьютеру на постоянное использование выделяется произвольный свободный IP-адрес из определённого администратором диапазона.
- **Динамическое распределение.** Этот способ аналогичен автоматическому распределению, за исключением того, что адрес выдаётся компьютеру не на постоянное пользование, а на определённый срок. Это называется арендой адреса. По истечении срока аренды IP-адрес вновь считается свободным, и клиент обязан запросить новый (он, впрочем, может оказаться тем же самым). Кроме того, клиент сам может отказаться от полученного адреса.
  Некоторые реализации службы DHCP способны автоматически обновлять записи DNS, соответствующие клиентским компьютерам, при выделении им новых адресов. Это производится при помощи протокола обновления DNS, описанного в RFC 2136. Протокол DHCP является клиент-серверным, то есть в его работе участвуют клиент DHCP и сервер DHCP. Передача данных производится при помощи протокола UDP, при этом сервер принимает сообщения от клиентов на порт 67 и отправляет сообщения клиентам на порт 68.

Модель сервиса IP

Internet Protocol («межсетевой протокол») – маршрутизируемый протокол сетевого уровня стека TCP/IP. Именно IP стал тем протоколом, который объединил отдельные компьютерные сети во всемирную сеть Интернет. Неотъемлемой частью протокола является адресация сети.
IP объединяет сегменты сети в единую сеть, обеспечивая доставку пакетов данных между любыми узлами сети через произвольное число промежуточных узлов (маршрутизаторов). Он классифицируется как протокол третьего уровня по сетевой модели OSI. IP не гарантирует надёжной доставки пакета до адресата – в частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (приходят две копии одного пакета), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прийти вовсе. Гарантию безошибочной доставки пакетов дают некоторые протоколы более высокого уровня – транспортного уровня сетевой модели OSI, – например, TCP, которые используют IP в качестве транспорта.
В современной сети Интернет используется IP четвёртой версии, также известный как IPv4. В протоколе IP этой версии каждому узлу сети ставится в соответствие IP-адрес длиной 4 октета (4 байта). При этом компьютеры в подсетях объединяются общими начальными битами адреса. Количество этих бит, общее для данной подсети, называется маской подсети (ранее использовалось деление пространства адресов по классам – A, B, C; класс сети определялся диапазоном значений старшего октета и определял число адресуемых узлов в данной сети, сейчас используется бесклассовая адресация).

- предотвращает «зацикливание» пакетов;
- фрагментирует пакеты если они слишком длинные;
- использует контрольную сумму, чтобы сократить возможность доставки в неправильное место назначение;
- две версии: IPv4 с 32 битным адресом IPv6 с 128 битным адресом
- позволяет добавлять новые опции к заголовку;
- работает над любой физической средой;
- над IP работают различные транспортные протоколы;
- поверх транспорта работают прикладные протоколы;
- IP используется всегда для передачи пакетов между различными сетями;
- очень простой сервис.

Модель сервиса TCP

Доступ к TCP-сервису происходит через сокет. Сокет состоит из IP-адреса хоста и 16-разрядного локального номера на хосте, называемого порт. Сокеты создаются как отправителем, так и получателем. Порт — это TSAP для TCP. Каждое соединение идентифицируется парой сокетов, между которыми оно установлено. Один и тот же сокет может быть использован для разных соединений. Никаких дополнительных виртуальных соединений не создается.
Порты до 256 номера зарезервированы для стандартных сервисов, которые постоянно активны и готовы к работе. Например, для обеспечения FTP-передачи файла соединение должно выполняться через 21-й порт, где находится FTP-демон, а для TELNET – через 23-й порт. Полный список таких портов можно найти в RFC 1700.

Все TCP-соединепня — дуплексные, т.е. передача по ним происходит независимо в оба направления. TCP-соединение поддерживает только соединение точка—точка. Не существует TCP-соединений типа «от одного ко многим».
TCP-агент поддерживает поток байтов, а не поток сообщений. Напомним, это означает, что гранины сообщений не поддерживаются автоматически в потоке. Например, если по TCP-соединению передается текст, разбитый на страницы, то TCP-агент не будет каким-либо образом обозначать конец каждой страницы.
После передачи приложением данных TCP-агенту, эти данные могут быть отправлены сразу на сетевой уровень, а могут быть буферизованы. Как поступить в этом случае решает TCP-агент. Однако в ряде случаев бывает необходимо, чтобы данные были отправлены сразу, например если эти данные представляют собой команду для удаленной машины. Для этого в заголовке TCP-сегмента имеется флаг PUSH, и если он установлен, то это говорит TCP-агенту о том, что данные должны быть переданы немедленно.
Наконец, если в заголовке TPDU-сегмента установлен флаг URGENT, то TCP-агент передает такой сегмент незамедлительно. Когда срочные данные поступают к месту назначения, то их передают получателю немедленно.

Модель сервиса ICMP

Управление функционированием Интернета на сетевом уровне происходит через маршрутизаторы с помощью протокола ICMP (Internet Control Message Protocol), описанного в RFC 792. Этот протокол обеспечивает доставку сообщений любой машине, имеющей IP-адрес, от маршрутизаторов и других хостов в сети. С помощью этого протокола реализуют обратную связь для решения проблем, возникающих при передаче. Он также выявляет и рассылает сообщения о десятках событий. Для доставки своих сообщений протокол ICMP использует пакеты IP-протокола. Приведем наиболее важные сообщения.
Сообщение **destination unreachable** охватывает множество случаев, например, случай, когда маршрутизатор не знает, как достигнуть необходимой подсети или хоста, или случай, когда дейтаграмма при доставке должна быть фрагментирована, но установлен флаг, который запрещает это делать.
Сообщение **time exceeded** посылает маршрутизатор, если он обнаружил дейтаграмму с истекшим времени жизни. Это сообщение также генерирует хост, если он не успел завершить обработку IP-пакета до истечения времени его жизни. (Далее слова «пакет», «IP-пакет», «дейтаграмма» будем понимать как синонимы, если специально не оговорено что-либо другое.)
Синтаксические или семантические ошибки в заголовке IP-пакета вызывают появление сообщения **parameter problem**.
Сообщение **source quench** обеспечивает управление потоком. Маршрутизатор или хост-получатель высылает этот пакет хосту-отправителю, если последнему необходимо понизить скорость передачи. Другими словами, это пример подавляющего пакета. Сообщения такого типа будут генерироваться до тех пор, пока скорость поступления пакетов от отправителя не достигнет значения, необходимого хосту- получателю. Это сообщение система может использовать для предотвращения перегрузки, поскольку оно возникает всякий раз, когда маршрутизатор вынужден сбросить пакет из-за переполнения своего буфера.
Сообщение **redirect** позволяет маршрутизатору отправить рекомендацию о лучшем маршруте и впредь посылать пакеты с определенным IP-адресом через другой маршрутизатор.
Сообщения **echo request** и **echo reply** позволяют проверить работоспособность хостов в сети: получатель сообщения echo request обязан ответить сообщением echo reply, причем с теми же параметрами, что и в echo request.
Сообщения **time-stamp request** и **time-stamp reply** позволяют измерять временную задержку в Интернете на сетевом уровне. Этот механизм необходим, например, для работы алгоритма маршрутизации по состоянию канала.

Свойство	Поведение
Сообщение о состоянии	Самодостаточное сообщение об ошибке или исключительном состоянии
Ненадежный	Сервис без соединения и подтверждения

ICMP предоставляет информацию о сетевом уровне хостам и маршрутизаторам. ICMP работает над IP и относится к транспортному уровню. ping и traceroute реализованы с помощью ICMP.

17. Физический уровень стека сетевых протоколов. Технологии Ethernet и WiFi. Алгоритмы работы, коллизии, управление множественным доступом к каналу.

Назначение физического уровня – передавать данные в виде потока битов от одной машины к другой. При этом для передачи данных можно использовать различные физические среды, каждую из которых характеризует следующие параметры:

- Ширина полосы пропускания.
- Пропускная способность.
- Задержка сигнала.
- Стоимость.
- Сложность прокладки.
- Сложность прокладки.
- Сложность обслуживания.
- Достоверность передачи.
- Затухание.
- Помехоустойчивость.

Кабели в стандарте IEEE 802.3.

Первым появился так называемый толстый Ethernet(10Base5). Коаксиальный кабель жёлтого цвета с отметками через каждые 2.5 м, указывающими, где можно производить подключение. Подключение выполняется через специальные розетки, которые монтируются прямо на кабеле. В эти розетки встраивался специальный прибор – трансивер, отвечающий за обнаружение несущей частоты и коллизий. Когда трансивер обнаруживает коллизию, он посылает специальный сигнал по кабелю, гарантирующий, что другие трансиверы услышат эту коллизию. Кабель обеспечивает пропускную способность 10 Мбит/с, а максимальная длина равна 500м(10Base5).

Вторым появился 10Base2. Это более простой в употреблении коаксиальный кабель с простым подключением через BNC-коннектор, представляющий собой Т-образное соединение коаксиальных кабелей. Его сегмент не должен превышать 200м и объединять более 30 машин.

Решение проблемы поиска обрыва, частичного повреждения кабеля или плохого контакта в коннекторе привели к созданию совершенно иной кабельной конфигурации на основе витой пары. В этом случае каждая машина соединяется витой парой со специальным устройством – хабом. Такой способ обозначается 10Base-T.

Трансиверы в 10Base5 размещаются прямо на кабеле и соединяются с компьютером трансиверным кабелем, длина которого не может превышать 50м. Трансиверный кабель состоит из пяти витых пар. Две из них используются для передчи данных к компьютеру и от него, две служат для передачи управляющей информации в обе стороны, а пятая – для подачи питания на трансивер.

Трансиверный кабель подключается к контроллеру в компьютере через интерфейс AUI. В контроллере имеется специальная микросхема, отвечающая за приём кадром и их отправку, проверку и формирование контрольной суммы. В некоторых случаях эта микросхема отвечает и за управление буферами на канальном уровне, очередью буферов на отправку и обеспечивает прямой доступ к памяти машины, а также решает другие вопросы доступа к сети.

В 10Base2 трансивер располагается на контроллере, и каждая машина должна иметь свой индивидуальный трансивер.

В 10Base-T трансивера нет вовсе. Машины соединяются хабом с витой парой, длина которой не должна превышать 100м. Вся электроника сосредоточена в нём.

Последний используемый в стандарте IEEE 802.3 тип кабеля – оптоволокно 10Base-F, которое относительно дорогое, но обеспечение низкого уровня шума и большая длина одного сегмента являются его преимуществом. Для увеличения длины сегментов в этом стандарте используются репитеры – устройства физического уровня, отвечающие за очистку, усиление и передачу сигнала. Репитеры не могут отстоять друг от друга более чем на 2.5 км, и на одном сегменте их не может быть более четырёх.

Физический уровень. Первой и ключевой технологией стандарта 802.11 является технология расширения спектра передачи методом **прямой последовательности** (Direct Sequence Spread Spectrum – DSSS). Использование DSSS позволяет беспроводным интерфейсам передавать данные со скоростью от 1 до 2 Мбит/с.

Идея метода. Пусть имеется канал с широкой полосой пропускания. Разобьем его на полосы. Каждому значению бита сопоставим определенный кот с длиной, равной числу полос, на которые разбили канал. Теперь будем передавать каждый бит, параллельно передавая его код, причем каждый элемент кода(чип) в своей полосе. Такой способ передачи позволяет эффективнее использовать полосу пропускания канала, и он более надежен по сравнению с традиционным способом передачи.

Канальный уровень. Минимально сеть wifi может содержать всего два устройства. В этом случае организуется выделенная есть, в которую входят все беспроводные интерфейсы этих устройств. Данное соединение можно сравнить с соединением типа точка-точка в проводной связи.

Для решения задачи совместимости в сети устанавливается выделенный узел – точка доступа, представляющая собой устройство, имеющее проводной интерефейс для подключения к проводной

сети, а также антенну, образующую вокруг себя зону покрытия точки доступа. Радиус покрытия точки доступа от 90 до 150м.
Когда абонентское устройство включается внутри сети, оно начинает прослушивать эфир в поисках совместимого устройства, с которым оно могло бы взаимодействовать. Этот этап называется **сканированием**. При активном сканировании генерируется широковещательный запрос от абонентского устройства, обязательно включающий в себя идентификатор сети (SSID), к которой он хочет присоединиться. Когда запрос достигает точки доступа, имеющий запрашиваемый идентификатор, эта сеть генерирует ответ на запрос. При **пассивном сканировании** абонентское устройство слушает эфир и ожидает появления кадров-маяков, которые периодически рассылаются точками доступа. Когда абонентское устройство получает кадр-маяк, в котором указан SSID, оно пытается присоединиться к указанной сети. Пассивное сканирование – постоянный процесс, при котором устройства могут присоединиться к точке доступа или отсоединяться по мере изменения мощности радиосигнала.
Структура кадра. В wifi определены 3 типа кадров: контрольные, управляющие и кадры данных. Структрура совпадает с IEEE802.3, за исключением некоторых отличий: размер поля данных равен 1500 байт, максимальный размер кадра составляет 2346 байт.

Базовая модель динамического предоставления доступа к каналу.

Пять основных предположений, составляющих основу моделей сетей ЭВМ, в которых в качестве СПД используется канал с множественным доступом:

1) Станции. Модель состоит из N независимых станций (компьютеров, телефонов, факс-машин и т. п.). На каждой станции работает пользователь или программа, генерирующие кадры для передачи. Предполагаем, что если кадр сгенерирован, то станция блокируется, и новый кадр не появится, пока не будет передан первый кадр. Это означает, что станции независимы, и на каждой из них работает только одна программа или один пользователь, генерирующие нагрузку с постоянной скоростью.
2) Единственность канала. Канал один и он доступен всем станциям. Все станции равноправны. Они получают кадры и передают кадры только через этот единственный канал. Аппаратные средства всех станций для доступа к каналу одинаковые, но программно можно устанавливать станциям приоритеты.
3) Коллизии. Если во время передачи кадра одной станцией другая станция начала передачу своего кадра, то такой случай будем называть коллизией. Предполагаем, что любая станция может обнаружить коллизию и что кадры, разрушенные при коллизии, должны быть посланы повторно позднее. Кроме коллизий других ошибок передачи нет.
4) Время. Возможны две модели времени – непрерывная и дискретная:
    a. непрерывное время. Передача кадра может начаться в любой момент. В сети нет единых часов, которые разбивают время на слоты. Другими словами, время является непрерывной функцией, отображающей интересующие нас действия в сети на множество вещественных чисел;
    b. дискретное время. Все время работы канала разбивается на одинаковые интервалы, называемые слотами, В слоте может оказаться нуль кадров, если это слот ожидания, один кадр, если в этом слоте передача кадра прошла успешно, и несколько кадров, если в этом слоте произошла коллизия.
5) Доступ к каналу. Возможны два способа доступа станции к каналу:
    a. с обнаружением несущей. Станция прежде чем использовать канал всегда определяет, занят он или нет с помощью несущей – сигнала определенной формы. Когда канал не занят, по нему все время передается такой сигнал, а если канал занят, то сигнал в нем отличается от несущей, и станция не начинает передачу;
    b. при отсутствии несущей. Станция ничего не знает о состоянии канала, пока не начнёт использовать его. Она сразу начинает передачу и лишь в ходе передачи обнаруживает коллизию, т.к. сигнал, который она «увидит» в канале, будет отличаться от того сигнала, который станция передала в канал.

Говоря о **динамическом доступе**, подразумевают, что отсутствует какая-либо фиксированная политика предоставления доступа к каналу для передачи в отличие от статических методов доступа. При этом любая станция может запросить доступ к каналу в любой момент времени, а методы, доступа лишь определяют правила удовлетворения этих запросов.

Методы множественного доступа ALOHA. Система состояла из наземных радиостанций, работающих на одной частоте и связывающих острова между собой. Идея ее конструкции заключалась в том, чтобы позволить в вещательной среде любому количеству пользователей неконтролируемо использовать один и тот же канал.

Чистая ALOHA: любой пользователь, желающий передать сообщение, сразу пытается это сделать. Благодаря тому, что в вещательной среде у него всегда есть обратная связь, т.е. он может определить, пытался ли кто-то еще передавать сообщение на его частоте, отправитель может установить возникновение конфликта при передаче. Обратная связь в среде ЛВС происходит практически мгновенно. Отправитель при этом должен слушать среду передачи до тех пор, пока последний бит его сообщения не достигнет самого отдаленного получателя. Обнаружив конфликт, отправитель ожидает некоторый случайный отрезок времени, после чего повторяет попытку

передачи. Интервал времени на ожидание должен быть случайным, иначе конкуренты, повторяя попытки передачи вызовут коллизию снова. Системы, в которых пользователи конкурируют за получение доступа к общему каналу, называются **системами с состязаниями.**
Неважно, когда произошел конфликт, оба кадра считаются испорченными и должны быть переданы повторно. Контрольная сумма, защищающая данные в кадре, не позволяет различать разные случаи наложения кадров.
Назовем **временем кадра** время, необходимое на передачу кадра стандартной фиксированной длины. Обозначим это время t. Предположим, что число пользователей неограниченно и все они порождают кадры по закону Пуассона со средним числом N кадров за t. Это означает, что вероятность события, при котором будет порождено n кадров за время t, можно записать в виде:

$$P[k] = \frac{G^k e^{-G}}{k!}$$

$$P[n] = \frac{\lambda^n e^{-\lambda}}{n!}, \lambda = N$$

Поскольку при N > 1 очередь на передачу будет только расти и все кадры будут страдать от коллизий, предположим, что 0 < N < 1. Также предположим, что вероятность за время кадра сделать к попыток передачи, как новых, так и ранее не переданных из-за коллизий кадров, распределяется по закону Пуассона со средним значением G. Понятно, что при этом должно выполняться соотношение □ ≥ □, иначе очередь будет расти бесконечно. При слабой загрузке (□ ~ 0) будет мало передач, а, следовательно, и коллизий, поэтому G N. При высокой загрузке должно выполняться соотношение G > N. При этом пропускная способность канала (S) будет равна числу кадров, которые надо передать, умноженному на вероятность успешной передачи. Если обозначить P0 вероятность отсутствия коллизий при передаче кадра, то можно записать □ = □□$_0$. Рассмотрим внимательно, сколько времени требуется отправителю, чтобы обнаружить коллизию. Пусть он начал передачу в момент времени □$_0$ и пусть требуется время t, чтобы кадр достиг самой отдаленной станции. Тогда, если в тот момент, когда кадр почти достиг этой отдаленной станции, она начнет передачу (ведь в системе ALOHA. станция сначала передает, а потом слушает), отправитель узнает об этом только через время, равное □$_0$ + 2□. Вероятность появления к кадров при передаче кадра с распределением Пуассона поэтому вероятность, что появится 0 кадров, равна $-$□. За двойное время кадра среднее число кадров равно 2G, откуда □$_0$ = □$^{-2}$□, а так как □ = □□$_0$ то пропускная способность канала □ = □□$^{-2}$□.
Максимальная пропускная способность достигается при G = 0,5 и при $S = \frac{1}{2e}$, что составляет примерно 18% номинальной пропускной способности системы.
Слотированная ALOHA.
Модификация чистой ALOHA, в которой все время работы канала разделяется на слоты. Размер слота при этом должен быть равен максимальному времени кадра. Ясно, что такая организация работы канала требует синхронизации. Кто-то, например одна из станций, испускает сигнал начала очередного слота. Поскольку передачу теперь можно начинать не в любой момент, а только по специальному сигналу, то время на обнаружение коллизии сокращается вдвое. Откуда □ = □□$^{-□}$. Максимум пропускной способности слотированной ALOHA наступает при G = 1, где S = $S = \frac{1}{e}$, т.е.составляет около 37% , что вдвое больше, чем у чистой ALOHA. Рассмотрим как G влияет на пропускную способность системы. Для этого подсчитаем вероятность успешной передачи кадра за k попыток. Так как □$^{-□}$ – это вероятность отсутствия коллизии при передаче, то вероятность того, что кадр будет передан ровно за k попыток, можно записать в виде $P_k = e^{-G}(1 - e^{-G})^{k-1}$
Среднее ожидаемое число повторных передач: $E = \sum_{k=1}^{\infty} kP_k = \sum_{k=1}^{\infty} ke^{-G}(1 - e^{-G})^{k-1}$
Эта экспоненциальная зависимость показывает, что с ростом G резко возрастает число повторных попыток, поэтому незначительное увеличение загрузки канала ведет к резкому падению его пропускной способности.
Протоколы множественного доступа с обнаружением несущей.
Протоколы, реализующие идею начала передачи только после определения, занят канал или нет, называются **протоколами с обнаружением несущей**– CSMA (Carrier Sensitive Multiple Access).
Настойчивые и ненастойчивые CSMA-протоколы.
Если канал занят, то станция ждет, а как только он освободился, пытается сразу начать передачу. Если при этом произошла коллизия, станция ожидает случайный промежуток времени и все начинает сначала. Этот протокол называется **настойчивым CSMA-протоколом первого уровня** или **1-настойчивым CSMA-протоколом**, поскольку станция, следуя этому протоколу, начинает передачу с вероятностью 1, как только обнаруживает, что канал свободен. Здесь важную роль играет задержка распространения сигнала в канале. Всегда существует вероятность того, что, как только одна станция начала передачу, другая станция также стала готова передавать. Если вторая станция проверит состояние канала прежде чем до нее дойдет сигнал от первой станции о том, что она заняла канал, то вторая станция сочтет канал свободным и начнет передачу. В результате

возникает коллизия. Чем больше время задержки сигнала, тем больше вероятность такого случая и тем хуже производительность канала.
Однако даже если время задержки сигнала будет равно нулю коллизии все равно могут возникать. Например, если во время передачи готовыми к передаче оказались две станции. В этом случае они подождут, пока ранее начатая передача будет закончена, а затем будут состязаться между собой. Тем не менее этот протокол более эффективен, чем любая из систем ALOHA, так как станция учитывает состояние канала, прежде чем начать действовать.
Другим вариантом CSMA-протокола является **ненастойчивый CSMA-протокол**. Основное отличие его от предыдущего состоит в том, что готовая к передаче станция опрашивает канал. Если канал свободен, то она начинает передачу. Если же канал занят, то она не будет настойчиво его опрашивать в ожидании, когда он освободится, а будет делать это через случайные отрезки времени. Это несколько увеличивает задержку при передаче сигнала для станции, но общая эффективность протокола возрастает. И, наконец, настойчивый CSMA-протокол уровня p. Который применяется квотированным каналам. Когда станция готова к передаче, она опрашивает канал. Если канал он свободен, то она с вероятностью p передает свой кадр и с вероятностью q = 1 – p ждет следующего слота. Так станция действует, пока не передаст кадр. Если во время передачи происходит коллизия, станция ожидает случайный промежуток времени и опрашивает канал снова. Если при опросе он опять оказывается занятым, станция ждет начала следующего слота, и весь алгоритм повторяется.
CSMA-протокол с обнаружением коллизий.
Протоколы этого класса широко используются в локальных сетях. Модель их работы следующая. В момент времени τ_0 одна станция заканчивает передачу очередного кадра, а все другие станции, у которых имеется кадр для передачи, начинают передачу. Естественно, в этом случае происходят коллизии, которые быстро обнаруживаются посредством сравнения отправленного сигнала с тем сигналом, который есть на линии. Обнаружив коллизию, станция сразу прекращает передачу на случайный промежуток времени, после чего все начинается сначала. Таким образом, в работе протокола CSMA/CD можно выделить три стадии: состязания, передачи и ожидания (когда нет кадров для передачи).
Рассмотрим подробнее алгоритм состязаний. Определим, сколько времени станции, начавшей передачу, требуется, чтобы обнаружить коллизию. Обозначим t время распространения сигнала до самой удаленной станции на линии. Для коаксиального кабеля длиной в 1 км t = 5 мкс. В этом случае минимальное время для определения коллизии будет равно 2t. Следовательно, станция не может быть уверена, что она захватила канал до тех пор, пока не убедится, что в течение 2t секунд не было коллизий. Поэтому мы будем рассматривать период состязаний как слотированную систему ALOHA со слотом 2t секунд на один бит. Захватив канал, станция может далее передавать кадр с любой скоростью.
Обнаружение коллизий– это аналоговый процесс, поэтому, чтобы обнаруживать их, необходимо использовать специальные кодировки на физическом уровне.

18. Коммутация пакетов, устройство пакетов. Как устроен и работает пакетный коммутатор (switch). Виды задержек в компьютерной сети и способы управления ими (приоритеты, веса и гарантированная скорость потока). Управление потоком при пакетной коммутации.

Рассмотрим пакетный коммутатор с буферизацией на выходе, как гарантию того, что пакеты из буфера будут обслужены с определенной скоростью.

Пакетные коммутаторы работают по принципу FIFO, однако поступающие пакеты могут быть разной длины, а значит, при равномерной работе, окажется, что кто-то получит бОльшую долю пропускной способности.

Если пакеты имеют равный размер, то время задержки каждого пакета не более, чем размер буфера делить на пропускную способность канала.

Задание строгих приоритетов нарушает справедливость: на каждом цикле работы сети есть пакеты с высоким приоритетом, то они обслуживаются в первую очередь. При этом пакеты с низким приоритетом оказываются «задавлены», в то время как с высоким даже не знают о них. Это плохо.

Взвешенная справедливая очередь (англ. Weighted fair queuing, WFQ) — механизм планирования пакетных потоков данных с различными приоритетами. Его целью является регулировать использования одного канала передачи данных несколькими конкурирующими потоками. В данном случае под потоком понимается очередь пакетов данных.

Это обобщение алгоритма честных планировщиков (англ. Fair Queuing) (FQ). Оба планировщика имеют отдельные FIFO-очереди для каждого потока данных. Так, если канал со скоростью R используется для N потоков, то скорость обработки каждого из них будет R/N при использовании честного планировщика.

Честный планировщик с приоритетными коэффициентами позволяет регулировать долю каждого потока. Если имеется N активных потоков, с приоритетами □1, ..., □□, то i-ый поток будет иметь скорость:

$$\frac{W_{w_i}}{\sum_{i=1}^{N} w_i}$$

- FIFO очередь – нет приоритетов, не гарантирована скорость.
- Строгие приоритеты: высокоприоритетный трафик «не видит» низкоприоритетного трафика в сети. Полезно, когда высокоприоритетного трафика ограниченное количество.
- Waited Fair Queuing (WFQ) позволяет каждому потоку обеспечить гарантированный сервис, планируя их в порядке bit-by-bit finishing time.

Коммутация пакетов: гарантирование задержки

Основные допущения:

- Пакеты не теряются.
- FIFO обслуживание.

Поскольку мы не можем управлять процессом поступления, его можно ограничить. Пусть число бит, которые могут поступить за период □ ограничены □ + □□ .Где, например, □ = □, □ = □1 – размер канала, тогда мы не столкнемся с проблемой переполнения буфера **(для конкретного случая).**

Одной из причин перегрузок является неравномерный трафик. Если бы этого не было, перегрузок можно было бы избежать. Поэтому используются механизмы формирования трафика, например, **алгоритм текущего ведра.**

Каждая станция, подключенная к сети имеет в своем интерфейсе буфер, подобный ведру и сбрасывающий пакеты при переполнении. Для регулирования скорости поступления пакетов можно, например, использовать системные часы и установить предел числа пакетов, которые можно направить в сеть в промежуток времени. Если пакеты имеют переменную длину – можно ограничивать число байтов, поступающих в сеть.

Иногда бывает полезно ускорить передачу пакетов в сеть, тогда используют **алгоритм текущего ведра с маркерами**. Вместе с пакетами в ведро поступают маркеры, а пакеты выходят только при наличии определенного числа маркеров. Тогда можно накапливать маркеры и кратковременно ускорять передачу пакетов в сеть. Особенность – при переполнении буфера маршрутизатору временно будет запрещено передавать пакеты в сеть.

□ + □□ = □□, тогда$S = \frac{C}{M-\rho}$, где □ – длительность временного увеличения трафика на входе, □ – скорость поступления маркеров Б/с, – максимальная скорость входного трафика Б/с, – емкость корзины в байтах.

Несмотря на то, что технически это возможно, лишь некоторые сети могут управлять e2e задержкой. Причины:

1) Слишком сложно и хлопотно.
2) В большинстве сетей комбинация прогнозирования и приоритетов дает вполне приемлемые результаты.

$$e2e \text{ задержка}, \tau = \sum_i \left(\frac{p}{r_i} + \frac{l_i}{c} + Q_i(t) \right)$$

- Если мы знаем длину очереди и дисциплину ее обслуживания, то мы можем ограничить величину задержки в ней.
- Выбрав длину очереди, и, используя WFQ, мы можем определить скорость обслуживания.
- Поэтому самое главное не допустить сброса пакетов. Для этого можно использовать текущий буфер.
- Таким образом, мы можем ограничить величину e2e задержки.

Управление потоком при пакетной коммутации

В компьютерных сетях неизбежны потери пакетов данных, в частности, из-за переполнения буферной памяти хотя бы одного из узлов, расположенных на пути от источника к приёмнику, включая последний. Такие потери, связанные с переполнениями, в дальнейшем именуются **перегрузками узлов сети**.

Существует множество способов предотвращения и устранения перегрузок; эти способы, в большинстве своем, основаны на управлении потоками данных. Особое место занимает обслуживание пакетов с учетом их приоритетов.

Способ 1. Управление потоком данных регулировкой длительности пауз между пакетами.

Прототип. В процессе передачи данных приемник замечает устойчивую нехватку прибывающих пакетов (например, отслеживая их порядковые номера) и посылает источнику данных управляющий пакет, содержащий команду XOFF приостановки потока данных. Адрес источника данных известен приемнику, так как поступающие к нему пакеты данных содержат информацию об адресах отправителя и адресата. При этом также посылаются запросы на повторную передачу потерянных пакетов.

Получив команду XOFF, источник данных полностью прекращает передачу пакетов и возобновляет ее либо через некоторое время, оговоренное в протоколе обмена данными, либо после получения от приемника команды XON возобновления передачи.

Недостатки:

1) Поток либо есть, либо его нет. Запаздывание выполнения команд может привести к неоправданному простаиванию передатчика и периодическому возникновению новых перегрузок, при которых некоторая часть пакетов, в том числе, принадлежащих другим потокам, будет утеряна.
2) При длительной перегрузке приемник пересылает передатчику серию одинаковых команд приостановки потока, что засоряет канал связи большим количеством повторяющихся служебных пакетов.
3) Команды приостановки работы передатчика формируются приемником только в том случае, когда число отвергнутых в связи с переполнением буфера пакетов достаточно велико.
4) Приостановки работы передатчика увеличивают среднюю и максимальную задержки передачи пакетов по трассе, что может снизить заданные в контракте между пользователем и провайдером показатели качества обслуживания канала.

Решение. Предлагаемое решение в значительной степени устраняет перечисленные недостатки благодаря плавной и «опережающей события» регулировке скорости передачи данных источником. Скорость регулируется изменением длительности пауз между пакетами: чем больше паузы, тем ниже скорость передачи данных, и наоборот. Отметим, что наличие паузы не означает, что сигнал в линии связи отсутствует – сигнал присутствует постоянно, но в нем нет флаговых кодов, обозначающих начало пакета, либо наоборот – передается непрерывный поток этих кодов.

Способ 2. Управление потоком данных оповещением источника пакетов о причинах перегрузки.

Прототип. В ответ на каждый пакет или на группу пакетов приемник посылает источнику ответные пакеты ACK или NACK. Ответ ACK подтверждает успешный прием, ответ NACK служит запросом повторной передачи одного пакета или группы пакетов. Если источник данных при повышении скорости передачи пакетов или на некоторой фиксированной скорости начинает получать чрезмерное число запросов повторной передачи, то, вероятнее всего, по крайней мере, один из узлов трассы вошел в режим перегрузки. В этом случае источник данных резко снижает скорость передачи пакетов или (и) увеличивает их длину, чтобы уменьшить долю передаваемых в потоке данных служебных битов, образующих заголовки. В дальнейшем источник данных постепенно либо методом случайных проб и ошибок повышает скорость передачи данных, продвигаясь к допустимой верхней границе с учетом некоторого разрешенного запаса повышения скорости. Такой способ называют **«медленным стартом»**.

Рассмотренный способ управления потоком данных не предотвращает предстоящую потерю пакетов, а позволяет реагировать только на свершившийся факт перегрузки промежуточного узла сети или приемника данных. В этом состоит его **основной недостаток**.

Другой прототип. Идея состоит в том, чтобы вовремя предупредить источник данных A об угрозе перегрузки одного или нескольких узлов вдоль трассы AB распространения пакетов данных D. Таким предупреждением служит бит Z, входящий в состав заголовка ответного пакета ACK или NACK.

Решение. Поставленная задача решается расширением одноразрядного признака Z до нескольких битов.

Узел может испытывать перегрузку по крайней мере по одной из следующих причин:

1) Сужение полосы (пропускной способности) канала АВ из-за появления «узкого места».
    - Увеличился до значительного уровня ранее малозаметный конкурирующий поток данных по маршруту М4М2М3, который использует тот же канал М2М3, что и маршрут АВ. В результате узел М2 перераспределил полосу этого канала в ущерб маршруту АВ.
    - Узел М2 изменил тип модуляции сигнала в канале М2М3, снизив скорость передачи в связи с ухудшением соотношения «сигнал/шум» в этом канале.
2) Процессор узла М2 по тем или иным причинам перестал справляться с объемом работы по анализу заголовков пакетов, следующих по маршруту АВ.

Первая и вторая причины надвигающейся перегрузки отображаются соответственно кодами □ = 01 и □ = 10, отсутствие опасности перегрузки соответствует коду □ = 00, обе причины одновременно вызывают формирование кода □ = 11.

Способ 3. Управление потоком данных с компенсацией инерционности контура обратной связи. Плавность управления достигается дроблением серий пакетов и более интеллектуальным алгоритмом формирования команд XON и XOFF возобновления и прекращения передачи потока.

19. Алгоритмы маршрутизации в Интернет: основные подходы. Структура сети Интернет, понятие автономной системы, протокол внешней маршрутизации BGP. Явление перегрузки и основные методы борьбы с ней. Перегрузка: AIMD в случае одного потока и в случае нескольких потоков.

Общие сведения.

Основной задачей сетевого уровня является маршрутизация пакетов. Пакеты маршрутизируются всегда, т.е. независимо от того, какую внутреннюю организацию имеет транспортная среда: с виртуальными каналами или дейтаграммную. Разница состоит лишь в том, что в первом случае этот маршрут устанавливается один раз для всех пакетов, а во втором – для каждого пакета отдельно. Первый случай называют иногда маршрутизацией сессии, поскольку маршрут устанавливается на все время передачи данных пользователя, т.е. на время сессии.

Алгоритм маршрутизации реализует программное обеспечение маршрутизатора на сетевом уровне, т.е. он отвечает за определение, по какой из линий, доступных маршрутизатору, отправлять пакет дальше. При этом независимо от выбора маршрута (для сессии или для каждого пакета в отдельности) алгоритм маршрутизации должен обладать следующими свойствами: корректностью, простотой, устойчивостью, стабильностью, справедливостью и оптимальностью.

1) Корректность – свойство алгоритма маршрутизации, определяющее, что при любых обстоятельствах этот алгоритм либо найдет маршрут для доставки пакета адресату, либо выдаст сообщение о невозможности его доставки. Третьего варианта быть не может. При этом крайне желательно, чтобы алгоритм также сообщил о причинах невозможности доставки пакета.
2) Простота – свойство, определяющее вычислительную сложность алгоритма маршрутизации: чем она меньше, тем алгоритм проще, и тем меньше ресурсов маршрутизатора тратится на решение задачи маршрутизации.
3) Устойчивость – свойство алгоритма маршрутизации сохранять работоспособность независимо от каких-либо сбоев, отказов в системе передачи данных или транспортной среде, а также изменений топологии (отключения хостов или машин транспортной среды, разрушения каналов и т.п.). Алгоритм маршрутизации должен адаптироваться ко всем таким изменениям, не требуя при этом перезагрузки транспортной среды или остановки абонентских машин.
4) Стабильность – весьма важное свойство алгоритма маршрутизации. Существуют алгоритмы, которые никогда не приводят к какому-либо определенному маршруту, как бы долго они ни работали. Это означает, что адаптация алгоритма к изменениям в топологии иди конфигурации транспортной среды мажет оказаться весьма продолжительной, и более того, она может оказаться сколь угодно долгой.
5) Справедливость – свойство, означающее, что все пакеты независимо оттого, из какого канала они поступили, будут обслуживаться маршрутизатором равномерно, т.е. никакому направлению не будет отдаваться предпочтение, для всех абонентов будет всегда выбираться оптимальный маршрут.

Следует отметить, что справедливость и оптимальность часто могут вступать в противоречие друг с другом при неудачном выборе критерия оптимизации.

Прежде чем искать компромисс между оптимальностью и справедливостью, необходимо решить, что является критерием оптимизации маршрута. Один из возможных критериев – средняя задержка пакета (обратите внимание, что именно средняя задержка).

Другой критерий – пропускная способность транспортной среды. Однако эти критерии конфликтуют. Согласно теории массового обслуживания, если система с очередями функционирует близко к своему насыщению, то задержка в очереди увеличивается. Как компромисс во многих сетях минимизируется число переходов между маршрутизаторами. Один такой переход называется скачком, или переходом (hop). Уменьшение числа скачков сокращает маршрут, а следовательно, сокращает задержку и минимизирует необходимую пропускную способность СПД для передачи пакета.

Алгоритмы маршрутизации можно разбить на два больших класса: адаптивные и неадаптивные. Неадаптивные алгоритмы не принимают в расчет текущую загрузку сети и ее текущую топологию. Все возможные маршруты вычисляются заранее и загружаются в маршрутизаторы при загрузке сети. Такая маршрутизация называется статической.

Адаптивные алгоритмы, наоборот, определяют маршрут исходя из текущей загрузки и топологии транспортной среды. Адаптивные алгоритмы различаются способом получения информации (локально от соседних маршрутизаторов или глобально от всех маршрутизаторов), временем изменения маршрута (через каждые T секунд либо только когда изменяется нагрузка, либо когда изменяется топология) и метрикой, используемой при оптимизации (расстояние, число скачков, ожидаемое время передачи и т.н.).

Свойство оптимального пути.

Обоснуем одно важное предположение о свойстве оптимального маршрута, которое будет использоваться в дальнейших рассуждениях. Это свойство состоит в том, что если маршрутизатор J

находится на оптимальном пути между маршрутизаторами I и K, то оптимальный маршрут между J и K принадлежит этому оптимальному пути. Это так, поскольку существование между J и K оптимального маршрута, отличного от части маршрута между I и K, противоречило бы утверждению об оптимальности маршрута между I и K.
Поскольку дерево захода – это дерево, то там нет циклов, и каждый пакет будет доставлен за конечное число шагов. На практике же все может оказаться сложнее: маршрутизаторы могут выходить из строя, могут появляться новые маршрутизаторы, каналы могут выходить из строя, разные маршрутизаторы могут узнавать об этих изменениях в разное время и т.д.
Фактически, Интернет состоит из множества локальных и глобальных сетей, принадлежащих различным компаниям и предприятиям, работающих по самым разнообразным протоколам, связанных между собой различными линиями связи, физически передающих данные по телефонным проводам, оптоволокну, через спутники и радиомодемы. Структура Интернет напоминает паутину, в узлах которой находятся компьютеры, связанные между собой линиями связи. Узлы Интернет, связанные высокоскоростными линиями связи, составляют базис Интернет. Оцифрованные данные пересылаются через маршрутизаторы, которые соединяют сети с помощью сложных алгоритмов, выбирая маршруты для информационных потоков.
Каждый компьютер в Интернет имеет свой уникальный адрес – IP-адрес. Этот номер может быть постоянно закреплен за компьютером или же присваиваться динамически – в тот момент, когда пользователь соединился с провайдером, но в любой момент времени в Интернет не существует двух компьютеров с одинаковыми IP-адресами.
Доменное имя – это уникальное имя, которое данный поставщик услуг избрал себе для идентификации. Для преобразования имени в адрес компьютер посылает запрос серверу DNS, начиная с правой части доменного имени и двигаясь влево.
Данные в Интернет пересылаются не целыми файлами, а небольшими блоками, которые называются пакетами. Каждый пакет содержит в себе адреса компьютеров отправителя и получателя, передаваемые данные и порядковый номер пакета в общем потоке данных. Благодаря тому, что каждый пакет содержит все необходимые данные, он может доставляться независимо от других, и довольно часто случается так, что пакеты добираются до места назначения разными путями. А компьютер-получатель затем выбирает из пакетов данные и собирает из них тот файл, который был заказан.
Порт – это число, которое добавляется к адресу компьютера, которое указывает на программу, для которой данные предназначены.
В Интернет используются не просто доменные имена, а универсальные указатели ресурсов URL (Universal Resource Locator).
URL включает в себя:

- метод доступа к ресурсу, т.е. протокол доступа (http, gopher, WAIS, ftp, file, telnet и др.);
- сетевой адрес ресурса (имя хост-машины и домена);
- полный путь к файлу на сервере.

Сервер в сети Интернет – это компьютер, обеспечивающий обслуживание пользователей сети: разделяемый доступ к дискам, файлам, принтеру, системе электронной почты. Обычно сервер – это совокупность аппаратного и программного обеспечения.
Сайт – обобщенное название совокупности документов в Интернет, связанных между собой ссылками.
Шлюз (gateway) – это компьютер или система компьютеров со специальным программным обеспечением, позволяющая связываться двум сетям с разными протоколами.
Домашняя страница – это персональная Web-страница конкретного пользователя или организации.
Автономная система (AS) в интернете – это система IP-сетей и маршрутизаторов, управляемых одним или несколькими операторами, имеющими единую политику маршрутизации с Интернетом.
Протокол BGP используется для передачи информации о внутренних маршрутах между автономными системами. Протокол BGP может быть использован для определения различных типов маршрутов:

- Inter-autonomous system routing – маршруты, которые соединяют данную автономную систему с одной или несколькими другими автономными системами.
- Intra-autonomous system routing – протокол может быть использован для определения маршрута внутри автономной системы, в том случае, когда несколько маршрутизаторов участвуют в процессе определения маршрута BGP.
- Pass-through autonomous system – протокол может быть использован для определения маршрутов, которые проходят через автономную систему, которая не участвует в процессе BGP.

Для обеспечения информационного обмена маршрутизаторы BGP используют сообщения стандартной формы. Для передачи этих сообщений в протоколе BGP предусматривается использование транспортного протокола TCP. Сообщения BGP передаются в следующих случаях:

- Начало сеанса (Open).
- Для периодической проверки состояния соседа (Keep Alive).
- При изменении содержания таблицы маршрутов автономной системы (Update).

- При возникновении аварийной ситуации (Notification).

Формат сообщения BGP. Каждое сообщение BGP состоит из заголовка и последующих специфических полей:

MARKER	
MARKER	
MARKER	
MARKER	
LENGTH	TYPE

В поле LENGTH помещается размер сообщения (вместе с заголовком), выраженный в байтах. В поле TYPE помещается код сообщения в соответствии со следующей таблицей:

TYPE	Сообщение
1	OPEN
2	UPDATE
3	NOTIFICATION
4	KEEPALIVE

В поле маркера может быть помещена информация, которая необходима для выполнения операции аутентификации абонента. Если установление подлинности абонента не требуется, маркер формируется значениями – все «1».

OPEN – первое сообщение, которое должно быть передано маршрутизатором BGP после установления соединения TCP.

UPDATE используется для представления маршрута соседнему маршрутизатору BGP. Это сообщение одновременно может быть использовано для уничтожения маршрутов, которые перестали существовать.

Когда в транспортной среде находится в одно и то же время слишком много пакетов, ее производительность начинает падать. Перегрузка может возникнуть в силу нескольких причин. Например, если сразу несколько потоков, поступающих по нескольким входным линиям, устремятся на одну и ту же выходную линию. Если буфер маршрутизатора переполнится, то пакеты начнут теряться. Перегрузки могут случаться и из-за недостаточной скорости процессора. Если процессор будет не в состоянии справиться своевременно с рутинными задачами (размещения пакета в буфере, корректировка таблиц и т.п.), то даже при наличии линий с достаточной пропускной способностью очередь будет расти. Аналогичная картина может случиться при быстром процессоре, но медленном канале и наоборот. Таким образом, источник проблемы – несбалансированность производительности компонентов системы. Перегрузка – это глобальная проблема в сети.

Управление перегрузками – это такая организация потоков в транспортной среде, при которой потоки соответствуют пропускной способности подсети и не превышают ее.

Основные принципы управления перегрузками.

В терминологии теории управления все методы управления перегрузками в сетях можно разбить на две большие группы: с открытым контуром управления и закрытым контуром управления. Методы с открытым контуром предполагают, что все продумано и предусмотрено заранее в конструкции системы, и если нагрузка находится в заданных пределах, то перегрузки не происходит. Если же нагрузка начинает превышать определенные пределы, то заранее известно, когда и где начнется сброс пакетов, в каких точках сети начнется перепланировка ресурсов, и т.п. Главное, что все эти меры будут приниматься вне зависимости от текущего состояния сети.

Решения, основанные на закрытом контуре, используют обратную связь. Эти решения включают три этапа:

- Наблюдение за системой для определения, где и когда началась перегрузка.
- Передача данных туда, где будут предприняты надлежащие меры.
- Перестройка функционирования системы для устранения проблемы.

При наблюдении за системой используются разные метрики для определения перегрузки. Основными среди них являются:

- Процент пакетов, сброшенных из-за нехватки памяти в буферах.
- Средняя длина очередей в системе.
- Число пакетов, для которых наступил time_out и для которых были сделаны повторные передачи.
- Средняя задержка пакета при доставке и среднее отклонение задержки при доставке пакета.

Следующий шаг при использовании обратной связи – передать информацию о перегрузке туда, где что-то может быть сделано, чтобы исправить положение. Например, маршрутизатор, обнаруживший перегрузку, может направить сообщение о перегрузке всем источникам сообщений. Ясно, что это увеличит нагрузку в сети, причем именно в тот момент, когда это менее всего желательно. Однако есть и другие возможности. Например, в каждом пакете зарезервировать специальный бит перегрузки, и если какой-то маршрутизатор обнаружил перегрузку, то он устанавливает этот бит, тем самым сообщая другим о ней (вспомним структуру кадра во Frame

Relay). Другое решение напоминает прием, используемый некоторыми радиостанциями: направлять несколько автомашин по дорогам, чтобы обнаруживать пробки, а затем сообщать о них по радиоканалам, предупреждая другие машины, призывая их пользоваться объездными путями. По аналогии с этим решением в сети рассылаются специальные пробные пакеты, которые проверяют нагрузку, и если где-то обнаружена перегрузка, то о ней сообщается всем и происходит перенаправление пакетов так, чтобы обогнуть перегруженные участки.

Методы, предотвращающие перегрузки.

Рассмотрение методов, предотвращающих перегрузки, начнем с методов для систем с открытым контуром. Эти методы ориентированы на минимизацию перегрузок при первых признаках их проявлений, а не на борьбу с перегрузками, когда они уже случились. Основные факторы, влияющие на перегрузки на канальном, сетевом и транспортном уровнях, перечислены в таблице:

Уровень	Факторы
Транспортный	Повторная передача Порядок передачи бит Уведомления Управление потоком Значение timeout
Сетевой	Виртуальные каналы vs. дейтаграммы внутри подсети Очередность пакетов и сервисы Сброс пакета Алгоритм маршрутизации Управление временем жизни пакетов
Канальный	Повторная передача Порядок передачи бит Уведомления Управление потоком

Методы:

1) Схема управления потоком (небольшое окно) сдерживает нарастание трафика и предотвращает появление перегрузок.
2) Методы управления очередями, организация очередей: одна общая на входе или одна общая на выходе; по одной на каждую входную линию или на каждую выходную; по одной очереди на каждую входную и выходную – все это влияет на появление перегрузок.
3) Выбор метода сброса пакетов также влияет на перегрузки. Правильная маршрутизация, равномерно использующая каналы в транспортной среде, позволяет избежать перегрузки.
4) Методы, регулирующие время жизни пакета в сети, также влияют на образование перегрузок.

Additive-increase/multiplicative-decrease (AIMD) алгоритм является алгоритмом управления перегрузками с обратной связью. AIMD сочетает линейный рост окна с экспоненциальным сокращением, когда происходит сброс пакета.

Пусть W – размер окна, тогда:

- Если пакет получен, то: $W = W + \frac{1}{W}$
- Если пакет сброшен, то: $W = \frac{W}{2}$

AIMD в случае одного потока:

1) Окно увеличивают, сокращают в соответствии с AIMD, можно определить как много байт канал еще может вместить.
2) Пилообразное поведение графика размера окна от времени – это нормальная форма динамики.
3) Скорость отправки постоянная: $R = \frac{W}{RTT}$. (RTT – Round-trip time), если есть достаточно буферного пространства (R * RTT > W).

AIMD в случае нескольких потоков:

1) Окно увеличивают, сокращают в соответствии с AIMD.
2) В «узком месте» будут скапливаться пакеты разных потоков.
3) Скорость отправки меняется в зависимости от размера окна.
   Доля теряемых пакетов:

$$p = \frac{1}{A}, \text{где } A = \frac{3}{8} W_{max}^2, R = \frac{A}{\frac{W_{max}}{2} RTT}.$$

$$R = \sqrt{\frac{3}{2}} \frac{1}{RTT\sqrt{p}}$$

4) AIMD очень чувствителен к частоте потери пакетов.
5) AIMD ущемляет потоки с большим RTT.

20. Качество программного обеспечения и методы его контроля. Тестирование и другие методы верификации.

Качество программногообеспечения определяется в стандарте ISO 9126 как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Различаются понятия внутреннегокачества, связанного с характеристиками ПО самого по себе, без учета его поведения; внешнего качества характеризующего ПО с точки зрения его поведения; икачества ПО при использовании в различных контекстах— того качества, которое ощущается пользователями при конкретных сценариях работы ПО.

6 основных характеристик качества ПО:

1) Функциональность
   Способность ПОв определенных условиях решатьзадачи, нужные пользователям. Определяет, что именно делает ПО, какие задачи оно решает.
   а) Функциональная пригодность
      Способность решать нужный набор задач.
   б) Точность (accuracy).
      Способность выдавать нужные результаты.
   в) Способность к взаимодействию (interoperability).
      Способность взаимодействовать с нужным набором других систем.
   г) Соответствие стандартам иправилам (compliance).
      Соответствие ПО имеющимся индустриальным стандартам, нормативным изаконодательным актам, другим регулирующим нормам.
   д) Защищенность (security).
      Способность предотвращать неавторизированный, т.е. без указания лица, пытающегося его осуществить, и не разрешенный доступ к данным и программам.
2) Надежность
   Способность ПО поддерживать определенную работоспособность в заданных условиях.
   а) Зрелость, завершенность
      Величина, обратная частоте отказов ПО. Обычно измеряется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени.
   б) Устойчивость к отказам (faulttolerance)
      Способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением.
   в) Способность к восстановлению (recoverability).
      Способность восстанавливатьопределенныйуровень работоспособности и целостностьданных после отказа, необходимые для этого время и ресурсы.
   г)Соответствие стандартам надежности (reliabilitycompliance).
      Этот атрибут добавлен в 2001 году.

3) Удобство использования (usability) или практичность.
Способность ПО быть удобнымв обучении и использовании, а также привлекательным для пользователей.

а) Понятность (understandability).
Показатель, обратный к усилиям, которые затрачиваются пользователями на восприятиеосновных понятий ПО и осознание их

б) Удобство обучения (learnability).
Показатель, обратный усилиям, затрачиваемым пользователями на обучение работе сПО.

в) Удобство работ
Показатель, обратный усилиям, предпринимаемым пользователями для решения своих задач с помощью ПО.

г)Привлекательность (attractiveness).
Способность ПО быть привлекательным для пользователей. Этот атрибут добавлен в 2001.

д) Соответствие стандартам удобства использования (usabilitycompliance).
Этот атрибут добавлен в 2001.

4) Производительность (efficiency) или эффективность
Способность ПОпри заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам. Можно определить ее и как отношение получаемых с помощью ПО результатов к затрачиваемым на это ресурсам.

а) Временная эффективность
Способность ПО выдавать ожидаемые результаты, а также обеспечивать передачу необходимого объема данных за отведенное время.

б) Эффективность использования ресурсов
Способность решать нужные задачи с использованием определенных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременнаяпамять, сетевые соединения, устройства ввода и вывода, и пр.

в)Соответствие стандартам производительности (efficiencycompliance).
Этот атрибут добавлен в 2001.

5) Удобство сопровождения (maintainability).
Удобство проведения всех видов деятельности, связанных с сопровождение программ

а) Анализируемость (analyzability) или удобство проведения анализа
Удобство проведения анализа ошибок, дефектов и недостатков, а также удобствоанализа необходимости изменений и их возможных последствий.

б) Удобство внесения изменений (changeability).
Показатель, обратный трудозатратам на выполнение необходимых изменений.

в) Стабильность (stability).
Показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений.

г) Удобство проверки (testability).
Показатель, обратный трудозатратам на проведение тестирования и других видовпроверки того, что внесенные изменения привели к нужным результатам.

д) Соответствие стандартам удобства сопровождения
Этот атрибут добавлен в 2001.

6) Переносимость (portability).
Способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения.Иногда эта характеристиканазывается в русскоязычной литературе мобильностью.

а)Адаптируемость (adaptability).
Способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных

б)Удобство установки (installability).
Способность ПО быть установленным или развернутым в определенном окружении

в) Способность к сосуществованию (coexistence).
Способность ПО сосуществовать с другими программами в общем окружении, деля с ними ресурсы.

г) Удобство замены (replaceability) другого ПО данным.
Возможность применения данногоПОвместо других программных систем для решения тех же задач в определенном окружении.

д) Соответствие стандартам переносимости (portabilitycompliance).
Этот атрибут добавлен в 2001.

Методы контроля качества

Верификация обозначает проверку того, что ПО разработано в соответствии со всемитребованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.

Валидация — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого родаситуации называют тестовыми или просто тестами.

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен.Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций.Тем не менее, тестирование может использоваться для достаточно уверенного вынесения оценок о качестве ПО. Для этого необходимо иметь *критерии полноты тестирования*, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними.

Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В таком случае считают, что все равно какую из ситуаций класса использовать в качестве теста.Такой критерий называют *критерием тестового покрытия*, а процент классов

эквивалентности ситуаций, случившихся во время тестирования, — достигнутым *тестовым покрытием*.
Таим образом, основные задачи тестирования: построить такой набор ситуаций, который достаточно представителен и позволял бы завершить тестирование с достаточной степенью уверенности в правильности ПО вообще, и убедиться, что в конкретной ситуации ПО работает правильно, в соответствии с требованиями.

Кроме того, особо выделяют нагрузочное ил стрессовое тестирование, проверяющее работоспособность ПО и показатели его производительности в условиях повышенных нагрузок.
Наоснове исходных данных, используемых для построения тестов, тестирование делят на следующие виды:

1) Тестирование методом черного ящика. (Тесты строятся на основе требований и ограничений, прописанных в нормативных актах)
2) Тестирование методом белого ящика. (Тесты создаются на основе знаний о структуре самой системы и о том, как она работает)
3) Тестирование, нацеленное на определенные ошибки. (Тесты строятся так, чтобы гарантированно выявить определенные ошибки)

Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено.

1) Модульное тестирование (Тестирование корректности отдельного модуля вне зависимости от его окружения.При этом проверяется, что еслимодуль получает на вход данные, удовлетворяющие определенным критериямкорректности, то и результаты его корректны.)
2) Интеграционное тестирование. (Предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работе модули взаимодействуют, не нарушая ограничений на это взаимодействие.)
3) Системное тестирование.(Предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователем задачи в разных ситуациях.)

Проверка на моделях

Проверка свойств на моделях - проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях.(Например, отсутствие взаимных блокировок)
Свойства безопасности (safetyproperties) утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. Свойства живучести, наоборот, утверждают, что нечто желательное при любом развитии событий произойдет в ходе его работы.(Например, гарантированная доставка сообщения)
В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде так называемых временнЫх логик. Операторы «всегда в будущем » и «когда-то в будущем».

Программа, как правило, моделируется с помощью конечного автомата. Проверка, выполняемая автоматически, состоит в том, что для всех достижимых при работе системы

состояний этого автомата проверяется нужное свойство. Если оно выполнено – все хорошо, если нет – ошибка.
Основная проблема – большое количество состояний в моделях, хорошо отражающих реальные системы.

21. Виды параллельной обработки данных, их особенности. Компьютеры с общей и распределенной памятью. Вычислительные кластеры: узлы, коммуникационная сеть, способы построения. Производительность вычислительных систем, методы оценки и измерения.

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность.

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени. УВЕЛИЧЕНИЕ КОЛИЧЕСТВА НЕЗАВИСИМО РАБОТАЮЩИХ УСТРОЙСТВ.

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций таких, как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п.
Процессоры первых компьютеров выполняли все эти "микрооперации" для каждой пары аргументов последовательно одна за однойдо тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых. УСЛОЖНИТЬ САМО УСТРОЙСТВО, ЧТОБЫ НА РАЗНЫХ ЭТАПАХ МОГЛИ НАХОДИТЬСЯ РАЗНЫЕ ДАННЫЕ.
Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передавал бы результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных вовремени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находится первые пять пар аргументов, а весь набор из ста пар будет обработан за 5+99=104 единицы времени - ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера). ТЕ СУЩЕСТВУЕТ НЕКОТОРАЯ ЗАДЕРЖКА ДЛЯ ТОГО ЧТОБЫ ЗАПОЛНИТЬ ВСЕ ЭТАПЫ КОНВЕЕРА, НО КОГДА ОНА ЗАПОЛНЕНА ДАЛЬШЕ ПРОИСХОДИТ УСКОРЕНИЕ ОБРАБОТКИ.

1. **Векторно-конвейерные компьютеры**. Конвейерные функциональные устройства и набор векторных команд - это две особенности таких машин. В отличие от традиционного подхода, векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные конвейеры, т.е. команда вида A=B+C может означать сложение двух массивов, а не двух чисел. Характерным представителем данного направления является семейство векторно-конвейерных компьютеров CRAY куда входят,

например, CRAY EL, CRAY J90, CRAY T90 (в марте 2000 года американская компания TERA перекупила подразделение CRAY у компании SiliconGraphics, Inc.).

2. **Массивно-параллельные компьютеры с распределенной памятью**. Идея построения компьютеров этого класса тривиальна: возьмем серийные микропроцессоры, снабдим каждый своей локальной памятью, соединим посредством некоторой коммуникационной среды - вот и все. Достоинств у такой архитектуры масса: если нужна высокая производительность, то можно добавить еще процессоров, если ограничены финансы или заранее известна требуемая вычислительная мощность, то легко подобрать оптимальную конфигурацию и т.п.

Однако есть и решающий "минус", сводящий многие "плюсы" на нет. Дело в том, что межпроцессорное взаимодействие в компьютерах этого класса идет намного медленнее, чем происходит локальная обработка данных самими процессорами. Именно поэтому написать эффективную программу для таких компьютеров очень сложно, а для некоторых алгоритмов иногда просто невозможно. К данному классу можно отнести компьютеры IntelParagon, IBM SP1, Parsytec, в какой-то степени IBM SP2 и CRAY T3D/T3E, хотя в этих компьютерах влияние указанного минуса значительно ослаблено. К этому же классу можно отнести и сети компьютеров, которые все чаще рассматривают как дешевую альтернативу крайне дорогим суперкомпьютерам.

3. **Параллельные компьютеры с общей памятью**. Вся оперативная память таких компьютеров разделяется несколькими одинаковыми процессорами. Это снимает проблемы предыдущего класса, но добавляет новые - число процессоров, имеющихдоступ к общей памяти, по чисто техническим причинам нельзя сделать большим. Вданное направление входят многие современные многопроцессорные SMP-компьютеры или, например, отдельные узлы компьютеров HP Exemplar и SunStarFire.

4. Последнее направление, строго говоря, не является самостоятельным, а скорее представляет собой комбинации предыдущих трех. Из нескольких процессоров (традиционных или векторно-конвейерных) и общей для них памяти сформируем вычислительный узел. Если полученной вычислительной мощности не достаточно, то объединим несколько узлов высокоскоростными каналами. Подобную архитектуру называют кластерной, и по такому принципу построены CRAY SV1, HP Exemplar, SunStarFire, NEC SX-5, последние модели IBM SP2 и другие. Именно это направление является в настоящее время наиболее перспективным для конструирования компьютеров с рекордными показателями производительности.

Различных вариантов построения кластеров очень много. Одно из существенных различий лежит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом решаемых задач.

Какими же числовыми характеристиками выражается производительностькоммуникационных сетей в кластерных системах? Необходимых пользователю характеристик две: латентность и пропускная способность сети. Латентность — это время начальной задержки при посылке сообщений. Пропускная способность сети определяется скоростью передачи информации по каналам связи. Если в программе много маленьких сообщений, то сильно скажется латентность. Если сообщения передаются большими порциями, то важна высокая пропускная способность

каналов связи. Из-за латентности максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной. На практике пользователям не столько важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений. После вызова пользователем функции посылки сообщения Send() сообщение последовательно проходит через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры. Этим, в частности, определяются и множество вариаций на тему латентности реальных систем. Установили MPI на компьютере плохо,
латентность будет большая, купили дешевую сетевую карту от неизвестного производителя, ждите дальнейших сюрпризов.
В заключение параграфа давайте попробуем и для данного класса компьютеров выделить факторы, снижающие производительность вычислительных систем с распределенной памятью на реальных программах. Начнем с уже упоминавшегося ранее закона Амдала. Для компьютеров данного класса он играет очень большую роль. В самом деле, если предположить, что в программе есть лишь 2% последовательных операций, то рассчитывать на более чем 50-кратное ускорение работы программы не приходится. Теперь попробуйте критически взглянуть на своюпрограмму. Скорее всего, в ней есть инициализация, операции ввода/вывода, какие-то сугубо последовательные участки. Оцените их долю на фоне всей программы и на мгновенье предположите, что вы получили доступ к вычислительной системе из 1000 процессоров. После вычисления верхней границы для ускорения программы на такой системе, думаем, станет ясно, что недооценивать влияние закона Амдала никак нельзя. Поскольку компьютеры данного класса имеют распределенную память, то взаимодействие процессоров между собой осуществляется с помощью передачи сообщений. Отсюда два других замедляющих фактора — латентность и скорость передачи данных по каналам коммуникационной среды. В зависимости от коммуникационной структуры программы степень влияния этих факторов может сильноменяться.
Если аппаратура или программное обеспечение не поддерживают возможности асинхронной посылки сообщений на фоне вычислений, то возникнут неизбежные накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов.

Для достижения эффективной параллельной обработки необходимо добиться максимально равномерной загрузки всех процессоров. Если равномерности нет, то часть процессоров неизбежно будет простаивать, ожидая остальных, хотя в это время они вполне могли бы выполнять полезную работу. Данная проблема решается проще, если вычислительная система однородна. Очень большие трудности возникают при переходе на неоднородные системы, в которых есть значительное различие либо между вычислительными узлами, либо между каналами связи. Существенный фактор — это реальная производительность одного процессора вычислительной системы. Разные модели микропроцессоров могут поддерживатьнесколько уровней кэш-памяти, иметь специализированные функциональные устройства и т. п. Возьмем хотя бы иерархию памяти компьютера Cray T3E: регистры процессора, кэш-память 1-го уровня, кэш-память 2-го уровня, локальная память процессора, удаленная память другого процессора. Эффективное использование такой структуры требует особого внимания при выборе подхода к решению задачи. Дополнительно каждый микропроцессор может иметь элементы векторно-конвейерной архитектуры. К сожалению, как и прежде, на работе каждой конкретной программы в той или иной мере сказываются все эти факторы. Однако в отличие от компьютеров других классов, суммарное воздействие изложенных

здесь факторов может снизить реальную производительность не в десятки, а в сотни и даже тысячи раз по сравнению с пиковой.
Потенциал компьютеров этого класса огромен, добиться на них можно очень многого. Крайняя точка — Интернет. Его тоже можно рассматривать как компьютер с распределенной памятью. Причем, как самый мощный в мире компьютер.

Пиковая производительность.
Главной отличительной особенностью многопроцессорной вычислительной системы является ее производительность, т.е. количество операций, производимых системой за единицу времени. Различают пиковую и реальную производительность.
Подпиковойпонимают величину, равную произведению пиковой производительности одного процессора на число таких процессоров в данной машине. При этом предполагается, что все устройства компьютера работают в максимально производительном режиме. Пиковая производительность компьютера вычисляется однозначно, и эта характеристика является базовой, по которой производят сравнение высокопроизводительных вычислительных систем. Чем больше пиковая производительность, тем (теоретически) быстрее пользователь сможет решить свою задачу. Пиковая производительность есть величина теоретическая и, вообще говоря, недостижимая при запуске конкретного приложения. Реальная же производительность, достигаемая на данном приложении, зависит от взаимодействия программной модели, в которой реализовано приложение, с архитектурными особенностями машины, на которой приложение запускается.
Существует два способа оценки пиковой производительности компьютера. Один из них опирается на число команд, выполняемых компьютером за единицу времени. Единицей измерения, как правило, является MIPS (MillionInstructionsPerSecond).
Производительность, выраженная в MIPS, говорит о скорости выполнения компьютером своих же инструкций. Но, во-первых, заранее не ясно, в какое количество инструкций отобразится конкретная программа, а во-вторых, каждая программа обладает своей спецификой, и число команд от программы к программе может меняться очень сильно. В связи с этим данная характеристика дает лишь самое общее представление о производительности компьютера.
Другой способ измерения производительности заключается в определении числа вещественных операций, выполняемых компьютером за единицу времени. Единицей измерения является Flops (Floatingpointoperationspersecond) – число операций с плавающей точкой, производимых компьютером за одну секунду. Такой способ является более приемлемым для пользователя, поскольку ему известна вычислительная сложность программы, и, пользуясь этой характеристикой, пользователь может получить нижнюю оценку времени ее выполнения.
Однако пиковая производительность получается только в идеальных условиях, т.е. при отсутствии конфликтов при обращении к памяти при равномерной загрузке всех устройств.
В реальных условиях на выполнение конкретной программы влияют такие аппаратно-программные особенности данного компьютера как: особенности структуры процессора, системы команд, состав функциональных устройств, реализация ввода/вывода, эффективность работы компиляторов.
Методы оценки производительности.
Основу для сравнения различных типов компьютеров между собой дают стандартные методики измерения производительности. В процессе развития вычислительной техники появилось несколько таких стандартных методик. Они позволяют разработчикам и

пользователям осуществлять выбор между альтернативами на основе количественных показателей, что дает возможность постоянного прогресса в данной области. Единицей измерения производительности компьютера является время: компьютер, выполняющий тот же объем работы за меньшее время является более быстрым. Время выполнения любой программы измеряется в секундах. Часто производительность измеряется как скорость появления некоторого числа событий в секунду, так что меньшее время подразумевает большую производительность.

Однако в зависимости от того, что мы считаем, время может быть определено различными способами. Наиболее простой способ определения времени называется астрономическим временем, временем ответа (responsetime), временем выполнения(executiontime) или прошедшим временем (elapsedtime). Это задержка выполнения задания, включающая буквально все: работу процессора, обращения к диску, обращения к памяти, ввод/вывод и накладные расходы операционной системы. Однако при работе в мультипрограммном режиме во время ожидания ввода/вывода для одной программы, процессор может выполнять другую программу, и система не обязательно будет минимизировать время выполнения данной конкретной программы.

Для измерения времени работы процессора на данной программе используется специальный параметр - время ЦП (CPU time), которое не включает время ожидания ввода/вывода или время выполнения другой программы. Очевидно, что время ответа, видимое пользователем, является полным временем выполнения программы, а не временемЦП. Время ЦП может далее делиться на время, потраченное ЦП непосредственно на выполнение программы пользователя и называемое пользовательским временем ЦП, и время ЦП, затраченное операционной системой на выполнение заданий, затребованных программой, и называемое системным временем ЦП.

В ряде случаев системное время ЦП игнорируется из-за возможной неточности измерений, выполняемых самой операционной системой, а также из-за проблем, связанных со сравнением производительности машин с разными операционными системами. С другой стороны, системный код на некоторых машинах является пользовательским кодом на других и, кроме того, практически никакая программа не может работать без некоторой операционной системы. Поэтому при измерениях производительности процессора часто используется сумма пользовательского и системного времени ЦП.

В большинстве современных процессоров скорость протекания процессов взаимодействиявнутренних функциональных устройств определяется не естественными задержками в этих устройствах, а задается единой системой синхросигналов, вырабатываемых некоторым генератором тактовых импульсов, как правило, работающим с постоянной скоростью.

Дискретные временные события называются тактами синхронизации (clockticks), просто тактами (ticks), периодами синхронизации (clockperiods), циклами (cycles) или циклами синхронизации (clockcycles).Разработчики компьютеров обычно говорят о периоде синхронизации, который определяется либо своей длительностью (например, 10 наносекунд), либо частотой (например, 100 МГц). Длительность периода синхронизации есть величина, обратная к частоте синхронизации.

Таким образом, время ЦП для некоторой программы может быть выражено двумя способами: количеством тактов синхронизации для данной программы, умноженным на длительность такта синхронизации, либо количеством тактов синхронизации для данной программы, деленным на частоту синхронизации.

Важной характеристикой, часто публикуемой в отчетах по процессорам, является среднее количество тактов синхронизации на одну команду - CPI (clockcyclesperinstruction). При известном количестве выполняемых команд в программе этот параметр позволяет быстро оценить время ЦП для данной программы. Таким образом, производительность ЦП зависит от трех параметров: такта (или частоты) синхронизации, среднего количества тактов на команду и количества выполняемых команд. Невозможно изменить ни один из указанных параметров изолированно от другого, поскольку базовые технологии, используемые для изменения каждого из этих параметров, взаимосвязаны: частота синхронизации определяется технологией аппаратных средств и функциональной организацией процессора; среднее количество тактов на команду зависит от функциональной организации и архитектуры системы команд; а количество выполняемых в программе команд определяется архитектурой системы команд и технологией компиляторов. Когда сравниваются две машины, необходимо рассматривать все три компоненты, чтобы понять относительную производительность. В процессе поиска стандартной единицы измерения производительности компьютеровбыло принято несколько популярных единиц измерения, вследствие чего несколько безвредных терминов были искусственно вырваны из их хорошо определенного контекста и использованы там, для чего они никогда не предназначались. В действительности единственной подходящей и надежной единицей измерения производительности является время выполнения реальных программ, и все предлагаемые замены этого времени в качестве единицы измерения или замены реальных программ в качестве объектов измерения на синтетические программы только вводят в заблуждение.

Тесты для оценки производительности систем:

Ливерморский набор циклов : 24 цикла разной степени векторизуемости
В Ливерморских циклах встречаются последовательные, сеточные, конвейерные, волновые вычислительные алгоритмы, что подтверждает их пригодность и для параллельных машин.

Набор тестов LINPACK.
В основе алгоритмов действующего варианта LINPACK лежит метод декомпозиции. Исходная матрица размером 100x100 элементов (в последнем варианте размером 1000x1000) сначала представляется в виде произведения двух матриц стандартной структуры, над которыми затем выполняется собственно алгоритм нахождения решения.

Тесты SPEC
Набор тестов CINT92, измеряющий производительность процессора при обработке целых чисел, состоит из шести программ, написанных на языке Си и выбранных из различных прикладных областей: теория цепей, интерпретатор языка Лисп, разработка логических схем, упаковка текстовых файлов, электронные таблицы и компиляция программ.

Набор тестов CFP92, измеряющий производительность процессора при обработке чисел с плавающей точкой, состоит из 14 программ, также выбранных из различных прикладных областей: разработка аналоговых схем, моделирование методом Монте-Карло, квантовая химия, оптика, робототехника, квантовая физика, астрофизика, прогноз погоды и другиенаучные и инженерные задачи. Две программы из этого набора написаны на языке Си, а остальные 12 - на Фортране. В пяти программах используется одинарная, а в остальных -двойная точность.

22. Закон Амдала, его следствия. Граф алгоритма. Критический путь графа алгоритма, ярусно-параллельная форма графа алгоритма. Этапы решения задач на параллельных вычислительных системах.

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f, где 0<=f<=1 (при этом доля понимается не по статическом числу строк кода, а по числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным (f=0) и полностью последовательным (f=1) программам. Так вот, для того, чтобы оценить, какое ускорение S
может быть получено на компьютере из 'p' процессоров при данном значении f, можно воспользоваться законом Амдала:

$$S \leq \frac{1}{f + \frac{1-f}{p}}$$

Если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более, чем в 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0).
Посмотрим на проблему с другой стороны: а какую же часть кода надо ускорить (а значит и предварительно исследовать), чтобы получить заданное ускорение? Ответ можно найти в следствии из закона Амдала: для того чтобы ускорить выполнение программы в q раз необходимо ускорить не менее, чем в q раз не менее, чем (1-1/q)-ю часть программы.

Пусть при фиксированных входных данных программа описывает некоторый алгоритм. Построим ориентированный граф. В качестве вершин возьмем любое множество, например, множество точек арифметического пространства, на котороевзаимнооднозначно отображается множество всех операций алгоритма. Возьмем любую пару вершин и, v. Допустим, что согласно описанному выше частичному порядку операция, соответствующая вершине и, должна поставлять аргумент операции, соответствующей вершине v. Тогда проведем дугу из вершины и в вершину y. Если соответствующие операции могут выполняться независимо друг от друга, дугу проводить не будем. В случае, когда аргументом операции являетсяначальное данное или результат операции нигде не используется, возможны различные договоренности. Например, можно считать, что соответствующие дуги отсутствуют. Мы будем поступать в зависимости от обстоятельств. Построенный таким образом граф будем называть графом алгоритма.
Независимо от способа построения ориентированного графа, те его вершины, которые не имеют ни одной входящей или выходящей дуги, будем называть соответственно входными или выходными вершинами графа.

Критический путь графа — путь максимальной длины в ориентированном ациклическом графе.

Его длина является минимальной из всех возможных высот у ярусно-параллельной формы данного ациклического графа.

Утверждение 4.1

Пусть задан ориентированный ациклический граф, имеющий n вершин. Существует число s < n, для которого все вершины графа можно так пометить одним изиндексов 1, 2, ..., s, что если дуга из вершины с индексом i идет в вершину с индексом j, тоi<j.
Выберем в графе любое число вершин, не имеющих предшествующих, и пометим их индексом 1. Удалим из графа помеченные вершины и инцидентные им дуги.
Оставшийся граф также является ациклическим. Выберем в нем любое число вершин, не имеющих предшествующих, и пометим их индексом 2. Продолжая этот процесс, в конце концов, исчерпаем весь граф. Так как при каждом шаге помечается не менее одной вершины, то число различных индексов не превышает числа вершин графа. Отсюда следует, что никакие две вершины с одним и тем же индексом не связаны дугой. Минимальное число индексов, которым можно пометить все вершины графа, на 1 больше длины его критического пути. И, наконец, для любого целого числа s, не превосходящего общего числа вершин, но большего длины критического пути, существует такая разметка вершин графа, при которой используются все s индексов.

Граф, размеченный в соответствии с утверждением 4.1, называется строгой параллельной формой графа. Если в параллельной форме некоторая вершина помечена индексом k, то это означает, что длины всех путей, оканчивающихся в данной вершине, меньше к. Существует строгая параллельная форма, при которой максимальная из длипутей, оканчивающихся в вершине с индексом k, равна k-1. Для этой параллельной формы число используемых индексов на 1 больше длины критического пути графа. Среди подобных параллельных форм существует такая, в которой все входные вершины находятся в группе с одним индексом, равным 1. Эта строгая параллельная форма называется канонической. Для заданного графа его каноническая параллельная форма единственна. Группа вершин, имеющих одинаковые индексы, называется ярусом параллельной формы, а число вершин в группе — шириной яруса. Число ярусов в параллельной форме называется высотой параллельной формы, а максимальная ширина ярусов — ееширином. Параллельная форма минимальной высоты называется максимальной.

Этапы решения задач на параллельных вычислительных системах

1) Задача
2) Метод
3) Алгоритм
4) Технология параллельного программирования
5) Программа
6) Компилятор
7) Компьютер
1) Задача

23. Технологии параллельного программирования MPI и OpenMP.

Наиболее распространенной технологией программирования параллельных

компьютеров с распределенной памятью в настоящее время является MPI.

MPI поддерживает работу с языками C и Fortran. В данной книге все примеры и описания всех функций будут даны с использованием языка C. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи. Полная версия интерфейса содержит описание более 120 функций.

Интерфейс поддерживает создание параллельных программ в стиле MIMD, что подразумевает объединение процессов с различными исходными текстами. Однако на практике программисты гораздо чаще используют SPMD-модель, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Все дополнительные объекты: имена функций, константы, предопределенные типы данных и т. п., используемые в MPI, имеют префикс MPI_. Например, функция посылкисообщения от одного процесса другому имеет имя MPi_Send. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. Все описания интерфейса MPI собраны в файле mpi.h, поэтому в начале MPI-программы должна стоять директива #include<mpi.h>.

MPI-программа — это множество параллельных взаимодействующих процессов.

Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается.

Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать группы процессов, предоставляя им отдельную среду для общения — коммуникатор. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя MPI_COMM_WORLD. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет уникальный атрибут номер процесса, который является целым неотрицательным числом. С помощью этого атрибута

происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда два основных атрибута процесса: коммуникатор и номер в коммуникаторе.

Если группа содержит п процессов, то номер любого процесса в данной группе лежит в пределах от 0 до п - 1. Подобная линейная нумерация не всегда адекватно отражает логическую взаимосвязь процессов приложения. Например, по смыслу задачи процессы могут располагаться в узлах прямоугольной решетки и взаимодействовать только со своими непосредственными соседями. Такую ситуацию пользователь может легко отразить в своей программе, описав соответствующую виртуальную топологию процессов. Эта информация может оказаться полезной при отображении процессов программы на физические процессоры вычислительной системы. Сам процесс отображения в MPI никак не специфицируется, однако система поддержки MPI в ряде случаев может значительно уменьшить коммуникационные накладные расходы, если воспользуется знанием виртуальной топологии.
Основным способом общения процессов между собой является посылка со-общений. Сообщение — это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и др. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до 32 767. Для работы с атрибутами сообщений введена структура MPistatus, поля которой дают доступ к значениям атрибутов. На практике сообщение чаще всего является набором однотипных данных, расположенных подряд друг за другом в некотором буфере. Такое сообщение может состоять, например, из двухсот целых чисел, которые пользователь разместил в соответствующем целочисленном векторе. Это типичная ситуация, на нее ориентированобольшинство функций MPI, однако такая ситуация имеет, по крайней мере, два ограничения. Во-первых, иногда необходимо составить сообщение из разнотипных данных. Конечно же, можно отдельным сообщением послать количество вещественных чисел, содержащихся в последующем сообщении, но это может быть и неудобно программисту, и не столь эффективно. Во-вторых, не всегда посылаемые данные занимают непрерывную область в памяти. Если в Fortran элементы столбцов матрицы расположены в памяти друг за другом, то элементы строк уже идут с некоторым шагом. Чтобы послать строку, данные нужно сначала упаковать, передать, а затем вновь распаковать.
Чтобы снять указанные ограничения, в MPI предусмотрен механизм для введения производных типов данных (deriveddatatypes). Описав состав и схему размещения в памяти посылаемых данных, пользователь в дальнейшем работает с такими типами так же, как и со стандартными типами данных MPI.

Режимы пересылки, приема данных, коллективные операции.

Одним из наиболее популярных средств программирования компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берется последовательная программа, а для создания ее параллельной

версиипользователю предоставляется набор директив, процедур и переменных окружения.

Стандарт OpenMP разработан для языков Фортран, C и C++. Поскольку все основные конструкции для этих языков похожи, то рассказ о данной технологии мы будем вести на примере только одного из них, а именно на примере языка Фортран.
Как, с точки зрения OpenMP, пользователь должен представлять свою параллельную программу?
Весь текст программы разбит на последовательные и параллельные области (см. рис.1). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Здесь следует сразу сказать, почему вместо традиционных для параллельного программирования процессов появился новый термин - нити (threads, легковесные процессы). Технология OpenMP опирается на понятие общей памяти, поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов.
Основная нить и только она исполняет все последовательные области программы.

При входе в параллельную область порождаются дополнительные нити. После порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

В параллельной области все переменные программы разделяются на два класса: общие (SHARED) и локальные (PRIVATE). Общая переменная всегда существует лишь в одном экземпляре для всей программы и доступна всем нитям под одним и тем жеименем. Объявление же локальной переменной вызывает порождение своего экземпляра данной переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.

По сути, только что рассмотренные два понятия: области и классы переменных, и определяют идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляется параллельными областями; именно эти областии только они исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными. Все остальное - это конкретизация деталей и описание особенностей реализации данной идеи на практике.