

# С/к Введение в современное программирование (v.5.5)

Физфак МГУ. 2006/7 уч. год.

## Лекция 3

Раз продолжают вопросы:

Цель курса — дать опорные сведения ("фунд. знания" и т.п.) для ориентации в сфере ИТ.

Включая основной минимум праткич. знаний.

В лекции три логические части:

**Очередная доза метафизика**

**Мат. основы, часть 1**

**Конкр. сведения**

### Напоминание

Сфера ИТ: "раковая опухоль избыточной сложности".

Главная причина: источник "ренды сложности" для корпорации ИТ

("коллективный эффект").

Еще не выяснили конкретные механизмы (что важно).

С другой стороны: **Избыточная сложность = уязвимость** (принцип

Калашникова).

Другие примеры:

Ford T "вынес" конкурентов;

T-34 и Sherman vs немецких танков.

Действие принципа ослаблено внутри сферы ИТ в конечном счете из-за быстрого расширения этой самой сферы. Опять же интересно, как и почему именно.

Общее практическое правило:

**Как огня бояться избыточной сложности.** -- надо бы пояснить, где и как.

## Из метафизики

Любая конструктивная деятельность (в т.ч. сочинение лекций) подразумевает компромиссы.

**Принцип неравномерности (Парето, 80/20) — основа эффективных компромиссов**

*40% граждан США платят 90% всего подоходного налога;*

*80% времени программа проводит в 20% кода*

*0.01% населения глупы настолько, что оправдывают рассылку миллионов копий "нигерийского" спама*

**Выделять главное, на котором и сосредоточиться (= "упростить")**

Выделять главное трудно.

Речь пойдет о сложности в общем метафизическом плане — **не** в смысле математической теории сложности алгоритмов.

## Почему в программировании приходится говорить о сложности

Огромный семантический разрыв между приложениями ("вычислить вторую поправку к ширине распада...") и железом ("mov ds, ax").

При том, что голова не чердак, всего не упомнишь (~7 "регистров"

кратковременной памяти).

**Люди постоянно ошибаются.** (Мозг — не digital device.)

## Что такое сложные задачи

— трудно явно сформулировать (очень много условий, "клиент" сам толком не знает, чего хочет -- даже чего можно хотеть).

— постановка обязательно будет меняться в процессе решения по мере прояснения.

— постановка меняется во времени (потому что задача существует в меняющейся реальности),

**... а жизнь непредсказуема** (причем со страшной силой).

— тенденция к уникальности (электроны тождественны, люди -- все разные).

### Следствия:

— Для сложной задачи трудно ожидать, что найдется готовое решение, в лучшем случае решение похожей задачи.

— Предсказать, как именно будет отличаться следующая задача, невозможно.

Значит, невозможно спроектировать библиотеку (предусмотреть все возможные управляющие параметры и т.п.) на все времена: зачастую проще заново написать свою процедуру под конкретные условия, чем бороться с процедурой с десятком параметров и сложными свойствами.

— Отсюда объективное основание для open source ("открытые исходники"): самое полное решение есть сам алгоритм. (Уточнение А.Шалыто: исходники по-настоящему полезны, когда они сопровождаются проектной документацией, т.е. объяснениями, *почему* сделано то или иное.)

Итого: **Обязательно придется менять программу.**

Фокс подчеркивал итерационный характер дизайна на всех уровнях.

Вплоть до готовности переписать все с нуля.

*"Мудрым пользуйся девизом — будь готов к любым сюрпризам"*

Более того:

**Изменения программы = главный источник ошибок** и ухудшения структуры (нарастание "энтропии").

Причина: Когда делается первый вариант, автор держит в голове всю структуру и помнит все идеи. Исправления/модификации делаются локально, а прогос, заплатками -- часто и не автором (или автором, который уже забыл детали первоначального замысла -- "Какого дьявола тут делает это?!").

**Программы живут дольше, чем предполагается ...**

Эксп. наблюдения:

Главные усилия — на изменения программы (при итеративном проектировании, адаптации и т.п.), а не на первоначальный вариант.

**Львиная доля всех затрат — сопровождение.**

101 Поэтому **evolvability** = **главный показатель качества программ**  
 102 (корректных, конечно)  
 103 **Принцип приоритета чтения** перед написанием при работе с программами  
 104 **Принцип минимализма** при проектировании программ  
 105 **Хорошая программа = простая и ясная** на каждом участке/уровне  
 106 абстракции

107 Практическое правило:  
 108 — Опасно закладывать в программу возможности "на всякий случай" --  
 109 вероятность "не угадать" велика, а потери -- немедленные.  
 110 Наоборот, делать программу попроще, но с предельно ясной структурой, чтобы  
 111 полетче было менять.

112 **Сильная недооценка трудности сопровождения —**  
 113 **постоянная ошибка.**

114 Типичное явление:  
 115 "закрытый" продукт, рассчитанный на большой рынок, с множеством "фич" —  
 116 ровно та ситуация, где невозможно не ошибиться.

117 Клиент  
 118 балдеет от множества обещанных фич,  
 119 начав пользоваться — не может отказаться.  
 120 Производитель  
 121 не может ничего изменить, даже если захочет.  
 122 продукт открывать не хочет, вынужден делать нашлапки.

123 Сюда же: **Закон насыщения степеней свободы ЯП**

124 **Как бороться со сложностью**

125 **Упорно и постоянно.**

126 **Принцип неравномерности (Парето, 80/20)**

127 **Выделять главное, на котором и сосредоточиться**

128 Выделять главное трудно.

129 **Divide et impera = Разделяй и властвуй** — первая основа  
 130 управления сложностью.

131 modularity, information hiding

132 **Принцип иерархии** — вторая основа управления сложностью.

133 Знаменитое пошаговое уточнение = комбинация этих двух.

134 **Принцип раннейшего обнаружения ошибок**

135 **Чем позже** обнаружена ошибка, **тем дороже** она обходится

136 **Приоритет статических проверок** (компилятор) перед динамическими

137 —> важность строгой статической типизации

138 —> важность автоматического управления памятью

139 **Приоритет проверок** в компиляторе, а не в тулзовинах

140 По поводу инструментов:

141 Раз голова не чердак, и всего не упомнишь, то

142 —> **важность минималистичных инструментов**

143 Раз люди постоянно ошибаются, то

144 —> **важность "защитных" свойств ЯП**

145 —> ЯП должен быть как можно проще (хотя бы подмножество...)

146 **Принцип защитного программирования (defensive**  
 147 **programming)**

148 **БЕРЕЖЕНОГО БОГ БЕРЕЖЕТ**

149 контроль *всюду, постоянно*, не отключать!

150 —> вставлять все возможные проверки (ASSERT)

151 —> "вырубать" программу как можно раньше (пока ошибка не "просочится"

152 дальше)

153 —> готовиться к худшему (переполнения, некорректные входные данные и т.п.)

154 *Будете много программировать — вспомните не раз.*

155 *Паранойя относительно возможных ошибок — необходимый признак зрелого*  
 156 *программиста.*

157 —> *Зрелый программист избегает оптимизаций.*

158 —> *Зрелый программист как огня боится C/C++ и проч. языков, где каждая*  
 159 *опечатка может обернуться ненаходимой ошибкой.*

160 **Из математических основ**

161 **замечание** как изучать математику

162 самое главное в любой мат. теории — в первой четверти монографии/главы

163 базовые понятия, связи между ними в базовых леммах

164 базовые примеры — **важно** научиться узнавать "паттерны"

165 дальше идут "методика изучения и исследование различий между ножками

166 кузнечиков" -- как правило, с целью увековечить вклад конкретного автора

167 Удачная книжка (но без автоматов [почему]):

168 *Ф.А.Новиков. Дискретная математика для программистов. 2-е изд.*

169 *Питер, 2005.*

170 Обычно начинают с 2-чной системы счисления -- очередной пример мифа.

171 Характерное, но **не главное.**

172 **Формальная логика**

173 Науку логику придумал великий Аристотель (с гордостью заявив, что не нашел

174 ничего, написанного до него; в основном, правила построения и

175 комбинирования суждений на естественных языках — "силлогизмы",

176 "эпихеремы" и т.п.; фактически смешивал то, что мы назовем логические и

177 внелогические предикаты).

178 *Aristotle (384-322 BC): one of the greatest of the ancient Athenian philosophers;*

179 *pupil of Plato; teacher of Alexander the Great. Один из величайших*

180 *интеллектуалов в истории человечества.*

181 Алгебру логики придумал Буль — *George Boole (1815-1864): English*  
 182 *mathematician.*

193 Логику и императивное программирование научно увязал **Эдсгер Дейкстра**  
 194 (Edsger Dijkstra, 1930-2002, Turing Award 1972), один из легендарных пионеров  
 195 computer science (CS), автор концепта "**структурное программирование**" и  
 196 ряда конкретных находок (семафоры; некот. алгоритмы).  
 197 Физик-теоретик по образованию.

198 Основополагающая книга Э.Дейкстра "Дисциплина программирования" (1976),  
 199 см. также "разжеванный" вариант -- Д.Грис "Наука программирования"  
 200 (перевод 1984).

201  
 202 Два значения TRUE, FALSE или сокращенно: Т, F (часто 1, 0 -- но к числам  
 203 отношение совершенно условное).

204 Кроме того, в контексте императивного программирования следует допустить  
 205 значение ? = "неопределено" (соответствует ситуациям типа когда вычисление  
 206 вызовет облом из-за обращения за границу массива [Дейкстра]).

207

## 208 Операции с булевскими операндами

209 три операции  $\neg \wedge \vee$  — будем обозначать как в КП:  $\sim$ ,  $\&$ , OR

210

211 Операция  $\sim$  (отрицание) — **унарная**, т.е. применяется к единственному  
 212 операнду справа от себя.

213 Операции  $\&$  и OR — **бинарные**, т.е. применяются к двум операндам, слева и  
 214 справа от себя.

215

216 если без значения ?, то

217  $\sim T = F$ ,  $\sim F = T$

218 откуда следует, что  $\sim(\sim a) = a$

219  $\&$  и OR

220 **коммутативны:**

221  $a \& b = b \& a$ ;  $a \text{ OR } b = b \text{ OR } a$

222 **ассоциативны:**

223  $(a \& b) \& c = a \& (b \& c)$

224 **идемпотентны** (idem = сам, potentia = сила; степень):

225  $a \& a = a$ ,  $a \text{ OR } a = a$

226  $F \& \dots = F$ ,  $T \text{ OR } \dots = T$

227 Тривиальные тождества:

228  $a \& \sim a = F$ ,  $a \text{ OR } \sim a = T$ .

229

230 (Чисто математич. вопросы: что взять за определения/аксиомы, что считать  
 231 следствиями.)

232

233 Для значения ?:

234  $F \& ? = F$ ,  $T \text{ OR } ? = T$

235 в остальных случаях результат равен ? (неопределен)

236

## 237 Таблицы истинности:

238 Способ проверки тождеств прямым перебором всех возможных комбинаций  
 239 значений переменных, входящих в выражения. Например, для импликации по  
 240 определению:

$a \quad b \quad \& \quad \text{OR} \quad \Rightarrow$

$T \quad T \quad T \quad T \quad T$

241  $T \quad F \quad F \quad T \quad F$

$F \quad T \quad F \quad T \quad T$

$F \quad F \quad F \quad F \quad T$

242 Откуда сразу получаем, что  $(a \Rightarrow b) = (\sim a) \text{ OR } b$ .

243 Также проверяется **дистрибутивность**  $\&$  и OR относительно друг друга (как \*  
 244 относительно +).

245 **Упр** Выписать соотношения дистрибутивности в явном виде.

246 **Упр** Составить таблицу истинности для бинарных операций  $a = b$  и  $a \# b$ .

247 Операцию  $a \# b$  еще называют "исключающее ИЛИ" = "exclusive OR" = **XOR**.

248

## 249 Логические выражения:

250 Составляются из (внелогических) переменных, константа Т и F, операций и  
 251 скобок, отражающих последовательность выполнения.

252

253 **Приоритеты логич. операций.** Много скобок в выражениях -- утомляет.

254 Разрешается их опускать.

255 Во-первых, коммутативность и ассоциативность позволяют опускать часть  
 256 скобок:

257 например, выражения  $((a \& b) \& c) \& d$  и  $a \& (b \& (c \& d))$  эквивалентны,  
 258 и оба можно представить как  $a \& b \& c \& d$ .

259

260 Как насчет выражений с разными операциями?

261 Из арифметики знаем о соглашении насчет порядка вычислений в выражениях  
 262 типа  $a + bc$ : здесь произведение  $bc$  вычисляется раньше, чем сложение —  
 263 говорим, что у умножения **выше приоритет**.

264

265 Аналогичное правило удобно принять для логических операций.

266

## 267 Общее правило в О/КП:

268 Если в выражении не хватает скобок для определения порядка вычисления  
 269 отдельных операций, то применяются два правила:

270 а) если операнд связывается с двумя операциями разного приоритета, то  
 271 раньше выполняется операция более высокого приоритета;

272 б) в остальных операциях вычисляются слева направо.

273 Пример (новая пара скобок = очередная операция):

274  $a \text{ OR } \sim b \text{ OR } c \& d \text{ OR } e$

275  $a \text{ OR } (\sim b) \text{ OR } c \& d \text{ OR } e$

276  $(a \text{ OR } (\sim b)) \text{ OR } c \& d \text{ OR } e$

277  $(a \text{ OR } (\sim b)) \text{ OR } (c \& d) \text{ OR } e$

278  $((a \text{ OR } (\sim b)) \text{ OR } (c \& d)) \text{ OR } e$

279  $(( (a \text{ OR } (\sim b)) \text{ OR } (c \& d)) \text{ OR } e)$

280

281 **NB** Жесткая фиксация порядка слева направо очень важна: любая оптимизация  
 282 машинных кодов, которая будет менять этот порядок, будет нарушать  
 283 определение языка. Это важно по кр. мере в двух случаях:

284 для корректной реализации правил укороченных (сокращенных)

285 вычислений логических выражений (см. ниже);

286 для гарантии результатов приближенных вычислений (ошибки округления).

287

288 Для логических операций в КП принято такое правило:  
 289 приоритет операций  $\sim$  & OR убывает.  
 290 Т.е., например, & связывает сильнее, чем OR.  
 291 **NB** Правило приоритетов отражается визуально: OR более "рыхлая" операция,  
 292 чем &.  
 293 Такое правило связано с тем, что на практике чаще всего встречается  
 294 перечисление комбинаций условий, при которых имеет место нечто:  
 295 ЕСЛИ то и то и то, ИЛИ то и то, но не то, ТОГДА ...  
 296 Это соответствует первой канонической форме логических выражений, см. ниже.  
 297 Сие обстоятельство проясняется с т.зр. теории множеств (см. теорему об  
 298 эквивалентности алгебры логики алгебре множеств).  
 299  
 300 **NB** В случае любых сомнений ставьте скобки — *береженого бог бережет*.  
 301  
 302 **Пример из числа наиболее важных:**  
 303  $(i < n) \& \sim(a[i] < 0)$   
 304 Два внелогических предиката связаны парой операция  $\sim$ , &.  
 305 Причем вычисление объекта  $a[i]$  (i-й элемент массива) может давать облом  
 306 при  $i \geq n$  (если последний элемент массива имеет индекс  $n - 1$ ) -- пример  
 307 значения ?  
 308  
 309 **NB** Выражения с одним или двумя операндами, связанными посредством & или  
 310 OR, встречаются очень часто. С тремя и более -- гораздо реже.  
 311  
 312 **Два самых важных соотношения (законы [де] Мо'ргана):**  
 313  $\sim(a \& b) = (\sim a) \text{ OR } (\sim b)$  и  $\sim(a \text{ OR } b) = (\sim a) \& (\sim b)$   
 314 Встречаются все время!  
 315 Растространяются на любое кол-во "сомножителей", например:  
 316  $\sim(a \& b \& c) = (\sim a) \text{ OR } (\sim b) \text{ OR } (\sim c)$   
 317  
 318 **Упр** Найти аналогичное соотношение для XOR.  
 319  
 320 **Каноническая форма** логического выражения — прямое следствие таблиц  
 321 истинности:  
 322 Пусть дано произвольное выражение с переменными  $a, b \dots c$ .  
 323 Тогда его можно выразить в виде некой последовательности связанных  
 324 операций OR операндов вида:  
 325  $(!a \& !b \& \dots \& !c)$ , где !a есть a или  $\sim a$ .  
 326 Построить легко: каждый операнд соответствует строке в таблице истинности, у  
 327 которой значение выражения равно Т, причем если в этой строке для некот.  
 328 переменной стоит Т, то соотв. переменная входит в фактор без отрицания, а  
 329 если F — то с оным.  
 330  
 331 **Упр** Построить канонические формы для  $a \Rightarrow b$ ,  $a = b$ ,  $a \text{ XOR } b$ .  
 332 Убедиться, что полученная форма для  $a \Rightarrow b$  эквивалентна формуле  $(\sim a) \text{ OR } b$ .  
 333  
 334 **Вторая каноническая форма** — & и OR меняются местами.  
 335 **Упр** Использовать законы де Моргана для доказательства возможности второй  
 336 канонической формы, если уже доказано насчет первой.  
 337 **Упр** Найти правило построения второй канонической формы по таблице  
 338 истинности.  
 339

340 **Упр** Построить вторую каноническую форму по таблице истинности для  $a \Rightarrow b$ ,  
 341  $a = b$ ,  $a \text{ XOR } b$ .  
 342  
 343 *Всего два значения, а сколько премудрости...*  
 344  
 345 **Кванторы**  
 346 "для любого"  
 347  $\forall i: F_i = F_0 \& F_1 \& \dots$   
 348 где  $F_i$  — некоторая формула, зависящая от  $i$ .  
 349 Подразумевается, что  $i$  пробегает известный диапазон.  
 350  
 351 "существует хотя бы один такой, что"  
 352  $\exists i: a_i = F_0 \text{ OR } F_1 \text{ OR } \dots$   
 353 Немедленно получаем **важное правило**:  
 354  $\sim(\forall i: F_i) = \exists i: (\sim F_i)$   
 355  
 356 **NB** Выражения с кванторами всегда подразумевают контекст — допустимое  
 357 множество "кванторизуемой" переменной. Например,  $0 \leq i < N$ . Может быть  
 358 указан явно, например, так:  
 359  $\forall i: (0 \leq i < N) \& F_i$   
 360  
 361 **Упр** Записать в виде логических выражений след. утверждения для числовых  
 362 последовательностей:  
 363 — Все элементы последовательности  $a_i$  неотрицательны.  
 364 — Последовательность  $a_i$  упорядочена по неубыванию.  
 365 — В последовательности  $a_i$  нет повторяющихся значений.  
 366 — Последовательность  $a_i$  мажорируется по абсолютной величине  
 367 последовательностью  $b_i$ .  
 368 Рассмотреть два варианта: последовательности бесконечны в обе стороны;  
 369 последовательности конечны (индекс пробегает значения от 0 до N).  
 370  
 371 **Формальная логика и жизнь: внелогические предикаты**  
 372  
 373 "Внелогические предикаты" = выражения/утверждения, имеющие булевские  
 374 значения.  
 375 Примеры:  
 376 "Сегодня ясная погода"  
 377 "Ваши знания недостаточны для получения зачета"  
 378 "Книги рассортированы в алфавитном порядке по авторам"  
 379 " $x < y$ "  
 380  
 381 **Упр** В плоскости  $a, b$  нарисовать области, в которых истинны/ложны след.  
 382 утверждения о вещественных корнях уравнения  $ax^2 + x + b = 0$ :  
 383 "ур-е имеет 2 разных корня"  
 384 "ур-е имеет 1 корень"  
 385 "ур-е не имеет корней"  
 386  
 387 **Упр** Является ли утверждение " $1 < 0$ " правильным логическим утверждением?  
 388 **NB** Различать "правильность" (синтаксическую корректность) и "истинность".  
 389

390 **Упр** Разобраться с булевским значением предиката "завтра будет хорошая  
391 погода".

392  
393 **Предикат** — это просто логическое выражение, возможно, с кванторами.  
394

395 **Внелогический предикат** — выражение из какой-нибудь другой оперы, но с  
396 логическим (BOOLEAN) значением.

397 **NB** Преобразования типа  $\sim(a < b) \Leftrightarrow (a \geq b)$

398 не относятся к сфере логики. Изучайте соответствующее либретто.  
399  
400

## 401 Элементарный (базовый) тип BOOLEAN

402  
403 Размер ячейки памяти под переменную этого типа — 1 байт.  
404 VAR a, b: BOOLEAN;

405 b, F5 --> BOOLEAN;

406 Всего два возможных значения — TRUE, FALSE.

407 Это предопределенные в языке константы.

408 Если VAR b: BOOLEAN; то можно написать присваивания  $b := \text{TRUE}$  или  $b :=$   
409 FALSE,

410  
411 **Пример** Переменная In.Done имеет тип BOOLEAN и принимает соотв. значение  
412 после каждого обращения к процедурам модуля In.  
413

414 Печать в Log: StdLog.Bool( *любое выражение с результатом типа BOOLEAN* ).  
415

416 Простейшие выражения, дающие результат типа BOOLEAN:

417  $x > 0$  или  $x \geq 0$  или  $x < 0$  или  $x \neq 0$  или  $x \# y$  (например, для VAR  
418 x, y: REAL).  
419

420 Две операции сравнения для значений BOOLEAN: =, # (XOR).

421 Три операции: ~, &, OR.

422 Приоритеты:  $\sim a$  OR  $\sim b$  &  $\sim c$  —  $(\sim a)$  OR  $(\sim b)$  &  $(\sim c)$  ).  
423

424 **NB О скобках.** Мы договорились о том, когда можно не писать скобок в чисто  
425 логических выражениях.

426 Как насчет выражений вида  $(a < b) \& (a < c)$  ?

427 Чтобы не писать скобки, нужно договориться о приоритетах.

428 Никакой естественной иерархии операций сравнения и логических нет. Можно  
429 ли сравнивать (<,>) логические значения? В этом нет никакого объективного  
430 смысла, только случайное правило, значит, зубрить, а поэтому возможны  
431 ошибки.

432 Потенциальное минное поле.  
433

434 **Кстати** Правила насчет приоритетов разнородных операций — типичный  
435 пример **мусорных знаний**. Если работаете на C, то придется зубрить. Но это  
436 не имеет **объективного** смысла.

437 (**NB** Объективность объективности рознь.)  
438

439 **Правило** Во избежание коварных ошибок, в О/КП арифметические неравенства  
440 должно заключать в скобки.

441 Смешивать без скобок сравнения и ~, &, OR — **низзя!** Сравнения нужно  
442 обязательно заключать в скобки.

443 **Упр** Проверить, как реагирует компилятор на выражение  $a < b \& a < c$ .

444 **Q** Как будете проверять?  
445

446 Другие операции с результатом BOOLEAN: любые сравнения, допустимые для  
447 других типов.

448 Например, для VAR a, b: INTEGER; c, d: BOOLEAN;

449 возможно  $c := a < b$ ; и  $c := (a < b) \& d$ ;  
450

451 **Упр** Запишите на КП предикаты (все переменные имеют тип INTEGER):

452 — Значения переменных a, b, c положительны.

453 — Число a положительно, но не кратно 3.

454 — Число n делится нацело на 11 и на 7, но не на 3.

455 — Точка (a, b) лежит в прямоугольнике, стороны которого параллельны  
456 координатным осям, растянутом вершинами (x0, y0), (x1, y1).

457 — Точка (a, b) лежит на прямой, заданной двумя точками (x0, y0), (x1, y1).

458 — Две прямые, заданные парами точек (x0, y0), (x1, y1) и (a0, b0), (a1, b1),  
459 параллельны.

460 — Точка (a, b) лежит на прямой, заданной точкой (x, y) и тангенсом наклона (n,  
461 m).  
462

463 **NB** В О/КП **нельзя** использовать логические значения в арифметических  
464 выражениях, и обратно  
465 (*ЗДЕСЬ ВАМ НЕ ЦЫ!*)  
466

## 467 Сокращенное вычисление логических выражений

468 short-circuited, lazy, McCarthy evaluation  
469

470 Это имеет отношение к следующим двум ситуациям:

471  $F \& (\text{что угодно}) = F$  и  $T \text{ OR } (\text{что угодно}) = T$ .  
472

473 Ясно, что в этих случаях второй операнд можно не вычислять вообще.  
474

475 **John McCarthy** — автор второго после фортрана ЯП LISP (1958),  
476 являвшегося также первым функциональным языком. Он придумал это  
477 правило для оптимизации (чтобы ускорить счет, whence "lazy"), что во  
478 всех ФЯ весьма актуально.

479 **Дейкстра** открыл, что именно это правило позволяет наиболее  
480 естественно записывать циклы, и что именно его нужно иметь в виду при  
481 выводе императивных программ из логических условий решаемой задачи.  
482 Строго говоря, то, что у Дейкстры "вывелось" то же правило, что и у  
483 Маккарти, есть в известном смысле случайность — ведь их цели были  
484 совершенно разные.

485 **NB** О (не)справедливости математической номенклатуры.  
486

487 "При вычислении бинарной логической операции сначала вычисляется ее  
488 первый (левый) операнд. Если результат определяется без вычисления второго  
489 (правого) операнда, то второй операнд и не вычисляется."  
490

491 Фишка в том, что:

492 операнды могут быть вне-логическими предикатами;

493 второй операнд вообще может быть неопределен (в т.ч. вычисление может  
494 вызывать TRAP).

```

495
496 Фундаментальный пример.
497 Поиск первого отрицательного числа во входном потоке:

498   VAR x: REAL;
499   ...
500   In.Open;
501   In.Real( x );
502   WHILE In.Done & ~( x < 0 ) DO
503     (* некое вычисление *)
504     In.Real( x )
505   END;
506   (* в этом месте ~In.Done OR ( x < 0 ) *)
507   (* обработка ситуации после цикла: *)
508   IF ~In.Done THEN
509     (* попытка чтения не удалась *)
510     (* поэтому текущее значение x представляет собой мусор
511     -- именно поэтому IF работает с первым операндом *)
512     (* т.е. закончили чтение раньше, чем нашли отрицат. число *)
513     StdLog.String('во входом потоке нет отрицат. чисел')
514   ELSE
515     (* x — последнее успешно прочитанное число, причем: *)
516     ASSERT( x < 0 )
517   END;
518
519 Этот пример так важен, что мы к нему еще вернемся.
520 Упр Продумать как следует, что происходит.
521 Можно ли переставить операнды в охране цикла?
522 Можно ли переставить ветви IF, т.е. написать IF In.Done и т.д.?

523 NB Логические выражения, определенные таким образом, уже
524 не удовлетворяют всем тождествам "чистой логики".
525 Разрушается и коммутативность, и ассоциативность.
526 Упр Как насчет дистрибутивности?
527 Каждый раз нужно внимательно проверять!
528 К счастью, самые главные соотношения справедливы:

529   ~( a & b ) = (~a) OR (~b) и ~( a OR b ) = (~a) & (~b)

530   ~( a & b & c ) = (~a) OR (~b) OR (~c)

531 Но нельзя менять порядок операндов!!!
532

```

```

533 Псевдо-процедура ASSERT (=удостовериться, что)
534 Например, пусть VAR x: INTEGER;
535   ASSERT( x > 0 );
536 В качестве параметра может стоять любое выражение, имеющее результат типа
537 BOOLEAN.
538 ASSERT вычисляет логическое значение:
539 если видит значение TRUE, то выполнение программы продолжается как ни в
540 чем ни бывало;
541 иначе выполнение программы прерывается (происходит TRAP или облом или
542 прерывание или авост = аварийная остановка — и ББ "выбрасывает кишки").
543
544 Можно использовать второй параметр — константу (0, 1 ... 127).
545   ASSERT( x > 0, 100 );
546 Просто помогает определить причину облома.
547 В документации к библиотечным процедурам следует указывать, какая ошибка
548 соответствует какому номеру трапа.
549
550 Золотое правило программной санитарии: авост при
551 любом подозрении на ошибку.
552 По той же причине компилятор никаких "предупреждений" не делает —
553 сразу вырубает.
554 Береженого бог бережет.
555
556 Правило Вставлять побольше ASSERT'ов с проверками.
557
558 Precondition = предусловие: Условие, которое выполняется на момент
559 начала выполнения процедуры.
560 Postcondition = постусловие: Условие, которое выполняется на момент
561 завершения процедуры.
562 Invariant = инвариант: Условия, выполнение которых гарантируется на
563 промежуточных шагах. В частности, в дальнейшем познакомимся с
564 инвариантами цикла.
565
566 MODULE Kurs2006Пример10; (* ввод данных с контролем *)
567   IMPORT Log := StdLog, In := FVTsysIn, Math;
568
569   PROCEDURE Do*;
570     VAR x, y, zero: REAL;
571   BEGIN
572     Log.String('проверяем триг. тождество: ');
573     In.Open;   ASSERT( In.Done, 20 );
574     In.Real( x );   ASSERT( In.Done, 21 );
575     In.Real( y );   ASSERT( In.Done, 22 );
576
577     zero := Math.Cos( x ) * Math.Cos( y ) - Math.Sin( x ) * Math.Sin( y );
578     zero := zero - Math.Cos( x + y );
579
580     ASSERT( ABS( zero ) < 1.0E-10, 60 );
581     Log.Real( zero ); Log.Ln;
582   END Do;
583 END Kurs2006Пример10.

```

584 !Kurs2006Пример10.Do 0.1234 5.6789 !  
 585 Все работает:  
 586 "Kurs2006Пример10" 256 0  
 587 проверяем триг. тождество: -9.324138683375338E-17  
 588  
 589 Но если вставить запятую или командер после первого числа,  
 590 !Kurs2006Пример10.Do 0.1234, 5.6789 !  
 591 то случится TRAP — откроется окошко сл. вида (показано только начало):  
 592  
 593 (Dev, Edit Mode; select lines; Ctrl+C; go to insertion position; Edit, Paste Special [Picture  
 594 (Metafile)])

#### TRAP 22 (precondition violated)

```

◆ Kurs2006Пример10.Do [0000062H] ◆
.x          REAL          0.1234
.y          REAL          5.255097112376005E-
308
.zero       REAL          1.124163204946969E-
303
◆ StdInterpreter.CallProc [000003A5H] ◆
.a          BOOLEAN       FALSE
.b          BOOLEAN       FALSE
.c          BOOLEAN       FALSE
.i          Meta.Item      →fields←
.imported   ARRAY 256 OF CHAR "" →...←
.importing  ARRAY 256 OF CHAR "" →...←
.mn         Meta.Name      "Kurs2006Пример10"
.mod        StdInterpreter.Ident "Kurs2006Пример10"
.object     ARRAY 256 OF CHAR "" →...←
.ok         BOOLEAN       TRUE
.parType    INTEGER        0
.pn         Meta.Name      "Do"
.proc       StdInterpreter.Ident "Do" →...←

```

595  
 596  
 597 Это текстовый документ ББ, в котором синие ромбики и черные стрелки —  
 598 активные объекты: кликая по ним получают интересные вещи.

599  
 600 Первая строка показывает номер прерывания и (по соглашениям ББ) дает  
 601 некий комментарий.

602 Вот выписка из документации (F1, Programming Conventions):

603 *Preconditions* are one of the most useful tools to detect unaccounted **ripple**  
 604 **effects**. Precondition checks allow to pinpoint semantic errors as early as  
 605 possible, i.e. as closely to their true source as possible. After larger design  
 606 changes, properly used assertions can help to dramatically reduce debugging  
 607 time ...

608 • Precondition assertions should be used consequently. Don't allow client code to  
 609 "enter" your module <в нашем случае речь идет о данных, попадающих в наш  
 610 модуль> if it doesn't fulfill the preconditions of your module's entry points  
 611 (procedures, methods). In this way, you avoid propagation of foreign errors into  
 612 your own code.

613

614 Добавим, что эти ASSERT играют роль комментариев, но в отличие от последних  
 615 жестко привязаны к текущему состоянию выполняемой программы.

616  
 617 **ASSERT = один из важнейший типов комментариев.**

618  
 619 Окошко показывает состояние всех процедур в цепочке вызовов:  
 620 авост произошел в процедуре Kurs2006Пример10.Do, которая была вызвана из  
 621 процедуры StdInterpreter.CallProc, которая ... и т.д.

622  
 623 Для каждой процедуры — строчка с ромбиками.  
 624 После строчки с ромбиками — список всех локальных переменных процедуры с  
 625 их значениями.  
 626 Видим, что x прочитался нормально, а значение y — не то, что мы ожидали бы  
 627 при продолжении работы.

628  
 629 Где именно произошло прерывание, покажет **ромбик справа**: если по нему  
 630 кликнуть, открывается исходник модуля на том самом операторе, во время  
 631 выполнения которого произошло прерывание (оператор будет выделен). В  
 632 данном случае — ASSERT после попытки прочесть y.

633  
 634 **NB** Если исходник хранится не по правилам, то может открыться что-то не то  
 635 или вообще ничего.

636  
 637 **Ромбик слева** — открывает новое окошко, показывающее состояние в момент  
 638 прерывания глобальных переменных данного модуля (Kurs2005Пример6 в  
 639 первом случае, StdInterpreter во втором).

640  
 641 *Кстати, раз уж речь зашла о прерываниях: в тела еще не законченных*  
 642 *процедур полезно вставлять HALT(126) — эквивалент ASSERT( FALSE,*  
 643 *126 ).*

644  
 645 **NB** После прерывания все наши модули остаются в памяти, все их глобальные  
 646 переменные сохраняются, и можно продолжать работу.

647  
 648 **Замечание.** По поводу номеров:

649  
 650 Use the following numbers to be consistent with the rest of the BlackBox  
 651 Component Builder:

652	Free	0 .. 19	use for temporary breakpoints
653	Preconditions	20 .. 59	validate parameters at procedure
654	entry		
655	Postconditions	60 .. 99	validate results at procedure end
656	Invariants	100 .. 120	validate intermediate states
657			(detect local error)
658	Reserved	121 .. 125	reserved for future use
659	Not Yet Implemented	126	procedure is not yet implemented
660	Reserved	127	reserved for future use
661			
662			

663 **Не жалейте ASSERT'ов: Береженого бог бережет**

664

## 665 Императивное программирование по Дейкстре

666 Программа = преобразователь предикатов:  
 667 Предполагаемое начальное состояние и желаемое конечное состояние  
 668 описываются некоторыми предикатами — предусловием и постусловием.  
 669 Про каждую элементарную инструкцию знаем, как она преобразовывает  
 670 предикаты.

671 Цель — построить последовательность инструкций, обеспечивающих  
 672 необходимое преобразование предусловия в постусловие.

673  
 674 **НВ** Тем, кто говорит, мол, что императивное программирование (в отличие,  
 675 например, от функционального) не имеет настоящей математической основы,  
 676 можно сообщить об их элементарном невежестве.

677  
 678 **Практическое правило** Для каждой процедуры необходимо четко понимать  
 679 пред- и постусловия — *до начала программирования*.

680

## 681 Из арифметики

### 682 Двоичная система

683  $1101_2 = 13_{10}$

684 Разные "имена" для одного и того же числа.

685 **Упр** Выписать двоичное представление всех чисел от 0 до 15.

686 С помощью N двоичных цифр (бит) можно записать  $2^N$  разных чисел.

687 Сначала для простоты будем интерпретировать как неотрицательные целые.

688

689 Знать наизусть степени 2 до 10:

690  $2^0 = 1$

691  $2^1 = 2$

692  $2^2 = 4$

693  $2^3 = 8$

694  $2^4 = 16$

695  $2^5 = 32$

696  $2^6 = 64$

697  $2^7 = 128$

698  $2^8 = 256$

699  $2^9 = 512$

700  $2^{10} = 1024$

701

702  $2^{10} = 1K \sim 10^3$

703  $2^{20} = 1M \sim 10^6$

704  $2^{30} = 1G \sim 10^9$

705  $2^{16} = 64K, 2^{15} = 32K$

706  $2^{32} = 4G, 2^{31} = 2G$

707

708 бит, байт, килобайт, мегабайт, гигабайт, терабайт ( $\sim 10^{12}$ ), петабайт ( $\sim 10^{15}$ )

709 (также выдумка стандартизаторов: кибибайт, мебибайт, GiB ...)

710

## 711 Шестнадцатеричная система

712 *Когда-то была и восьмиричная.*

713 Цифры: 0, 1, .. 9, A, B, .. F

714 Одна 16-ричная цифра = 4 бита.

715 Важно помнить:  $0_{16} = 0000_2, F_{16} = 1111_2$

716 Содержимое байта: две 16-ричные цифры (напр., FF).

717 Содержимое слова: восемь 16-ричных цифр (напр., FF 76 00 91).

718

718 **"Дополнительный код"** для представления отрицат. чисел

719 **НВ** Первый пример нетривиального кодирования некоторого множества  
 720 значений с помощью набора бит.

721 Пусть речь идет о кодировании 3-битных чисел:

722 0, 1, 2, **3, 4**, 5, 6, **7** отображается на 0, 1, 2, **3, -4**, -3, -2, **-1**

723 000 0

724 001 1

725 010 2

726 011 3

727 100 -4

728 101 -3 = -4 + 1

729 110 -2

730 111 -1

731 Аналогично при большем числе битов.

732 **Легко** определять знак: "старший бит" (крайне левый) = 1 —> отрицат. число

733 **НВ** Отрицательных чисел всегда на одно больше, чем положительных.

734 **Упр** Обдумайте, как производится сложение чисел разных знаков.

735 **История** Когда-то бывало, что отрицат. числа представляли по-другому.

736

736 **КП** Спец. константы  $MAX(INTEGER) = 2^{31} - 1$ ,  $MIN(INTEGER) = -2^{31}$ .

737 В О/КП 16-ричную нотацию используют **только** для побитной записи

738 содержимого слов (в дополнительном коде).

739 **0DH** INTEGER 13

740 **0FFFF000H** INTEGER -65536

741 Нуль в начале — чтобы не путать с идентификаторами.

742 Н в конце — чтобы указать на 16-ричность. Ср.:

743 010 INTEGER 10

744 010H INTEGER 16

745 **НВ**

746 **0FFFF000L** LONGINT 4294901760 (64-битное положит > MAX(INTEGER))

747 **НВ** Если встретились 16-ричные цифры, а в конце нет Н или L — то это ошибка.

748 **НВ** Порядок цифр здесь **не связан** с порядком расположения соотв. байтов в

749 памяти (ср. big endian, little endian).

750 **НВ** Нужно было некоторое проектировочное усилие + опыт, чтобы ограничиться

751 этими и именно этими правилами (в других языках "от балды").

752 **Упр** Написать процедуры, в которых используются присваивания 16-ричных

753 констант переменным INTEGER и в Log печатаются соотв. числовые значения с

754 помощью StdLog.Int(...).

755 **НВ** Знать hex-арифметику — для нас совершенно лишнее (для редкого случая

756 есть калькуляторы).

757 **НВ** Возможно задание литер 16-ричными числами — **но** КП тогда требует в

758 конце ставить X вместо H (см. ниже).