

ИНФОРМАТИКА-21

МЕЖДУНАРОДНЫЙ ОБЩЕСТВЕННЫЙ НАУЧНО-ОБРАЗОВАТЕЛЬНЫЙ ПРОЕКТ

<http://www.inr.ac.ru/~info21>

Введение в систему Блэкбокс и Компонентный Паскаль

Составление по материалам проекта «Информатика-21»: Н. Д. Прыгунов, 2003
Редакция 2009-11-16

ОГЛАВЛЕНИЕ

[Краткая история Паскаля](#)

Алгол
Паскаль
Модуля-2
Simula, Smalltalk и Cedar
Оберон [Oberon]
Компонентный Паскаль
BlackBox

[О дисциплине программирования](#)

История изгнания оператора GOTO
О структурном программировании и отсутствии GOTO
О проверках выхода индексов за границы массивов
Почему порочно "наивное" программирование с пошаговым отладчиком

Система BlackBox

[Первичная установка системы Блэкбокс](#)

[Русификация системы Блэкбокс](#)

[Вторичные папки Блэкбокса](#)

Чтение файла
Запись файла
Ограничения

[Запуск Блэкбокса и выполнение первой программы](#)

Простейшая программа
Выполнение программы с помощью командира
Сохранение файла с программой
Продолжение работы с примером
Простейший цикл разработки программы
 Подготовительная стадия
 Основной цикл разработки
Некоторые заповеди грамотного программирования

[Формат документов Блэкбокса](#)

Импорт/экспорт стандартных текстовых файлов
Вставные визуальные объекты (views)

[Использование рабочего журнала](#)

[Средства, предоставляемые модулем StdLog](#)

Печать основных типов
Переход на новую строку и табуляция
Печать литерных цепочек

Документация в Блэкбоксе

Документация на русском языке
Справки по интерфейсам модулей
Документация по конкретному модулю
Общая документация и поиск в ней
Примеры программ

Работа с текстами программ

Форматирование текста
Отступы
Соглашения по оформлению программ
Шрифт по умолчанию
Использование цвета и т.п.
Копирование атрибутов текста
Ключевые слова из прописных букв
Организация файлов Блэкбокса
Соглашения об именах модулей и именах соответствующих файлов в Блэкбоксе

Компилирование и отладка

Сообщения компилятора об ошибках
Загрузка и перезагрузка модулей
Минимальные сведения
Когда происходит загрузка модуля в оперативную память
Что происходит при загрузке модуля
Как выгрузить старую и загрузить новую версию разрабатываемого модуля
Как проверить, какие модули загружены
Аварийная остановка программы
Основные ситуации, приводящие к аварийной остановке
Пример аварийной остановки

Примеры программ на Компонентном Паскале

Функция, проверяющая простоту задаваемого целого
Уплотнение цепочки литер, чтобы исключить идущие подряд пробелы
Сортировка вставками

Сообщение о языке Компонентный Паскаль

Отличия Компонентного Паскаля от Паскаля

К р а т к а я и с т о р и я П а с к а л я

© Оригинальный текст представляет собой часть документации системы BlackBox Component Builder v.1.4 компании [Oberon microsystems](#).

© Перевод на русский язык: [Ф.В.Ткачëв](#), 2001. Замечания переводчика даны в угловых скобках <>. Некоторые термины оригинала приведены в квадратных скобках [].

Алгол

Язык Компонентный Паскаль является кульминацией нескольких десятилетий исследовательской работы. Это самый младший член семейства алголоподобных языков. Алгол, определенный в 1960, был первым языком высокого уровня с синтаксисом, который был легко читаем, четко структурирован и описан формальным образом. Несмотря на его успешное использование в качестве нотации для математических алгоритмов, в нем недоставало важных типов данных, таких как указатели и литеры.

Паскаль

В конце 60-х гг. было выдвинуто несколько предложений об эволюционном развитии Алгола. Самым успешным оказался Паскаль, определенный в 1970 г. профессором Никлаусом Виртом из ЕТН, швейцарского Федерального Технологического Института в Цюрихе [Eidgenössische Technische Hochschule]. Наряду с очищением языка от некоторых непрозрачных средств Алгола, в Паскале была добавлена возможность объявления новых структур данных, построенных из уже существующих более простых. Паскаль также поддерживал динамические структуры данных, т.е. такие, которые могут расти или уменьшаться во время выполнения программы. Паскаль получил сильный импульс к распространению, когда в ЕТН был выпущен компилятор, порождавший простой промежуточный код для виртуальной машины (Р-код) вместо кода для конкретного процессора. Это существенно упростило перенос Паскаля на другие процессорные архитектуры, т.к. для этого нужно было только написать новый интерпретатор для Р-кода вместо всего нового компилятора. Один из таких проектов был предпринят в Университете Калифорнии в Сан-Диего. Замечательно, что эта реализация (UCSD Pascal) не требовала большого компьютера [mainframe] и могла работать на новых тогда персональных компьютерах Apple II. Это дало распространению Паскаля второй важный импульс. Третьим был выпуск компанией Borland продукта ТурбоПаскаль, содержавшего быстрый и недорогой компилятор вместе с интегрированной средой разработки программ для компьютеров IBM PC. Позднее Борланд возродил свою версию Паскаля, выпустив среду быстрой разработки приложений Дельфи.

Паскаль сильно повлиял на дизайн и эволюцию многих других языков, от Ады до Visual Basic.

Модула-2

В середине 70-х гг., вдохновленный годичным академическим отпуском, проведенным в исследовательском центре PARC компании Xerox в Пало Альто, Вирт начал проект по созданию нового компьютера класса рабочая станция <проект Lilith; см. [Краткая история Модулы и Лилит \(на англ.\)](#) — прим. перев.>. Компьютер должен

был полностью программироваться на языке высокого уровня, так что язык должен был обеспечить прямой доступ к аппаратному уровню. Далее, он должен был поддерживать коллективное программирование и современные принципы разработки программного обеспечения, такие как абстрактные типы данных. Эти требования были реализованы в языке программирования Модуля-2 (1979).

Модуля-2 сохранила хорошо зарекомендовавшие себя средства Паскаля и добавила систему модулей, а также контролируемые возможности обойти систему типов языка для целей программирования низкого уровня (например, при написании драйверов). Модули могли добавляться к операционной системе непосредственно во время работы. На самом деле вся операционная система представляла собой набор модулей без выделенного ядра [kernel] или подобного объекта. Модули могли компилироваться и загружаться раздельно, причем обеспечивалась полная проверка типов и версий их интерфейсов.

Успех Модуля-2 был наиболее значителен в задачах с высокими требованиями на надежность, таких как системы управления движением.

Simula, Smalltalk и Cedar

Однако Вирт продолжал интересоваться прежде всего настольными компьютерами, и опять важный импульс пришел из центра PARC компании Херох. В этом центре были изобретены рабочая станция, лазерный принтер, локальная сеть, графический дисплей и многие другие технологии, расширяющие возможности использования компьютеров человеком. Кроме того, в центре PARC были популяризированы некоторые более старые и малоизвестные технологии, такие как мышь, интерактивная графика и, наконец, объектно-ориентированное программирование. Эта последняя концепция (хотя и не сам термин) была впервые использована в языке высокого уровня Simula (1966) — еще одним из семейства алголоподобных языков. Как и предполагает имя, язык Simula использовал объектные технологии прежде всего для целей моделирования [simulation]. Однако язык Smalltalk (1983), разработанный в центре PARC компании Херох, использовал объектные технологии как универсальное средство. Проект Smalltalk был также пионерским в плане дизайна пользовательского интерфейса: графический пользовательский интерфейс, каким мы его теперь знаем, был разработан для системы Smalltalk.

В центре PARC эти идеи повлияли на другие проекты, например, паскалеподобный язык Cedar. Как и Smalltalk и позднее Оберон, Cedar представлял собой не только язык программирования, но и операционную систему. Операционная система Cedar была весьма впечатляющей и мощной, однако сложной и нестабильной.

Оберон [Oberon]

Проект Оберон был начат в 1985 в ЕТН Виртом и его коллегой Юргом Гуткнехтом [Jurg Gutknecht]. Это была попытка выделить все существенное из системы Cedar в виде универсальной, но все же обозримой операционной системы для рабочих станций. Получившаяся система оказалась очень маленькой и эффективной, прекрасно работала в оперативной памяти размером всего 2 МВ и требовала при этом лишь 10 МВ пространства на диске. Важной причиной малого размера системы Оберон был ее компонентный дизайн: вместо интеграции всех желаемых средств в один монолитный программный колосс, менее часто используемые программные компоненты (модули) могли быть реализованы как расширение ядра системы. Такие компоненты

загружались, только когда они были действительно нужны, и они могли совместно использоваться всеми приложениями.

Вирт понял, что компонентно-ориентированное программирование требовало некоторых средств объектно-ориентированного программирования, таких как упрятывание информации [information hiding], позднее связывание [late binding] и полиморфизм [polymorphism].

Упрятывание информации было сильной чертой Модулы-2. Позднее связывание поддерживалось в Модуле-2 посредством процедурных переменных. Однако там не было полиморфизма.

По этой причине Вирт добавил расширенное переопределение типов [type extension]: тип записей мог быть объявлен как расширение *<потомок>* другого типа записей *<предка>*. Тип-потомок можно было использовать всюду вместо его предков.

Но компонентно-ориентированное программирование выходит за рамки объектно-ориентированного. В системе, построенной из компонент, компонента может разделять свои структуры данных с произвольным числом других компонент, о которых она ничего не знает. Эти компоненты обычно также не знают о существовании друг друга. Такое взаимное незнание делает управление динамическими структурами данных, и в частности правильное освобождение уже ненужной памяти, принципиально более трудной проблемой, чем в закрытых программных системах. Следовательно, необходимо оставить на долю реализации языка всю работу по определению момента, когда какая-то область памяти более не нужна, чтобы повторно использовать ее без ущерба для безопасности системы. Системный сервис, выполняющий такую автоматическую утилизацию памяти, называется сборщик мусора [garbage collector]. Сбор мусора предотвращает две из числа наиболее труднонаходимых и попросту опасных ошибок в программах: утечки памяти [memory leaks] (когда более не используемая память не освобождается) и висячие ссылки [dangling pointers] (преждевременное освобождение памяти). Висячие ссылки позволяют одной компоненте разрушить структуры данных, принадлежащие другим. Такое нарушение защиты по типам [type safety] должно быть предотвращено, т.к. компонентные системы могут содержать много независимо написанных компонент неизвестного качества (например, полученных из Интернета).

Хотя алголоподобные языки всегда имели высокую репутацию в отношении безопасности, введение полной защиты типов (и, следовательно, сбора мусора) было квантовым прыжком. Именно по этой причине полная совместимость с Модулой-2 оказалась невозможной. Получившаяся модификация Модулы-2 была названа как и вся система — Оберон.

Система модулей в Обероне, как и в Модуле-2, обеспечивала упрятывание информации для целых семейств типов, а не только для отдельных объектов. Это позволило определять и гарантировать инварианты для нескольких взаимодействующих объектов. Другими словами, разработчики получили возможность разрабатывать механизмы защиты более высокого уровня, отталкиваясь от базовых средств защиты на уровне модулей [module safety] и защиты по типам, обеспечиваемых хорошей реализацией Оберона.

Такие ортодоксальные объектно-ориентированные языки, как Smalltalk, пренебрегали как типизацией переменных (там вообще нет понятия тип переменной), так и упрятыванием информации (ограничивая ее объектами и классами), что было большим шагом назад в технологии программирования. Оберон примирил миры объектно-ориентированного и модульного программирования.

Последнее требование компонентно-ориентированного программирования — возможность динамически загружать новые компоненты. В Обероне единица загрузки та же, что и единица компиляции — модуль.

Компонентный Паскаль

В 1992 г. сотрудничество с профессором Х.П. Мёссенбёком (H.P. Mossenbock) привело к нескольким добавлениям к первоначальному языку Оберон ("Оберон-2"). Так возник фактический стандарт языка.

В 1997 г. компания Oberon microsystems, Inc., отпочковавшаяся [spin-off] от ETH (с Виртом в составе совета директоров), сделала некоторые небольшие добавления к Оберону-2 и назвала его Компонентный Паскаль.

Данное название выбрано, чтобы четче выразить как его нацеленность (компонентно-ориентированное программирование), так и его происхождение (Паскаль). Это промышленная версия Оберона, являющаяся наследницей первоначального Паскаля и Модулы-2.

Главная идея уточнений по сравнению с Обероном-2 была в том, чтобы дать проектировщику компонентного каркаса [component framework] (т.е. интерфейсов модулей, определяющих абстрактные классы для конкретной проблемной области) более полный контроль над ее проектируемыми свойствами в плане безопасности. Положительным результатом стало то, что теперь легче обеспечить целостность больших компонентных систем, что особенно важно во время итеративных циклов проектирования, когда библиотека разрабатывается, и позднее, когда архитектура системы должна быть переработана, чтобы обеспечить дальнейшую эволюцию и поддержку.

BlackBox

Компания Oberon microsystems разрабатывала компонентную библиотеку BlackBox Component Framework начиная с 1992 г. (сначала библиотека называлась Oberon/F).

Эта библиотека написана на Компонентном Паскале и упрощает разработку компонент графического пользовательского интерфейса. Она поставляется с несколькими компонентами, включая текстовый редактор, систему визуального проектирования, средство доступа к базам данных SQL, интегрированную среду разработки, а также систему поддержки выполнения программ на Компонентном Паскале. Весь пакет представляет собой развитый, но весьма нетребовательный к системным ресурсам инструмент быстрой разработки компонентных приложений, названный BlackBox Component Builder. Он нетребователен к системным ресурсам, т.к. полностью построен из модулей Компонентного Паскаля — включая ядро со сборщиком мусора, а также самого компилятора для языка Компонентный Паскаль. Это — иллюстрация как мощи концепции компонентного программного обеспечения вообще, так и адекватности языка Компонентный Паскаль в частности.

Недавно диапазон приложений системы BlackBox Component Builder был значительно расширен за счет среды кросс-программирования *<т.е. программирование для процессоров, отличных от того, на котором работает система; обычно это специализированные процессоры для встроенных систем — прим. перев.>* Denia, которая является компонентой, расширяющей BlackBox.

Denia позволяет выполнять кросс-программирование на Компонентном Паскале для новой операционной системы реального времени JBed, которая тоже полностью реализована на Компонентном Паскале. JBed предназначен для встроенных систем и

приложений с жесткими требованиями реального времени [hard real-time requirements], например, в робототехнике и промышленной автоматизации. <О сложности этого класса приложений свидетельствует тот факт, что компания Майкрософт отказалась от продвижения своих операционных систем в этом сегменте рынка — прим. перев.>

О дисциплине программирования

(Почему в Обероне/Компонентном Паскале ограничена "свобода творчества" программиста.)

Особенность Оберона/Компонентного Паскаля — безусловный приоритет, отдаваемый безопасности программ, в том числе защите как пользователей, так и самого программиста от его же собственных случайных ошибок.

С этой целью из языка и системы программирования просто-напросто исключен ряд средств и опций, обычно имеющих в "профессиональных" системах. Прежде всего бросается в глаза:

- Отсутствие оператора безусловного перехода GOTO.
- Невозможность отключить проверки индексов на предмет выхода за границы массивов.
- Отсутствие пошагового отладчика.

Такая суровая дисциплина на первый взгляд кажется чрезмерной и ставит в тупик недостаточно подготовленного программиста. Но на самом деле упомянутые выше средства (как и некоторые другие из числа более новых; см., например, [Сообщение...](#)) — всего лишь тупики полувекового поиска наиболее эффективных технологий программирования.

Не всякое выдуманное за 50 лет средство полезно и эффективно, но понимание этого в каждом конкретном случае приходит только с опытом.

Как правило создатели новых языков и систем программирования не могут не продемонстрировать миру свою изощренность, включая в них новомодные прикрасы. А программисты не могут не доказать начальству и коллегам свое остроумие в немедленном и всевозможном использовании новинок.

Проблема в том, что все бывшие новинки — раз уж они были включены в языки и системы программирования — сохраняются в дальнейшем, даже если практика указывает на их порочность.

Сохраняются в силу необходимости обеспечивать совместимость с наследием прошлого:

- программным наследием — горами уже написанных программ, от которых оказывается нелегко избавиться;
- человеческим капиталом — заслуженными программистами, переучить которых также трудно, как и переписать старые программы.

Что касается GOTO и проч., то, с одной стороны, доказано, что отказ от упомянутых средств *не ограничивает* создаваемые программы ни в плане алгоритмических возможностей, ни в плане эффективности (скорее даже наоборот).

С другой стороны, такой отказ *гарантирует*, что получившаяся программа будет иметь ряд чрезвычайно желательных свойств — четкую структуру, верифицируемость и повышенную надежность. Ни один настоящий профессионал не позволит себе

пренебречь этими свойствами, даже если это потребует от него определенной интеллектуальной дисциплины.

Главная проблема в том, что надлежащая **дисциплина программирования** не может возникнуть сама по себе: она требует целенаправленного обучения.

(Попробуйте, например, освоить грамотный горнолыжный поворот — т.наз. угловинтовое движение: без квалифицированного тренера получится только поворот "броском зада". Разве грамотное программирование проще?)

Возвращаясь к списку средств, исключенных из Оберона/Компонентного Паскаля: более подробно каждое из них обсуждается отдельно. Там же содержатся и дальнейшие аргументы в пользу "сурового" дизайна средств программирования.

История изгнания оператора GOTO

С оператором GOTO к настоящему времени достигнута полная ясность: в языке программирования для серьезной работы оператора GOTO быть не должно. История его изгнания весьма поучительна.

Наличие GOTO в машинном языке понятно: там важна экономия средств, а с помощью одного этого оператора можно смоделировать любые структуры управления (операторы IF, FOR, WHILE и т.д.).

По той же причине в 50-е годы GOTO был автоматически включен в первые языки программирования (например, фортран): ведь еще не было ясно, ни что такое хорошие программы, ни как их строить.

К концу 60-х гг. анализ уже накопленного опыта выявил (см. знаменитое письмо выдающегося теоретика систематического программирования Э.Дейкстры ["Go To Statement Considered Harmful"](#)), что неконтрольное использование GOTO приводит к "спагетти" — программам, хаотически напичканным операторами GOTO, в которых отследить все возможные состояния вычислительного процесса — а следовательно, и убедиться в корректности программы — становится практически невозможно.

Вслед за работами Дейкстры и др. возникла методология **структурного программирования** (Здесь часто говорят о "программировании без GOTO", хотя это [слишком грубое упрощение](#)).

В структурном программировании был найден минимальный набор управляющих структур для систематического построения эффективных программ (IF, WHILE и т.п.; все они представлены в Обероне/Компонентном Паскале).

В 70-е гг. "проблема GOTO" еще активно дискутировалась, но к настоящему времени вопрос исчерпан: GOTO не нужен и вреден, и защищать его не возьмется ни один серьезный специалист. Наличие GOTO в старых языках — следствие необходимости обеспечить совместимость с "программным наследием"; а его наличие в новых — признак некомпетентности проектировщика.

Но почему оператор GOTO следует вообще исключать из языка программирования, ведь, казалось бы, им можно просто не пользоваться, оставив лишь на всякий пожарный случай? Ответ заключается в следующем.

В эволюции больших программных проектов наблюдается феномен **насыщения степеней свободы языка программирования**, т.е. тенденция к использованию всех возможных средств языка, причем проконтролировать это трудно.

Насыщение происходит по ряду причин: участие многих программистов, значительная часть которых относится к категории самоучек; использование программ, написанных вне данного проекта; соблазн поставить быструю "заплатку" — соблазн зачастую непреодолимый под давлением жесткого графика.

Но **такие заплатки как ржавчина**: стоит ржавчине появиться в одной точке, и она начинает расти, разъедая конструкцию.

Проблема усугубляется в случае программистов-самоучек — а таких большинство, т.к. цифровая революция развивается слишком быстро, требуя большего числа специалистов, нежели может поставлять система образования. Причем, она сама в значительной степени построена из самоучек: нередко "образование" сводится к семестровому курсу, в котором объясняется синтаксис фортрана или С под сопровождением благоглупостей вроде "программирование — просто раздел прикладной математики" или "настоящему программисту все равно, на каком языке писать".

Программисты-самоучки, особенно начинающие, нередко убеждены, что "крутизна" программирования в том, чтобы как можно хитроумней использовать как можно больше средств языка в каждой программе: ведь им пришлось учить про них к экзамену!

Из документации, где просто перечислены все средства — как и из большинства руководств, сляпанных по принципу разжевывания документации — невозможно узнать, что то или иное средство было просто сделанной когда-то ошибкой дизайнера.

Поэтому наилучшее решение проблемы GOTO — радикальное: просто исключить его из языка.

Пример. Пусть дан массив из n целочисленных элементов $a[0] \dots a[n-1]$.

Описание на Обероне/Компонентном Паскале:

```
VAR a: ARRAY n OF INTEGER;
```

Пусть про этот массив известно, что один из элементов равен x . Требуется найти этот элемент и записать его позицию в целую переменную i .

Типичный начинающий программист (и многие "профессионалы") сразу напишет FOR с вложенной проверкой и начнет искать способ выйти из цикла:

```
FOR i := 0 TO n-1 DO
  IF a[i] = x THEN ... END
END
```

Но в Обероне/Компонентном Паскале из цикла FOR выйти просто так нельзя. Если не отказываться от цикла FOR (см. ниже решение этой задачи с циклом WHILE), то единственный способ это сделать — заключить цикл в процедуру и использовать оператор RETURN, позволяющий выйти из процедуры в произвольной точке:

```
PROCEDURE Найти (VAR pos: INTEGER);
  VAR i: INTEGER;
BEGIN
  FOR i := 0 TO n-1 DO
    IF a[i] = x THEN pos := i; RETURN END
  END
END Найти
```

Тогда в основной программе вместо цикла нужно вызвать эту процедуру:

```
Найти( i )
```

Еще можно заменить FOR на оператор LOOP, в теле которого допускается выход на конец цикла (оператор EXIT).

Но ни в том, ни в другом случае нельзя "выпрыгнуть" вообще в произвольную точку вне цикла: RETURN "прыгает" на конец процедуры, а EXIT — на конец цикла. Таким образом программа остается достаточно структурированной.

Кстати, вот классическое "правильное" решение в стиле Дейкстры, выраженное на Обероне/Компонентном Паскале:

```
  i := 0;
  WHILE (i < n) & (a[i] # x) DO
    i := i + 1
  END
```

Напомним, что вычисление логических выражений в Обероне/Компонентном Паскале производится по правилу "короткого замыкания": если результат логической операции можно определить по первому операнду, то второй не вычисляется.

В данном случае по достижении $i = n$ первый операнд ($i < n$) даст FALSE, так что результатом операции & (логическое И) тоже будет FALSE, и выход из цикла произойдет сразу, без вычисления второго операнда.

Заметим, что вычисление второго операнда привело бы к ошибке и аварийной остановке программы из-за выхода за верхнюю границу массива. (О проверках выхода за границы массивов в Обероне/Компонентном Паскале см. далее.)

Правило весьма удобно при систематической разработке алгоритмов и приводит к лаконичным программам; одно это правило исключает множество ситуаций, где наивный инстинкт требует использовать GOTO. Много примеров на этот счет — в том числе гораздо менее тривиальных, чем приведенный — дано в книгах Гриса и Дейкстры в [нашем списке](#).

Заметим еще, что по приведенной стандартной схеме решается большое число задач, сводящихся к задаче поиска:

```
  <инициализация цикла>
  WHILE <условие ограничения поиска> & ~(<условие окончания
поиска>) DO
    i := i + 1
  END
```

Логическое значение, выражающее успех поиска, равно значению выражения <условие ограничения поиска> после выхода из цикла.

О структурном программировании и отсутствии GOTO

Говорить о структурном программировании как о "программировании без GOTO" — слишком грубое упрощение. Если просто взять наивную программу с парой вложенных циклов и выходами из них с помощью GOTO и механически исключить GOTO, вводя дополнительные логические переменные, то "исправленная" программа может оказаться еще менее ясной, чем исходная.

Вот пример подобного рода программы без единого GOTO, навеянной на Компонентном Паскале энергичным программистом-самоучкой:

```
PROCEDURE (obj:setGuider) generate*,NEW;
  VAR exit,reset:BOOLEAN;i:INTEGER;
```

```

BEGIN
  exit:=FALSE; reset:=FALSE; i:=LEN(obj.content)-1;
  WHILE exit=FALSE
    DO
      IF i>=0 THEN
        obj.content[i].checkFull();
        IF obj.content[i].full=FALSE THEN
          obj.content[i].add();

          exit:=TRUE;
        ELSE
          reset:=TRUE;
          obj.content[i].reset();
          DEC(i);
        END;
        IF (reset=TRUE) & (exit=TRUE) THEN
          obj.setMax(i);
        END;
      ELSE
        exit:=TRUE;
        obj.end:=TRUE;
      END;
    END;
  END generate;

```

Отметим достаточно типичное неумение пользоваться логическими выражениями: WHILE exit = false DO ... вместо просто WHILE ~exit DO ...

А вот эквивалентная хорошо структурированная программа:

```

PROCEDURE ( sg: SetGuider ) Next*, NEW;
  VAR i: INTEGER; c: POINTER TO ARRAY OF Set;
BEGIN
  c := sg.content;
  i := LEN(c) - 1;
  WHILE ( i >= 0 ) & c[i].IsFull() DO
    c[i].Reset(); i := i - 1
  END;
  IF i < 0 THEN
    sg.end := TRUE
  ELSE
    c[ i ].Inc();
    IF i < LEN( c ) - 1 THEN sg.SetMax( i ) END
  END
END Next;

```

Здесь сразу видно, что цикл WHILE устроен по принципу поиска в массиве первого с конца элемента, не обладающего свойством c[i].IsFull():

ПОКА (не вышли из массива) И (текущий эл-т не удовлетворяет условию поиска) ДЕЛАТЬ ...,

а оператор IF в конце — стандартный обработчик двух возможных состояний после выхода из цикла:

ЕСЛИ (вышли из массива, т.е. дошли до конца, не найдя элемента) ТО ...

А теперь представьте себе, что вам нужно модифицировать эту программу под новую задачу: очевидно, что во втором случае это сделать не только намного легче, но и минимален риск внести при исправлении ошибку. Именно в этом и заключается цель применения структурного программирования.

Впрочем, польза от механического исключения GOTO есть: по крайней мере так можно доказать, что без GOTO можно обойтись в любых программах (см. книгу Лингера и др. в [списке литературы](#)).

О проверках выхода индексов за границы массивов

В старых языках программирования (фортран и т.п.) при обращении к элементам массива компилятор, как правило, не предусматривает проверок, выходит ли индекс за границы массива.

В другом старом языке — С, где с массивами можно работать с помощью указателей — подобные проверки вообще могут быть невозможны.

В таких условиях, если происходит ошибочное обращение к несуществующим элементам массива, то программа просто считывает содержимое ячеек памяти, никакого отношения к данному массиву не имеющих, либо записывает туда какую-то информацию, портя содержимое других переменных, возможно, в других программах, и затем продолжает свою работу, уже скорее всего бессмысленную. В конце - концов либо выдается невнятный результат, либо программа аварийно останавливается в какой-то точке, не имеющей никакого отношения к ошибке. Либо что-то неожиданное происходит с другими программами на компьютере.

Понятно, что искать ошибку такого рода отнюдь не легко — ведь при каждом новом вызове программа может загружаться в разные области памяти и, соответственно, ошибочно обращаться к разным областям памяти.

Когда было обнаружено, насколько велика доля ошибок, связанных с выходом за границы массивов, в компиляторы стали добавлять возможность включать в код проверку индексов на предмет выхода за границы массива. Но это обычно делается, только если компилятор работает в специальном "отладочном" режиме. Разумеется, "настоящий" программист считает отладочный режим средством для новичков - "чайников".

Оберон/Компонентный Паскаль не предусматривает никакого специального отладочного режима, а проверки индексов на предмет выхода за границы массива отключить просто нельзя. В результате "настоящий" программист чувствует, что ему навязана роль "чайника", и выдвигает следующее "практическое" возражение: дескать, включение таких проверок увеличивает и замедляет скомпилированную программу.

На самом деле экспериментальное изучение программ доказывает, что и тот и другой эффект в подавляющем большинстве случаев ничтожен (при условии тщательного дизайна языка и компилятора, как это имеет место для Оберона/Компонентного Паскаля).

Жертвовать же надежностью *всех* программ ради ускорения пусть даже на десяток процентов в редком случае — безответственно (гораздо чаще имеет место замедление на уровне 1%, если его вообще удастся достоверно измерить).

Если же такая редкая программа предназначена для частого использования, то и оптимизировать ее нужно особыми методами, например, переписав соответствующий

фрагмент непосредственно в машинных кодах. Цикл, в котором проверка индексов заметно сказывается на производительности, вряд ли будет сложным, так что написать несколько команд в машинных кодах труда не составит для специалистов, которые обычно занимаются такими программами.

О том, как включать фрагменты в машинных кодах непосредственно в программу на Компонентном Паскале, см. в документации Блэкбокса *Platform-Specific Issues*, раздел *Code procedures*).

Возможно и возражение метафизическое, мол, ограничения такого рода (как и исключение [оператора GOTO](#), и т.п.), навязываемые "профессионалу", ограничивают его свободу творчества.

Но, **во-первых**, величина и сложность подавляющего большинства полезных программ оставляет достаточный простор для творчества и за вычетом мелкой возни с индексами массивов.

Во-вторых, программирование — сколько бы ни было в нем от искусства — это прежде всего эффективное создание правильных, надежных и эффективных программ (эффективность не имеет смысла, если программа неправильная или приводит к взрыву ракеты или потере финансовой отчетности).

Посмотрим, с этой точки зрения, на вариант ошибки такого рода из числа наихудших. А именно, на ситуацию, когда данные, записываемые в массив, поступают из Интернета.

В таком случае, если программа не проверяет выход за границу буфера — массива, предназначенного для записи приходящей из Интернета информации, — можно заставить программу записать в какую-то область памяти исполняемый код и даже передать ему управление, тем самым перехватив управление компьютером. Именно в этом состоит излюбленный хакерами способ проникать в компьютеры, подключенные к Интернету — атака посредством переполнения буфера (buffer overflow).

Сообщениями о таких атаках пестрят новостные ленты компьютерного мира. Например, вспомним одну поучительную историю. В августе 2001 г. вице-президент Майкрософт Джим Олчин (Jim Allchin) объявил во время доклада на открытии конференции Intel Developers Forum в Сан Хосе, что в новой операционной системе Windows XP все возможные проблемы из разряда переполнения буфера были устранены посредством специального анализа исходных текстов на предмет безопасности (security audit).

Но в декабре того же года была найдена "дыра" в одной из программ в составе Windows XP (в программах поддержки стандарта подключения внешних устройств Universal Plug and Play) — причем дыра оказалась именно из категории переполнения буфера.

- **Вопрос:** Что мешает Майкрософт компилировать Windows XP с включенной опцией контроля выхода за границы массивов?

Ответ: Неадекватность языков и компиляторов, использованных в написании операционной системы (С и С++).

- **Вопрос:** Почему аудит исходных текстов, о котором объявил Джим Олчин, не обнаружил этой проблемы?

Ответ: Потому что вовсе не равна нулю вероятность ошибиться живым людям, выполнявшим аудит 50 миллионов строк программ на нечитабельном языке типа С.

• **Вопрос:** Почему столь важный продукт, как операционная система, используемая на сотне миллионов компьютеров во всем мире, строится на столь гнилом фундаменте, как язык программирования, для которого нельзя написать компилятор, генерирующий эффективный и безопасный код, с тем чтобы переложить тривиальный механический труд по нудной проверке индексов массивов с человека, которому свойственно ошибаться при выполнении механических процедур, на компьютер, который именно для таких процедур и придуман?

Ответ: Потому что подавляющее большинство программистов Майкрософт, начиная с Билла Гейтса, — как бы хорошо ни была организована их работа и какими бы талантливыми индивидуумами они ни были сами по себе — классические программисты-самоучки, втянутые в круговорот компьютерной революции большими деньгами и задумывающиеся о методологии программирования только под сильной угрозой со стороны конкурентов.

В начале 2002 г. Майкрософт на месяц приостановила нормальную работу, чтобы программистский персонал мог специально сосредоточиться на проблемах безопасности и надежности программ. Если оценить количество программистов Майкрософт в 20 тысяч человек при зарплате от \$150 тыс. в год и выше, то стоимость месячника повышения квалификации выйдет не меньше 250 миллионов долларов.

Майкрософт в состоянии это себе позволить. Остальным, видимо, все же дешевле перейти на инструменты программирования, где проверки индексов массивов не отключаются.

Впрочем, и сама компания Майкрософт в настоящее время переходит на платформу .NET, в которой главный язык программирования — т.наз. С# — смоделирован во многом, в том числе и в отношении безопасности программирования, по образцу Оберона.

Почему порочно "наивное" программирование с пошаговым отладчиком

Пошаговый отладчик — обязательный элемент традиционного "наивного" подхода к программированию, практикуемого подавляющим большинством программистов-самоучек (в том числе зарабатывающих программированием на жизнь и поэтому называющих себя "профессионалами").

Этот подход заключается в том, чтобы написать первый вариант программы более или менее наугад, а потом "отлаживать" ее, ставя программные заплатки на основе наблюдений за поведением программы с помощью пошагового отладчика.

С профессиональной точки зрения неадекватность подхода "начнем как попало, потом отладим" заключается в следующем:

- в низкой производительности программиста, так как поиск решения — т.е. построение программы — осуществляется в значительной степени наугад (хорошо известно выражение "агония отладки" — "the agony of debugging");
- в неприемлемо низком качестве получающихся "залатанных" программ (трудность сопровождения, низкая надежность; подробнее об этом см. [ниже](#)).

С точки зрения обучения программированию, хорошо известно, что **лучше всего усваивается то, что изучается в начале курса**. Поэтому начинающие программисты быстро и прочно привыкают полагаться на пошаговый отладчик при создании программы, слепленной как попало.

Переучиваться всегда намного труднее, чем научиться правильной технике с самого начала, хотя "правильное" обучение может быть довольно скучным делом. В этом смысле дело здесь обстоит как в спорте: при серьезном обучении начинающих, скажем, горнолыжников достаточно долго обучают скучным элементам техники боковых соскальзываний и т.п., прежде чем разрешить делать первые повороты на склоне.

Видимо, и начинающих программистов следует достаточно долго с мелом и карандашом обучать элементам логики, правильному построению простейших циклов и т.п., прежде чем подпускать к клавиатуре с заданием написать работающую программу.

Альтернативой наивному подходу являются систематические методы, в которых программа выводится из требуемых пред- и пост-условий (см. книги Вирта, Гриса и Дейкстры в [списке литературы](#)).

Такое проектирование опирается на систематическую верификацию (доказательство, проверку) логических свойств программы, которые должны удовлетворяться в ключевых точках программы (пред- и пост-условия для процедур, инварианты циклов и т.п.). Тогда и "отладка" в идеале сводится к устранению синтаксических ошибок и мелких описок.

Для такого систематического программирования более чем достаточны средства, предлагаемые в Блэкбоксе (при аварийной остановке — например, если нарушено логическое условие в ASSERT — Блэкбокс дает возможность исследовать состояние локальных переменных всех процедур в цепочке вызовов, приведших к аварийной остановке, а также глобальных переменных всех модулей).

Отсутствие пошагового отладчика дисциплинирует процесс программирования подобно тому, как невозможность применить GOTO стимулирует использование методов, приводящих к программам, имеющим хорошую структуру.

Чем плохи "златанные" программы

1. Такие программы очень трудно сопровождать. Хорошо известна эмпирическая закономерность, что время жизни программ превосходит ожидания их авторов: после успешного решения одной задачи возникает желание воспользоваться решением еще раз. Но задачи меняются, и старые программы приходится "сопровождать" — приспособливать к меняющимся условиям. Опыт показывает, что сопровождение отнимает львиную долю усилий программистов.

Сопровождать программу можно только четко понимая, что и как она делает. Даже если сопровождением занимается автор программы, то отнюдь не факт, что он помнит детали своих старых кодов (простой пример из жизни: перенос макросов для MS Word между версиями 6.0 и 2002, предугадать который было совершенно невозможно при их написании в 1994 г.).

Разобраться с "златанной" программой как правило гораздо труднее, чем с программой, построенной систематическими методами.

Поэтому, кстати, и вероятность внесения новых, причем трудно устранимых ошибок при модификации тоже велика, а дальнейшее "залатывание" — даже если оно и

завершится чем-то похожим на успех — еще сильнее запутает структуру программы и заложит основу для еще больших проблем в будущем. Этот известный эффект можно назвать *законом повышением энтропии* в структуре больших программ.

2. Такие программы страдают низкой надежностью. С "залатанной" программой никогда нет уверенности, что в ней предусмотрены все ситуации, которые могут возникнуть при выполнении. Поэтому такую программу нельзя считать надежной.

Конечно, абсолютно надежной нельзя считать никакую достаточно сложную программу. Но, во-первых, надежность программ повышается действительно радикально, если использовать соответствующие методы и инструменты (языки и т.п.). В интервью директора Центра по надежности программного обеспечения университета Карнеги-Меллон (США) отмечалось, что 95% из более чем 50 тысяч программных дефектов, исследованных центром за 9 месяцев 2002 г., относятся к категории "предотвратимых", т.е. таких, которые исключаются при использовании надлежащих средств программирования.

Во-вторых, нужно учесть и стоимость ошибок для тех, кто программой будет пользоваться: даже потерей пяти минут работы из-за сбоя редактора трудно пренебрегать, если умножить эффект на миллионы пользователей. Тем более если речь идет о потере [космической ракеты](#) стоимостью сотни миллионов долларов.

Система BlackBox

Систему Блэббокс проще всего установить простым копированием на жесткий или флэш диск уже готового комплекта. Готовые комплекты (для школ, университетов и т.п.) в виде архивов свободно доступны, например, с сайта проекта Информатика-21 (<http://www.inr.ac.ru/~info21/>). Так можно в разных папках установить любое число независимых вариантов конфигурации Блэббокса для разных целей. При этом не будут установлены ассоциации с расширениями имен файлов в операционной системе, но обычно это и не нужно. Именно такой способ установки Блэббокса и рекомендуется.

Ниже описывается способ установки Блэббокса из оригинального дистрибутива, при котором создаются и все ассоциации. После такой установки можно просто скопировать любой из комплектов с сайта проекта Информатика-21 поверх инсталляции.

Первичная установка системы Блэббокс

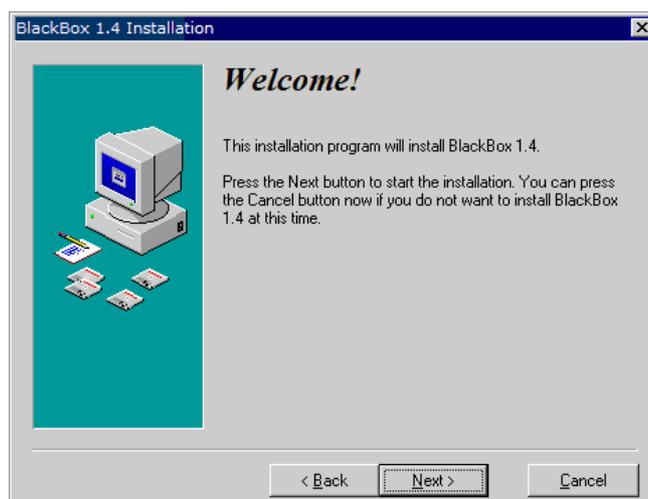
Настоящая инструкция касается операционных систем MS Windows 95 и более новых. О Windows 3.1 сказано [отдельно](#).

Блэббокс потребует чуть меньше 20 МВ на диске для полной установки, для минимальной (без поддержки MS Office) — чуть больше 10 МВ.

Получив дистрибутив, достаточно просто выполнить его.

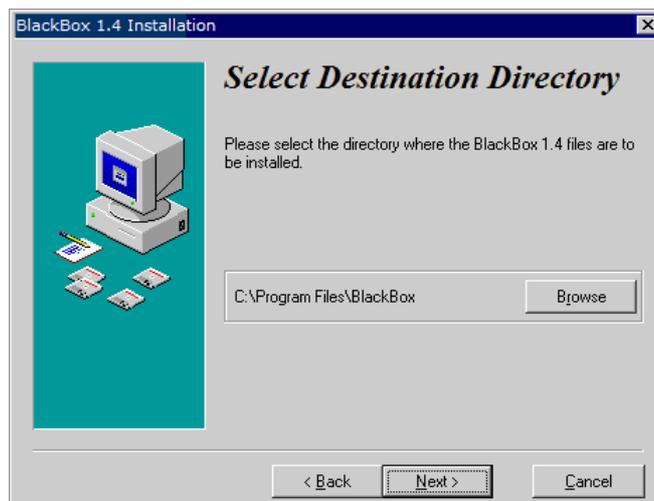
Появится синяя заставка, а также откроется окошко с текстом лицензии и кнопками ОК и Cancel. Нажимая кнопку ОК, вы подтверждаете, что прочли, поняли и обязуетесь выполнять [условия лицензии](#).

После этого откроется новое окошко Welcome с информацией, что Вы устанавливаете Блэббокс:



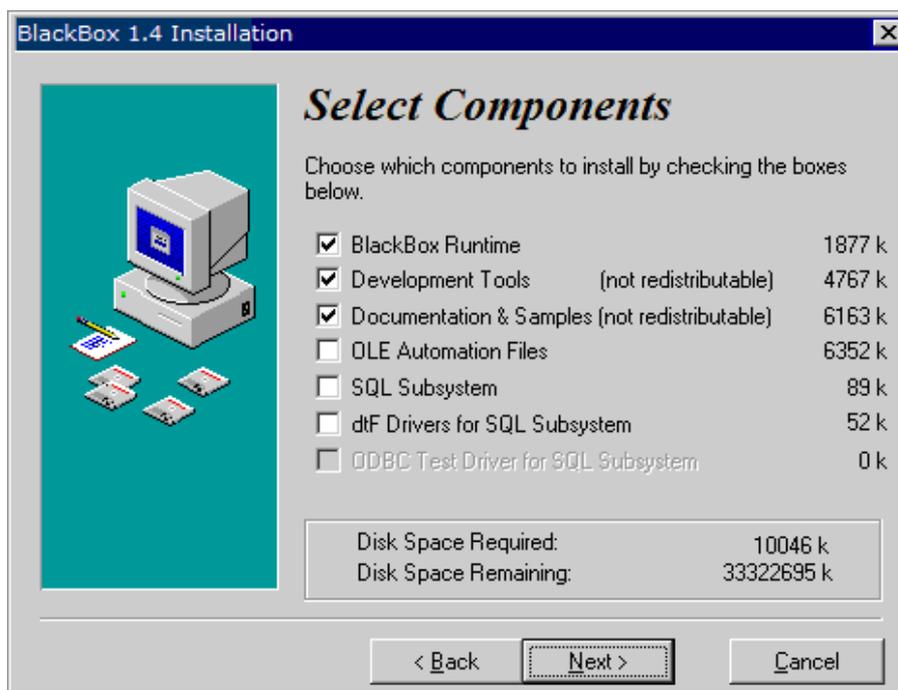
Чтобы продолжать установку, кликните Next.

Затем будет предложено определить, куда устанавливать Блэббокс:



По умолчанию предлагается папка C:\Program Files\BlackBox -- вы можете сделать собственный выбор (например, поместить систему на другой диск). Для этого достаточно кликнуть по Browse. Выбранную папку для дальнейшего полезно называть *первичная директория Блэббокса*. После этого кликните по кнопке Next.

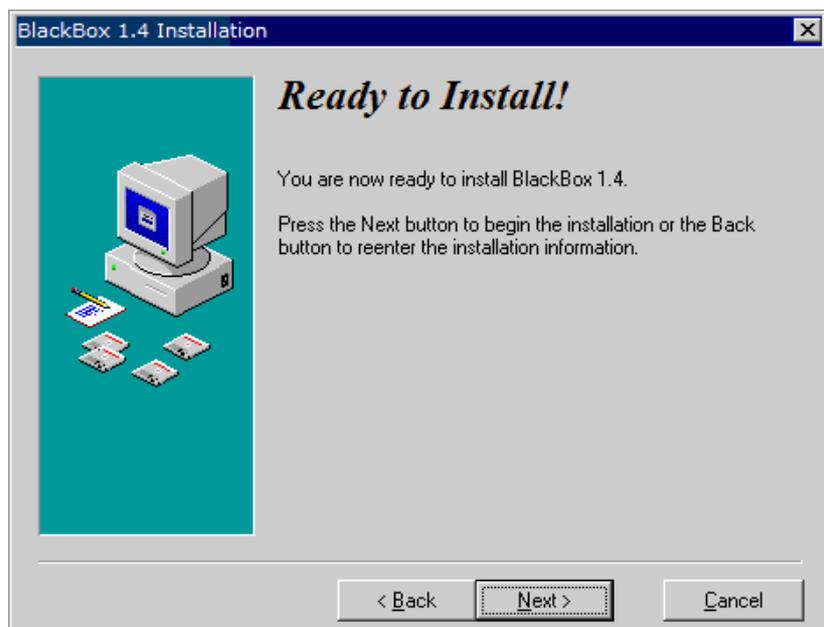
Появится диалог для выбора компонент системы, которые нужно установить. По умолчанию выбраны все (20 MB). В условиях средней школы можно мышкой разотметить последние три опции, как показано ниже — тогда для установки достаточно 10 MB на диске:



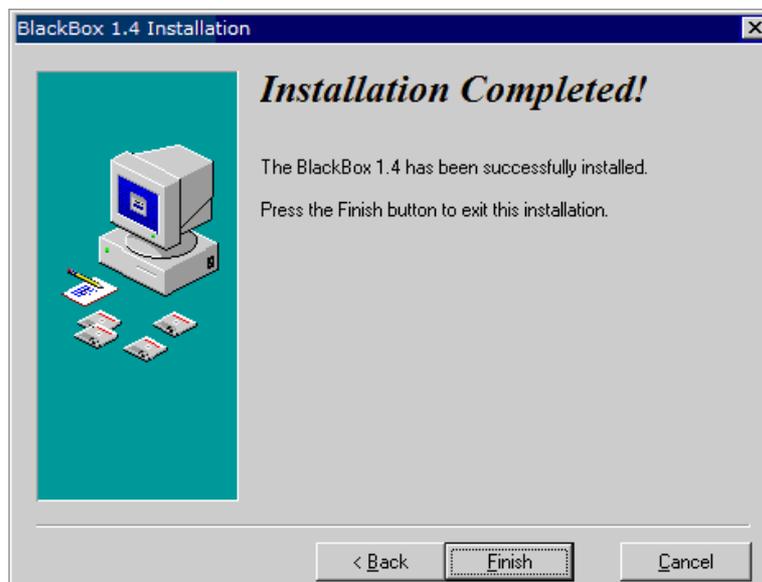
Если Вы собираетесь писать программы, взаимодействующие с Microsoft Office, нужно отметить компоненту OLE Autotmation Files.

Заметим, что доустановить нужные компоненты можно в любой момент, повторив установку.

После клика по Next появится предупреждение о готовности начать инсталляцию:



которая запускается еще одним кликом по Next. Инсталляция завершится быстро:



Нажмите кнопку Finish. На верстаке обнаружите желтую иконку, с помощью которой запускается Блэкбокс:



Рекомендуется пользоваться комплектами Блэкбокса с сайта Информатики-21. В частности, т.наз. базовый комплект в минимальной степени отличается от оригинальной дистрибуции (только русификация компилятора и инструментов системы + переводы документации), — но содержит несколько простых, но очень полезных добавлений (например, возможность автоматического раскрытия синтаксических конструкций по первым буквам с правильной структурой отступов).

Русификация системы Блэкбокс

Все комплекты системы Блэкбокс с сайта проекта Информатика-21 русифицированы: поддерживают полный русский алфавит в идентификаторах Компонентного Паскаля, снабжены полными переводами документации Блэкбокса (папки `Docu/ru/` в каждой подсистеме), а редактор и все инструменты Блэкбокса корректно обрабатывают слова и идентификаторы с кириллицей.

Для профессиональной работы и учебной работы на уровне университетов этого достаточно.

В школьной конфигурации выполнена полная русификация — ошибок компилятора (`Dev/Rsrc/Errors.odc`), всех меню (файлы `Rsrc/Menus.odc` в каждой подсистеме) и т.п. Кроме того, там по умолчанию включен фильтр, позволяющий компилировать программы на Компонентном Паскале с русскими ключевыми словами. Заметим, что этот фильтр можно настроить и на любой другой национальный язык — достаточно добавить соответствующий словарь (документ Блэкбокса) и, в случае необходимости, поправить команду компиляции в документах, описывающих меню.

Можно создать и любой промежуточный вариант: достаточно скопировать документ `Dev/Rsrc/Errors.odc` из школьной версии в университетскую, чтобы в последней сообщения компилятора стали русскими. Можно также в любой момент отредактировать этот документ на свой вкус, причем активизировать новый вариант можно, не выходя из Блэкбокса (`Dev, Flush Resources`).

Вторичные папки Блэкбокса

Механизм вторичных папок полезен в ряде ситуаций: независимая работа нескольких учеников на одном компьютере; ведение нескольких независимых проектов; одновременная работа с несколькими версиями системы Блэкбокс без дублирования всей системы — например, в одной версии меню и сообщения компилятора могут быть переведены на русский язык, а в другом оставлены в английском варианте. Изначально этот вариант был придуман для работы с конфигурацией Блэкбокса, установленной на защищенном сетевом диске; поэтому еще используется название «серверный режим работы Блэкбокса».

Проще всего организовать такую работу, копируя во вторичные папки (а это могут быть любые папки) запускающий файл `StartBlackBox.vbs`, создаваемый командой меню `Create vbs`, добавленной во все комплекты Блэкбокса проекта Информатика-21. При этом запуск Блэкбокса осуществляется активацией (например, двойным кликом) запускающего файла в нужной вторичной папке. Ниже в отдельных разделах описан механизм чтения файлов, а также сохранения исправленных или модифицированных файлов, который применяет Блэкбокс при работе со вторичными папками; этот механизм нужно обязательно понять при работе в этом режиме.

Если по какой-то причине воспользоваться механизмом запускающего файла невозможно, то нужно воспользоваться инструкциям, приводимым ниже.

Напомним, что первичная директория — это та, в которую была произведена первоначальная инсталляция Блэкбокса. Будем предполагать, что это `C:\Program Files\BlackBox\`.

Выберем любую другую директорию в качестве вторичной. Например, пусть это будет `Z:\Вторичная\`.

Создадим на рабочем столе ярлык (shortcut) для файла `BlackBox.exe` — главного загружаемого файла Блэкбокса (например, можно скопировать ярлык, появившийся на рабочем столе при инсталляции Блэкбокса) — и изменим в ярлыке параметры как показано на картинке:



Строка **Target**: содержит следующий текст:

```
"C:\Program Files\BlackBox\BlackBox.exe" /Use "Z:\Вторичная"
```

Строка **Start in**: содержит следующий текст:

```
"Z:\Вторичная"
```

Теперь можно вызвать Блэкбокс посредством этого ярлыка.

Тогда говорим, что Блэкбокс вызывается во вторичной директории (в данном случае это `Z:\Вторичная`). Обычный вызов двойным кликом мышки по `BlackBox.exe` удобно обозначать как вызов в первичной директории.

При вызове во вторичной директории Блэкбокс будет пытаться работать с файлами (если их адреса не заданы в абсолютной форме) так, как если бы вторичная директория была на самом деле первичной, и только не найдя там нужного файла, выполнять повторный поиск в первичной директории.

Более подробно алгоритмы работы с файлами описываются ниже.

Чтение файла

Например, в первичной директории Блэкбокса есть папка Docu, содержащая файл CP-Lang.odc с английской версией "Сообщения о языке Компонентный Паскаль".

Если вызвать Блэкбокс в первичной директории, то нажатие клавиши F1 и затем двойной клик по гиперссылке Language Report откроет "Сообщение" на английском языке.

Теперь во вторичной директории тоже создадим папку Docu, а в ней — файл CP-Lang.odc с каким угодно содержимым (например, с текстом романа "Война и мир"). Вызовем Блэкбокс во вторичной директории как описано выше.

Теперь при попытке открыть "Сообщение .." (F1, затем клик мышкой по Language Report) откроется текст "Войны и мира": файл во вторичной директории "закрыл" для Блэкбокса соответствующий файл в первичной.

Еще пример: пусть мы вызвали Блэкбокс во вторичной директории и пишем новый модуль, в котором импортируется другой модуль, скажем, i21sysEdit.

Тогда при компиляции нового модуля Блэкбоксу будет нужен символьный файл модуля i21sysEdit. Если бы Блэкбокс был вызван в первичной директории, то нужный символьный файл — это Edit.osf в папке i21sys\Mod\ в первичной директории, т.е файл с полным адресом C:\Program Files\BlackBox\i21sys\Mod\Edit.osf.

Но т.к. Блэкбокс был вызван во вторичной директории, то он сначала попытается открыть файл с тем же относительным адресом (i21sys\Mod\Edit.osf), но во вторичной директории, т.е. файл с полным адресом Z:\Вторичная\i21sys\Mod\Edit.osf. И только если такого файла не окажется, Блэкбокс обратится к соответствующему файлу в первичной директории, т.е. C:\Program Files\BlackBox\i21sys\Mod\Edit.osf.

В общем случае, если Блэкбоксу нужен какой-то файл с некоторым относительным адресом в первичной директории, то сначала Блэкбокс попытается найти файл с точно таким же именем и относительным адресом во вторичной директории, и только если файл там не будет обнаружен, будет выполнен поиск в первичной директории.

Запись файла

При записи на диск нового файла Блэкбокс запишет его во вторичной директории с тем же относительным адресом, как и в случае, когда Блэкбокс вызывается непосредственно из первичной директории.

Например, при компиляции модуля i21sysEdit.odc Блэкбокс всегда будет записывать скомпилированный модуль в файл с относительным адресом i21sys\Code\Edit.ocf — независимо от того, в первичной или вторичной директории был вызван Блэкбокс, а также независимо от того, где хранится исходный файл модуля.

Это означает, что в первичной и вторичной директории могут храниться разные версии этого модуля, и тогда при вызове Блэкбокса в первичной или вторичной директории будут активироваться (например, при вызове процедур модуля из меню, при импорте модуля из других модулей и т.п.) соответствующие разные версии модуля.

Ограничения

- 1) Описанный механизм работы с файлами можно обойти, если задавать их адреса в абсолютной форме. Например, в документе, открываемом при нажатии F1, можно поменять гиперссылку на "Сообщение..." так, чтобы там был указан абсолютный адрес (в первоначальной версии там указан относительный адрес). В этом случае будет вызываться один и тот же файл при вызове Блэкбокса и в первичной, и во вторичной директориях.
- 2) Описанный механизм не работает для файла BlackBox.exe (впрочем, обратиться к нему при работе в Блэкбоксе не так просто). Единственный способ работать с разными версиями этого файла — скопировать целиком инсталляцию Блэкбокса в другую папку и там заменить BlackBox.exe. Фактически при этом получится еще одна первичная директория, и работать с ней можно как с первоначальной первичной директорией, созданной при инсталляции Блэкбокса. В частности, можно создавать для нее вторичные директории и т.п.

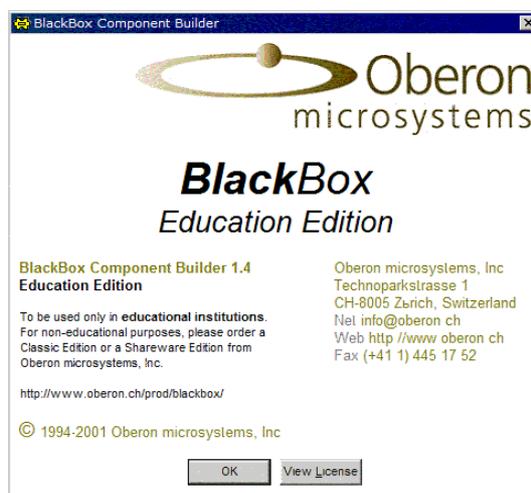
Запуск Блэкбокса и выполнение первой программы

Ниже дано быстрое введение в работу в системе Блэкбокс. Инструкции ориентированы на использование русифицированной версии Блэкбокса в школе. Для опытных программистов в конфигурациях Блэкбокса от Информатики-21 предусмотрен документ «readme, профи!.odc» в основной папке Блэкбокса, его и следует прочесть (из Блэкбокса).

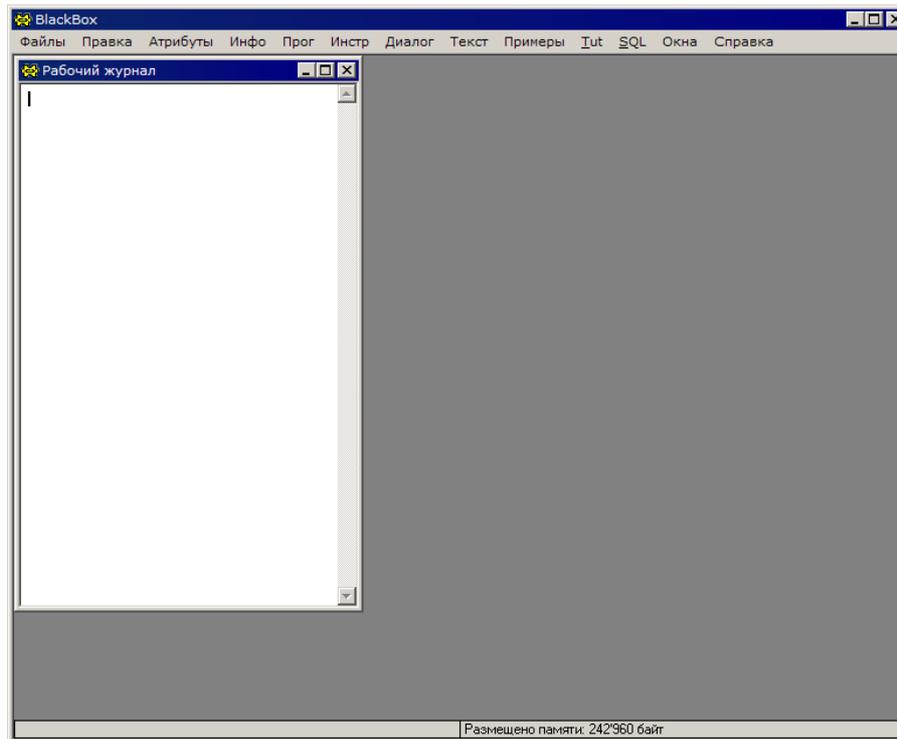
Запуск Блэкбокса производится, например, с помощью желтой иконки, которая возникает на верстаке после первичной установки:



После запуска появится стандартная заставка:



в которой надо кликнуть по ОК. После этого увидим примерно следующее (порядок меню соответствует англоязычной версии):



Пустое окошко — "рабочий журнал" (Log), представляющий собой обычный документ в стандартном формате, используемом в Блэкбоксе (см. Формат документов Блэкбокса). Туда система пишет свои сообщения о компиляции и проч. Туда же удобно направлять простейший вывод (см. Использование рабочего журнала).

Простейшая программа

Простейшая программа на Компонентном Паскале состоит из одного "модуля", состоящего из единственной выполняемой процедуры. Текст модуля может выглядеть так:

```
MODULE Привет;  
  IMPORT StdLog;  
  
  PROCEDURE Сделать*;  
  BEGIN  
    StdLog.String("Привет!")  
  END Сделать;  
  
END Привет.
```

Смысл приведенного текста:

1. В тексте описан единственный модуль, имеющий название Привет. Конец модуля обозначен точкой, которая обязательно должна присутствовать. Перед точкой повторяется имя модуля.
2. В первом операторе модуля (IMPORT StdLog) мы сообщаем компилятору, что собираемся использовать процедуры, содержащиеся в модуле StdLog. Это называется "импортировать" модуль.

Импортируемый модуль должен уже присутствовать в системе в скомпилированном виде (StdLog входит в стандартную дистрибуцию Блэкбокса и предоставляет средства печати в рабочий журнал).

3. В нашем модуле Привет описана единственная процедура, не имеющая параметров, с названием Сделать. В конце процедуры имя должно быть повторено (END Сделать); это правило (как и многие другие) принято, чтобы было легче локализовать ошибки.
4. В заголовке процедуры после ее имени стоит звездочка (показана красным цветом). Это символ экспорта. Он означает, что мы разрешаем этой процедуре быть "видимой снаружи".

В этом случае ее можно будет вызывать из других модулей, если импортировать в них наш модуль Привет так же, как мы импортировали StdLog. Если не поставить звездочку экспорта, то мы вообще не сможем вызвать и выполнить процедуру извне модуля.

5. Процедура Сделать не имеет локальных переменных (о чем говорит отсутствие описаний перед служебным словом BEGIN). Ее выполняемое тело находится между BEGIN и END Сделать.

Тело содержит один оператор StdLog.String("Привет!"), который является вызовом процедуры String, содержащейся в модуле StdLog. Эта процедура имеет один параметр, который должен быть символьной цепочкой произвольной длины (в нашем случае цепочка "Привет!"). Эту цепочку процедура пишет в конец рабочего журнала.

Пара замечаний:

- Слова, написанные большими буквами, являются служебными словами языка Компонентный Паскаль. Они должны быть написаны именно большими буквами (большие и маленькие буквы в Компонентном Паскале различаются). Остальные имена (StdLog, String, Привет, Сделать) — продукт творчества программистов.

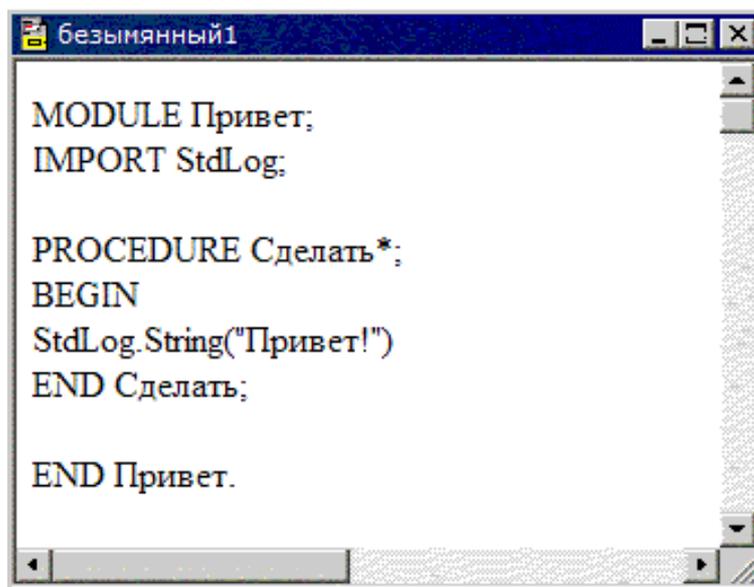
О том, как с удобством печатать служебные слова из больших букв, см. раздел «Ключевые слова из прописных букв»;

- Компилятор игнорирует форматирование текста программы (отступы в начале строк, выбор шрифта — Verdana, Arial, жирный, цвет и т.п.). Ему важно только, чтобы можно было различить служебные слова и идентификаторы.

Теперь воссоздадим эту программу в Блэкбоксе.

Для этого выполним в нем сначала команду меню Файлы --> Новый (в нерусифицированном варианте Files --> New).

Получим новое пустое окошко, озаглавленное безымянный1 (untitled1). Скопируем в него приведенный выше текст модуля. К сожалению, форматирование (отступы) при этом может быть частично потеряно, и результат может выглядеть так:

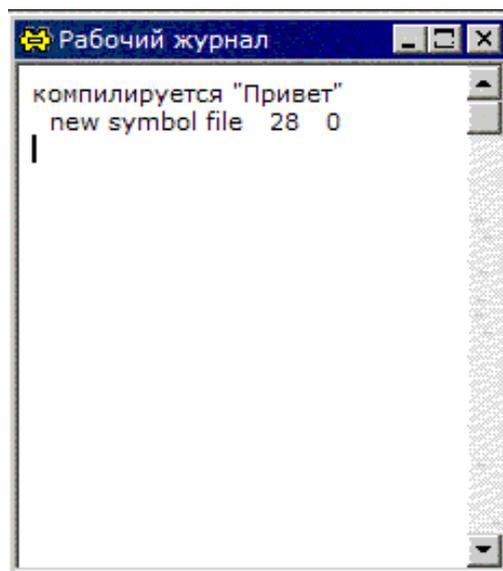


```
MODULE Привет;  
IMPORT StdLog;  
  
PROCEDURE Сделать*;  
BEGIN  
  StdLog.String("Привет!")  
END Сделать;  
  
END Привет.
```

Как уже говорилось, детали форматирования компилятору не важны. (О том, как с удобством форматировать отступы, см. Форматирование текста).

Скопировав, выполним команду меню Прог --> Компилировать (Dev --> Compile или просто Ctrl+K с латинским К, переключать на русскую клавиатуру не обязательно).

В рабочем журнале появится примерно следующее:



```
компилируется "Привет"  
new symbol file 28 0  
|
```

Сообщение во второй строке означает, что модуль скомпилировался успешно (числа несут информацию о размере получившегося машинного кода — в данном случае 28 байт) и что модуля с таким именем в системе перед данной компиляцией еще не было (слово new).

После компиляции перевод модуля Привет на машинный язык хранится на диске в определенном месте (см. ниже), и Блэкбокс может выполнять процедуры из этого модуля.

Фактически, успешная компиляция модуля автоматически включает модуль в систему, тем самым ее "расширяя".

Кстати, большая часть Блэкбокса представляет собой подобные расширения небольшого ядра. Никакой жесткой грани между модулями Блэкбокса и модулями, добавляемыми программистами, нет.

Скомпилированный файл имеет имя `Привет.ocf` и хранится — в соответствии с принятыми в Блэкбоксе [соглашениями](#) (см. Организация файлов в Блэкбоксе) — в папке `Code` в первичной директории Блэкбокса.

Если последняя была выбрана по умолчанию, то полный адрес файла `C:\Program Files\BlackBox\Code\Привет.ocf`.

- Скомпилированные Блэкбоксом файлы **не** являются ни стандартными `exe`-, ни `dll`-файлами. Иначе важнейшие удобства Блэкбокса стали бы невозможны.

П о д р о б н е е :

В Блэкбоксе можно создавать **exe** и **dll** файлы. См. примеры в документации к модулю `DevLinker`.

Из Блэкбокса можно обращаться к независимо созданным **dll**. Об этом позже ...

- При вызове командой `Ctrl+K` компилятор считает, что текст модуля стоит в начале документа, т.е. ищет ключевое слово `MODULE`.
До слова `MODULE` можно ставить правильные комментарии, заключенные в составные скобки (`*` и `*`), а также любые визуальные объекты. Кроме того, компилятор проигнорирует и все, что стоит после точки, оканчивающей модуль.
- О том, как компилятор сообщает об ошибках компиляции см. ниже.

Выполнение программы с помощью «командира»

Опишем простой и удобный способ выполнить процедуру из скомпилированного модуля (есть и другие способы).

Поставим курсор в окошке с нашим модулем в самый конец текста после точки, закрывающей модуль (`Ctrl+End`).

Выполним команду меню `Инстр --> Вставить командир (Tools --> Insert Commander; с клавиатуры Ctrl+Q)`. Появится черный кружок с белым восклицательным знаком — так называемый **командир** (см. картинку ниже).

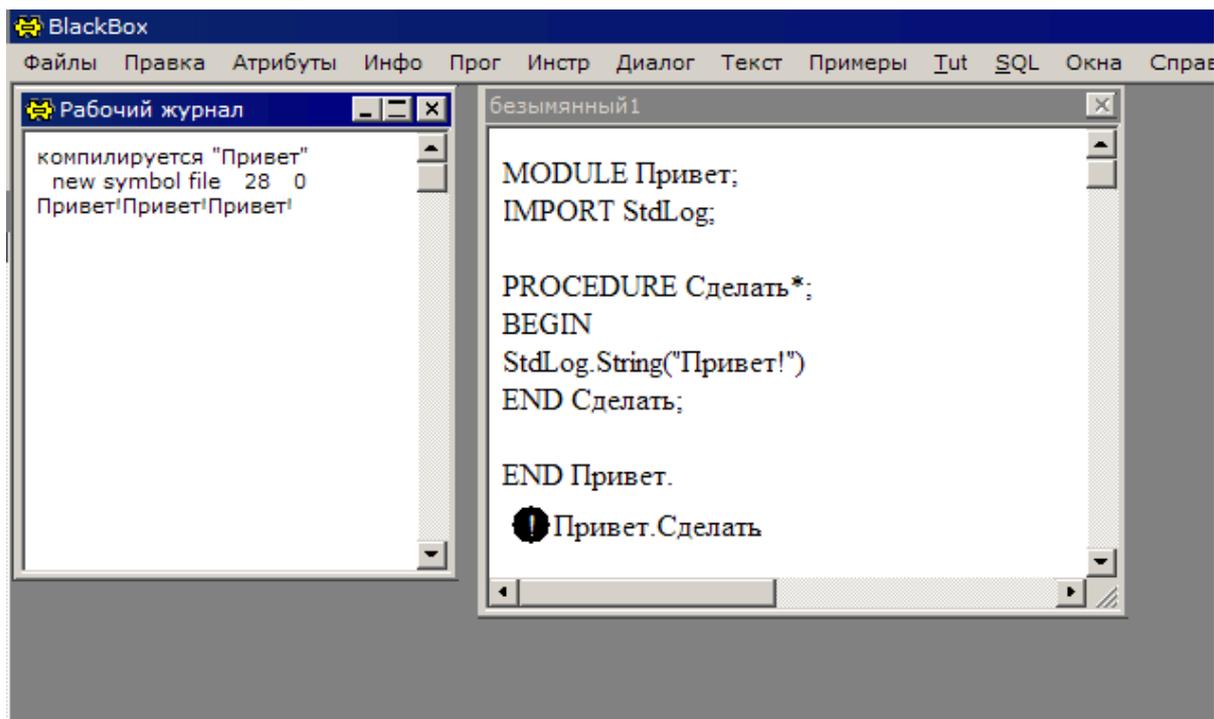
После командира без пробелов напечатать **Привет.Сделать** (имя модуля, точка, имя вызываемой процедуры — это "*ФИО*" нашей процедуры в системе).

Роль фамилии играет имя модуля. Важно следить за прописными и строчными буквами, т.к. в Компонентном Паскале они различаются.

Теперь кликнем мышкой по командиру. Это вызовет загрузку скомпилированного модуля в память и выполнение процедуры. Можно кликнуть еще пару раз — процедура выполнится еще дважды.

- После первой загрузки модуль остается в памяти, причем он уже слинкован с остальными активными модулями (например, [StdLog](#)).
Поэтому дальнейшие обращения к нашему модулю происходят столь же эффективно, как и при статической компиляции. О перезагрузке модулей см. Загрузка и перезагрузка модулей.

После этого экран будет выглядеть примерно так:

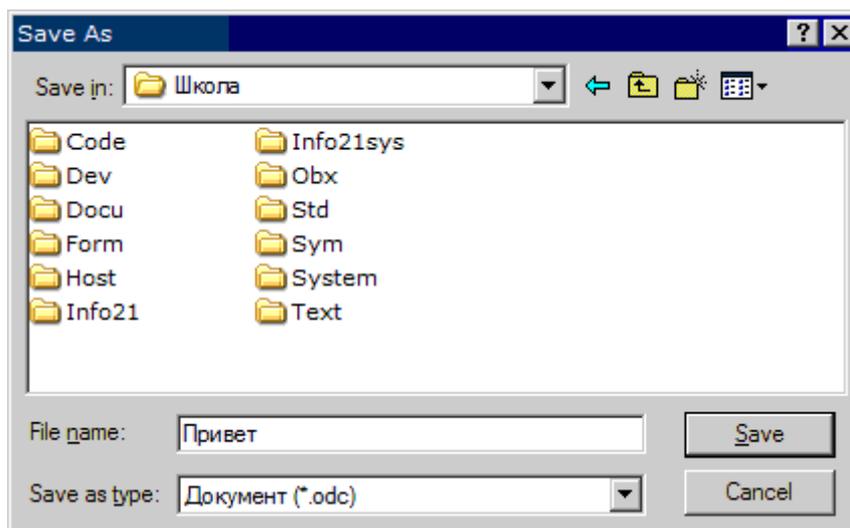


Три идущих подряд цепочки символов Привет! соответствуют трем кликам по команде — и трем выполнениям процедуры Сделать.

Сохранение файла с программой

Сохранить документ в окошке безымянный1 можно в любом месте на диске обычным образом (выполните команду меню Файлы --> Сохранить как...; Files --> Save as...).

Для простоты сохраним модуль прямо в той папке, в которой откроется диалог Файлы --> Сохранить как..., с именем Привет (ниже показан соответствующий диалог непосредственно перед кликом по Save):



- Дополнительные сведения о средствах модуля StdLog (например, о переходе на новую строку можно найти в разделе «Средства, предоставляемые модулем StdLog»).

- При серьезной работе все-таки полезно придерживаться определенной дисциплины при сохранении исходных текстов (см. «Организация файлов Блэкбокса»).

Продолжение работы с примером.

Суть данного раздела сводится к объяснению простого практического правила:

Для выполнения процедур из новой версии модуля нужно кликать по командиру **при нажатой клавише Ctrl**.

Подробнее:

Предположим, что мы выполнили процедуру из нашего модуля, а потом модифицировали его, успешно скомпилировали и пытаемся снова выполнить ту же (или новую) процедуру из него посредством простого клика по командиру.

В этой ситуации Блэкбокс будет вызывать (или пытаться вызвать) процедуру из старой версии модуля, а не из новой.

Чтобы произошло обращение к новой версии модуля, нужно делать клик с нажатой клавишей Ctrl.

Вспоминать об этом правиле обычно приходится в следующих ситуациях:

1. Вызываемая процедура ведет себя так, как будто никаких изменений сделано не было, хотя измененный модуль только что был успешно скомпилирован.

2. Блэкбокс печатает в рабочем журнале сообщение типа:

ошибка команды: команда Сделать2 не найдена в Привет.

Это обычно означает, что программист добавил в модуль Привет процедуру Сделать2 ("команда" в терминологии Блэкбокса — это процедура без параметров) и попытался ее вызвать, забыв нажать Ctrl, так что в памяти осталась старая версия модуля, в которой Блэкбокс и пытался найти новую процедуру — и, естественно, не нашел, о чем и сообщил.

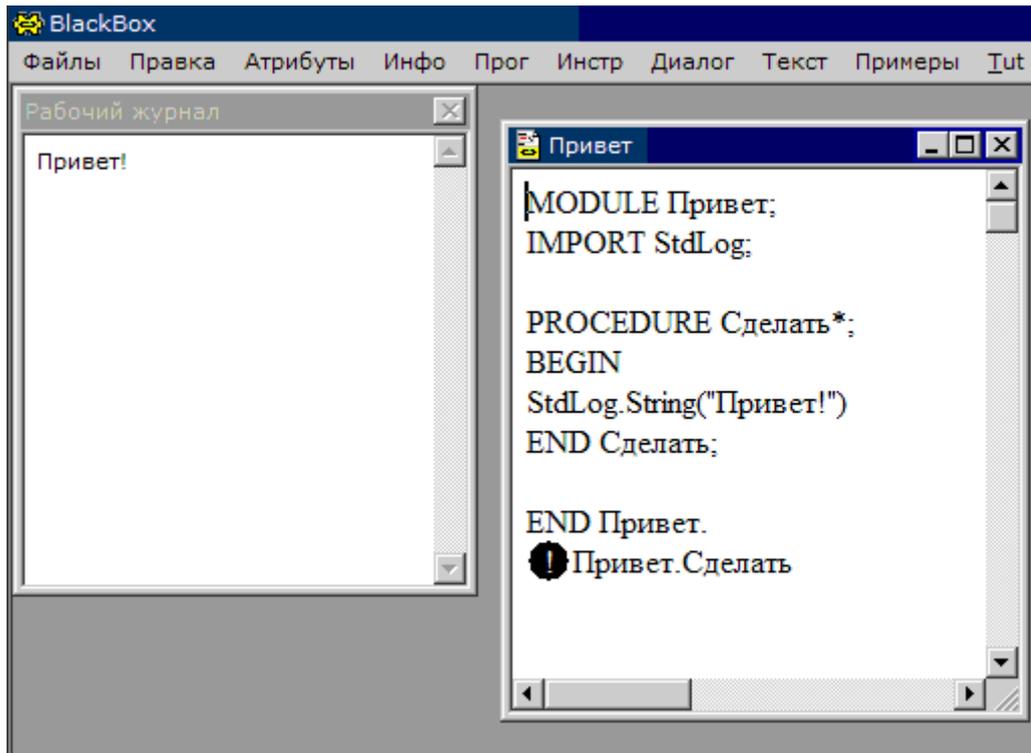
Еще одна причина, порождающая ситуацию 2 — имя процедуры после командира задано с ошибкой.

Если вышеприведенное правило твердо усвоено, и если программист работает (т.е. вносит изменения и т.п.) с единственным модулем, то объяснения, данные ниже, читать не обязательно: достаточно просмотреть только [последний раздел](#).

Для дальнейшего будем предполагать, что мы создали, выполнили и сохранили модуль Привет как это описано [ранее](#).

Запустив Блэкбокс, откроем модуль Привет (Ctrl+O или Файлы --> Открыть ...). Поскольку скомпилированный модуль хранится на диске в определенном месте (о котором знает Блэкбокс), можно немедленно выполнить процедуру Привет.Сделать, нажав на командир.

Получится примерно следующее:

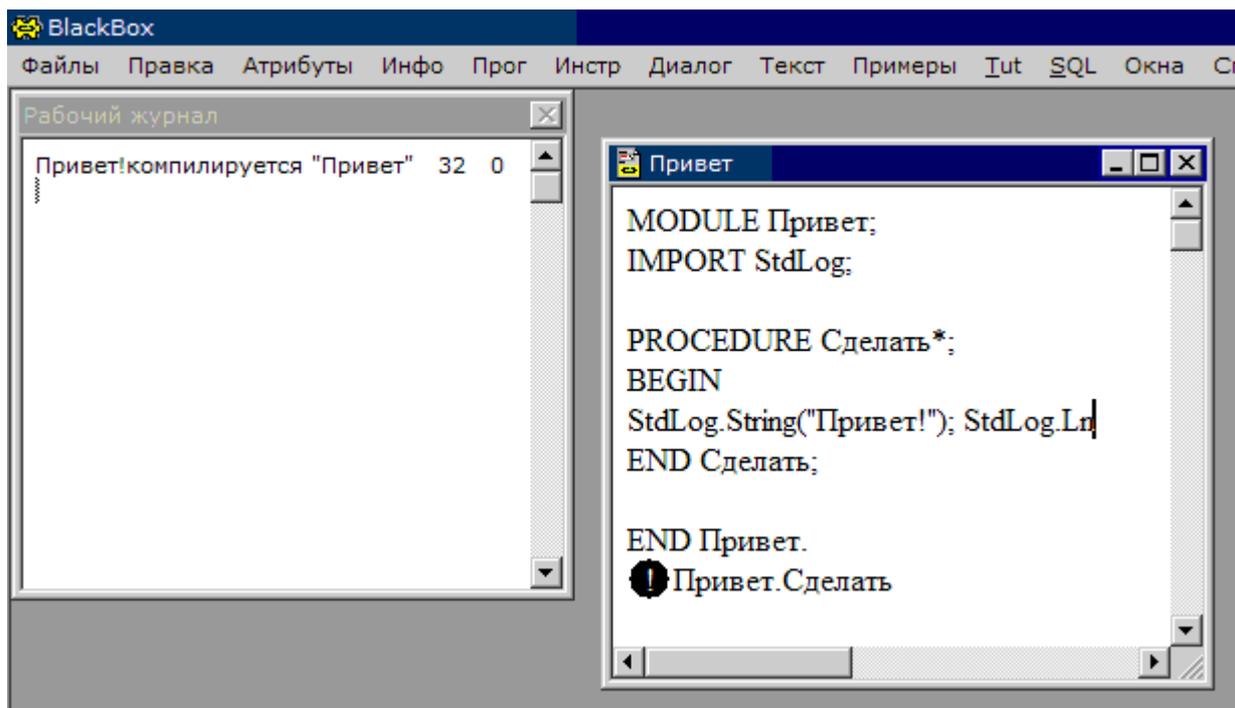


Теперь исправим текст модуля так, чтобы после печати слова "Привет!" происходил перевод строки (вспомним, что при повторных вызовах процедуры кликами по командире, "Привет!" печатался подряд на одной строке — см. [картинку](#)).

Для этого поставим курсор после закрывающей скобки и напечатаем следующую последовательность литер:

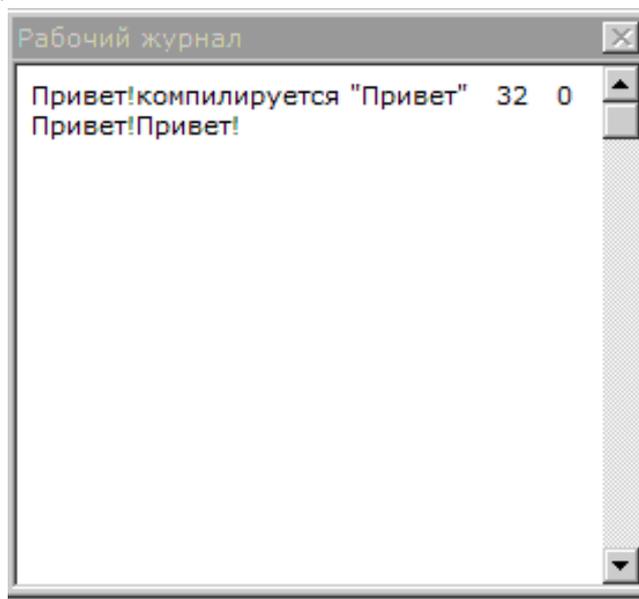
```
; StdLog.Ln
```

Это означает, что после вызова процедуры `StdLog.String` с фактическим параметром "Привет!" должна быть вызвана процедура `StdLog.Ln` (т.е. процедура `Ln`, находящаяся в модуле `StdLog`), не имеющая параметров. Эта процедура выполняет перевод строки в конце рабочего журнала, так что любая дальнейшая печать туда будет идти в новую строку. После этого модуль выглядит как в окошке справа:



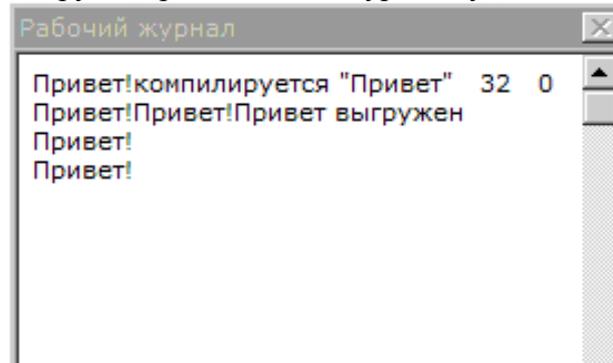
Сразу скомпилируем его (Ctrl+K). Рабочий журнал будет выглядеть примерно как в окошке слева.

Если теперь просто кликать по командиру, то будет вызываться *старая* версия нашего модуля — поскольку модуль остается в памяти, пока его не выгрузят, и именно версия в памяти вызывается в первую очередь. После пары кликов рабочий журнал будет выглядеть так:



Видно, что никакого перевода строки после восклицательного знака нет — вызывается старый вариант модуля (переход на новую строку после нуля произошел потому, что строчка оказалась слишком длинная; достаточно растянуть окошко в ширину, чтобы весь текст встал в одну строку).

Чтобы выгрузить старую версию модуля из памяти и вызвать новую, достаточно в момент клика по командиру нажимать на клавишу Ctrl. Сделаем это. Потом, отпустив Ctrl, кликнем по командиру еще раз. Рабочий журнал будет выглядеть примерно так:



Видим, что, во-первых, в последней строке журнала непосредственно после восклицательного знака (см. предыдущую картинку) произошла печать системного сообщения Блэкбокса 'Привет выгружен'.

Сообщение означает, что та (старая) версия модуля Привет, которая находилась в памяти, была выгружена.

Заметим, что после сообщения Блэкбокс делает перевод строки, так как будто после этого Блэкбокс действовал как при простом клике по командиру, т.е.

- (1) попытался вызвать процедуру Сделать в Привет, для этого он
- (2) проверил, есть ли в памяти модуль Привет;
- (3) не обнаружив его, Блэкбокс стал искать его на диске;
- (4) найдя его там, загрузил в память и после этого
- (5) вызвал процедуру Сделать.

Естественно, что на диске хранилась уже новая версия скомпилированного модуля Привет (с переводом строки), и именно эта новая версия была загружена и выполнена — т.е. была напечатана цепочка литер "Привет!", а затем переведена строка (это видно из того, что следующая печать идет с новой строки).

При втором клике Блэкбокс действовал по тому же алгоритму: сначала нашел в памяти модуль Привет (уже новый), а в нем — процедуру Сделать, и выполнил ее. Процедура напечатала "Привет!" уже с новой строки (перевод строки был выполнен в предыдущем вызове процедуры).

Чтобы убедиться в том, что после последнего восклицательного знака был сделан перевод строки, достаточно заставить Блэкбокс написать в рабочий журнал какое-нибудь сообщение, например, скомпилировав модуль еще раз (Ctrl+K).

Сообщение о компиляции будет напечатано с новой строки. Это следует сравнить с предыдущей картинкой, где сообщение '**компилируется "Привет" 32 0**' напечатано сразу после восклицательного знака, т.к. старая версия процедуры не делала перевод строки.

Все вышесказанное суммируется в следующем разделе.

Простейший цикл разработки программы

В простейших случаях (например, на школьных уроках) работают с единственным модулем небольшого объема (скажем, до 500 строк). В нем может быть несколько процедур, одна из которых (а может быть и несколько) будет экспортирована и поэтому доступна для непосредственного исполнения (например, с помощью [командира](#)).

О запуске простой программы уже [говорилось](#).

Разработка простейшей программы внешне выглядит примерно так:

Подготовительная стадия

1. Открыть новый документ. Выбрать для него имя и сохранить его с этим именем.
2. Написать в документе первый вариант модуля. (Первый вариант может и должен быть "скелетом", не содержащим никаких выполняемых операторов или даже объявлений.) Скомпилировать его (Ctrl+K с латинским K, или через меню: Прог --> Компилировать).
3. Устранять опечатки и синтаксические огрехи и нажимать Ctrl+K, пока модуль не скомпилируется.
4. В конце того же документа после точки, завершающей модуль, на новой строке вставить командир (Ctrl+Q или Инстр --> Вставить командир) и вслед за ним имя модуля, точка, имя процедуры, которую будем вызывать.
5. Вызвать процедуру, кликнув мышкой по командиру.

Основной цикл разработки

- i. Сделать очередное (небольшое) уточнение текста модуля.
- ii. Компилировать (Ctrl+K) и устранять ошибки, пока новая версия не скомпилируется без ошибок.
- iii. Если на данном шаге возможен частичный контроль правильности программы, то выполнить создаваемую процедуру. Для этого кликнуть мышкой по командиру, **одновременно нажимая клавишу Ctrl**, — нажатие на Ctrl заставит систему выгрузить из памяти старую версию модуля прежде чем вызвать новую.
- iv. Сохранить документ (Ctrl+S).
- v. Повторять шаги i-iv до тех пор, пока задача не будет решена.

Эта схема соответствует *разработке программы методом пошагового уточнения*; см. [заповеди грамотного программирования](#).

Некоторые заповеди грамотного программирования

1. Начинать проектировать модуль с интерфейсов процедур и формулировки того, что предполагается о состоянии вычислительного процесса при входе в процедуру, и что ожидается на выходе из нее (пред- и пост-условия).
2. Каждое уточнение модуля должно быть небольшим и обозримым.
3. Каждое уточнение должно приводить к синтаксически правильной программе.
4. После каждого уточнения программа должна быть семантически правильной в

той степени, в какой это требуется на данном шаге уточнения.

5. Почаще нажимать Ctrl+K!

Пункты 2 - 4 составляют фундаментальный метод разработки программ, известный как пошаговое уточнение — *step-wise refinement* (см. статью Вирта в *Comm. ACM*, Vol. 14, No. 4, April 1971).

Пункт 1 является исходной точкой для пошагового уточнения. Здесь еще говорят о проектировании, управляемом интерфейсами (*interface-driven design*), а также о проектировании по контрактам (*design by contract*).

Создание программы должно начинаться с задания интерфейсов процедур (прежде всего экспортируемых) и пред- и пост-условий — точно так же как решение любой задачи начинается с ее постановки.

Проектирование интерфейсов и задание пред- и пост-условий — ключевой элемент постановки задачи в программировании.

Выполнение **пунктов 2 - 4** позволяет сохранять интеллектуальный контроль за правильностью создаваемой программы, помогая устранять ошибки (как синтаксические огрехи вроде опечаток — п.3, так и смысловые — п.4) на возможно более ранней стадии.

Рекомендация **пункта 5** имеет смысл только в контексте **пп.2 - 4**: она бессмысленна, если процесс разработки не удовлетворяет требованиям **пп.2 - 4** (а также при работе с медленными компиляторами для сложных языков).

Здесь — одно из важнейших преимуществ, доставляемое минимализмом языка Оберон/Компонентный Паскаль и четкостью его дизайна, обеспечивающего исключительную быстроту компилятора. Быстрый компилятор подобного типа был впервые построен Виртом для предшественника Оберона — языка Модула-2 с систематическим применением принципа пошагового уточнения.

Правило 5 означает, в частности, что если компилятор обнаружил несколько ошибок, то устранять их нужно, как правило, по одной (обычно начиная с первой), каждый раз повторяя компиляцию (см. Сообщения компилятора об ошибках).

- Пошаговое уточнение считается самым полезным приемом систематического создания программ (Э.Дейкстра. *Дисциплина программирования*. М., Мир, 1978, с. 274). Весьма популярные в последнее время "гибкие" методологии (*Extreme Programming* и т.п.) во многом являются организационным воплощением идеи пошагового уточнения на уровне больших программных проектов.
- Чтобы применять методы типа пошагового уточнения требуется известная дисциплина и квалификация программистов — отсюда необходимость систематического "правильного" обучения. Однако в настоящее время признано, что резкое повышение эффективности программирования при использовании методов этой категории более чем окупает усилия, необходимые для их освоения.

Ф о р м а т д о к у м е н т о в Б л э к б о к с а

Когда мы пишем программу в окошке системы Блэкбокс, то мы создаем "документ" в особом формате.

Такие документы хранятся на диске в файлах с расширениями .odc (исторически это означает Oberon DoCument; напомним, что Компонентный Паскаль является уточнением Оберона).

В основном мы работаем с текстовыми документами, но есть и другие разновидности документов: например, вид и формат любых диалогов в Блэкбоксе (кроме стандартных диалогов операционной системы) также задаются документами (т.наз. формы — Forms).

- *Компилятор игнорирует любое форматирование (цвет, шрифт и т.п.), а также любые вставные визуальные объекты (о которых см. ниже), в том числе такие, которые сами по себе являются текстовыми документами.*

Импорт/экспорт стандартных текстовых файлов

Текстовые документы можно импортировать в документы Блэкбокса из файлов других форматов (см. диалог Файлы --> Открыть...), а также экспортировать в файлы других форматов (Файлы --> Сохранить как...).

Чаще всего приходится иметь дело с обычными текстовыми файлами в разных кодировках, экспорт и импорт которых производятся стандартными действиями, одинаковыми для всех программ под MS Windows.

- *Не путать экспорт/импорт файлов с экспортом/импортом процедур и т.п. из модулей!*

Блэкбокс допускает простой механизм добавления новых внешних форматов (т.наз. конвертеры — см. модуль Converters).

Например, несложно добавить конвертеры для кодировки KOI8-R и т.п. Однако написание нового конвертера может быть непростым делом, например, для файлов Microsoft Office (в стандартной дистрибуции Блэкбокса такие конвертеры отсутствуют; для переноса текстов между Блэкбоксом и программами MS Office с сохранением форматирования можно использовать формат rtf — Rich Text Format).

Вставные визуальные объекты (views или «вьюшки»)

Текстовые документы Блэкбокса могут содержать как разнообразное форматирование (см. Редактирование и форматирование), так и "плавающие" в тексте "картинки" — визуальные объекты не обязательно текстовой природы (общий термин Блэкбокса для обозначения таких объектов — view).

Такие документы называются "составными" (compound documents). В частности, в документ Блэкбокса можно вставлять обычным образом (Файлы --> Вставить) любые картинки, которые можно вставить в документы MS Word — картинки, нарисованные в программах типа Paint, цифровые фотографии и т.п.

В этом отношении документы Блэкбокса подобны, например, документам программы Microsoft Word, однако построены документы Блэкбокса гораздо более экономно и

гибко. При этом программист может добавлять к ним новые свойства и новое поведение ("функциональность"), хотя для этого требуется известная квалификация.

Примером специального визуального объекта, имеющего не только изображение, но и поведение, является так называемый командер (см. ниже) для запуска программ.

При клике мышкой по такому командиру Блэкбокс сообщает о факте клика командиру (точнее, процедуре, определенным образом связанной с ним) и последний может "среагировать" так, как его запрограммировали:

- командир смотрит на свой "контекст";
- определяет, что это текстовый документ;
- читает символы, следующие сразу за ним;
- анализирует, составляют ли они допустимое имя процедуры;
- отдает команду Блэкбоксу выполнить эту процедуру.

Что произойдет дальше, зависит уже не от командира, а от того, найдет ли Блэкбокс эту процедуру и что она сделает при вызове.

Другой пример — маркеры [ошибок компилятора](#), также являющиеся вставными визуальными объектами с поведением (см. Сообщения компилятора об ошибках).

В качестве визуального объекта можно также вставлять и другие текстовые документы.

Попробуйте скопировать любой фрагмент текста — например, с этой странички — и вставить его в Рабочий журнал Блэкбокса посредством команды меню Правка --> Вставить объект.

В этом случае вставляемый текстовый фрагмент будет содержаться внутри особого окошка фиксированного размера (этот размер можно менять мышкой; если окошка не видно, то кликните во вновь появившийся текст).

Окошко стоит в основном тексте как одна большая буква, не смешиваясь с основным текстом, но допуская редактирование (достаточно кликнуть мышкой внутрь такого фрагмента и что-нибудь напечатать).

Можно внутрь вставленного таким образом фрагмента вставить другие фрагменты, картинки и т.п. — никаких специальных ограничений для этого нет.

Использование рабочего журнала

Рабочий журнал (Log) представляет собой текстовый документ в стандартном формате Блэкбокса. Печать из процедур модуля [StdLog](#) всегда осуществляется в конец рабочего журнала.

Можно в любой момент очистить его командой меню Инфо --> Очистить журнал.

В рабочем журнале можно делать любое [форматирование](#) и редактирование: печатать туда текст с клавиатуры и менять его произвольным образом.

Поэтому его удобно использовать для оформления результатов не слишком большого объема. Установку положений табуляции для выравнивания можно делать стандартными средствами, имеющимися в Блэкбоксе (вставкой соответствующих линеек; см. модуль TextRulers и документацию к нему).

Приятно выделять части содержимого рабочего журнала цветом, например, при исследовании алгоритмов, когда производится выдача результатов нескольких модификаций одного и того же алгоритма, и т.п.

- Следует помнить, что рабочий журнал открывается как "временный" документ, т.е. при закрытии Блэббокса система не напоминает о его сохранении и его содержимое теряется.

Для сохранения информации достаточно открыть новый документ (Ctrl+N или Файл --> Новый), скопировать туда нужную информацию из журнала (например, с помощью стандартных комбинаций Ctrl+C, Ctrl+V) и затем сохранить получившийся новый документ.

Средства модуля StdLog

Модуль StdLog позволяет печатать информацию в рабочий журнал (об использовании рабочего журнала см. [здесь](#)). Ниже описаны простейшие и наиболее часто используемые средства этого модуля.

Напомним: чтобы использовать средства модуля StdLog в создаваемой программе, нужно импортировать его с помощью инструкции `IMPORT StdLog`. Удобно вводить сокращенное имя: `IMPORT Log := StdLog`, и обращаться к процедурам в виде `Log.Int(4)` и т.д. (см. [примеры](#)).

Печать всегда идет в конец рабочего журнала.

Печать основных типов

В StdLog есть простые процедуры для печати всех основных типов Компонентного Паскаля.

Каждая такая процедура имеет имя, соответствующее названию типа, и единственный параметр соответствующего типа, передаваемый по значению (т.е. на его место можно ставить любое выражение соответствующего типа, включая переменные и константы).

Чаще всего используются следующие процедуры:

- **Int** для печати значений любых целых типов (INTEGER, LONGINT, SHORTINT, BYTE). Сначала печатается пробел, потом, если число отрицательное, минус, потом значащие цифры.

Примеры:

`StdLog.Int (0)` напечатает цепочку '0',

`StdLog.Int (-123)` напечатает цепочку '-123',

`StdLog.Int (MIN(LONGINT))` напечатает цепочку '-9223372036854775808'
(минимальное длинное целое).

- **Real** для печати значений для вещественных типов (REAL, SHORTREAL). Печатаются все ненулевые значащие цифры после десятичной точки.

Примеры:

`StdLog.Real (123)` напечатает цепочку '123.0' (целое значение автоматически преобразуется в вещественное, которое и печатается),

`StdLog.Real (-123/1000000000)` напечатает цепочку '-1.23E-7'
(операция / имеет вещественный результат),

`StdLog.Real (Math.Pi)` напечатает цепочку '3.141592653589793' (число пи; получается вызовом функции Pi без параметров в модуле Math).

Реже используются процедуры:

- **Bool** для логических значений типа BOOLEAN,
- **Set** для типа SET,
- **Char** для отдельных литер CHAR. Эту процедуру можно использовать для печати любых символов в шрифтах стандарта Unicode:
 - StdLog.Char(''') напечатает простую одиночную кавычку (фактический параметр представляет собой простую одиночную кавычку, заключенную в простые двойные; простые — те, что есть на стандартной клавиатуре, в отличие от типографских кавычек в шрифтах Unicode; ср. примеры ниже),
 - StdLog.Char("") напечатает простую двойную кавычку (фактический параметр представляет собой простую двойную кавычку, заключенную в простые одиночные; литерные строки в Компонентном Паскале можно заключать как в двойные, так и в одиночные кавычки),
 - StdLog.Char(2018X) и StdLog.Char(2019X) напечатают открывающую и закрывающую одиночные кавычки (в шрифтах Unicode),
 - StdLog.Char(201CX) и StdLog.Char(201DX) напечатают открывающую и закрывающую двойные кавычки (в шрифтах Unicode).

Переход на новую строку и табуляция

Переход на новую строку в рабочем журнале осуществляется выполнением процедуры **StdLog.Ln** без параметров.

Символ табуляции печатается в помощью процедуры **StdLog.Tab** без параметров. Символы табуляции особенно полезны в сочетании со средством Блэкбокса под названием "линейки" — "rulers" (см. документацию к модулю TextRulers).

Печать литерных цепочек

Очень полезна процедура **StdLog.String**, единственный параметр которой может быть задан цепочкой литер, например, StdLog.String("Привет!").

Фактический параметр может быть также задан произвольным литерным массивом (ARRAY OF CHAR), тогда значение массива интерпретируется как цепочка литер по правилам Компонентного Паскаля (см. [Сообщение о языке ...](#) и [разд. 6.6](#)).

В модуле есть процедуры, позволяющие управлять представлением целых и вещественных чисел (IntForm и RealForm), а также процедуры, выполняющие ряд других функций.

Полную информацию о модуле можно получить, выбрав (например, двойным кликом мышкой) имя модуля StdLog в любом открытом документе Блэкбокса (например, напечатав идентификатор StdLog в рабочем журнале), и выполнив команду меню Инфо --> Документация.

Если нужно только посмотреть интерфейс модуля (видимые снаружи объекты — сигнатуры процедур и т.п.), то выбрав его имя, достаточно нажать Ctrl+D (из меню: Инфо --> Клиентский интерфейс).

Документация в Блэкбоксе

Документация на русском языке

На момент написания данного текста закончен полный первичный перевод на русский язык всей документации Блэкбокса. Этот перевод включен во все конфигурации Блэкбокса от проекта Информатика-21 вместе с соответствующими командами меню.

Следует иметь в виду, что редактирование столь большого объема документов, насыщенного нестандартной терминологией и выполненного большой командой переводчиков, требует больших усилий и потребует еще какого-то времени, чтобы добиться согласованности терминологии. Но даже не вполне отшлифованный перевод является большим подспорьем.

Справки по интерфейсам модулей

Наиболее часто приходится справляться насчет деталей интерфейсов модулей, которыми требуется воспользоваться и назначение которых уже известно.

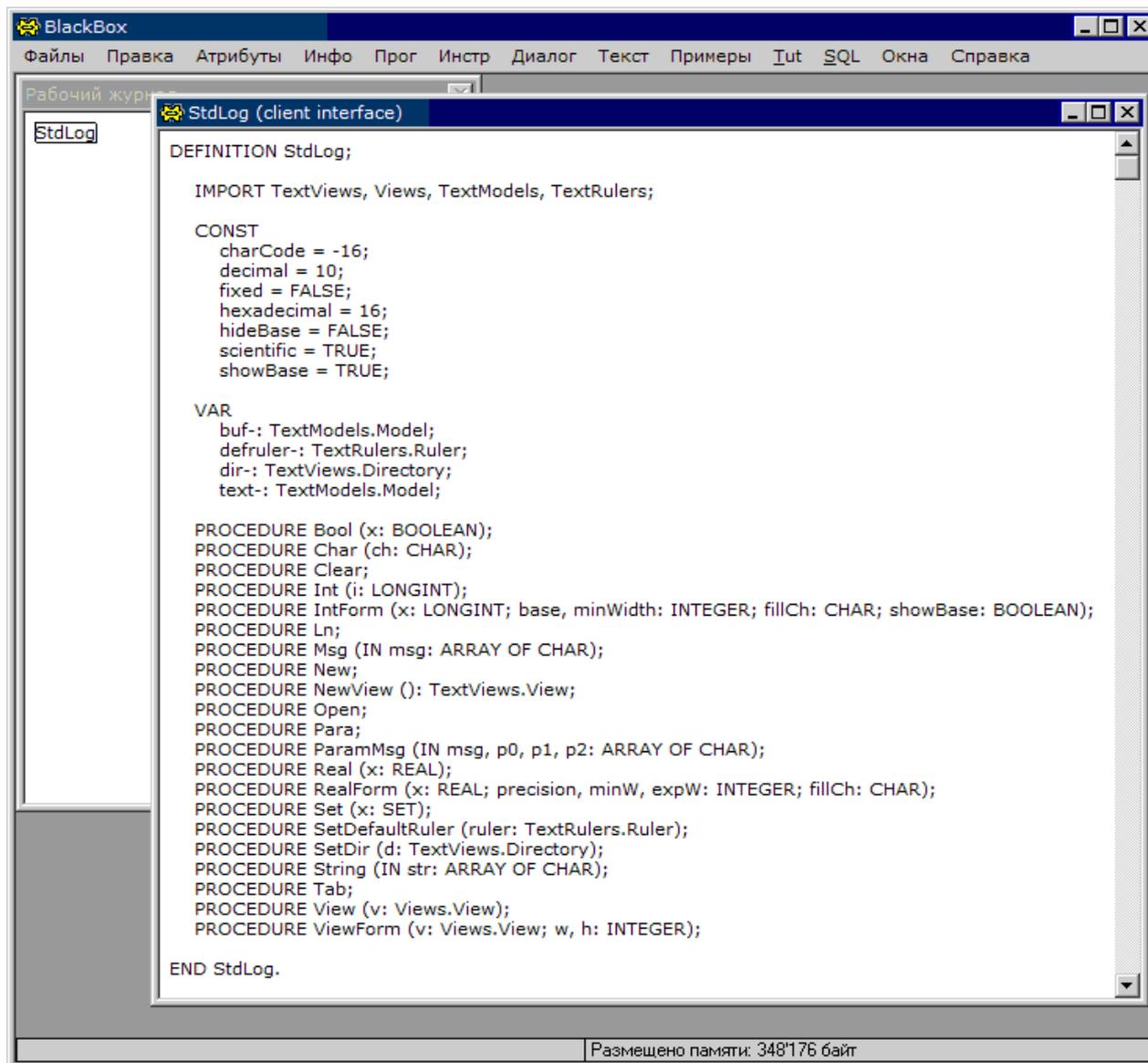
Например, модуль StdLog содержит средства печати в рабочий журнал, причем предусмотрены процедуры для значений разных основных типов языка (INTEGER, REAL и т.д.).

Достаточно в любом открытом текстовом документе Блэкбокса (например, в рабочем журнале) написать идентификатор модуля, выбрать его (например, дважды кликнув по нему мышкой) и нажать Ctrl+D (из меню: Инфо --> Клиентский интерфейс; в оригинале Info --> Client Interface). В ответ на это Блэкбокс найдет модуль, выяснит его интерфейс и представит последний в новом окошке в виде текстового документа:

На рисунке ниже показаны внешние интерфейсы модуля:

1. все модули, которые импортируются в данный (оператор IMPORT);
2. экспортируемые константы (раздел CONST);
3. экспортируемые переменные (раздел VAR); все они в данном случае экспортируются только для чтения (минусы в качестве значков экспорта после идентификаторов);
4. экспортируемые процедуры вместе с полными описаниями формальных параметров.

В общем случае могут экспортироваться определенные в модуле типы данных (на жаргоне объектно-ориентированного программирования — классы) вместе со связанными с ними процедурами (методами).



После изучения интерфейсов можно искать — если нужно — документацию по конкретной процедуре.

Документация по конкретному модулю

Чтобы открыть документацию по конкретному модулю (если таковая имеется), достаточно в любом открытом текстовом документе Блэкбокса (например, в рабочем журнале) написать идентификатор модуля.

Выбрать его (например, дважды кликнув по нему мышкой) и выполнить команду меню Инфо --> Документация (в оригинале Info --> Documentation).

Блэкбокс найдет соответствующий текст и откроет его в отдельном окошке в режиме браузера; в этом режиме "работают" гиперссылки (см. модуль StdLinks), обычно отмеченные голубым цветом и подчеркиванием.

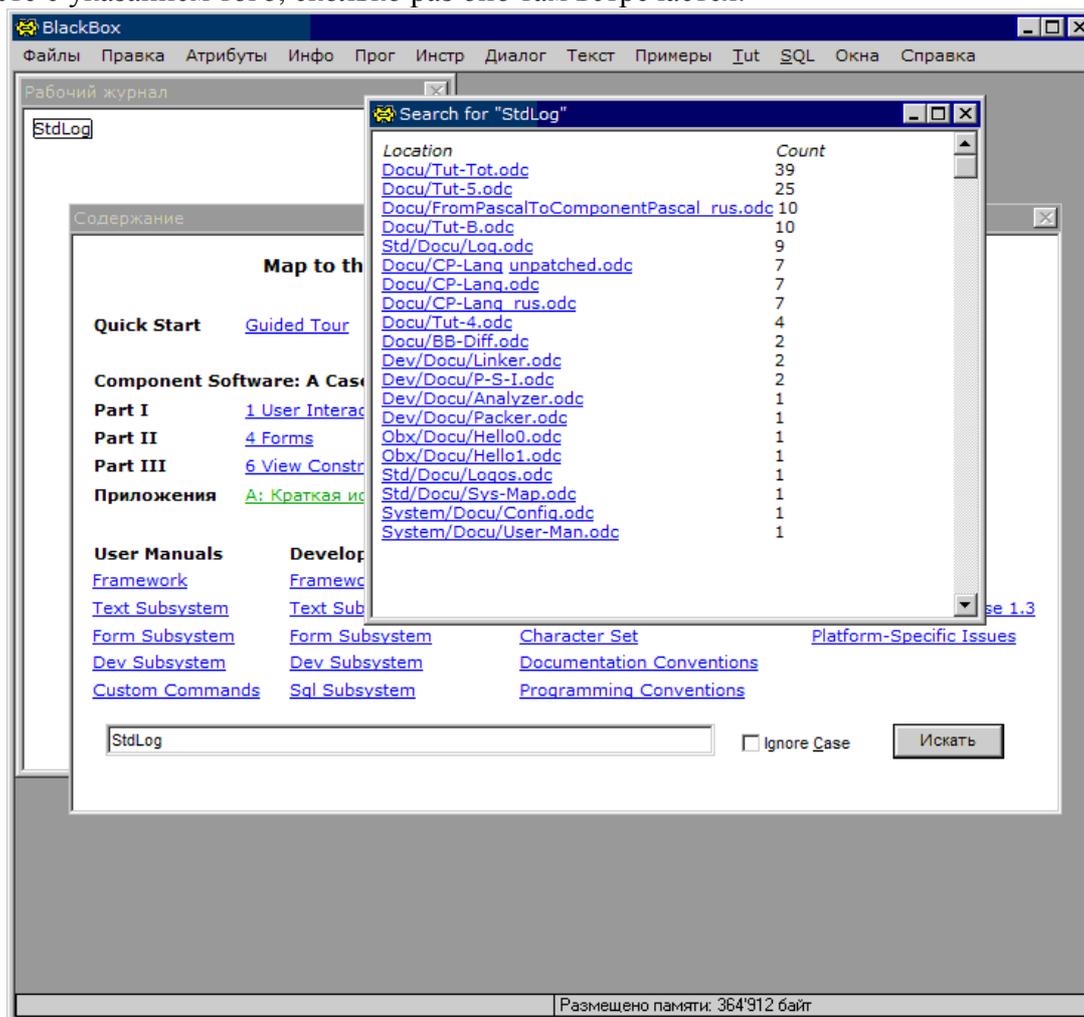
Об используемых в документации соглашениях можно прочесть в разделе документации Documentation Conventions (нажать F1 и искать там).

Общая документация и поиск в ней

Нажатие F1 в Блэкбоксе открывает (в режиме браузера) документ с гиперссылками, по которым можно найти всю документацию, поставляемую в оригинальной дистрибуции.

В русифицированной версии там же добавлены гиперссылки на переведенные документы. Там же внизу есть мини-диалог для поиска в документации.

Например, напечатав там имя модуля StdLog и кликнув по кнопке Искать, получим новое окно с гиперссылками на все модули в папках Doci, где встречается это имя, вместе с указанием того, сколько раз оно там встречается:



Примеры программ

1. Примеры программ "школьного" типа даны в подсистеме i21примеры (см. исходные тексты в папке BlackBox\i21примеры\Mod\).
2. В документации Блэкбокса (F1) содержится целая книга с примерами.
3. Несколько полезных текстов есть в папке BlackBox\i21sys\Mod\.
4. Большое количество очень полезных примеров поставляется в подсистеме Obx (см. документы в папке BlackBox\Obx\Mod\).
5. Исходники Блэкбокса — богатейший источник нетривиальных примеров.

Работа с текстами программ

Форматирование текста

Программисты не просто постоянно читают свои и чужие программы на бумаге и на экране монитора, они их тщательно исследуют символ за символом, проводя за этим занятием огромную часть своего рабочего времени.

Систематическое использование [отступов](#), следование определенным [соглашениям](#) по использованию пробелов, выбор [шрифта](#), использование [цвета](#) и т.п. — все это настолько существенно облегчает работу при минимуме усилий, что не может считаться второстепенными факторами.

Примерно треть человеческого мозга вовлечена в обработку зрительной информации — глупо "отключать" от мыслительного процесса эту треть.

Прежде всего отметим, что текстовый редактор Блэкбокса является весьма мощным:

- Во-первых, его команды следуют стандартам MS Windows (использование Shift+стрелки для выделения фрагмента текста, комбинации клавиш Ctrl+X, Ctrl+C, Ctrl+V для манипуляций с фрагментами текста, и т.д.).
- Во-вторых, **допускается произвольное количество отмен (Ctrl+Z или Правка --> Отменить)**, так что редактируемый документ всегда можно вернуть в то состояние, в котором он был на момент последнего выполнения команды Файлы --> Сохранить (Ctrl+S).
- В-третьих, документы Блэкбокса являются [составными](#).
- В-четвертых, все средства редактора доступны из программ, написанных на Компонентном Паскале, т.е. сам Компонентный Паскаль играет роль полноценного и весьма эффективного макроязыка для редактора.

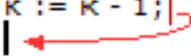
Отступы

Форматирование текста программ с помощью отступов — важнейший способ выявления структуры программ. См., например, [простейшую программу](#), в которой имеется два уровня отступов: содержимое модуля сдвинуто на один отступ относительно открывающего (MODULE Привет;) и закрывающего (END Привет.) операторов, а тело процедуры (состоящее из одного оператора StdLog.String("Привет!")) сдвинуто еще на один отступ относительно "скобок" BEGIN и END. См. также другие примеры программ на Компонентном Паскале.

Блэкбокс предоставляет два простых и удобных средства для управления отступами. (Предполагается, что отступы делаются с помощью табуляции (клавиша Tab), а не пробелами.)

Во-первых, если мы только что закончили печатать очередную строку и курсор стоит в конце строки, то нажатие клавиши Enter не только приведет к созданию новой строки, но и вставит в ее начале столько же символов табуляции, сколько их было в начале у предыдущей:

```
PROCEDURE КопироватьХвост ( VAR откуда, куда: ARRAY OF INTEGER );  
  VAR о, к: INTEGER;  
  BEGIN  
    о := LEN( откуда ); к := LEN( куда );  
    WHILE о >= 0 DO  
      куда[ к ] := откуда[ о ];  
      к := к - 1;  
    END  
  END КопироватьХвост;
```



На картинке переход курсора показан красной стрелкой.

Во-вторых, нередко возникает необходимость менять уровень отступа для группы строк (например, после того, как группа строк была вынесена во вновь создаваемую процедуру). Это удобно делать с помощью функциональных клавиш F11 и F12.

Сначала с помощью мышки выделим нужные строки (достаточно кликнуть мышкой в любое место первой строки и, не отпуская кнопки, "потянуть" курсор мышки в любое место последней строки). После этого нажатие F11/F12 будет сдвигать выделенные строки влево/вправо на один отступ (в начале строк будет удаляться/вставляться один символ табуляции).

Соглашения по оформлению программ

Хорошие соглашения по оформлению программ описаны в документации Блэкбокса (F1, Programming conventions), где даны примеры хорошего и плохого оформления отдельных элементов программ.

Кроме того, можно поизучать тщательно написанные примеры программ в подсистеме Form (документы в папке Form\Mod\).

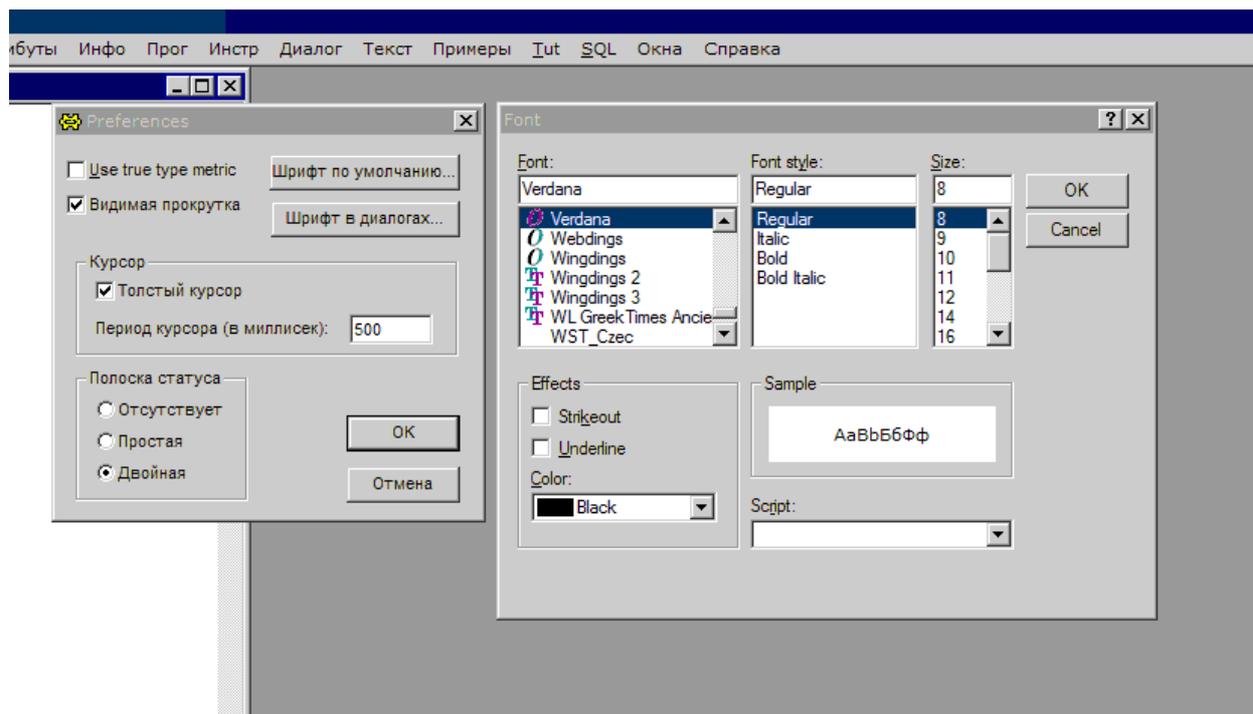
Шрифт по умолчанию

Блэкбокс использует шрифт Ариал по умолчанию во всех вновь создаваемых документах. Этот выбор обусловлен соображениями переносимости.

На экране лучше смотрится, например, шрифт Verdana, особенно при малой величине букв (скажем, 8 пунктов — ведь на экране желательно уместить побольше текста, поэтому выбирают шрифт помельче).

В школе при работе с небольшими программами можно предпочесть более крупный шрифт — например, 12 пунктов.

Чтобы поменять шрифт, используемый по умолчанию в текстовых документах Блэкбокса, нужно выполнить команду меню Правка --> Настройки..., после чего откроется примерно такой диалог, как показан в левой части картинки:



Клик по кнопке Шрифт по умолчанию... вызовет стандартный диалог операционной системы (показан справа). Выбрав шрифт и размер, кликнуть ОК в обоих диалогах.

Полоску статуса в школьных условиях можно предпочесть "простую" — например, чтобы видеть полностью сообщения компилятора (их можно сделать более подробными, отредактировав файл Dev\Rsrc\Errors.odc).

При серьезной работе полоску статуса лучше выбрать двойную (как показано на левом диалоге). В левой половине Блэкбокс пишет сообщения компилятора и т.п., а в правой сообщает о размере памяти, распределенной под данные (динамические структуры данных, созданные загруженными модулями, например, для открытых документов, и т.п.).

Использование цвета и т.п.

Цвет — мощная помощь в разметке визуальной информации, но от излишней пестроты "рябит в глазах", что тоже не хорошо.

В Компонентном Паскале — в отличие от менее продуманных языков — нет нужды выделять ключевые слова цветом, т.к. они пишутся большими буквами (см. Ключевые слова из прописных букв). Это позволяет эффективно задействовать цвет для более полезных целей.

Например, удобно использовать цвет для новых, еще не проверенных вставок в программный текст. Можно определенным цветом выделять комментарии, и т.п.

Хорошо выделять, скажем, красным цветом символы экспорта (звездочки и минусы после идентификаторов) в виду их большой важности (см. Примеры программ на Компонентном Паскале).

Цвет (как и шрифт, размер, и т.п.) выделенного фрагмента текста легко менять командами меню (Атрибуты --> Красный и т.д.).

Кроме того, рекомендуется использовать курсив для комментариев, а жирным шрифтом выделять идентификаторы процедур в их заголовках, а также операторы

RETURN и **EXIT**, т.к. последние могут передавать управление за пределы программных конструкций, внутри которых они находятся (см. Примеры программ...).

Копирование атрибутов текста

Предположим, Вы уже задали определенные атрибуты для некоторого фрагмента текста (цвет, шрифт, размер, курсив ...), и теперь хотите, чтобы другой фрагмент имел точно такие же атрибуты.

Этого можно добиться стандартным способом (таким же, как, например, в MS Word): скопировать атрибуты одного фрагмента текста (Ctrl+Shift+C, или из меню Правка --> Копировать атрибуты), а затем выделить нужный фрагмент и применить к нему скопированные атрибуты (Ctrl+Shift+V или Правка --> Применить атрибуты).

Можно повторять этот процесс, применяя скопированные атрибуты к другим фрагментам текста.

Наконец, можно открыть несколько окон на один и тот же документ (Окна --> Новое окно). Каждое окно может показывать свой фрагмент текста независимо от остальных окон. Однако следует помнить, что закрытие первичного окна (первого окна, в котором был показан документ) приводит к закрытию всех вторичных окон; это поведение отличается от некоторых других программ.

Ключевые слова из прописных букв

В Компонентном Паскале ключевые слова должны быть набраны большими буквами. Чтобы облегчить эту задачу, в пакете русификации предусмотрено специальное средство: нажатие клавиши F5 приводит к тому, что:

1. Все маленькие буквы (латинские и/или русские) влево от курсора до первого символа, не являющегося маленькой буквой (чаще всего это пробел), преобразуются в большие. Например: then превращается в THEN, а a1bc --> a1BC.
2. Если превращаемое слово является первым ключевым словом оператора IF, WHILE, PROCEDURE ... и т.п., то происходит подстановка всего "скелета" из ключевых слов для данного оператора.

Вот пример такой подстановки (слева — до, справа — после подстановки; положение курсора показано вертикальной черточкой):

```
procedure|      PROCEDURE |( );
                VAR
                BEGIN
                END ;
```

При этом пустая строка между BEGIN и END содержит ровно столько (невидимых) символов табуляции, чтобы начало печатаемого в ней текста было углублено на один отступ вправо относительно слов PROCEDURE, BEGIN и END — на ту же глубину, что и ключевое слово VAR.

Обратите внимание, что курсор после подстановки стоит так, чтобы можно было немедленно печатать идентификатор-имя процедуры.

Для тех — и только тех, — кто чувствует себя уверенно с Компонентным Паскалем и Блэкбоксом:

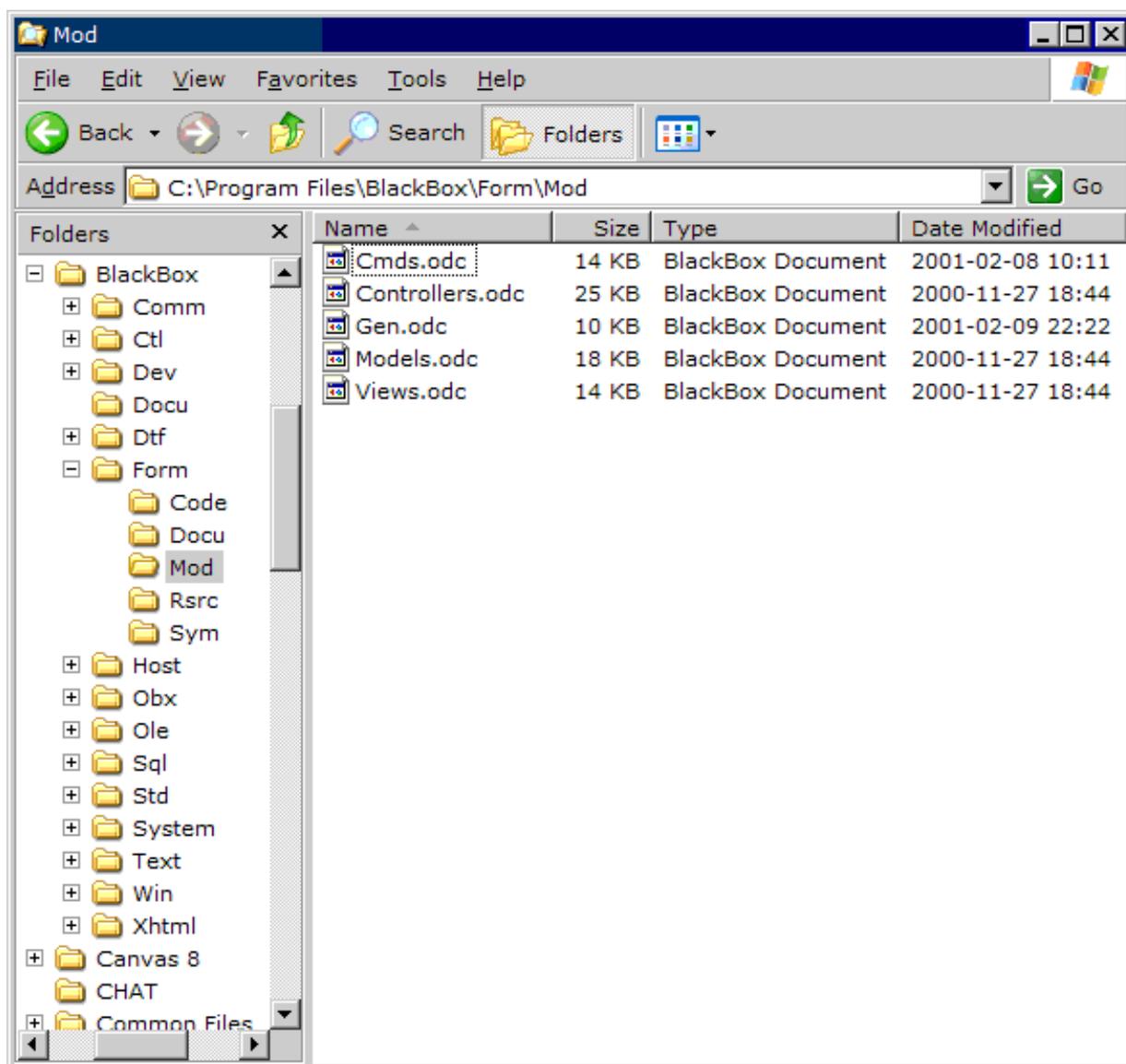
Текст процедуры, осуществляющей такие подстановки, содержится в модуле `i21sysEdit` (файл `i21sys\Mod\Edit.odc`).

Имя процедуры — `CapitalizeExpand` (см. в теле процедуры оператор `IF` с большим количеством проверок типа `ELSIF string$ = ...`).

Имея исходный текст в качестве образца, процедуру нетрудно модифицировать так, чтобы добавить новые подстановки или подправить уже имеющиеся под свой вкус. После этого достаточно скомпилировать ее и [перезагрузить модуль](#) `i21sysEdit` (о перезагрузке модулей см. Загрузка и перезагрузка модулей).

Организация файлов Блэкбокса

Будем предполагать, что Блэкбокс был установлен в первичную директорию, заданную по умолчанию — `C:\Program Files\BlackBox\` — структура которой и показана на следующей картинке (окно Проводника/Windows Explorer):



Первый загружаемый EXE-файл (BlackBox.exe) находится в папке BlackBox, но это нам сейчас не важно (Здесь и ниже будем указывать имя папки сокращенно, опуская C:\Program Files\).

Все файлы, с которыми работает Блэкбокс, распределены по так называемым **подсистемам**, каждой из которых соответствует отдельная папка в папке BlackBox.

Имя подсистемы совпадает с именем соответствующей папки: Comm, Ctl, ... Xhtml. (Например, подсистема Comm содержит программы для коммуникации с внешним миром по Интернету в соответствии с протоколом TCP/IP; подсистема Text содержит средства работы с текстами — правка, форматирование и т.п.; и т.д.)

Имя каждого модуля (идентификатор, стоящий в программном тексте после ключевого слова MODULE) в данной подсистеме должно начинаться с имени подсистемы (см. об этом [ниже](#)).

В каждой подсистеме могут иметься еще до **5 стандартных папок**, как это показано на картинке для подсистемы Form:

- 1) В папке **Mod** (modules) хранятся **исходные тексты модулей**, входящих в подсистему. Это документы Блэкбокса в его стандартном формате; файлы имеют расширение .odc.
Например, для подсистемы Form имеется 5 исходников, как показано на картинке. Каждый файл должен содержать текст одного модуля в начале текста. После точки, закрывающей модуль, может быть что угодно — любые тексты, картинки и т.п. Имя модуля должно состоять из имени подсистемы и имени соответствующего файла (см. [ниже](#)).
Поэтому можно, например, говорить о модуле Cnds подсистемы Forms: программный текст этого модуля хранится в файле Cnds.odc в папке BlackBox\Forms\Mod\.
- 2) В папке **Code** (code files) хранятся т.наз. **кодovые файлы** (расширения .ocf), содержащие оттранслированные машинные коды для каждого из модулей данной подсистемы.
Например, модулю Cnds, текст которого хранится в файле Cnds.odc в папке Mod, соответствует кодovый файл Cnds.ocf в папке Code.
Кодовые файлы Блэкбокса аналогичны динамически загружаемым библиотекам (DLL) ОС MS Windows, но, в отличие от последних, построены предельно экономно.
- 3) В папке **Sym** (symbol files) хранятся т.наз. **символьные файлы** (расширения .osf), содержащие информацию о внешних интерфейсах соответствующих модулей.
Например, модулю Cnds соответствует символьный файл Cnds.osf в папке Sym. Символьные файлы Блэкбокса аналогичны header-файлам языков C/C++, но, в отличие от последних, создаются компилятором автоматически.
Это исключает целый класс серьезных ошибок и сильно повышает надежность как самого Блэкбокса, так и создаваемых в нем программ. Более того, как компилятор, так и загрузчик проверяют согласованность соответствующих кодovых и символьных файлов, и обмануть систему нелегко.
 - Чтобы пользоваться процедурами и другими ресурсами, предоставляемыми каким-то модулем, достаточно иметь

соответствующие кодовый и символьный файл. Исходный текст не нужен.

Например, почти все подсистемы, входящие в дистрибуцию Блэкбокса, не содержат даже папок Mod. (Полные исходные тексты подсистемы Form включены в дистрибуцию в качестве примеров.) Вообще чтобы добавить новую подсистему в Блэкбокс, достаточно скопировать в папку Блэкбокс соответствующую папку, содержащую, как минимум, кодовые и символьные файлы. После этого можно сразу пользоваться средствами соответствующих модулей.

- Папки Code и Sym создаются Блэкбоксом автоматически. Так что программисту достаточно создать папку Mod, а в ней — файлы с исходными текстами модулей.
- 4) В папке **Docu** (documentation) хранится **документация**. Подразумевается, что для каждого модуля подсистемы есть соответствующий файл в папке Docu. Например, модулю Cmds соответствует файл с документацией Cmds.odc в папке Docu. Разумеется, там может храниться и дополнительная документация, описывающая подсистему в целом, а не отдельные модули.
- О том, как организована документация в Блэкбоксе, см. соответствующий раздел. При программировании для своих личных нужд редко возникает необходимость создавать папку Docu.
- 5) В папке **Rsrc** (resources) обычно хранятся **вспомогательные ресурсы** — формы диалогов, тексты сообщений об ошибках и т.п.
- Например, сообщения компилятора об ошибках хранятся в виде текста в файле Errors.odc в папке Rsrc подсистемы Dev.
- Русификация этих сообщений сводилась к замене английских сообщений на русские в этом тексте. Заметим, что благодаря чрезвычайной легкости создания графических диалогов в Блэкбоксе нередко удобно создавать диалоги для вспомогательных задач. Поэтому создавать папку Rsrc приходится чаще, чем папку Docu.
- Все файлы, описанные выше, могут быть открыты в Блэкбоксе. Что при этом будет показано, зависит от типа файла.

Программист, проектируя новую подсистему, волен придумать в ней дополнительные папки для своих нужд.

В папке BlackBox могут содержаться две папки Code и Sym, не входящие ни в какую подсистему. В них хранятся кодовые и символьные файлы модулей, имена которых не следуют соглашениям Блэкбокса.

Соглашения об именах модулей и именах соответствующих файлов в Блэкбоксе

В некоторых важных случаях Блэкбокс должен уметь найти исходный текст модуля. Например, при аварийной остановке тестируемой программы Блэкбокс пытается открыть исходный программный текст модуля, в котором произошла ошибка, и показать оператор, вызвавший ее. Это будет возможным, если следовать соглашениям об именах модулей и файлов.

В Блэкбоксе принято соглашение, что имя модуля содержит в качестве префикса имя своей подсистемы.

Блэкбокс определяет наличие префикса по следующему п р а в и л у:

Если в идентификаторе модуля прописная (большая) буква следует за строчной (малой) буквой, цифрой или символом подчеркивания (который разрешен в идентификаторах), то все, что предшествует этой прописной букве считается именем подсистемы.

Часть идентификатора, начинающаяся с этой прописной буквы, считается именем файла данного модуля (т.е. используется для имен кодовых и символьных файлов, а также для поиска документации и исходных текстов).

Заметим еще, что в Блэкбоксе *имена модулей* по соглашению начинаются с *прописной буквы*.

П р и м е р :

Имя модуля CommObxStreamsClient разбивается на префикс Comm (имя подсистемы) и собственно имя модуля и соответствующих файлов — ObxStreamsClient.

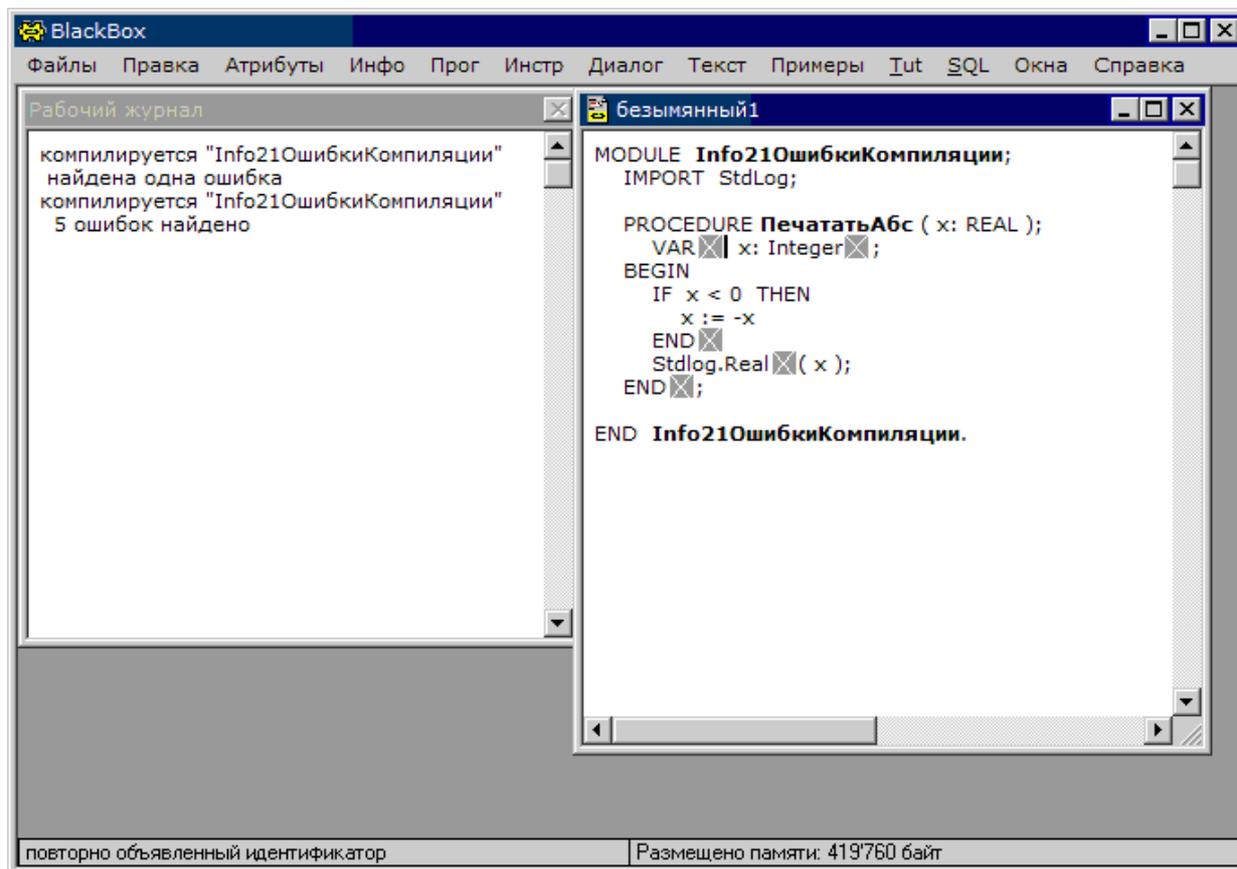
Тогда:

исходный текст хранится в Comm/Mod/ObxStreamsClient.odc,
кодовый файл — в Comm/Code/ObxStreamsClient.ocf,
а символьный файл — в Comm/Sym/ObxStreamsClient.ocf.

Компилирование и отладка

Сообщения компилятора об ошибках

Вот что получается, если попытаться скомпилировать командой Ctrl+K простой модуль с синтаксическими ошибками (правое окошко):



В рабочем журнале слева есть две записи о двух попытках компиляции с указанием количества ошибок, найденных компилятором каждый раз. Только вторая запись имеет отношение к текущему состоянию окошка с текстом модуля справа: каждой из 5 ошибок соответствует черный квадратик.

По завершении компиляции курсор стоит сразу после первой ошибки (вертикальная линия). В самом низу в статусной полоске слева видно сообщение компилятора о данной ошибке: "повторно объявленный идентификатор". Действительно, идентификатор `x` уже был описан в качестве формального параметра процедуры.

Устраняйте самую первую ошибку — и сразу компилируйте снова, нажимая Ctrl+K (см. [заповеди](#).)

Не следует пытаться сразу устранить все ошибки: компилятор после первой ошибки может "сбиться" и неправильно интерпретировать дальнейший текст — например, увидеть ошибки там, где их на самом деле нет, или пропустить настоящие ошибки. Разумеется, некоторые очевидные ошибки можно устранить одновременно.

Скорость компиляции достаточно велика, чтобы при такой методе неудобств не возникало.

Чтобы все-таки посмотреть, что за ошибки нашел компилятор в остальных случаях, достаточно кликнуть (один раз) мышкой по любому черному квадрату: в левой части статусной полосы возникнет соответствующее сообщение.

Вот пять этих сообщений для пяти ошибок, показанных на картинке (в квадратных скобках — пояснения):

1. повторно объявленный идентификатор [Смысл этого сообщения уже обсуждался выше.];
2. необъявленный идентификатор [Ключевые слова должны целиком писаться заглавными буквами; здесь явно имелось в виду INTEGER.];
3. пропущена ";" [Этот END закрывает оператор IF. Обычно после каждого оператора стоит точка с запятой, а опустить ее можно только в том случае, если дальше следует другое END.];
4. идентификатор не обозначает тип записей [Пример, когда компилятор "сбился": в начале конструкции подразумевался идентификатор StdLog с заглавной L, обозначающий импортируемый модуль (см. оператор IMPORT в начале модуля), а вся конструкция StdLog.Real должна была обозначать вызов процедуры Real, содержащейся в этом модуле.

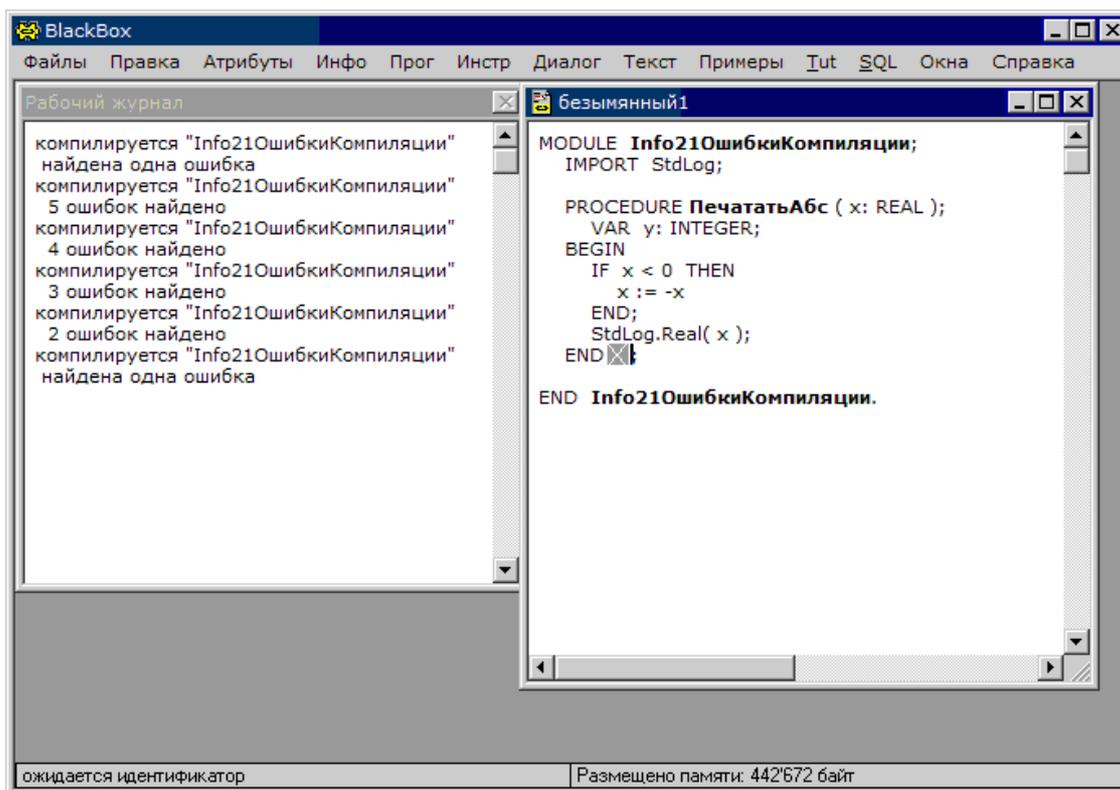
Но компилятор воспринял конструкцию как левую часть присваивания полю Real переменной-записи Stdreal, что синтаксически выглядит так же. Поскольку Stdreal не была объявлена как запись, то и полей у нее быть не может.

В подобных случаях (их в Компонентном Паскале немного) для каждого сообщения компилятора полезно было бы иметь список типичных ошибок, способных породить данное сообщение. Пока, к сожалению, такого списка не составлено.]

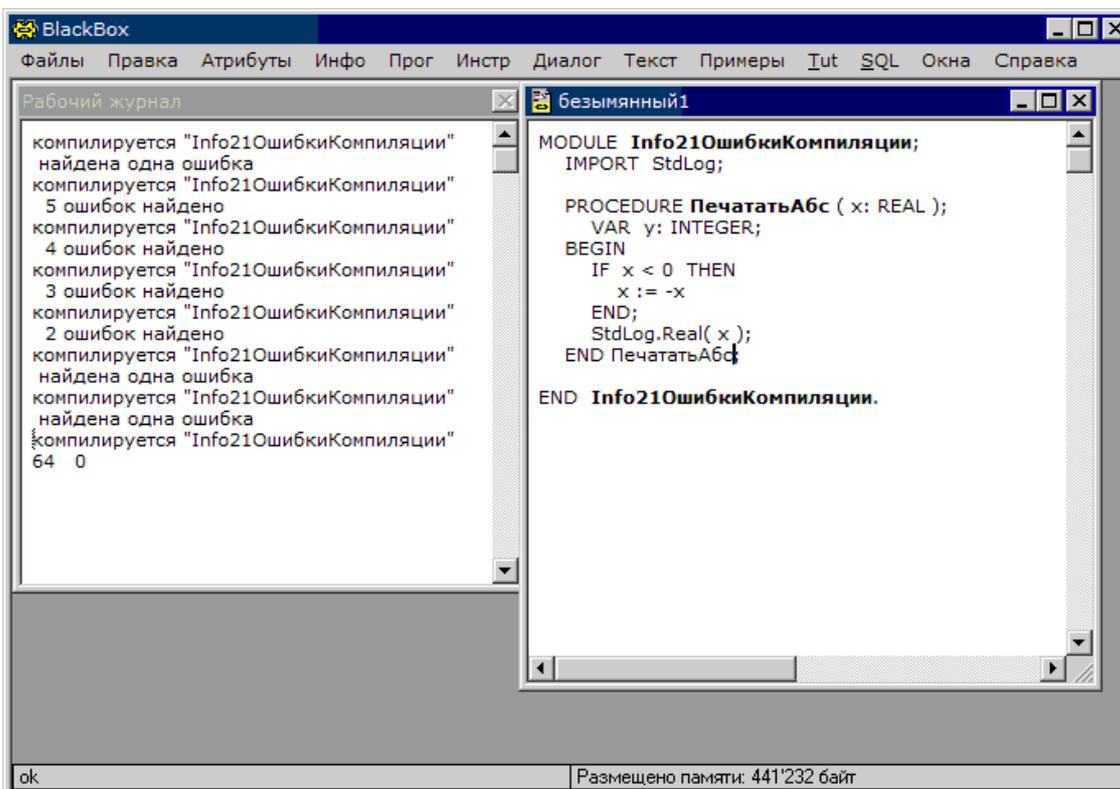
5. ожидается идентификатор [В Компонентном Паскале/Обероне после END, закрывающего тело процедуры, должно быть повторено ее имя.]

После устранения первых трех ошибок (каждый раз производилась компиляция, что видно из записей в рабочем журнале), будет видно, что после вставки точки запятой (ошибка 3 в первоначальном списке), компилятор точнее интерпретирует данную ошибку (зато он перестал видеть пятую ошибку из списка).

Исправляя Stdlog на StdLog и компилируя, получаем:



Компилятор снова увидел ошибку 5 из списка. Остается ее исправить (достаточно скопировать в это место имя процедуры) и нажать Ctrl+K:



В рабочем журнале теперь нет сообщения о найденных ошибках — зато указан размер получившегося машинного кода (64 байта; второе число — 0 — относится к

глобальным переменным модуля, объявляемым до всех процедур модуля; таких переменных в данном примере нет).

- Никакой классификации ошибок компиляции на "предупреждения", "ошибки", "критические ошибки" и т.п., в Блэкбоксе не делается: безусловная защита от больших и мелких ошибок программиста является высшим приоритетом. Все потенциальные источники ошибок, которые могут быть обнаружены компилятором, должны быть устранены как можно раньше: почистить код, устранив "предупреждения", выходит несравненно дешевле, чем вылавливать в большой программе ошибки, индуцированные «предупреждениями» — или проводить расследование, выясняя причины катастрофы стоимостью сотни миллионов долларов (например катастрофа ракетносителя Ариан-5).

Загрузка и перезагрузка модулей

Минимальные сведения

Нередко (а в школьном курсе практически всегда) программист работает с единственным модулем. Последовательность действий для такого случая и главный практический рецепт (нажимать Ctrl при клике по командиру) уже были описаны в разделе «Простейший цикл разработки программы».

Информация, данная ниже, нужна лишь в более сложных ситуациях, когда работа идет с несколькими модулями одновременно.

Когда происходит загрузка модуля в оперативную память

Откомпилированные модули в Обероне/Компонентном Паскале загружаются в память в следующих случаях (если модуль еще не загружен):

1. **Явная загрузка** — при первом обращении к процедуре, содержащейся в модуле, с помощью [командира](#), из диалога или из меню. Именно это имеет место в простейших случаях типичных школьных программ. (Напомним, что вызываемая процедура должна быть экспортирована, т.е. в ее заголовке сразу после идентификатора должна стоять звездочка.)
2. **Неявная загрузка** — осуществляется автоматически, если некоторый модуль импортируется из загружаемого, и если при этом этот некоторый модуль еще не загружен в память. Например, в примере «Привет» импортируется модуль StdLog. Если StdLog еще не загружен, то перед тем как загрузить модуль Привет, будет загружен модуль StdLog. А если StdLog импортирует какие-то другие модули, то они будут загружены в память еще до StdLog. И так далее.

Загруженный модуль остается в памяти до тех пор, пока он не будет явно выгружен по команде программиста (см. ниже).

Что происходит при загрузке модуля

При загрузке модуля (назовем его Данный) прежде всего происходит проверка и неявная загрузка нужных модулей, как описано выше в п.2.

После загрузки модуля Данный происходит невидимое для программиста редактирование его "внешних связей" (линкование — linking), т.е. запись в соответствующие ячейки памяти адресов импортированных модулей (уже находящихся в памяти), а также соответствующих дескрипторов типов и т.п.

Это позволяет в дальнейшем осуществлять вызов процедур из этих модулей без дополнительных накладных расходов по сравнению с технологией статического линкования.

Наконец, в качестве последнего шага происходит выполнение тела модуля (необязательный фрагмент текста, подобный выполняемому телу процедур; см. Примеры программ...; в простых случаях этот фрагмент отсутствует).

Там обычно происходит инициализация глобальных переменных модуля, сохраняющих свои значения между вызовами процедур модуля пока модуль находится в памяти.

В Компонентном Паскале предусмотрена также *возможность включать в модуль аналогичный фрагмент, выполняемый перед выгрузкой модуля из памяти* (см. документацию).

Как выгрузить старую и загрузить новую версию разрабатываемого модуля

Прежде всего следует помнить, что *нельзя выгрузить модуль, если его импортируют другие модули, находящиеся в памяти* (т.е. если у него есть клиенты; см. также [ниже](#)).

1. Уже описан способ, когда при вызове новой версии модуля кликом по командире, из памяти выгружается старая версия модуля перед загрузкой новой — для этого достаточно при клике нажимать на клавишу Ctrl.
2. Если открыт документ с исходным текстом данного модуля и этот документ находится в фокусе (является активным; другими словами нажатие буквенных клавиш вызвало бы печать в текст данного модуля), то этот модуль можно выгрузить из памяти командой меню Прог --> Выгрузить.
3. Еще одна удобная возможность — команда меню Прог --> Компилировать и выгрузить (Ctrl+Shift+K), которая компилирует новую версию модуля и выгружает из памяти старую.
4. Наконец, в любом окне, где в тексте есть имя модуля, который требуется выгрузить (часто это рабочий журнал, где имя модуля появляется в сообщениях Блэкбокса, например, после компиляции), можно дважды кликнуть по имени и выполнить программу меню Прог --> Выгрузить модули из списка.

Как проверить, какие модули загружены

Команда меню Инфо --> Загруженные модули заставляет Блэкбокс сгенерировать и открыть в новом окошке документ, содержащий список всех модулей, находящихся в данный момент в памяти.

Начало списка выглядит примерно так (состав списка и порядок модулей в нем может меняться):

имя модуля	байт использовано	клиенты	скомпилирован	загружен	Обновить
Info21sysEdit	6195	0	2002-05-04 08:46:23	2002-05-26 22:47:12	
Info21sysStringsRus	1969	1	2002-05-04 08:46:22	2002-05-26 22:47:12	
Info21sysWindows	998	1	2002-05-04 08:46:22	2002-05-26 18:55:34	
DevProfiler	12590	0	2001-05-22 14:11:16	2002-05-26 22:12:34	
DevCompiler	6885	0	2001-05-22 14:10:56	2002-05-26 18:56:53	
DevCPV486	35815	1	2001-05-22 14:10:55	2002-05-26 18:56:53	
DevCPC486	46245	1	2001-05-22 14:10:54	2002-05-26 18:56:53	
DevCPL486	17736	2	2001-05-22 14:10:54	2002-05-26 18:56:53	
DevCPH	5085	1	2001-05-22 14:10:54	2002-05-26 18:56:53	
DevCPE	21140	4	2001-05-22 14:10:53	2002-05-26 18:56:53	
DevCPP	31854	1	2001-05-22 14:10:53	2002-05-26 18:56:53	
DevCPS	8289	1	2001-11-06 15:11:37	2002-05-26 18:56:53	
DevCPB	46573	2	2001-05-22 14:10:52	2002-05-26 18:56:52	
DevCPT	80452	9	2001-05-22 14:10:52	2002-05-26 18:56:52	
DevCPM	13306	9	2001-05-22 14:10:51	2002-05-26 18:56:52	
DevSelectors	6957	1	2001-05-22 14:10:55	2002-05-26 18:56:52	
DevMarkers	6700	2	2001-05-22 14:10:39	2002-05-26 18:56:52	
DevDebug	30323	0	2001-11-06 14:59:45	2002-05-26 18:55:34	
DevCommanders	5715	2	2001-11-06 14:59:24	2002-05-26 18:55:34	
Привет	52	0	2002-05-26 19:20:33	2002-05-26 19:32:59	
Config	2027	0	2002-05-04 08:42:28	2002-05-26 18:55:34	
Init	411	0	2001-05-22 14:11:06	2002-05-26 18:55:34	
Controls	58265	6	2001-11-06 15:03:33	2002-05-26 18:55:34	
Windows	12258	20	2001-05-22 14:10:12	2002-05-26 18:55:34	
Documents	22936	23	2001-05-22 14:10:11	2002-05-26 18:55:34	
Printing	4348	16	2001-05-22 14:10:10	2002-05-26 18:55:34	
Dates	2718	11	2001-05-22 14:10:06	2002-05-26 18:55:34	

Первые две колонки — имена загруженных модулей и их размеры в байтах (включая память, занятую под их глобальные переменные).

Важная третья колонка указывает, сколько модулей импортирует данный (другими словами, сколько у данного модуля модулей-"клиентов", использующих содержащиеся в нем средства — процедуры и т.п.).

Дальше указаны дата и время последней компиляции и загрузки для каждого модуля.

Гиперссылка [Обновить](#) вверху справа позволяет обновлять информацию в этом окошке одним кликом, не закрывая его.

- Выгрузить можно только модули, не имеющие на данный момент клиентов. Другими словами, выгружать модули нужно в порядке обратном по сравнению с их загрузкой. Попытка выгрузить модуль, имеющий ненулевое количество клиентов, порождает в рабочем журнале сообщение '**не удалось выгрузить модуль ...**'

Обычно в окошке 'Загруженные модули' модули, связанные между собой отношениями импорта, группируются. Пример такой группы — первые три модуля в списке (i21sys...).

Такие модули легко выгрузить одновременно следующим образом:

- Достаточно в этом окошке выделить мышкой соответствующие строки (вместе с цифровой информацией), а затем выполнить команду меню Прог --> Выгрузить модули из списка.

Блэкбокс будет их выгружать в том порядке, в каком они встречаются в тексте. Поскольку в списке они упорядочены так, что модули-клиенты стоят всегда выше тех модулей, которые они импортируют, то выгрузка модулей будет происходить в правильном порядке.

Вместо выделения нужных модулей в окошке 'Загруженные модули' можно также создать отдельный документ, в котором записать (в правильном порядке) имена модулей, с которыми ведется работа.

Выделяя весь документ (Ctrl+A), а затем выполняя Прог --> Выгрузить модули из списка, можно заставить Блэкбокс выгрузить только требуемые модули.

- Обычная ошибка — выполнять команду меню Прог --> Выгрузить вместо Прог --> Выгрузить модули из списка, и наоборот.

Аварийная остановка программы

Аварийная остановка ("авост", "облом" и т.п.) программы на Блэкбоксе может произойти по ряду причин.

Простейший пример — деление на нуль. Другой пример — оператор HALT, намеренно использованный программистом и имитирующий аварийную остановку. Другие ситуации описаны ниже.

Общая идея состоит в следующем (см. ниже Пример аварийной остановки):

1. Модуль, в котором произошла ошибка, просто остается в памяти, так что его процедуры можно вызывать снова (см. Загрузка и перезагрузка модулей).
Однако вызов любых процедур (в т.ч. и той, в которой произошла ошибка) будет осуществляться как обычно — т.е. с первого выполняемого оператора. Блэкбокс не предусматривает возможность продолжить выполнение прерванной процедуры.

2. Блэкбокс открывает окошко со специальной диагностикой, представляющей собой документ Блэкбокса с особыми гиперссылками.

Предоставляемая диагностика дает возможность получить, кликая мышкой, полную удобочитаемую информацию о состоянии глобальных и локальных переменных всех модулей и процедур в цепочке вызовов, приведших к аварийной остановке. (Поэтому еще говорят: "Блэкбокс выбросил потроха".)

Кроме того, с помощью простого клика мышкой можно посмотреть те операторы в программных текстах, где произошла ошибка — но только если исходные тексты хранятся в соответствии с [соглашениями](#) (см. Организация файлов...), принятыми в Блэкбоксе.

Разумеется, для модулей, поставляемых в дистрибуции Блэкбокса без исходных текстов — как и для других модулей, чьи исходные тексты хранятся отдельно, эта возможность не работает.

Следует подчеркнуть следующее:

- Никакого особого отладочного режима в Блэкбоксе нет и со стороны программиста **не нужно никаких специальных усилий** (кроме клика мышкой), чтобы найти то место в программе, где произошел авост. (Однако авост будет локализован неправильно, если текст модуля изменился после последней компиляции.)
- Принятый подход является весьма удобным гибридом между архаичным дампом памяти и обычным пошаговым отладчиком. Расшифровывать дампы нелегко, а практика отладки как попало написанных программ наугад с помощью пошагового отладчика порочна и неприемлема [по ряду причин](#) (см. О дисциплине программирования).

Основные ситуации, приводящие к аварийной остановке программы

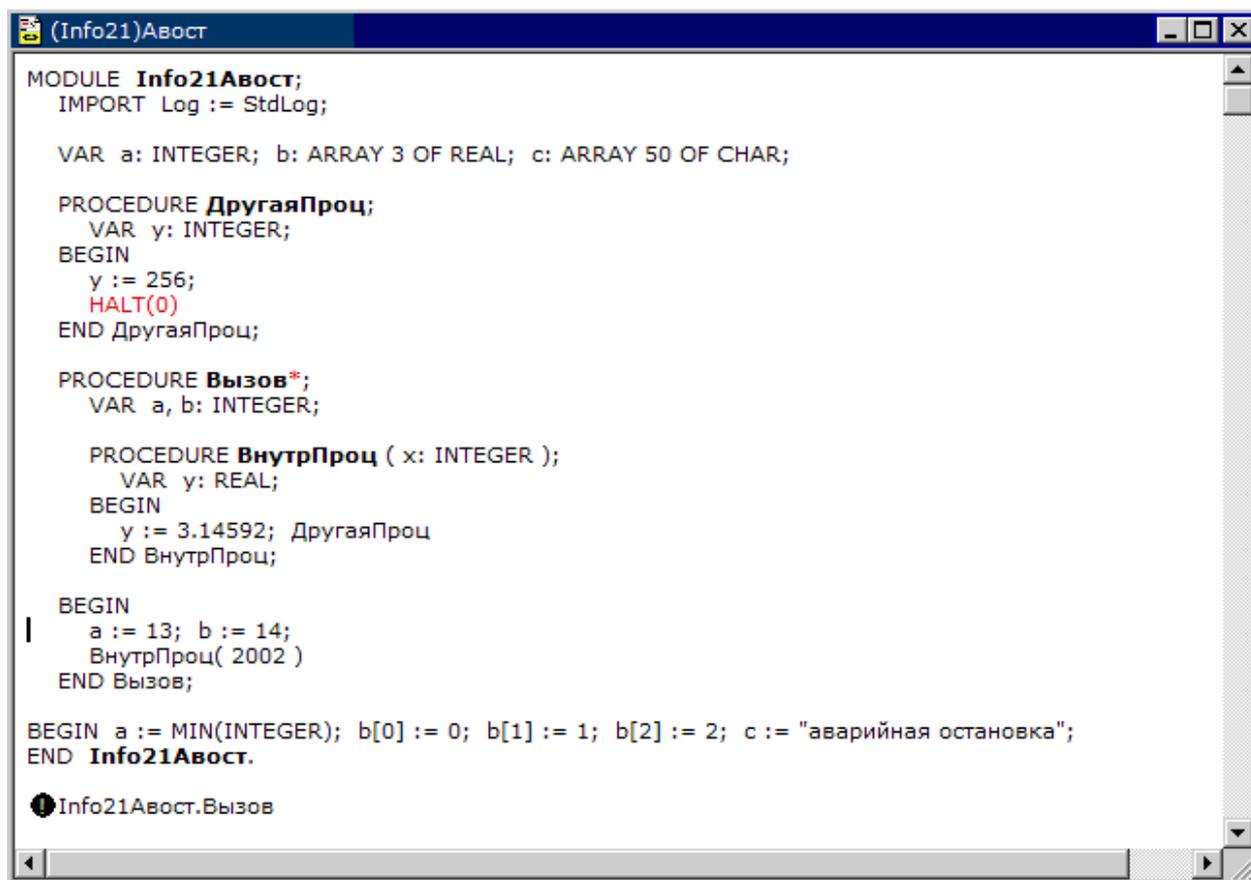
1. Логическое выражение в первом аргументе процедуры ASSERT имеет результат FALSE.
Эта процедура предназначена для контроля пред- и пост-условий, а также инвариантов циклов и т.п.; программиста.
Аварийная остановка в этом случае означает, что предположения программиста о поведении программы в данной точке не соответствуют логическому условию в процедуре ASSERT — либо программа не делает того, что требует программист, либо программист допустил ошибку в записи условия.
2. Попытка выйти за границы массива — обычно в результате неправильно заданного условия окончания цикла.
3. Попытка разыменования указателя, имеющего значение NIL в данной точке программы — либо из-за того, что указатель не был надлежащим образом инициализирован (не была вызвана процедура NEW), либо из-за того, что неправильно контролируется обход динамической структуры (например, ссылку на NIL обычно содержит последний элемент линейного списка).
4. Программа выполнила процедуру HALT, которая по определению служит для вызова аварийной остановки.
5. В правой или (реже) левой части оператора присваивания оказалась не выполненной охрана типа (динамический тип переменной оказался несоответствующим типу, указанному в охране).
6. В операторе CASE или WITH не предусмотрен вариант ELSE, а при выполнении программы ни один из явно указанных вариантов выбора не реализовался.

Пример аварийной остановки

Пусть имеется следующий модуль, показанный на картинке (текст имеется в пакете русификации).

Клик по командиру внизу за текстом модуля вызывает процедуру Вызов, которая вызывает свою внутреннюю процедуру ВнутрПроц.

Эта процедура в свою очередь вызывает процедуру ДругаяПроц. Процедура ДругаяПроц содержит оператор HALT(0), который имитирует аварийную остановку (числовой параметр предусмотрен для удобства программиста и особой роли не играет).



```

MODULE Info21Авост;
  IMPORT Log := StdLog;

  VAR a: INTEGER; b: ARRAY 3 OF REAL; c: ARRAY 50 OF CHAR;

  PROCEDURE ДругаяПроц;
    VAR y: INTEGER;
  BEGIN
    y := 256;
    HALT(0)
  END ДругаяПроц;

  PROCEDURE Вызов*;
    VAR a, b: INTEGER;

    PROCEDURE ВнутрПроц ( x: INTEGER );
      VAR y: REAL;
    BEGIN
      y := 3.14592; ДругаяПроц
    END ВнутрПроц;

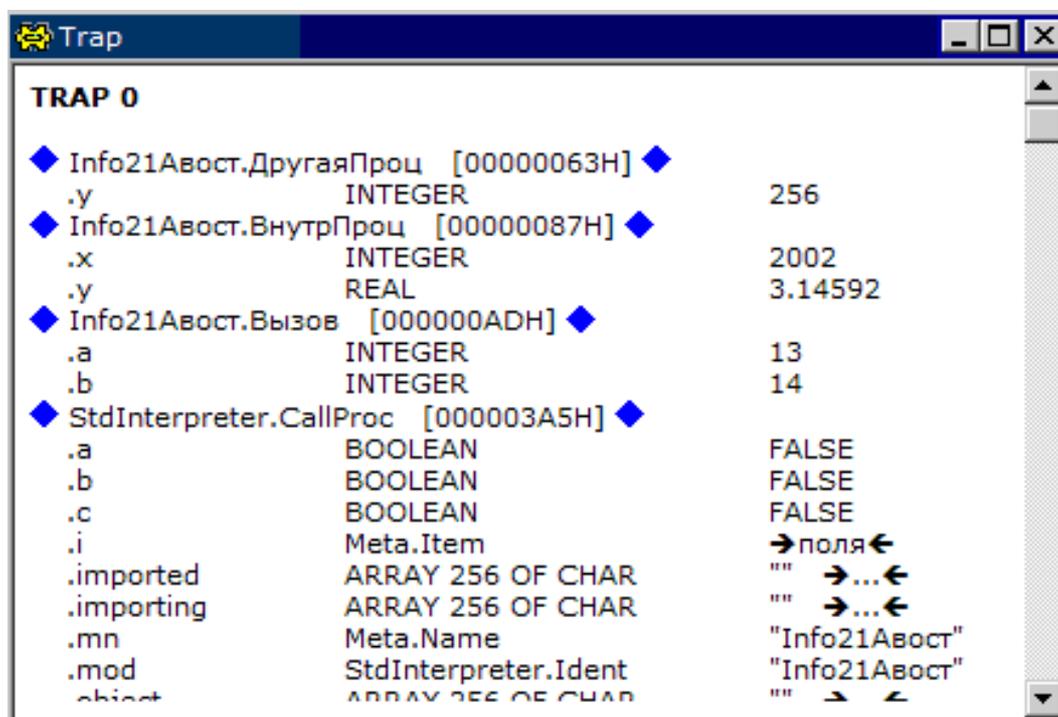
  BEGIN
    a := 13; b := 14;
    ВнутрПроц( 2002 )
  END Вызов;

  BEGIN a := MIN(INTEGER); b[0] := 0; b[1] := 1; b[2] := 2; c := "аварийная остановка";
END Info21Авост.

Info21Авост.Вызов

```

Выполнение HALT(0), кроме собственно аварийной остановки, открывает диагностическое окошко примерно следующего вида (Trap = ловушка):



```

TRAP 0
◆ Info21Авост.ДругаяПроц [00000063H] ◆
.y      INTEGER      256
◆ Info21Авост.ВнутрПроц [00000087H] ◆
.x      INTEGER      2002
.y      REAL         3.14592
◆ Info21Авост.Вызов [000000ADH] ◆
.a      INTEGER      13
.b      INTEGER      14
◆ StdInterpreter.CallProc [000003A5H] ◆
.a      BOOLEAN      FALSE
.b      BOOLEAN      FALSE
.c      BOOLEAN      FALSE
.i      Meta.Item    →поля←
.imported  ARRAY 256 OF CHAR  "" →...←
.importing  ARRAY 256 OF CHAR  "" →...←
.mod       Meta.Name   "Info21Авост"
.obj       StdInterpreter.Ident "Info21Авост"

```

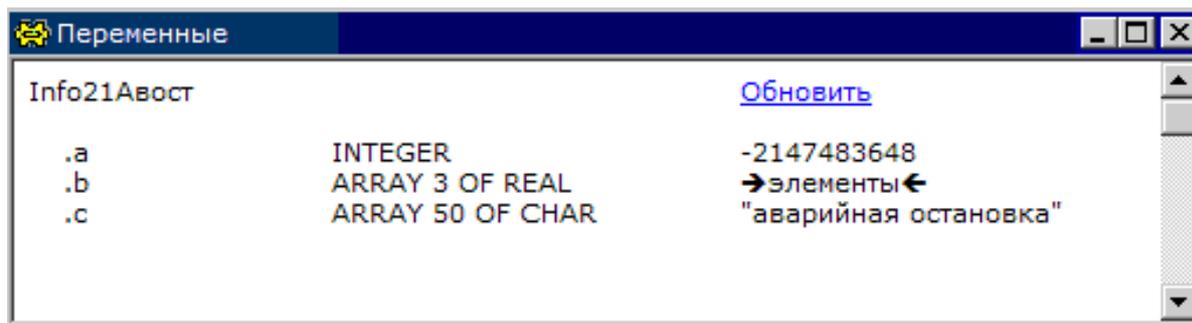
В окошке представлена вся цепочка вызовов, приведшая к авосту (включая и системные процедуры Блэкбокса, которые, собственно, и осуществили вызов после клика по командиру).

После имени каждой процедуры дан алфавитный список ее внутренних переменных вместе с их типами и значениями в читабельной форме.

Слева и справа от имени каждой процедуры — синие ромбы, являющиеся гиперссылками (гиперссылками являются и черные стрелки; см. об этом дальше).

Клик по ромбу слева от имени процедуры — скажем, ДругаяПроц — открывает окошко, в котором показаны значения глобальных переменных модуля, в котором определена данная процедура.

В нашем конкретном случае это модуль Info21Авост:



Глобальные переменные модуля объявляются в нем до всех процедур (см. выше текст модуля).

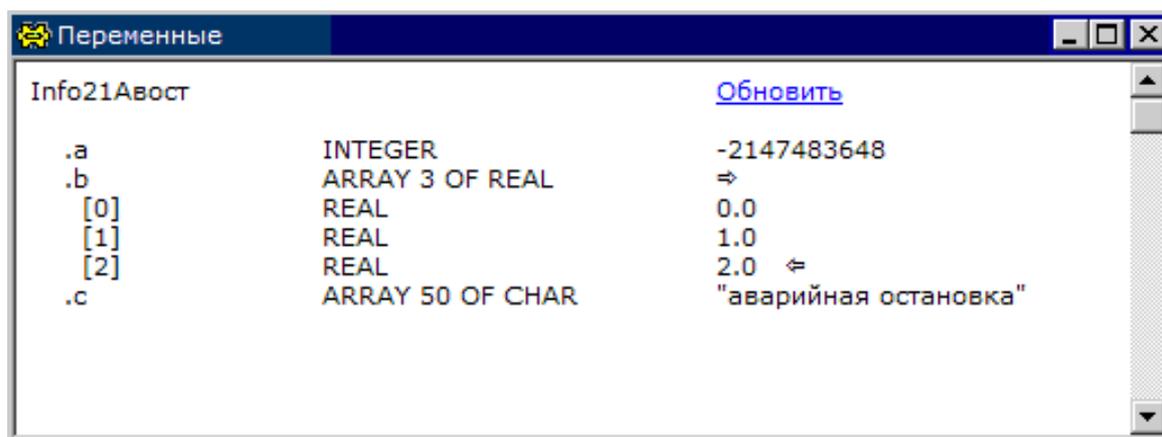
В данном случае интересно посмотреть, как показаны массивы.

Массивы литер (ARRAY OF CHAR) играют особую роль в Компонентном Паскале, т.к. служат для хранения литерных цепочек.

Например, массиву **c** в тексте модуля была присвоена литерная цепочка "аварийная остановка", которая непосредственно показана в диагностическом окошке.

Все другие массивы трактуются более единообразно.

Например, в нашем случае переменная **b** это массив из трех целых. Клик по любой из черных стрелок раскрывает список значений элементов массива:



Клик по любой белой стрелке закрывает список, возвращая картинку в предыдущее состояние.

Черные и белые стрелки — визуальное оформление т.наз. "складок" — folders — стандартного средства, которое можно использовать в любых документах Блэкбокса, в т.ч. в текстах программ. Например, для "упрятивания" длинных комментариев или

временной вставки/удаления фрагментов программ, предназначенных, скажем, для отладочных проверок и т.п.

Если же в окошке Тгар кликнуть по синему ромбу справа от имени процедуры (скажем, первой в списке), то откроется текст соответствующего модуля (если он еще не был открыт), в котором будет выделен оператор, вызвавший авост. В данном случае это будет оператор HALT(0).

Если же Блэкбокс по какой-то причине не найдет текста модуля (например, в случае модуля StdInterpreter в показанной цепочке вызовов), то сначала Блэкбокс предложит программисту самому "вручную" открыть текст модуля (Блэкбокс вызовет стандартный диалог операционной системы для открытия файла).

А если программист откажется (Cancel, Отмена и т.п.), то в рабочем журнале появится сообщение "исходник для StdInterpreter не найден".

- Диагностические окна позволяют исследовать произвольные динамические структуры данных, созданные программой перед авостом.
- Значения глобальных переменных любого загруженного модуля можно исследовать в любой момент и без авоста: достаточно в любом окне (например, в рабочем журнале или в окне со списком загруженных модулей: Инфо --> Загруженные модули) выделить имя нужного модуля (скажем, двойным кликом) и выполнить команду меню Инфо --> Глобальные переменные.

Примеры программ на Компонентном Паскале

Приводимые примеры покажут читателю, как выглядят простейшие программы на Компонентном Паскале.

Исходные тексты всех примеров находятся в папке i21примеры\Mod\ во всех комплектах от Информатики-21. Имена модулей на приводимых ниже картинках могут слегка отличаться от имен в текущих комплектах.

Их можно непосредственно открывать из Блэкбокса обычным порядком (Ctrl+O и т.д.). Там же (i21примеры\Docu*.odc) можно найти документы, детально описывающие разработку этих модулей методом пошагового уточнения.

Функция, проверяющая простоту задаваемого целого:

```
MODULE ПошаговаяПростые;  
  IMPORT Math;  
  
  PROCEDURE Простое* ( n: INTEGER ): BOOLEAN;  
    VAR кандидат: INTEGER; корень: REAL;  
  BEGIN  
    кандидат := 2;  
    корень := Math.Sqrt( n );  
    WHILE (кандидат <= корень) & ~( n MOD кандидат = 0 ) DO  
      кандидат := кандидат + 1  
    END;  
    RETURN ~(кандидат <= корень)  
  END Простое;  
  
END ПошаговаяПростые|
```

Звездочка после имени процедуры в заголовке — символ экспорта — делает процедуру видимой и доступной из других модулей.

Проверка эквивалентна задаче поиска (в данном случае, поиск делителя) и выполняется по стандартной схеме для таких задач.

Уплотнение цепочки литер, чтобы исключить идущие подряд пробелы:

```
MODULE ПошаговаяУплотнение;
  IMPORT StdLog;

  PROCEDURE Проверить ( VAR а: ARRAY OF CHAR );
    VAR кандидат: INTEGER;
  BEGIN
    кандидат := 0;
    WHILE (кандидат < LEN(а)) & (а[кандидат] # 0X) DO
      кандидат := кандидат + 1
    END;
    ASSERT( кандидат < LEN(а) )
  END Проверить;

  PROCEDURE Уплотнить* ( VAR а: ARRAY OF CHAR );
    VAR i, j: INTEGER;
  BEGIN Проверить( а );
    i := 0; j := 0;
    WHILE а[j] # 0X DO
      i := i + 1;
      IF (а[i] # ' ') OR (а[j] # ' ') THEN
        j := j + 1;
        а[j] := а[i]
      END
    END
  END Уплотнить;

  PROCEDURE Демо* ( цепочка: ARRAY OF CHAR );
    VAR а: ARRAY 200 OF CHAR;
  BEGIN
    а := цепочка$;
    StdLog.String('до сжатия:'); StdLog.Ln;
    StdLog.String(а); StdLog.Ln;
    Уплотнить( а );
    StdLog.String('после сжатия:'); StdLog.Ln;
    StdLog.String(а); StdLog.Ln;
  END Демо;

END ПошаговаяУплотнение.
```

Процедура Проверить проверяет наличие в массиве литер специальной литеры-ограничителя, задаваемой 16-ричным кодом 0X.

Сортировка вставками

```
MODULE ПошаговаяВставки;
  IMPORT StdLog, In;

  PROCEDURE Вставить ( VAR a: ARRAY OF INTEGER; i: INTEGER );
    VAR t: INTEGER;
  BEGIN
    IF a[i] < a[i-1] THEN
      t := a[i]; a[i] := a[i-1]; a[i-1] := t;
      IF i - 1 # 0 THEN
        Вставить( a, i - 1 )
      END;
    END;
  END Вставить;

  PROCEDURE Сорт* ( VAR a: ARRAY OF INTEGER );
    VAR i: INTEGER;
  BEGIN
    FOR i := 1 TO LEN(a) - 1 DO
      Вставить( a, i )
    END;
  END Сорт;

  PROCEDURE Демо*;
    VAR x, n: INTEGER; a: POINTER TO ARRAY OF INTEGER;
  BEGIN In.Open; n := 0; In.Int( x );
    WHILE In.Done DO
      INC( n ); In.Int( x )
    END;
    NEW( a, n );
    In.Open; n := 0; In.Int( x );
    WHILE In.Done DO
      a[n] := x; INC( n ); In.Int( x )
    END;
    Сорт( a );
    FOR n := 0 TO LEN(a) - 1 DO
      StdLog.Int( a[n] )
    END;
  END Демо;

END ПошаговаяВставки.
```

Разумеется, можно обойтись и без рекурсии в процедуре Вставить (это так называемая хвостовая рекурсия, которая легко заменяется на цикл WHILE).

Процедура Демо показывает использование средств модуля In. Их следует использовать в преподавании вместо ввода с клавиатуры старого Паскаля.

В процедуре Демо сначала подсчитывается количество чисел в потоке ввода (первый цикл WHILE), затем размещается массив нужной длины (процедура NEW(a, n)).

Затем производится повторное считывание чисел из потока ввода (In.Open и второй цикл WHILE) и их запись в размещенный массив.

Наконец, вызывается процедура сортировки и производится печать отсортированного массива в [рабочий журнал](#) (специальное окошко для системных сообщений, обычно постоянно открытое при работе в Блэкбоксе).

Сообщение о языке Компонентный Паскаль

Copyright © 1994-2001 by Oberon microsystems, Inc., Switzerland.

All rights reserved. No part of this publication may be reproduced in any form or by any means, without prior written permission by Oberon microsystems. The only exception is the free electronic distribution of the education version of BlackBox (see the accompanying copyright notice for details).

Oberon microsystems, Inc.
Technoparkstrasse 1
CH-8005 Zuerich
Switzerland

Oberon is a trademark of Prof. Niklaus Wirth.
Component Pascal is a trademark of Oberon microsystems, Inc.
All other trademarks and registered trademarks belong to their respective owners.

Authors
Oberon microsystems, Inc.
March 2001

Authors of Oberon-2 report
H. Moessenboeck, N. Wirth
Institut fuer Computersysteme, ETH Zuerich
October 1993

Author of Oberon report
N. Wirth
Institut fuer Computersysteme, ETH Zuerich
1987

© Перевод на русский язык: Ф.В.Ткачев, Институт ядерных исследований РАН, Москва, 2001-2003, 2009.

Перевод был первоначально выполнен для спецкурса «Современное программирование. Введение в объектные и компонентные технологии», читаемого переводчиком на физическом факультете МГУ с 2001 г.

Перевод сверен с документацией BlackBox v.1.4 Final.

Терминология перевода ориентируется на существующие переводы книг Н.Вирта на русский язык (переводчики: Д.Б.Подшивалов, В.А. Серебряков и В.М. Ходукин, В.М. Курочкин). В некоторых случаях в квадратных скобках даны английские выражения оригинала, а в угловых — вставки переводчика.

Дополнительные замечания о терминологии можно найти [здесь](#).

С.З.Свердлов ранее и независимо выполнил перевод на русский язык описания языка Оберон-2, предшественника Компонентного Паскаля (перевод С.З.Свердлова доступен в Сети по адресу <http://www.univologda.ac.ru/oberon/o2rus.htm>).

Содержание

- [1. Введение](#)
- [2. Синтаксис](#)
- [3. Словарь и изображение](#)
- [4. Описания и правила видимости](#)
- [5. Описания констант](#)
- [6. Описания типов](#)
 - [6.1 Элементарные типы](#)
 - [6.2 Массивовые типы](#)
 - [6.3 Записевые типы](#)
 - [6.4 Указательные типы](#)
 - [6.5 Процедурные типы](#)
 - [6.6 Типы цепочек литер](#)
- [7. Описания переменных](#)
- [8. Выражения](#)
 - [8.1 Операнды](#)
 - [8.2 Операции](#)
- [9. Операторы](#)
 - [9.1 Присваивания](#)
 - [9.2 Вызовы процедур](#)
 - [9.3 Операторные последовательности](#)
 - [9.4 Условные операторы IF](#)
 - [9.5 Операторы выбора CASE](#)
 - [9.6 Цикл с условием продолжения WHILE](#)
 - [9.7 Цикл с условием окончания REPEAT](#)
 - [9.8 Цикл с шагом FOR](#)
 - [9.9 Безусловный цикл LOOP](#)
 - [9.10 Операторы возврата RETURN и выхода EXIT](#)
 - [9.11 Операторы конкретизации типа WITH](#)
- [10. Описания процедур](#)
 - [10.1 Формальные параметры](#)
 - [10.2 Методы](#)
 - [10.3 Предопределенные процедуры](#)
 - [10.4 Финализация](#)
- [11. Модули](#)
- [Приложение А: Определения терминов](#)
- [Приложение В: Синтаксис Компонентного Паскаля](#)
- [Приложение С: Диапазоны значений элементарных типов](#)
- [Приложение D: Обязательные требования к среде выполнения](#)

1. Введение

Компонентный Паскаль — усовершенствованная версия языка Оберон-2, разработанная компанией Oberon microsystems. Компания Oberon microsystems благодарит Х. Мёссенбёка и Н. Вирта за дружеское разрешение использовать их Сообщение о языке Оберон-2 в качестве основы данного документа.

Компонентный Паскаль — язык общего назначения в традиции языков Паскаль, Модуля-2 и Оберон. Его самые важные черты — блочная структура, модульность, отдельная компиляция, статическая типизация [static typing] и строгий контроль согласованности типов (в том числе через границы модулей), расширенное переопределение типов [type extension] вместе с методами, динамическая загрузка модулей, а также автоматический сбор мусора.

Переопределение типов делает Компонентный Паскаль объектно-ориентированным языком. Объект — переменная абстрактного типа данных, состоящая из частных данных (т.е. его состояния) и процедур, оперирующих с этими данными. Абстрактные типы данных определяются как записи, допускающие <расширенное> переопределение [extensible records]. В Компонентном Паскале для большей части понятий объектно-ориентированных языков используется установившийся словарь императивных языков, чтобы минимизировать число терминов для сходных понятий. <Так как буквальный перевод терминологии оригинала оказывается громоздким в силу разной структуры английского и русского языков, в переводе для краткости добавлены термины «потомок» [extentions] и «предок» [base type] для описания типов. К тому же в оригинале употребляются base type и basic type, что при небрежном переводе чревато путаницей.>

Полная безопасность **контроль типов переменных [complete type safety]** и требование динамической объектной модели делают Компонентный Паскаль компонентно-ориентированным языком.

Данное сообщение не является учебником. Его краткость намеренна. Его функция — служить справочником для программистов. Большинство недоговоренностей оставлено намеренно, либо потому, что соответствующее уточнение может быть выведено из сформулированных правил языка, либо потому, что уточнение потребовало бы конкретизировать определения там, где универсальная конкретизация не кажется полезной.

Приложение А определяет некоторые термины, используемые для выражения правил проверки типов в Компонентном Паскале. В тексте такие термины даны курсивом, чтобы указать на их специальный смысл (например, *одинаковый* тип).

Рекомендуется минимизировать использование процедурных типов и супер-вызовов, т.к. они считаются устаревшими средствами. Они пока сохранены, чтобы облегчить использование существующих программ на Обероне-2. Поддержка этих средств может быть уменьшена в последующих выпусках продукта. В дальнейшем тексте то, что касается этих устаревших средств, отмечено красным цветом.

2. Синтаксис

Для описания синтаксиса Компонентного Паскаля используется расширенный формализм Бэкуса-Наура (РФБН). Альтернативы разделяются символом |. Квадратные скобки [и] означают необязательность заключенного в них выражения, а фигурные скобки { и } означают его возможное повторение (0 или более раз). В случае необходимости для группирования лексем используются круглые скобки (и). Нетерминальные лексемы начинаются с большой буквы (например, Statement). Терминальные лексемы либо начинаются с маленькой буквы (например, ident), либо записаны только большими буквами (например, BEGIN), либо обозначаются цепочками литер (например, «:=»).

3. Словарь и изображение

Изображение (терминальных) лексем посредством литер использует стандарт ISO 8859-1, т.е. расширение Latin-1 набора литер ASCII. Такие лексемы суть идентификаторы, числа, операции и ограничители. Следует соблюдать следующие лексические правила: Пробелы и концы строк не должны появляться внутри лексем (за исключением комментариев, а также пробелов в литерных цепочках). Они игнорируются, если они не нужны для разделения двух последовательных лексем. Большие и маленькие буквы различаются.

1. *Идентификаторы* суть последовательности букв, цифр и символов подчеркивания. Первая литера не должна быть цифрой.

ident = (letter | «_») {letter | «_» | digit}.

letter = «A» .. «Z» | «a» .. «z» | «А»..«Ц» | «Ш»..«ц» | «ш»..«я».
 digit = «0» | «1» | «2» | «3» | «4» | «5» | «6» | «7» | «8» | «9».

Примеры: x Scan Oberon2 GetSymbol firstLetter

<В русифицированной версии системы Блэкбокс 1.5, доступной в рамках проекта Информатика-21 по адресу <http://www.inr.ac.ru/~info21/software.htm>, в идентификаторах допустимы все русские буквы.>

2. Числа суть целые или вещественные константы (без знака). Типом целой константы является INTEGER, если значение константы принадлежит диапазону значений типа INTEGER, или LONGINT в противном случае (см. 6.1). Если константа задана с суффиксом 'H' или 'L', представление является 16-ричным, в противном случае представление десятичное. Суффикс 'H' используется для записи 32-битных констант в диапазоне -2147483648 .. 2147483647. Разрешается не более восьми значащих 16-ричных цифр. Суффикс 'L' используется для записи 64-битных констант.

Вещественное число всегда содержит десятичную точку. Оно также может содержать десятичный масштабный множитель. Буква E означает «умножить на 10 в степени». Вещественное число всегда имеет тип REAL.

number = integer | real.
 integer = digit {digit} | digit {hexDigit} («H» | «L»).
 real = digit {digit} «.» {digit} [ScaleFactor].
 ScaleFactor = «E» [«+» | «-»] digit {digit}.
 hexDigit = digit | «A» | «B» | «C» | «D» | «E» | «F».

Примеры:

1234567 INTEGER1234567
 0DH INTEGER13
 12.3 REAL 12.3
 4.567E8 REAL 456700000
 0FFFF0000H INTEGER-65536
 0FFFF0000L LONGINT 4294901760

3. Литеры [*characters*] обозначаются своим порядковым номером в 16-ричной нотации, за которым следует буква X.

character = digit {hexDigit} «X».

4. Литерные цепочки [*strings*] — последовательности литер, заключенные в одиночные (') или двойные (") кавычки. Открывающая кавычка должна всегда совпадать с закрывающей, и не содержаться внутри цепочки. Число литер в цепочке называется ее *длиной*. Цепочка длины 1 может использоваться всюду, где разрешена литерная константа, и наоборот.

string = «"» {char} «"» | «'» {char} «'».

Примеры: "Component Pascal" "Don't worry!" "x"

5. Операции и ограничители [*operators and delimiters*] суть специальные литеры, пары литер или ключевые слова, перечисленные ниже. Ключевые слова содержат только большие буквы и не могут использоваться как идентификаторы.

+	:=	ABSTRACT	EXTENSIBLE	POINTER
-	^	ARRAY	FOR	PROCEDURE
*	=	BEGIN	IF	RECORD

/	#	BY	IMPORT	REPEAT
~	<	CASE	IN	RETURN
&	>	CLOSE	IS	THEN
.	<=	CONST	LIMITED	TO
,	>=	DIV	LOOP	TYPE
;	..	DO	MOD	UNTIL
	:	ELSE	MODULE	VAR
\$		ELSIF	NIL	WHILE
()	EMPTY	OF	WITH
[]	END	OR	
{	}	EXIT	OUT	

6. *Комментарии* могут вставляться между любой парой лексем в программе. Они представляют собой произвольные последовательности литер, открывающиеся скобкой (*** и закрывающиеся скобкой ***). Комментарии могут быть вложены друг в друга. Они не влияют на смысл программы.

4. Описания и правила видимости

Каждый идентификатор, встречающийся в программе, должен быть введен с помощью описания [declaration], за исключением случаев, когда это предопределенный [predeclared] идентификатор. Описания также задают некоторые перманентные свойства объекта, такие как является ли он константой, типом, переменной или процедурой. Тогда идентификатор используется для ссылок на связанный с ним объект.

Область видимости объекта *x* текстуально распространяется от точки его описания до конца блока (модуля, процедуры или записи), к которому принадлежит описание и по отношению к которому объект, таким образом, считается *локальным*. Из этой области исключаются области видимости объектов с таким же именем, описанных в блоках, вложенных в данный. Правила видимости таковы:

1. Идентификатор может обозначать только один объект в данной области видимости (т.е. никакой идентификатор не может быть объявлен в блоке дважды);
2. На объект можно сослаться только в его области видимости;
3. Описание типа *T*, содержащее ссылки на другой тип *T1* могут стоять в точках, где *T1* еще не известен. Описание типа *T1* должно следовать далее в том же блоке, в котором локализован *T*;
4. Идентификаторы полей записей (см. 6.3) или методов (см. 10.2) могут встречаться только в составных именах [designators].

За идентификатором, описанном в блоке модуля, в его описании может следовать метка экспорта («*» или «-»), чтобы указать, что он экспортируется. Идентификатор *x*, экспортированный из модуля *M*, можно использовать в других модулях при условии, что они импортируют *M* (см гл. 11). В таких модулях идентификатор обозначается как *M.x* и называется *уточненным идентификатором* [qualified identifier]. Переменные и поля записей, отмеченные в своих описаниях символом «-», доступны в импортирующих модулях *только для чтения* [are read-only] (в случае переменных и полей) или *только для реализации* [implement-only] (в случае методов).

Qualident = [ident «.»] ident.

IdentDef = ident [«*» | «-»].

Следующие идентификаторы являются предопределенными; их значения описаны в указанных разделах:

ABS	(10.3)	INTEGER	(6.1)
ANYPTR	(6.1)	FALSE	(6.1)
ANYREC	(6.1)	LEN	(10.3)
ASH	(10.3)	LONG	(10.3)
ASSERT	(10.3)	LONGINT	(6.1)
BITS	(10.3)	MAX	(10.3)
BOOLEAN	(6.1)	MIN	(10.3)
BYTE	(6.1)	NEW	(10.3)
CAP	(10.3)	ODD	(10.3)
CHAR	(6.1)	ORD	(10.3)
CHR	(10.3)	REAL	(6.1)
DEC	(10.3)	SET	(6.1)
ENTIER	(10.3)	SHORT	(10.3)
EXCL	(10.3)	SHORTCHAR	(6.1)
HALT	(10.3)	SHORTINT	(6.1)
INC	(10.3)	SHORTREAL	(6.1)
INCL	(10.3)	SIZE	(10.3)
INF	(6.1)	TRUE	(6.1)

5. Описания констант

Описание константы связывает идентификатор с некоторым неизменяемым значением.

ConstantDeclaration = IdentDef «= \Rightarrow » ConstExpression.

ConstExpression = Expression.

Константное выражение — это выражение, которое может быть вычислено при простом текстовальном просмотре без фактического выполнения программы. Его операнды суть константы (гл. 8) или предопределенные функции (10.3), которые могут быть вычислены при компиляции. Примеры константных выражений:

N = 100

limit = 2*N - 1

fullSet = {MIN(SET) .. MAX(SET)}

6. Описания типов

Тип данных определяет множество значений, которое может принимать переменная этого типа, а также применимые к ней операции. Описание типа связывает идентификатор с типом. В случае структурированных типов (массивовых и записевых) оно также определяет структуру переменных этого типа.

Структурированный тип не может содержать себя.

TypeDeclaration = IdentDef «= \Rightarrow » Type.

Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.

Примеры:

Table = ARRAY N OF REAL

Tree = POINTER TO Node

Node = EXTENSIBLE RECORD

key : INTEGER;

left, right: Tree

END

```
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
    width: INTEGER;
    subnode: Tree
END
Object = POINTER TO ABSTRACT RECORD END;
Function = PROCEDURE (x: INTEGER): INTEGER
```

6.1 Элементарные типы

Элементарные типы обозначаются предопределенными идентификаторами. Соответствующие операции определены в 8.2, а предопределенные процедуры-функции — в 10.3. Значения элементарных типов таковы:

- | | |
|--------------|--|
| 1. BOOLEAN | логические значения TRUE и FALSE |
| 2. SHORTCHAR | литеры набора Latin-1 (0X .. 0FFX) |
| 3. CHAR | литеры набора Unicode (0X .. 0FFFFX) |
| 4. BYTE | целые от MIN(BYTE) до MAX(BYTE) |
| 5. SHORTINT | целые от MIN(SHORTINT) до MAX(SHORTINT) |
| 6. INTEGER | целые от MIN(INTEGER) до MAX(INTEGER) |
| 7. LONGINT | целые от MIN(LONGINT) до MAX(LONGINT) |
| 8. SHORTREAL | вещественные числа от MIN(SHORTREAL) до MAX(SHORTREAL), значение INF |
| 9. REAL | вещественные числа от MIN-REAL) до MAX-REAL), значение INF |
| 10.SET | множества целых чисел из диапазона от 0 до MAX(SET) |

Типы 4-7 суть *целые типы*, типы 8 и 9 суть *вещественные типы*, а вместе они называются *числовыми типами*. Они образуют иерархию; старший тип *включает* младший (включение касается значений):

REAL >= SHORTREAL >= LONGINT >= INTEGER >= SHORTINT >= BYTE

Типы 2 и 3 суть литерные типы со следующей иерархией типов:

CHAR >= SHORTCHAR

6.2 Массивовые типы [array types]

Массив — структура, состоящая из некоторого количества элементов, имеющих один и тот же тип, называемый *типом элементов*. Количество элементов массива называется его *длиной*. Элементы массива выбираются с помощью индексов, являющихся целыми числами из диапазоне от 0 до длина минус 1.

```
ArrayType      = ARRAY [Length {«,» Length}] OF Type.
Length        = ConstExpression.
```

Тип вида

ARRAY L0, L1, ..., Ln OF T

интерпретируется как сокращенная запись для

```
ARRAY L0 OF
  ARRAY L1 OF
    ...
      ARRAY Ln OF T
```

Массивы, описанные без длины, называются *открытыми массивами*. Их использование ограничено базовыми типами для указателей (см. 6.4), типами элементов открытых массивов, а также типами формальных параметров (см. 10.1). Примеры:

ARRAY 10, N OF INTEGER

ARRAY OF CHAR

6.3 Записевые типы [record types]

Запись — это структура, состоящая из фиксированного количества элементов, называемых *полями*, которые могут иметь разные типы. Описание записевого типа указывает имя и тип каждого поля. Область видимости идентификаторов полей распространяется от точки их описания до конца данного типа, но они также видимы внутри составных имен, обозначающих поля переменных данного типа (см. 8.1). Если записевый тип экспортируется, то идентификаторы полей, которые нужно иметь видимыми вне описывающего модуля, должны быть соответствующим образом помечены. Они называются *публичными полями*; непомеченные элементы называются *приватными полями*.

```
RecordType = RecAttributes RECORD [«(»BaseType«)»]
            FieldList {«;» FieldList} END.
RecAttributes = [ABSTRACT | EXTENSIBLE | LIMITED].
BaseType = Qualident.
FieldList = [IdentList «:» Type].
IdentList = IdentDef {«;» IdentDef}.
```

Использование каждого записевого типа ограничено наличием или отсутствием одного из следующих атрибутов:

ABSTRACT, EXTENSIBLE и LIMITED.

Переменные записевого типа, помеченного как ABSTRACT, не могут быть размещены [instantiated]: не может существовать ни переменных, ни полей такого типа. Абстрактные типы <т.е. описанные с атрибутом ABSTRACT> используются только как типы-предки для других записевых типов (см. ниже).

Переменные записевого типа, имеющего атрибут LIMITED, могут размещаться [allocated] только внутри того модуля, где описан данный тип. Ограничение применимо как к статическому размещению посредством описания переменных (гл. 7), так и к динамическому размещению с помощью стандартной процедуры NEW (10.3).

Записевый тип, помеченный как ABSTRACT или EXTENSIBLE, допускает расширенное переопределение, т.е. можно определить новый записевый тип как расширение такого типа. В примере

```
T0 = EXTENSIBLE RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

T1 является (непосредственным) *потомком* типа *T0*, а *T0* — (непосредственным) *предком* для *T1* (см. Приложение А). Тип-потомок *T1* состоит из полей типа-предка и полей, объявленных в *T1*. Все идентификаторы, описанные в типе-потомке, должны быть отличны от идентификаторов, описанных в его предке. Базовый тип абстрактного записевого типа должен быть абстрактным.

Кроме того, в качестве типа-предка можно использовать указательный тип. В этом случае в качестве типа-предка описываемого записевого типа используется базовый записевый тип данного указателя.

Нельзя экспортировать записевый тип, являющийся потомком скрытого (т.е. не экспортированного) записевого типа.

Каждая запись является неявным потомком предопределенного типа ANYREC. ANYREC не содержит никаких полей и может использоваться только в описаниях указателей и параметров-переменных.

Сводка атрибутов:

атрибут	переопределение	размещение
отсутствует	нет	да
EXTENSIBLE	да	да
ABSTRACT	да	нет
LIMITED	только в определяющем модуле	

Примеры описаний записевых типов:

```
RECORD
```

```
    day, month, year: INTEGER
```

```
END
```

```
LIMITED RECORD
```

```
    name, firstname: ARRAY 32 OF CHAR;
```

```
    age: INTEGER;
```

```
    salary: REAL
```

```
END
```

6.4 Указательные типы [pointer types]

Переменные, имеющие указательный тип P , принимают в качестве значений указатели на переменные некоторого типа T . T называется базовым типом [pointer base type] для P и должен быть записевым или массивовым типом. Указательные типы наследуют отношение потомок-предок для своих базовых типов: если тип $T1$ является потомком типа T , а $P1$ является типом `POINTER TO T1`, то $P1$ также является потомком типа P .

PointerType = POINTER TO Type.

Если p — переменная типа $P = \text{POINTER TO } T$, то вызов предопределенной процедуры $NEW(p)$ (см. 10.3) размещает переменную типа T в свободной памяти. Если T — записевый тип или массивовый тип с фиксированной длиной, то размещение следует выполнять оператором $NEW(p)$; если T — n -мерный открытый массив, то размещение следует выполнять оператором $NEW(p, e_0, \dots, e_{n-1})$, где T размещается с длинами, задаваемыми выражениями e_0, \dots, e_{n-1} . В обоих случаях указатель на размещенную переменную присваивается переменной p . p имеет тип P . *Разыменованная* [referenced] переменная p^{\wedge} (читается: p -разыменованная), имеет тип T . Любой указатель может принимать значение `NIL`, которое не указывает ни на какую переменную вообще.

Все поля и элементы вновь размещенной записи или массива очищаются; в частности, значения все содержащиеся в них указательные и процедурные переменные устанавливаются в `NIL`.

Предопределенный тип `ANYPTR` определяется как `POINTER TO ANYREC`. Поэтому любой указатель на какой-либо записевый тип является потомком типа `ANYPTR`. Процедуру `NEW` применять к переменным типа `ANYPTR` нельзя.

6.5 Процедурные типы

Переменные процедурного типа T имеют в качестве значения некоторую процедуру или `NIL`. Если переменной типа T присвоена процедура P , то списки формальных параметров (см. 10.1) для P и T должны соответствовать (см. Приложение А). P не может быть ни предопределенной процедурой, ни методом, ни быть локальной в другой процедуре.

ProcedureType = PROCEDURE [FormalParameters].

6.6 Типы цепочек литер [string types]

Значения типов цепочек литер суть последовательности литер, оканчивающиеся нуль-литерой (0X). *Длина* цепочки — это количество литер в ней, исключая нуль-литеру.

Цепочки могут либо быть константами, либо храниться в массиве литер. Для типов цепочек нет предопределенных идентификаторов, т.к. они для описаний не нужны.

Константные цепочки, состоящие только из литер в диапазоне 0X..0FFX, и цепочки, хранящиеся в массиве элементов типа SHORTCHAR, имеют тип Shortstring, все остальные — тип String.

7. Описания переменных

Описания переменных вводят переменные, определяя для них идентификатор и тип данных.

VariableDeclaration = IdentList «:» Type.

Переменные записевых и указательных типов имеют как *статический тип* (тот тип, с которым они описаны — будем просто называть его их типом) и *динамический тип* (тип их значения при выполнении программы). Для указателей и параметров-переменных записевых типов динамический тип может быть потомком их статического типа. Статический тип определяет, какие поля записи доступны. Динамический тип используется для вызова методов (см. 10.2).

Примеры описаний переменных (см. примеры в гл. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF
    RECORD
        name: ARRAY 32 OF CHAR;
        count: INTEGER
    END
t, c: Tree
```

8. Выражения

Выражения это конструкции, описывающие вычислительные правила, в соответствии с которыми комбинируются константы и текущие значения переменных для вычисления других значений посредством применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для выражения конкретных связей между операциями и операндами.

8.1 Операнды

За исключением конструкторов множеств и буквальных констант (т.е. чисел, литер или цепочек), операнды обозначаются *составными именами* [*designators*]. Составное имя содержит идентификатор, обозначающий константу, переменную или процедуру. Этот идентификатор может быть уточнен идентификатором модуля (см. гл. 4 и 11), и кроме того за ним могут следовать *селекторы*, если обозначаемый объект является элементом массива или полем записи.

```
Designator           = Qualident {«.» ident | «[» ExpressionList «]» | «^» |
                    «(» Qualident «)» | ActualParameters} [ «$» ].
ExpressionList       = Expression {«.» Expression}.
ActualParameters     = «(» [ExpressionList] «)».
```

Если a — составное имя, обозначающее массив, то $a[e]$ обозначает тот элемент массива a , у которого индекс равен значению выражения e в данный момент. Тип значения e должен быть целым. Составное имя, имеющее вид $a[e_0, e_1, \dots, e_n]$, является сокращением для $a[e_0][e_1] \dots [e_n]$. Если r является составным именем, обозначающим запись, то $r.f$ обозначает поле f записи r или метода f динамического типа записи r (10.2). Если a или r доступны только для чтения, то это же справедливо для $a[e]$ и $r.f$.

Если p — составное имя, обозначающее указатель, то p^{\wedge} обозначает переменную, на которую ссылается p . Составные имена $p^{\wedge}.f$, $p^{\wedge}[e]$ и $p^{\wedge}\$$ можно сокращать до $p.f$, $p[e]$ и $p\$$, т.е. селекторы записи, массива и литерной цепочки подразумевают разыменованное [dereferencing]. Разыменованное также подразумевается, если указатель присваивается переменной типа запись или типа массив (9.1), если указатель используется как фактический параметр, соответствующий формальному параметру типа запись или типа массив (10.1), или если указатель используется как аргумент стандартной процедуры LEN (10.3).

Охрана типа [type guard] $v(T)$ удостоверяет, что динамический тип переменной v есть T (или потомок типа T), т.е. выполнение программы аварийно прекращается, если динамический тип переменной v не T (и не потомок T). Тогда внутри составного имени переменная v считается имеющей статический тип T . Охрана применима, если

1. v является IN или VAR параметром записевого типа или v является указателем на запись, и если
2. T является потомком статического типа переменной v

Если обозначаемый объект — константа или переменная, то составное имя ссылается на ее значение в данный момент. Если это процедура, то составное имя ссылается на эту процедуру, исключая случаи, когда за ним следует (возможно, пустой) список параметров, и в этих случаях оно подразумевает вызов процедуры и представляет получающийся результат. Фактические параметры должны соответствовать формальным как в обычных вызовах процедур (см. 10.1).

Если a — составное имя, обозначающее массив литер, то $a\$$ обозначает литерную цепочку, оканчивающуюся нуль-литерой, содержащуюся в a . Если a не содержит литеру 0X, то использование $a\$$ приводит к ошибке при выполнении программы. Селектор $\$$ применяется неявно, если a используется как операнд конкатенации (8.2.4), операции отношения (8.2.5) или одной из predetermined процедур LONG и SHORT (10.3).

Примеры составных имен (см. примеры в гл. 7):

i	(INTEGER)
$a[i]$	(REAL)
$w[3].name[i]$	(CHAR)
$t.left.right$	(Tree)
$t(\text{CenterTree}).subnode$	(Tree)
$w[i].name\$$	(String)

8.2 Операции [operators]

В выражениях синтаксически различаются операторы четырех классов с разными приоритетами (т.е. силой связывания]). Операция \sim имеет наивысший приоритет, за ней следуют мультипликативные операции, аддитивные операции и, наконец, отношения. Операции с одинаковым приоритетом связывают операнды слева направо. Например, $x-y-z$ обозначает $(x-y)-z$.

Expression	= SimpleExpression [Relation SimpleExpression].
SimpleExpression	= [«+» «-»] Term {AddOperator Term}.
Term	= Factor {MulOperator Factor}.
Factor	= Designator number character string NIL Set «(» Expression «)» «~» Factor.
Set	= «{» [Element {«,» Element}] «}».
Element	= Expression [«.» Expression].
Relation	= «=» «#» «<» «<=» «>» «>=» IN IS.
AddOperator	= «+» «-» OR.
MulOperator	= «*» «/» DIV MOD «&».

Имеющиеся операции перечислены в нижеследующих таблицах. Некоторые операции применимы к операндам различных типов, обозначая различные операции. В этих случаях фактическая операция определяется типами операндов. Операнды должны быть *совместимы по выражению* по отношению к операции (см. Приложение А).

8.2.1 Логические операции

OR	логическое ИЛИ	$p \text{ OR } q$	«если p , то TRUE, иначе q »
&	логическое И	$p \text{ \& } q$	«если p , то q , иначе FALSE»
~	отрицание	$\sim p$	«не p »

Эти операции применимы к операндам типа BOOLEAN и дают результат типа BOOLEAN. Второй операнд логического ИЛИ вычисляется только если результат первого равен FALSE. Второй операнд логического И вычисляется, только если результат первого равен TRUE.

8.2.2 Арифметические операции

+	сумма
-	разность
*	произведение
/	вещественное частное
DIV	целое частное
MOD	остаток

Операции +, -, * и / применимы к операндам числовых типов. Типом результата будет REAL в случае операции деления (/) или если один из операндов имеет тип REAL. В противном случае типом результата будет SHORTREAL, если тип одного из операндов — SHORTREAL, LONGINT если тип одного из операндов — LONGINT, или INTEGER в любом другом случае. Если результат вещественной операции слишком велик, чтобы представлять вещественное число, то он заменяется на предопределенное значение INF с тем же знаком, что и исходный результат. Заметим, что это правило применимо и к $1.0/0.0$, но не к $0.0/0.0$, т.к. последнее выражение не имеет определенного результата вообще и приводит к ошибке при выполнении программы. Если используются одноместные операции, - обозначает переменную знака, а + обозначает тождественную операцию. Операции DIV и MOD применимы только к целым операндам. Они связаны следующими формулами:

$$x = (x \text{ DIV } y) * y + (x \text{ MOD } y)$$

$$0 \leq (x \text{ MOD } y) < y \text{ или } 0 \geq (x \text{ MOD } y) > y$$

Заметим, что: $x \text{ DIV } y = \text{ENTIER}(x / y)$

Примеры:

x	y	x DIV y	x MOD y
5	3	1	2
-5	3	-2	1
5	-3	-2	-1
-5	-3	1	-2

Заметим, что:

$$(-5) \text{ DIV } 3 = -2$$

но

$$-5 \text{ DIV } 3 = -(5 \text{ DIV } 3) = -1$$

8.2.3 Операции над множествами

- + объединение
- разность ($x - y = x * (-y)$)
- * пересечение
- / симметричная разность ($x / y = (x-y) + (y-x)$)

Операции над множествами применимы к операндам типа SET и дают результат типа SET. Одноместный минус обозначает дополнение множества x , т.е. $-x$ обозначает множество целых от 0 до MAX(SET), которые не являются элементами множества x . Операции над множествами не являются ассоциативными ($(a+b)-c \neq a+(b-c)$).

Конструктор множества определяет значение множества перечислением его элементов между фигурными скобками. Элементы должны быть целыми в диапазоне 0..MAX(SET). Диапазон $a..b$ обозначает все целые i такие, что $i \geq a$ и $i \leq b$.

8.2.4 Операции над цепочками

- + конкатенация

Операция конкатенации применяется к операндам, имеющим типы цепочек. Получающаяся цепочка состоит из литер первого операнда, за которыми следуют литеры второго операнда. Если оба операнда имеют тип Shortstring, то типом результата будет Shortstring, в противном случае тип результата будет String.

8.2.5 Отношения

- = равно
- # неравно
- < меньше
- <= меньше или равно
- > больше
- >= больше или равно
- IN принадлежность множеству
- IS проверка типа

Отношения дают результат типа BOOLEAN. Отношения =, #, <, <=, > и >= применимы к числовым типам, литерным типам и типам цепочек литер. Отношения = и # применимы также к BOOLEAN и SET, а также к указательным и процедурным типам (включая значение NIL). $x \text{ IN } s$ означает « x является элементом s ». x должен быть целым в диапазоне 0..MAX(SET), а s иметь тип SET. $v \text{ IS } T$ означает «динамический тип переменной v равен T (или потомку типа T)» и называется *проверкой типа*. Она применима, если

1. v является IN или VAR параметром записевого типа или указателем на записевой тип, и если

2. T является потомком статического типа v

Примеры выражений (см. примеры в гл. 7):

1991	INTEGER
$i \text{ DIV } 3$	INTEGER
$\sim p \text{ OR } q$	BOOLEAN
$(i+j) * (i-j)$	INTEGER
$s - \{8, 9, 13\}$	SET
$i + x$	REAL
$a[i+j] * a[i-j]$	REAL
$(0 \leq i) \& (i < 100)$	BOOLEAN
$t.\text{key} = 0$	BOOLEAN
$k \text{ IN } \{i..j-1\}$	BOOLEAN
$w[i].\text{name} \leq "John"$	BOOLEAN
$t \text{ IS } \text{CenterTree}$	BOOLEAN

9. Операторы [statements]

Операторы обозначают действия. Есть элементарные и структурированные операторы. Элементарные операторы не содержат частей, которые сами являлись бы операторами. Это: присваивание, вызов процедуры, оператор возврата RETURN и оператор выхода EXIT. Структурированные операторы состоят из частей, которые сами являются операторами. Они используются для выражения последовательного, условного, выборочного и повторяющегося выполнения. Оператор может быть пустым, и в этом случае он обозначает отсутствие действия. Пустой оператор разрешен, чтобы ослабить правила пунктуации в операторных последовательностях.

Statement = [Assignment | ProcedureCall | IfStatement | CaseStatement |
WhileStatement | RepeatStatement |
ForStatement | LoopStatement | WithStatement |
EXIT | RETURN [Expression]].

9.1 Присваивания

Присваивание заменяет текущее значение переменной новым, определяемым неким выражением. Выражение должно быть *совместимым по присваиванию* с переменной (см. Приложение А). Операция присваивания записывается как «:=» и произносится *становится равной*.

Assignment = Designator «:=» Expression.

Если выражение e типа T_e присваивается переменной v типа T_v , происходит следующее:

1. если T_v и T_e имеют записевой тип, то присваиваются все поля этого типа;
2. если T_v и T_e имеют указательные типы, то динамическим типом v становится динамический тип e ;
3. если T_v — массив *литерного типа*, а e — цепочка литер длины $m < LEN(v)$, то $v[i]$ становится равным e_i для $i = 0..m-1$, а $v[m]$ становится равным 0X. Если $m \geq LEN(v)$, то в процессе выполнения генерируется ошибка.

Примеры присваиваний (см. примеры в гл. 7):

```
i := 0
p := i = j
x := i + 1
```

```

k := Log2(i+j)
F := Log2      (* см. 10.1 *)
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].name := "John"
t := c

```

9.2 Вызовы процедур

Вызов процедуры активизирует процедуру. Он может содержать список фактических параметров, которые замещают соответствующие формальные параметры, определенные в описании процедуры (см. гл. 10). Соответствие устанавливается по положению параметров в списках фактических и формальных параметров. Имеется два типа параметров: *параметры-переменные* и *параметры-значения*.

Если формальный параметр является параметром-переменной, то соответствующий фактический параметр должен быть составным именем некоторой переменной. Если он обозначает элемент структурированной переменной, то соответствующие селекторы вычисляются при подстановке фактического параметра вместо формального, т.е. до выполнения процедуры. Если формальный параметр — параметр-значение, то соответствующий фактический параметр должен быть выражением. Это выражение вычисляется до активизации процедуры, а получающееся значение присваивается формальному параметру (см. также 10.1).

ProcedureCall = Designator [ActualParameters].

Примеры:

```

WriteInt(i*2+1)    (* см. 10.1 *)
INC(w[k].count)
t.Insert("John")  (* см. 11 *)

```

9.3 Операторные последовательности

Операторная последовательность обозначает последовательность действий, указанных отдельными операторами, разделенными точками с запятой.

StatementSequence = Statement {«;» Statement}.

9.4 Условный оператор IF

```

IfStatement =
    IF Expression THEN StatementSequence
    {ELSIF Expression THEN StatementSequence}
    [ELSE StatementSequence]
    END.

```

Оператор IF задает условное выполнение охраняемых операторных последовательностей. Логическое выражение, предшествующее операторной последовательности, называется его *охраной*. Охраны вычисляются в том порядке, в котором они встречаются в тексте, до тех пор, пока одна из них не даст значение TRUE, после чего выполняется соответствующая операторная последовательность. Если ни одна охрана не будет удовлетворена, выполняется операторная последовательность, следующая за лексемой ELSE, если таковая имеется.

Пример:

```

IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = "'") OR (ch = "\"") THEN ReadString
ELSE SpecialCharacter

```



```
FOR v := beg TO end BY step DO statements END
```

эквивалентен следующему

```
temp := end; v := beg;
IF step > 0 THEN
    WHILE v <= temp DO statements; v := v + step END
ELSE
    WHILE v >= temp DO statements; v := v + step END
END
```

temp имеет *одинаковый* тип с переменной *v*. *step* должно быть ненулевым константным выражением. Если выражение *step* не указано, оно берется равным 1.

Примеры:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY -1 DO a[i] := a[i-1] END
```

9.9 Безусловный цикл LOOP

Оператор LOOP описывает повторяющееся выполнение операторной последовательности. Оно прекращается выполнением оператора выхода EXIT внутри этой последовательности (см. 9.10).

```
LoopStatement = LOOP StatementSequence END.
```

Пример:

```
LOOP
    ReadInt(i);
    IF i < 0 THEN EXIT END;
    WriteInt(i)
END
```

Циклы LOOP полезны для выражения повторяющихся вычислений с несколькими точками выхода или в тех случаях, когда условие выхода находится в середине повторяющейся операторной последовательности.

9.10 Операторы возврата RETURN и выхода EXIT

Оператор возврата RETURN указывает прекращение процедуры. Он обозначается лексемой RETURN, за которой следует выражение, если речь идет о процедуре-функции. Тип выражения должен быть *совместим по присваиванию* (см. Приложение А) с типом результата, указанным в заголовке процедуры (см. гл. 10).

Процедуры-функции требуют наличия оператора возврата, указывающего значение-результат. В собственно процедурах оператор возврата неявно подразумевается в конце тела процедуры. Поэтому любой явный оператор возврата появляется как дополнительная (вероятно, исключительная) точка выхода.

Оператор выхода обозначается лексемой EXIT. Он означает, что выполнение охватывающего оператора LOOP должно быть прекращено, а выполнение программы должно быть продолжено с оператора, следующего за этим оператором LOOP. Оператор EXIT контекстуально, хотя и не синтаксически, связан с содержащим его оператором LOOP.

9.11 Операторы конкретизации типа WITH

Оператор WITH выполняет операторную последовательность в зависимости от результата проверки типа и применяет охрану типа к каждому вхождению проверяемой переменной внутри операторной последовательности.

```
WithStatement      = WITH [ Guard DO StatementSequence ]
                    {«|» [ Guard DO StatementSequence] }
```

[ELSE StatementSequence] END.

Guard = Qualident «:» Qualident.

Если v — параметр-переменная записевого типа или указательная переменная, и если ее статический тип T_0 , то значение оператора

WITH v : T_1 DO S_1 | v : T_2 DO S_2 ELSE S_3 END

таково: если динамическим типом переменной v оказался T_1 , то выполняется операторная последовательность S_1 , в которой v рассматривается как если бы ее статическим типом был T_1 ; в противном случае если динамическим типом v оказался T_2 , то выполняется S_2 , в которой v рассматривается как если бы ее статическим типом был T_2 ; в противном случае выполняется S_3 . T_1 и T_2 должны быть потомками типа T_0 . Если ни одна проверка типа не дала положительного результата и если отсутствует ELSE (вместе с соответствующей операторной последовательностью), то программа аварийно останавливается.

Пример:

WITH t : CenterTree DO i := t .width; c := t .subnode END

10. Описания процедур

Описание процедуры состоит из *заголовка процедуры* и *тела процедуры*. Заголовок задает идентификатор процедуры и ее *формальные параметры*. Для методов он также задает принимающий параметр [receiver parameter] и атрибуты (см. 10.2). Тело содержит объявления и операторы. Идентификатор процедуры повторяется в конце ее описания.

Есть два вида процедур: *обычные процедуры* и *процедуры-функции*.

Процедура-функция активизируется составным именем функции, выступающим в качестве части выражения, и дает результат, являющийся операндом выражения. Обычные процедуры активизируются вызовом процедуры. Процедура является процедурой-функцией, если список ее формальных параметров задает некоторый тип результата. Тело процедуры-функции должно содержать оператор возврата, который определяет ее результат.

Все константы, переменные, типы и процедуры, описанные внутри тела процедуры, являются *локальными* по отношению к этой процедуре. Поскольку процедуры тоже могут описываться как локальные объекты, описания процедур могут быть вложены друг в друга. Вызов процедуры внутри ее описания означает рекурсивную активизацию.

Локальным переменным указательных **или процедурных** типов присваивается значение NIL перед выполнением тела процедуры.

Объекты, описанные в контексте, окружающем процедуру, видимы также и в тех частях процедуры, где они не перекрыты локально определенным объектом с точно тем же именем.

ProcedureDeclaration = ProcedureHeading [«;» ProcedureBody ident].

ProcedureHeading = PROCEDURE [Receiver] IdentDef
[FormalParameters] MethAttributes.

ProcedureBody = DeclarationSequence
[BEGIN StatementSequence] END.

DeclarationSequence = {CONST {ConstantDeclaration «;»} |
TYPE {TypeDeclaration «;»} |
VAR {VariableDeclaration «;»} }
{ProcedureDeclaration «;» | ForwardDeclaration «;»}.

ForwardDeclaration = PROCEDURE «^» [Receiver] IdentDef
[FormalParameters] MethAttributes.

Если в описании процедуры указан *принимающий параметр* [*receiver*], то процедура считается методом, связанным с типом принимающего параметра (см. 10.2). *Упреждающее описание* служит для того, чтобы разрешить ссылки на процедуру, чье фактическое описание содержится дальше по тексту. Списки формальных параметров упреждающего и фактического описаний должны *соответствовать* (см. Приложение А), а имена соответствующих параметров должны быть идентичными.

10.1 Формальные параметры

Формальные параметры суть идентификаторы, описанные в списке формальных параметров процедуры. Они соответствуют фактическим параметрам, указанным в вызове процедуры. Соответствие между формальными и фактическими параметрами устанавливается при вызове процедуры. Есть два вида параметров: *параметры-значения* и *параметры-переменные*, последние отмечены в списке формальных параметров одним из описателей VAR, IN или OUT. Параметры-значения представляют собой локальные переменные, которым в качестве начального значения присваивается значение соответствующего фактического параметра. Параметры-переменные соответствуют фактическим параметрам, являющимся переменным, и они представляют эти переменные. Параметры-переменные могут использоваться только для ввода данных (IN), только для вывода данных (OUT) или для ввода и вывода (VAR). Описатель IN можно использовать только для параметров, являющихся массивами или записями. Внутри процедуры IN-параметры доступны только для чтения. Как и локальные переменные, OUT-параметры указательных **или процедурных** типов инициализируются в NIL. Значения других выходных параметров должны рассматриваться как неопределенные до первого присваивания в процедуре. Область видимости формального параметра распространяется от его описания до конца процедурного блока, в котором он описан. Процедура-функция без параметров должна иметь пустой список параметров. Она должна вызываться с помощью составного имени, у которого есть пустой список фактических параметров. Тип результата процедуры не может быть ни записью, ни массивом.

FormalParameters = «(» [FPSection {«;» FPSection}] «)» [«:» Type].

FPSection = [VAR | IN | OUT] ident {«;» ident} «:» Type.

Пусть *f* — формальный параметр, и пусть *a* — соответствующий фактический параметр. Если *f* — открытый массив, то *a* должен быть *совместим по массивам с f*, и длины *f* берутся из *a*. В противном случае *a* должен быть *совместим по параметрам с f* (см. Приложение А).

Примеры описаний процедур:

```
PROCEDURE ReadInt (OUT x: INTEGER);
  VAR i: INTEGER; ch: CHAR;
BEGIN
  i := 0; Read(ch);
  WHILE ("0" <= ch) & (ch <= "9") DO
    i := 10 * i + (ORD(ch) - ORD("0")); Read(ch)
  END;
  x := i
END ReadInt

PROCEDURE WriteInt (x: INTEGER);      (* 0 <= x < 100000 *)
  VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN
```

```

    i := 0;
    REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i) UNTIL x = 0;
    REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt

PROCEDURE WriteString (IN s: ARRAY OF CHAR);
    VAR i: INTEGER;
BEGIN
    i := 0; WHILE (i < LEN(s)) & (s[i] # 0X) DO Write(s[i]); INC(i) END
END WriteString

PROCEDURE Log2 (x: INTEGER): INTEGER;
    VAR y: INTEGER; (* assume x > 0 *)
BEGIN
    y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
    RETURN y
END Log2

PROCEDURE Modify (VAR n: Node);
BEGIN
    INC(n.key)
END Modify

```

10.2 Методы

Процедуры, описанные глобально, могут быть связаны с каким-либо записевым типом, описанным в том же модуле. Такие процедуры называют *методами*, связанными с данным записевым типом. Связь выражается посредством указания типа *принимающего параметра* в заголовке описания процедуры. Получающий параметр может быть VAR или IN параметром типа *T* или параметром-значением типа POINTER TO *T*, где *T* — записевый тип. Метод связан с типом *T* и считается в нем локальным.

```

ProcedureHeading = PROCEDURE [Receiver] IdentDef
                  [FormalParameters] MethAttributes.
Receiver         = «(» [VAR | IN] ident «:» ident «)».
MethAttributes   = [«,» NEW] [«,» (ABSTRACT | EMPTY | EXTENSIBLE)].

```

Если метод *M* связан с типом *T0*, он также неявно связан с любым потомком *T1* типа *T0*. Однако если метод *M'* (с тем же именем, что и у *M*) описан как связанный с *T1*, он становится связан с *T1* вместо *M*. *M'* считается переопределением *M* для *T1*. Списки формальных параметров *M* и *M'* должны *соответствовать*, кроме случаев, когда *M* — процедура-функция, возвращающая указательный тип. В последнем случае тип результата функции *M'* должен быть потомком типа результата *M* (ковариантность) (см. Приложение А). Если *M* и *T1* экспортируются (см. гл. 4), то *M'* тоже должен экспортироваться.

Если *M* не экспортируется, то *M'* тоже не должен экспортироваться. Если *M* и *M'* экспортируются, их метки экспорта должны быть одинаковыми.

Для ограничения и документирования предполагаемого использования метода используются следующие атрибуты:

NEW, ABSTRACT, EMPTY и EXTENSIBLE

<соответственно: новый, абстрактный, пустой и переопределяемый>.

Необходимо использовать атрибут NEW для всех вновь вводимых методов, и его нельзя использовать для переопределяющих методов. Этот атрибут помогает

обнаружить несогласованности в определениях методов, связанных с типом и его потомками; такие несогласованности могут возникнуть при переименованиях методов.

Описания абстрактных и пустых методов состоят только из заголовка процедуры. Абстрактные методы не могут вызываться. Записевый тип, с которым связан абстрактный метод, должен быть абстрактным. Метод, переопределяемый абстрактным методом, должен быть абстрактным. Абстрактный метод экспортируемого записевого типа должен экспортироваться. Вызов пустого метода не имеет никакого эффекта. Пустые методы не могут быть процедурами-функциями и не могут иметь OUT параметров. Записевый тип, с которым связаны новые пустые методы, должен быть переопределяемым или абстрактным. Метод, переопределенный пустым методом, должен быть пустым или абстрактным. Абстрактные или пустые методы некоторого типа обычно переопределяются (реализуются) в его потомках. Они не должны вызываться супер-вызовами. Конкретный (не абстрактный) записевый тип, являющийся потомком абстрактного типа, должен реализовать все абстрактные методы своего типа-предка.

Конкретные методы (имеющие процедурное тело) должны быть либо переопределяемыми, либо конечными (т.е. описанными без атрибутов). Конечный метод не может переопределяться в потомках своего типа. Записевый тип, содержащий переопределяемые методы, должен быть переопределяемым или абстрактным.

Если v — составное имя, а M — метод, то $v.M$ обозначает метод M , связанный с динамическим типом v . Заметим, что такой метод может отличаться от одноименного метода, связанного со статическим типом переменной v . v передается принимающему параметру метода M' по правилам передачи параметров, указанным в 10.1.

Если r — принимающий параметр, описанный как имеющий тип T , то $r.M^{\wedge}$ означает метод M , связанный с типом-предком типа T (супер-вызов). В упреждающем и фактическом описаниях метода принимающие параметры должны быть соответствующих типов. Списки формальных параметров обоих описаний должны соответствовать (Приложение А), а имена соответствующих параметров должны быть идентичны.

Методы, помеченные символом «-», считаются экспортированными *только для реализации*. Такой метод может быть переопределен в любом импортирующем модуле, но может вызываться только в модуле, содержащем описание метода. (В настоящее время компилятор допускает супер-вызовы методов, экспортированных только для реализации, вне их определяющего модуля. Это допускается временно для облегчения переноса программ.)

Примеры:

```
PROCEDURE (t: Tree) Insert (node: Tree), NEW, EXTENSIBLE;
  VAR p, father: Tree;
BEGIN p := t;
  REPEAT father := p;
    IF node.key = p.key THEN RETURN END;
    IF node.key < p.key THEN p := p.left ELSE p := p.right END
  UNTIL p = NIL;
  IF node.key < father.key THEN
    father.left := node
  ELSE
    father.right := node
  END;
```

```
node.left := NIL; node.right := NIL
END Insert
```

```
PROCEDURE (t: CenterTree) Insert (node: Tree); (* redefinition *)
BEGIN
  WriteInt(node(CenterTree).width);
  t.Insert^ (node) (* calls the Insert method of Tree *)
END Insert
```

```
PROCEDURE (obj: Object) Draw (w: Window), NEW, ABSTRACT
```

```
PROCEDURE (obj: Object) Notify (e: Event), NEW, EMPTY
```

10.3 Предопределенные процедуры

Следующая таблица дает список предопределенных процедур. Некоторые процедуры являются обобщенными, т.е. они применимы к разным типам операндов. v представляет переменную, x и y — выражения, а T — тип. Первая подходящая строка дает правильный тип результата.

Процедуры-функции

Имя	Тип аргументов	Тип результата	Функция
ABS(x)	\leq INTEGER вещественный тип, LONGINT	INTEGER тип x	абсолютное значение
ASH(x, y)	x : \leq INTEGER x : LONGINT y : целый тип	INTEGER LONGINT	арифметический сдвиг ($x * 2^y$)
BITS(x)	INTEGER	SET	$\{i \mid \text{ODD}(x \text{ DIV } 2^i)\}$
CAP(x)	литерный тип	тип x	x — буква из набора Latin-1: соответствующая большая буква
CHR(x)	целый тип	CHAR	литера с порядковым номером x
ENTIER(x)	вещественный тип	LONGINT	наибольшее целое, не превосходящее x
LEN(v, x)	v : массив; x : целая константа	INTEGER	длина v в измерении x (первое измерение = 0)
LEN(v)	массивовый тип String <цепочка>	INTEGER INTEGER	эквивалентно LEN($v, 0$) длина цепочки (не считая 0X)
LONG(x)	BYTE SHORTINT INTEGER SHORTREAL SHORTCHAR Shortstring	SHORTINT INTEGER LONGINT REAL CHAR String	тождество
MAX(T)	T = элементарный тип	T	максимальное значение для типа T

	T = SET	INTEGER	максимальный элемент множества
MAX(x, y)	<= INTEGER целый тип	INTEGER LONGINT	большее значение из x и y
	<= SHORTREAL числовой тип	SHORTREAL REAL	
	SHORTCHAR литерный тип	SHORTCHAR CHAR	
MIN(T)	T = элементарный тип	T	минимальное значение для типа T
	T = SET	INTEGER	0
MIN(x, y)	<= INTEGER целый тип	INTEGER LONGINT	меньшее значение из x и y
	<= SHORTREAL числовой тип	SHORTREAL REAL	
	SHORTCHAR литерный тип	SHORTCHAR CHAR	
ODD(x)	целый тип	BOOLEAN	$x \text{ MOD } 2 = 1$
ORD(x)	CHAR	INTEGER	порядковый номер литеры x
	SHORTCHAR	SHORTINT	порядковый номер литеры x
	SET	INTEGER	$(\text{SUM } i: i \text{ IN } x: 2^i)$
SHORT(x)	LONGINT	INTEGER	тождество
	INTEGER	SHORTINT	тождество
	SHORTINT	BYTE	тождество
	REAL	SHORTREAL	тождество (возможно усечение)
	CHAR	SHORTCHAR	проекция
	String	Shortstring	проекция
SIZE(T)	любой тип	INTEGER	количество байт, требуемое для T

SIZE не может применяться в константных выражениях, т.к. его значение зависит от фактической реализации компилятора.

Собственно процедуры

Имя	Типы аргументов	Функция
ASSERT(x)	x: логическое выражение	остановить программу, если не x
ASSERT(x, n)	x: логическое выражение; n: целая константа	остановить программу, если не x
DEC(v)	целый тип	$v := v - 1$
DEC(v, n)	v, n: целый тип	$v := v - n$
EXCL(v, x)	v: SET; x: целый тип, $0 \leq x \leq \text{MAX}(\text{SET})$	$v := v - \{x\}$
HALT(n)	целая константа	остановить программу
INC(v)	целый тип	$v := v + 1$
INC(v, n)	v, n: целый тип	$v := v + n$
INCL(v, x)	v: SET; x: целый тип,	$v := v + \{x\}$

	$0 \leq x \leq \text{MAX}(\text{SET})$	
NEW(v)	указатель на запись или фиксированный массив	разместить $v \wedge$
NEW(v, x0, ..., xn)	v: указатель на открытый массив; xi: целый тип	разместить $v \wedge$ с длинами x0.. xn

В ASSERT(x, n) и HALT(n) интерпретация n определяется конкретной реализацией.

10.4 Финализация

Предопределенный метод с именем FINALIZE связан с каждым записевым типом как если бы он был описан как связанный с типом ANYREC:

```
PROCEDURE (a: ANYPTR) FINALIZE-, NEW, EMPTY;
```

Метод FINALIZE может быть реализован для любого указательного типа. Этот метод вызывается в неопределенное время после того, как объект этого типа (или соответствующего базового типа) стал недоступен через другие указатели (перестал быть глобально доступным) и до того как память, занимаемая объектом, утилизируется системой.

Не рекомендуется вновь делать этот объект глобально доступным внутри этого метода, и этот метод не вызывается повторно, если объект снова станет недоступным. Порядок финализации нескольких недоступных объектов не определен.

11. Модули

Модуль это набор описаний констант, типов, переменных и процедур вместе с некоторой операторной последовательностью для присваивания начальных значений переменным <a также с некоторой операторной последовательностью финализации>. Модуль представляет собой текст, компилируемый как целое.

```
Module      = MODULE ident «;» [ImportList] DeclarationSequence
              [BEGIN StatementSequence]
              [CLOSE StatementSequence] END ident «.».
```

```
ImportList  = IMPORT Import {«,» Import} «;».
```

```
Import     = [ident «:=»] ident.
```

Список импорта указывает имена импортируемых модулей. Если модуль A импортируется модулем M , и A экспортирует идентификатор x , то внутри M на x ссылаются как $A.x$. Если A импортируется как $B := A$, то на объект x следует ссылаться как $B.x$. Благодаря этому возможно использовать краткие псевдонимы для имен модулей в уточненных идентификаторах. Модуль не может импортировать сам себя. Идентификаторы, которые нужно экспортировать (т.е. которые должны быть видимы в модулях-клиентах), должны быть надлежащим образом помечены в описаниях (см. главу 4).

Операторная последовательность, следующая за BEGIN, выполняется, когда модуль добавляется к системе (загружается), что выполняется после загрузки импортируемых модулей. Следовательно, циклический импорт модулей невозможен. Отдельные экспортированные процедуры могут быть активизированы системой, и такие процедуры служат *командами*.

Содержимое переменных, описанных в модуле, очищается перед выполнением тела модуля. Это подразумевает, что все переменные указательных или процедурных типов инициализируются в NIL.

Операторная последовательность, следующая за CLOSE, выполняется, когда модуль удаляется из системы.

Пример:

```
MODULE Trees;      (* exports: Tree, Node, Insert, Search, Write, Init *)
  IMPORT StdLog;
  TYPE
    Tree* = POINTER TO Node;
    Node* = RECORD   (* exports read-only: Node.name *)
      name-: POINTER TO ARRAY OF CHAR;
      left, right: Tree
    END;
  PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR), NEW;
    VAR p, father: Tree;
  BEGIN
    p := t;
    REPEAT father := p;
      IF name = p.name^ THEN RETURN END;
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    UNTIL p = NIL;
    NEW(p); p.left := NIL; p.right := NIL;
    NEW(p.name, LEN(name$) + 1); p.name^ := name$;
    IF name < father.name^ THEN father.left := p ELSE father.right := p END
  END Insert;
  PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree, NEW;
    VAR p: Tree;
  BEGIN
    p := t;
    WHILE (p # NIL) & (name # p.name^) DO
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    END;
    RETURN p
  END Search;
  PROCEDURE (t: Tree) Write*, NEW;
  BEGIN
    IF t.left # NIL THEN t.left.Write END;
    StdLog.String(t.name); StdLog.Ln;
    IF t.right # NIL THEN t.right.Write END
  END Write;
  PROCEDURE Init* (t: Tree);
```

```

BEGIN
    NEW(t.name, 1); t.name[0] := 0X; t.left := NIL; t.right := NIL
END Init;

BEGIN
    StdLog.String("Trees loaded"); StdLog.Ln
CLOSE
    StdLog.String("Trees removed"); StdLog.Ln
END Trees.

```

Приложение А: Определения терминов

Литерные типы	SHORTCHAR, CHAR
Целые типы	BYTE, SHORTINT, INTEGER, LONGINT
Вещественные типы	SHORTREAL, REAL
Числовые типы	целые и вещественные типы
Типы цепочек литер	Shortstring, String
Основные типы	BOOLEAN, SET, литерные и числовые типы

Одинаковые типы [Same types]

Две переменные a и b с типами Ta и Tb имеют *одинаковый* тип, если

1. Ta и Tb оба обозначены одним и тем же идентификатором типа, или
2. Ta описан в описании типа вида $Ta = Tb$, или
3. a и b появляются в одном списке идентификаторов в описании переменных, полей записи или формальных параметров.

Эквивалентные типы [Equal types]

Два типа Ta и Tb *эквивалентны*, если

1. Ta и Tb имеют *одинаковый* тип, или
2. Ta и Tb суть типы открытых массивов с *эквивалентными* типами элементов, или
3. Ta и Tb суть *процедурные типы, чьи списки формальных параметров соответствуют*.
4. Ta и Tb суть указательные типы с *эквивалентными* базовыми типами.

Соответствие списков формальных параметров [Matching formal parameter lists]

Два списка формальных параметров *соответствуют*, если

1. они имеют одинаковое число параметров, и
2. они имеют либо *эквивалентные* типы результатов функции, либо не имеют никаких, и
3. параметры в соответствующих позициях имеют *эквивалентные* типы, и
4. параметры в соответствующих позициях суть оба либо параметры-значения, либо IN, OUT или VAR параметры.

Включение типов [Type inclusion]

Числовые и литерные типы *включают* (значения) меньших типов того же класса в соответствии со следующими иерархиями:

REAL >= SHORTREAL >= LONGINT >= INTEGER >= SHORTINT >= BYTE
 CHAR >= SHORTCHAR

Типы-потомки и типы-предки [Type extension]

Если дано описание типа $T_b = RECORD (T_a) \dots END$, то T_b является непосредственным потомком типа T_a , а T_a — непосредственным предком типа T_b . Какой-либо тип T_b является потомком [extension] типа T_a (T_a является предком для T_b), если

1. T_a и T_b суть одинаковые типы, или
2. T_b — непосредственный потомок некоторого потомка типа T_a , или
3. T_a имеет тип ANYREC.

Если $P_a = POINTER TO T_a$ и $P_b = POINTER TO T_b$, то P_b — потомок типа P_a (P_a — предок для P_b), если T_b является потомком типа T_a .

Совместимость по присваиванию [Assignment compatible]

Выражение e типа T_e является совместимым по присваиванию с переменной v типа T_v , если выполняется одно из следующих условий:

1. T_e и T_v эквивалентны и не являются ни типами открытых массивов, ни записевыми типами, описанными с атрибутами ABSTRACT, EXTENSIBLE или LIMITED;
2. T_e и T_v — числовые или литерные типы, и T_v включает T_e ;
3. T_e и T_v — указательные типы, и T_e — потомок типа T_v ;
4. T_v — указательный или процедурный тип, и e есть NIL;
5. T_v — числовой тип, а e — константное выражение, чье значение содержится в T_v ;
6. T_v — массив CHAR, T_e — String или Shortstring, и $LEN(e) < LEN(v)$;
7. T_v — массив SHORTCHAR, T_e — Shortstring, и $LEN(e) < LEN(v)$;
8. T_v — процедурный тип, а e — имя процедуры, чьи формальные параметры соответствуют параметрам T_v .

Совместимость по массивам [Array compatible]

Фактический параметр a типа T_a является совместимым по массивам с формальным параметром f типа T_f , если

1. T_f и T_a — эквивалентны, или
2. T_f — открытый массив, T_a — произвольный массив, и типы их элементов совместимы по массивам, или
3. T_f — открытый массив CHAR, и T_a имеет тип String, или
4. T_f — открытый массив SHORTCHAR, и T_a имеет тип Shortstring.

Совместимость по параметрам [Parameter compatible]

Фактический параметр a типа T_a является совместимым по параметрам с формальным параметром f типа T_f , если

1. T_f и T_a — эквивалентны, или
2. f — параметр-значение, а T_a — совместим по присваиванию с T_f , или
3. f — IN или VAR параметр, T_f и T_a — записевые типы, и T_a — потомок типа T_f .

Совместимость по выражению [Expression compatible]

Для некоторой операции типы ее операндов *совместимы по выражению*, если они подчиняются правилам, суммированным в следующей таблице. Первая подходящая строка дает правильный тип результата. Тип T1 должен быть расширением типа T0:

операция	первый операнд	второй операнд	тип результата
+ - * DIV MOD	<= INTEGER целый тип	<= INTEGER целый тип	INTEGER LONGINT
/	целый тип	целый тип	REAL
+ - * /	<= SHORTREAL числовой тип	<= SHORTREAL числовой тип	SHORTREAL REAL
+	SET Shortstring тип цепочек	SET Shortstring тип цепочек	SET Shortstring String
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	числовой тип литерный тип тип цепочек	числовой тип литерный тип тип цепочек	BOOLEAN BOOLEAN BOOLEAN
= #	BOOLEAN SET NIL, указательный тип T0 или T1 процедурный тип T, NIL	BOOLEAN SET NIL, указательный тип T0 или T1 процедурный тип T, NIL	BOOLEAN BOOLEAN BOOLEAN BOOLEAN BOOLEAN
IN	целый тип	SET	BOOLEAN
IS	T0	тип T1	BOOLEAN

Константные выражения вычисляются при компиляции с максимальной точностью (LONGINT для целых типов, REAL для вещественных типов) и результат трактуется как буквальное <т.е. заданный непосредственно своим изображением> численное значение.

Приложение В: Синтаксис Компонентного Паскаля

Module <модуль>	=	MODULE ident «;» [ImportList] DeclSeq [BEGIN StatementSeq] [CLOSE StatementSeq] END ident «.».
ImportList <список импорта>	=	IMPORT [ident «:=»] ident {«,» [ident «:=»] ident} «;».
DeclSeq <последовательность описаний>	=	{ CONST {ConstDecl «;» } TYPE {TypeDecl «;» } VAR {VarDecl «;» } } {ProcDecl «;» ForwardDecl «;»}.
ConstDecl <описание константы>	=	IdentDef «=» ConstExpr.
TypeDecl <описание типа>	=	IdentDef «=» Type.
VarDecl <описание переменной>	=	IdentList «:=» Type.

ProcDecl	=	PROCEDURE [Receiver] IdentDef [FormalPars]
MethAttributes		
<описание процедуры>		[«;» DeclSeq [BEGIN StatementSeq] END ident].
MethAttributes	=	[«,» NEW] [«,» (ABSTRACT EMPTY EXTENSIBLE)].
<атрибуты методов>		
ForwardDecl	=	PROCEDURE «^» [Receiver] IdentDef [FormalPars]
MethAttributes.		
<упреждающее описание>		
FormalPars	=	«(» [FPSection {«;» FPSection}] «)» [«:» Type].
<формальные параметры>		
FPSection	=	[VAR IN OUT] ident {«,» ident} «:» Type.
<раздел формальных параметров>		
Receiver	=	«(» [VAR IN] ident «:» ident «)».
<принимающий параметр>		
Type	=	Qualident
<тип>		ARRAY [ConstExpr {«,» ConstExpr}] OF Type
		[ABSTRACT EXTENSIBLE LIMITED]
		RECORD [«(»Qualident«)»] FieldList {«;» FieldList}
END		POINTER TO Type
		PROCEDURE [FormalPars].
FieldList	=	[IdentList «:» Type].
<список полей>		
StatementSeq	=	Statement {«;» Statement}.
<операторная последовательность>		
Statement	=	[Designator «:=» Expr
<оператор>		Designator [«(» [ExprList] «)»]
		IF Expr THEN StatementSeq
		{ELSIF Expr THEN StatementSeq}
		[ELSE StatementSeq] END
		CASE Expr OF Case {« » Case}
		[ELSE StatementSeq] END
		WHILE Expr DO StatementSeq END
		REPEAT StatementSeq UNTIL Expr
		FOR ident «:=» Expr TO Expr [BY ConstExpr]
		DO StatementSeq END
		LOOP StatementSeq END
		WITH [Guard DO StatementSeq]
		{« » [Guard DO StatementSeq] }
		[ELSE StatementSeq] END
		EXIT
		RETURN [Expr]
].
Case	=	[CaseLabels {«,» CaseLabels} «:» StatementSeq].
<вариант выбора>		
CaseLabels	=	ConstExpr [«.» ConstExpr].

<метки выбора>	
Guard	= Qualident «:» Qualident.
<охрана>	
ConstExpr	= Expr.
<константное выражение>	
Expr	= SimpleExpr [Relation SimpleExpr].
<выражение>	
SimpleExpr	= [«+» «-»] Term {AddOp Term}.
<простое выражение>	
Term	= Factor {MulOp Factor}.
<слагаемое>	
Factor	= Designator number character string NIL Set
<сомножитель>	«(» Expr «)» «~» Factor.
Set	= «{» [Element {«,» Element}] «}».
<множество>	
Element	= Expr [«..» Expr].
Relation	= «=» «#» «<» «<=» «>» «>=» IN IS.
<отношение>	
AddOp	= «+» «-» OR.
<аддитивная операция>	
MulOp	= «*» «/» DIV MOD «&».
<мультипликативная операция>	
Designator	= Qualident {«.» ident «[» ExprList «]» «^» «(»
Qualident «)»	«(» [ExprList] «)»} [«\$»].
<составное имя>	
ExprList	= Expr {«,» Expr}.
<список выражений>	
IdentList	= IdentDef {«,» IdentDef}.
<список идентификаторов>	
Qualident	= [ident «.»] ident.
<уточненный идентификатор>	
IdentDef	= ident [«*» «-»].
<определение идентификатора>	

Приложение С: Диапазоны значений [domains] элементарных типов

Тип	Диапазон значений
BOOLEAN	FALSE, TRUE
SHORTCHAR	0X .. 0FFX
CHAR	0X .. 0FFFFX
BYTE	-128 .. 127
SHORTINT	-32768 .. 32767
INTEGER	-2147483648 .. 2147483647
LONGINT	-9223372036854775808 .. 9223372036854775807
SHORTREAL	-3.4E38 .. 3.4E38, INF (32-битный формат IEEE)
REAL	-1.8E308 .. 1.8E308, INF (64-битный формат IEEE)
SET	подмножество из 0 .. 31

Приложение D: Обязательные требования к среде выполнения

Определение Компонентного Паскаля опирается на три фундаментальных предположения.

- 1) Во время исполнения программ доступна информация, позволяющая проверять динамический тип объекта. Это нужно для реализации проверок типов и охраны типов.
- 2) Отсутствует процедура DISPOSE <освобождение памяти, занятой более не используемыми объектами>. Память не может быть освобождена по явной инструкции программиста, поскольку это создало бы проблемы безопасности, связанные с утечками памяти [memory leaks] и с висячими указателями [dangling pointers]. За исключением таких встроенных систем, где не используется динамическое управление памятью, или где ее можно разместить только однажды и никогда не нужно освобождать, требуется автоматический сбор мусора.
- 3) Модули и по крайней мере их экспортированные процедуры (команды) и экспортированные типы должны быть доступны динамически. В случае необходимости это может вызывать загрузку модулей. Программный интерфейс, используемый для загрузки модулей или для доступа к указанной мета-информации, не определяется языком, но компилятор должен сохранять эту информацию при генерации кода.

За исключением полностью слинкованных приложений, в которых при исполнении не нужно загружать никакие модули, для модулей требуется динамический загрузчик. Встроенные системы являются важными примерами приложений, которые могут быть полностью слинкованы.

Реализация, которая не удовлетворяет этим требованиям к компилятору и среде выполнения, не считается удовлетворяющей определению Компонентного Паскаля.

Приложение В: Отличия Компонентного Паскаля от Паскаля

[ENGLISH](#)

© Английский оригинал: Oberon microsystems, 1994-2001.

© Перевод на русский язык: Ф.В.Ткачев, апрель 2001, март 2009.

Исключенные средства

- **Типы-диапазоны**

Используйте один из стандартных целых типов.

- **Перечислимые типы**

Используйте вместо них целые константы.

- **Произвольные диапазоны для массивов**

Массивы теперь всегда определены над целым диапазоном 0..max-1.

Пример

```
A = ARRAY 16 OF INTEGER (* разрешены индексы из диапазона 0..15 *)
```

- **Нет общих множеств**

Тип SET теперь описывает набор целых чисел, который может включать элементы 0..31.

- **Нет явного оператора DISPOSE**

Неиспользуемая более память автоматически собирается сборщиком мусора. Вместо DISPOSE, просто присвойте переменной значение NIL.

- **Нет вариантных записей**

Используйте расширение (расширенное переопределение) записей.

- **Нет упакованных структур**

Используйте типы SHORTCHAR или BYTE для значений, умещающихся в байт.

- **Нет GOTO**

- **Нет стандартных функций PRED и SUCC**

Используйте DEC и INC для целых значений.

- **Нет встроенных средств ввода/вывода**

Нет файловых типов. Ввод/вывод обеспечивается библиотечными процедурами.

Измененные средства

- **Стандартная процедура ENTIER вместо ROUND**

- **Синтаксис для констант типа REAL**

3.0E+4, но не 3.0e+4

- **Синтаксис для объявлений указательных типов**

P = POINTER TO R

вместо

P = ^R

• Синтаксис для оператора CASE

"|" вместо ";" в качестве разделителя случаев.
Предложение ELSE.

Пример

```
CASE i * 3 - 1 OF
  0: StdLog.String("нуль")
  | 1..9: StdLog.String("от единицы до девяти")
  | 10, 20: StdLog.String("десять или двадцать")
ELSE StdLog.String("что-то еще")
END
```

• Имя процедуры должно быть повторено

Пример

```
PROCEDURE DrawDot (x, y: INTEGER);
BEGIN
END DrawDot;
```

• Большие и малые буквы различаются

Пример "proc" не то же самое, что "Proc".

• Синтаксис литерных цепочек

Литерные цепочки-константы заключаются между " или между '. В одной цепочке не может быть одновременно одиночных и двойных кавычек. Литерные цепочки-константы единичной длины могут присваиваться литерным переменным.

Пример

```
"That's great" 'Write "hello world" to the screen'
ch := "x"
ch := 'x'
```

• Комментарии

Комментарии заключаются между (* и *) и могут быть вложены.

• Скобки для множеств

Константы-множества задаются между { и } вместо [и].

Пример {0..2, 4, j..2 * k}

• Синтаксис функций

Используйте ключевое слово PROCEDURE также и для функций вместо FUNCTION.

Процедуры, возвращающие значение, всегда должны иметь (возможно пустой) список параметров в своих объявлениях и в вызовах.

Результат функции возвращается явно оператором RETURN, вместо присваивания имени функции.

Пример

```
PROCEDURE Fun (): INTEGER;
BEGIN
  RETURN 5
END Fun;
```

вместо

```
FUNCTION Fun: INTEGER;
BEGIN
```

```
        Fun := 5
    END;

    n := Fun()  вместо  n := Fun
```

• Объявления

Последовательность объявлений теперь имеет вид
{ ConstDecl | TypeDecl | VarDecl } { ProcDecl | ForwardDecl }
вместо
[ConstDecl] [TypeDecl] [VarDecl] {ProcDecl}.

Упреждающее (forward) объявление необходимо, если процедура используется до ее определения.

Пример

```
    PROCEDURE ^ Proc;
    вместо
    PROCEDURE Proc; FORWARD;
```

• Процедурные типы

Процедуры могут быть не только переданы в качестве параметров, но и присваиваться переменным процедурных типов.

Пример

```
    TYPE P = PROCEDURE (x, y: INTEGER);
    VAR v: P;
    v := DrawDot; (* присваивание *)
    v(3, 5);      (* вызов DrawDot(3, 5) *)
```

• Явные END вместо составных операторов

BEGIN может появляться только перед последовательностью операторов, но не внутри ее. IF, WHILE и LOOP всегда заканчиваются ключевым словом END.

• Оператор WITH

Оператор WITH является охраной типа, действующей в соответствующем программном фрагменте, он не подразумевает наличие скрытой переменной и не открывает новый диапазон видимости для переменных.

См. детали в описании языка.

• ELSIF

Операторы IF могут иметь несколько ветвей.

Пример

```
    IF name = "top" THEN
        StdLog.Int(0)
    ELSIF name = "bottom" THEN
        StdLog.Int(1)
    ELSIF name = "charm" THEN
        StdLog.Int(2)
    ELSIF name = "beauty" THEN
        StdLog.Int(3)
    ELSE
        StdLog.String("strange")
    END
```

- **BY вместо только DOWNTO в FOR**

Циклы FOR могут использовать любое константное значение в качестве приращения (положительного или отрицательного).

Пример

```
FOR i := 15 TO 0 BY -1 DO StdLog.Int(i, 0) END
```

- **Логические выражения используют «сокращенное» вычисление**

Вычисление логического выражения прекращается, как только его результат определен.

Пример

```
Следующее выражение не вызывает ошибки при выполнении, когда p = NIL:  
IF (p # NIL) & (p.name = "quark") THEN
```

- **Константные выражения**

В объявлениях констант допустимы не только буквальные константы, но и константные выражения.

Пример

```
CONST  
    zero = ORD("0");  
    one = zero + 1;
```

- **Разные операции**

используется вместо <> для проверки на неравенство.

& используется вместо AND для логической конъюнкции.

~ используется вместо NOT для логического отрицания.

- **Явное преобразование к меньшему типу с помощью SHORT**

Включение типов для числовых типов позволяет присваивать значения меньшего типа переменной большего типа. Присваивание в обратном направлении должно использовать стандартную процедуру SHORT.

Пример

```
int := shortint;  
shortint := SHORT(int)
```

Новые средства

- **Шестнадцатеричные числа и литеры**

Пример

```
100H (* десятичное 256 *)  
0DX (* возврат каретки *)
```

- **Дополнительные числовые типы**

Добавлены типы LONGINT, SHORTINT, BYTE, SHORTREAL.

- **Симметрическая разность множеств**

Множества могут вычитаться.

- **Новые стандартные процедуры**

Добавлены новые стандартные процедуры INC, DEC, INCL, EXCL, SIZE, ASH, HALT, ASSERT, LEN, LSH, MAX, MIN, BITS, CAP, ENTIER, LONG и SHORT.

- **LOOP с EXIT**

Имеется новый оператор цикла с явным оператором выхода. См. детали в сообщении о языке.

- **ARRAY OF CHAR могут сравниваться**

Литерные массивы могут сравниваться с помощью операций =, #, <, >, <= и >=.

- **Открытые массивы, многомерные массивы**

Можно определять массивы, не указывая их размера, возможно, с несколькими измерениями.

Пример

```
VAR a: POINTER TO ARRAY OF CHAR;  
NEW(a, 16)
```

```
PROCEDURE ScalarProduct (a, b: ARRAY OF REAL; VAR c: ARRAY OF REAL);
```

```
TYPE Matrix = ARRAY OF ARRAY OF REAL;  
PROCEDURE VectorProduct (a, b: ARRAY OF REAL; VAR c: Matrix);
```

- **Разыменование указателей не обязательно**

Операция разыменования ^ может быть опущена.

Пример

```
root.next.value := 5  
вместо  
root^.next^.value := 5
```

- **Модули**

Модули суть единицы компиляции, упрятывания информации, а также загрузки. Упрятывание информации -- одна из главных черт объектно-ориентированного программирования.

Возможны разные уровни упрятывания информации: полное упрятывание, экспорт только для чтения/реализации, полный экспорт.

См. детали в сообщении о языке.

- **Расширенное переопределение (расширение) типов**

Типы записей (указательные типы) могут переопределяться, обеспечивая таким образом полиморфизм. Полиморфизм -- одно из главных средств объектно-ориентированного программирования.

- **Методы**

Процедуры могут быть связаны с типами записей (указательными типами), таким образом обеспечивая позднее связывание [late binding]. Позднее связывание является одним из главных средств объектно-ориентированного программирования. Такие процедуры еще называются *методами*.

- **Операция с цепочками литер**

Литерная цепочка, содержащаяся в литерном массиве, может быть выбрана посредством селектора \$.

- **Атрибуты записей**

По умолчанию записи не могут быть расширены (переопределены), но могут быть помечены как EXTENSIBLE, ABSTRACT или LIMITED.

- **Атрибуты методов**

По умолчанию методы не могут быть расширены (переопределены), но могут быть помечены как EXTENSIBLE, ABSTRACT или EMPTY. Вновь вводимые методы должны быть помечены как NEW.