RUSSIAN(Русский) ENGLISH

Переключить цветовую схему
Эта функция использует куки.
Если браузер не принимает куки,
используется светлая тема.

Оглавление

Введение

Язык программирования **AL/IV** - это
минимальный
императивный
объектно-ориентированный
платформенно-независимый
язык высокого уровня
со статической типизацией,
претендующий на чрезвычайно высокую степень надежности кода
управляемого уровня.

Важные семантические особенности:

NONE-объекты вместо NULL-ов

- Поля **NONE**-объектов всегда равны **NONE**.
- Запись в поля **NONE**-объектов игнорируется.
- Чтение полей всегда дает нули и **NONE**-объекты соответствующих типов.
- **NONE**-объекты возвращаются в случае обращения за пределами массивов,
- и при потере объекта, на который имелась ссылка.
- Все объектные переменные всегда инициализированы значениями **NONE**
- и никогда не имеют значение **NULL**.

Разделение указателей на владеющие (сильные) и использующие_ (слабые)

- Сильные указатели не могут быть использованы для организации замкнутых цепочек взаимного владения.
- Не нужен механизм сборки мусора.
- Объект автоматически уничтожается, когда обнуляется его счетчик использования сильными указателями.
- При уничтожения объекта все слабые ссылки на него автоматически перенаправляются на **NONE**-объекты соответствующего класса.
- Новые объекты либо должны адресоваться сильным указателем, либо передаваться во владение ранее созданному объекту.

Полная изоляция данных потока от других параллельно работающих вычислительных потоков

- Не требуются механизмы для исключительного доступа к разделяемым объектам.
- Поток видит только свои объекты и не имеет доступа к объектам других потоков.
- Невозможны взаимные блокировки потоков из-за доступа к данным.
- Глобальные переменные в языке AL/IV отсутствуют, поэтому проблемы с одновременным доступом к ним не возникают.

Невозможность зацикливания и бесконечной рекурсии

- Циклы возможны только в форме
  FOR var IN arr[] : ...
  (нет циклов типа **WHILE / REPEAT-UNTIL**).
  Есть бесконечные циклы **FOR, INFINITE**, но их использование жёстко ограничено. Любой цикл **FOR** обязательно завершается, поэтому вечное зацикливание становится невозможным.
- Рекурсивные функции обязательно помечаются маркером **RECURSIVE** , и глубина рекурсии контролируется: в случае опасности бесконечной рекурсии выполняется возврат к рекурсивному вызову первого уровня в цепочке вызовов, и она возвращает **NONE**-значение.

Оператор PUSH для переменных

- В операторе **PUSH** для переменной сохраняется прежнее значение и может присваиваться новое.
- По окончании блока, для переменной гарантированно восстанавливается ее значение.

Параметры простых типов всегда передаются "по значению" и не могут изменяться в теле функции

- Повторное использование простых входных параметров для хранения промежуточных значений запрещено.

- Отсутствует возможность изменять значение скалярного параметра (кроме изменения полей объектного параметра).
- Для массива допускается изменение элементов, а для динамического массива-параметра - добавление и удаление элементов.

Поля и переменные класса "только для чтения"

- Упрощают создание "свойств", которые может изменять только владелец (класс и его наследники).

Перечисления

- Являются прямыми аналогами перечислений других языков высокого уровня.
- Позволяют работать с именованными константами и флажками.
- В качестве коллекции предлагается использовать булевские массивы с индексами типа этих перечислений.
- Элементы перечислений заключаются в одинарные кавычки.
- Оператор выбора **CASE** по перечислению требует указания всех возможных вариантов. В случае (например) расширения перечисления новыми значениями компилятор укажет как на ошибочные на все такие операторы CASE, в которых еще не определены действия для новых элементов.

Обработка исключений не требуется и не предполагается

- Передача управления в рекурсивную функцию верхнего уровня при превышении допустимого уровня рекурсии выполняется только для ускорения обнаружения сбоя и предотвращения "зависания" в случае большого числа вложенных циклов.
- Большинство ошибок подавляется путём замены предполагаемого результата значением **NONE**.
- В случае возникновения ошибок (работа с файлами, исключения в нативном коде и др.) они фиксируются в системном массиве ошибок, и в дальнейшем могут анализироваться в коде.

Встроенные в язык средства тестирования

- Функции **TEST** позволяют протестировать исходный код.
- Степень покрытия кода тестами оценивается компилятором.
- Модуль, недостаточно покрытый тестами, должен быть помечен модификатором **UNTESTED**.

Возможность переопределения операторов арифметики

- Переопределить можно только четыре операции (+, -, *, /).
- Любая функция, использующая переопределенные операторы, должна явно декларировать это в заголовке с указанием классов, операторы которых используются.

Важные синтаксические особенности:

Один класс - один модуль

- Один исходный текстовой файл содержит ровно один класс.
- Все публичные имена и параметры функций начинаются с заглавной буквы, все скрытые поля и локальные переменные - со строчной буквы.
- Подчёркивание на конце имени используется только для имён объектных полей класса, для индикации того, что поле является слабой ссылкой на объект.

Двойное именование типов, переменных, функций, констант

- Большинство объектов (переменных, функций, типов данных, перечислимых констант) имеют несколько имен.
- При декларации, полное имя делится на части символом '|', при этом первая часть до разделителя является кратким именем объекта.
- Кроме того, могут быть указаны другие варианты цепочек именования, отделенные символом '||'. Первый символ в цепочке должен быть совпадать (без учета регистра буквы). Например, **Name|_first|_variant || nam|e**
- Все сущности (за исключением переменных цикла **FOR**) обязаны иметь длинное имя не менее 8 символов.
- Ключевые слова языка записываются в верхнем регистре и не могут употребляться для именования функций и переменных.

Классы, перечисления и записи всегда упоминаются в фигурных скобках

- Имеется 5 простых типов данных: **BOOL**, **BYTE**, **INT|EGER**, **REAL**, **STR|ING**.
- Классы и перечисления именуются в фигурных скобках, например, **{My|_class}**.
- Имена классов в фигурных скобках начинаются с **П**рописной буквы. Имена перечислений и записей - со **с**трочной.
- Нет типа char (используется **STR**).

- Неявное преобразование типа возможно только из **BYTE** в **INT**, и из **INT** в **REAL**.

Ограничение длины строк кода

- Исходный код не должен содержать строки длиннее 80 символов (не считая комментариев и завершающих пробелов, и приравнивая символы табуляции к одному символу).
- Это правило не распространяется на строки, расположенные за финальной директивой **END** и на строки внутри длинных комментариев */ ... /*.

Правило продолжения оператора на другую строку

- Отсутствует специальный символ, завершающий простой оператор. Символ ';' используется для завершения блочных операторов **CASE**, **FOR** и т.п.
- Оператор обычно занимает одну строку.
- Код продолжается в следующей строке, если строка завершается одним из символов ',', '(', '['
- либо следующая строка не является пустой и не начинает новый оператор, т.е. не начинается с '{', '<', '[' или буквы,
- и не является завершением для блока или функции, т.е. не начинается с символов ';', '.'

Возможность вызова функции в постфиксной форме для статических однопараметрических функций

- Вызов **sin(x)** эквивалентен **x.sin**.

Массивы всегда упоминаются с квадратными скобками: A[ ]

- Имеются только одномерные массивы.
- Массив может быть динамическим или фиксированным.
- Для фиксированного массива в качестве индексов может использоваться перечисление (как тип).

Строка кода содержит только один оператор

- Символ '**;**'используется для завершения блочного оператора (**CASE**, **FOR**, **PUSH** , **DEBUG**, **SILENT**).
- Символ '.' завершает функцию, перечисление, блок констант, список импорта.
- Символ завершения ';' или '.' должен быть последним таким символом в строке (то есть два символа подряд '; ;' не допускаются), но указание '; .' в конце последней строки функции возможно.
- Любой простой оператор (кроме **BREAK**, **CONTINUE** или **STOP**) может быть завершён оператором выхода из функции '**==>**'

Ограничения на качество кода:

- Допускается не более трёх уровней вложения операторов **CASE** / **FOR** (Ограничения не затрагивают операторы **PUSH**, **DEBUG** , **SILENT**).
- Допускается четвертый уровень блочных операторов **CASE** / **FOR** , но с ограничением на один вложенный оператор.
- Допускается не более 7 простых и 7 блочных операторов подряд в одном блоке, после чего требуется наличие комментария вида **------------------ 'комментарий'**
- В случае нарушения любого из приведённых правил, класс должен быть помечен маркером плохого кода (**BAD**).

Перегрузка операторов

- Есть возможность переопределять основные операторы арифметики для классов и записей. Использование операторов в функции декларируется в заголовке функции.

В языке отсутствуют:

Препроцессор

- Нет макросов.
- ИСКЛЮЧЕНИЕ: Есть повторное использование кода - оператор **LIKE**. Но он макросом с параметрами не является, не допускает вложенных вызовов и обращений за пределами класса.
- Нет условной компиляции.
- Нет включения файлов.

Указатели функций

- Взамен указателей функций, имеется механизм виртуальных методов.

Массив как результат функции

- Функция может возвращать только скалярное значение.
- При необходимости возвратить массив, результат оформляется как параметр.

Множественное наследование
- Класс может быть унаследован только от одного класса - предка.

Оператор go to
- Есть возможность продолжения или выхода из глубоко вложенного цикла (**CONTINUE x** , **BREAK y**).

Символьный тип данных
- Для представления символа используется строчная переменная или константа длиной 1 символ.

Указание разрядности типов данных
- Разрядность простых типов данных не указывается.
- Она всегда одинакова для всех типов **INT** и **REAL** и зависит от целевой платформы, компилятора и настроек проекта (для целочисленных переменных, кроме счетчиков циклов, по умолчанию - 64 бита, с ключом /int32 - 32 бита)

Неявные преобразования разнородных типов данных
- Нет неявных преобразований переменных, кроме преобразования целого в вещественное (и выражений в строку - в операторах << вывода в консоль).
- Нет приведения типов данных.

Обработка исключений
- В программе практически невозможны исключения, связанные с неверной адресацией или порчей структур в оперативной памяти (благодаря использованию **NONE**-объектов, гарантированной инициализации переменных, проверкам выхода за границы массива).
- Во всех случаях, когда в обычных системах генерируется исключение, в AL/IV не выполняются никакие действия, и возвращается пустой результат (**NONE**).
- Соответственно, в AL/IV не требуется обработка исключений, и отсутствуют для этого средства.

I. Синтаксис операторов

I. 1. Форматирование операторов

I. 1. a. Один оператор - одна строка

Простые операторы не требуют завершения специальным знаком (таким, как ';').
Знак ';' используется для завершения блока.
Например:

```
d|escriminant = B * B - 4 * A * C
CASE d.Sign ?
     [-1]: NONE
     [0] : R[] << -B / 2 / A
     [+1]: R[] << (-B - d.Sqrt) / 2 / A
           R[] << (-B + d.Sqrt) / 2 / A  ;
```

Строка не может быть длиннее 80 символов (не учитывая комментарии до конца строки и пробелы и табуляции в конце строки). Символ табуляции считается за один символ.
Оператор может быть записан в нескольких строках,
но строка считается продолжающейся на следующую только если:

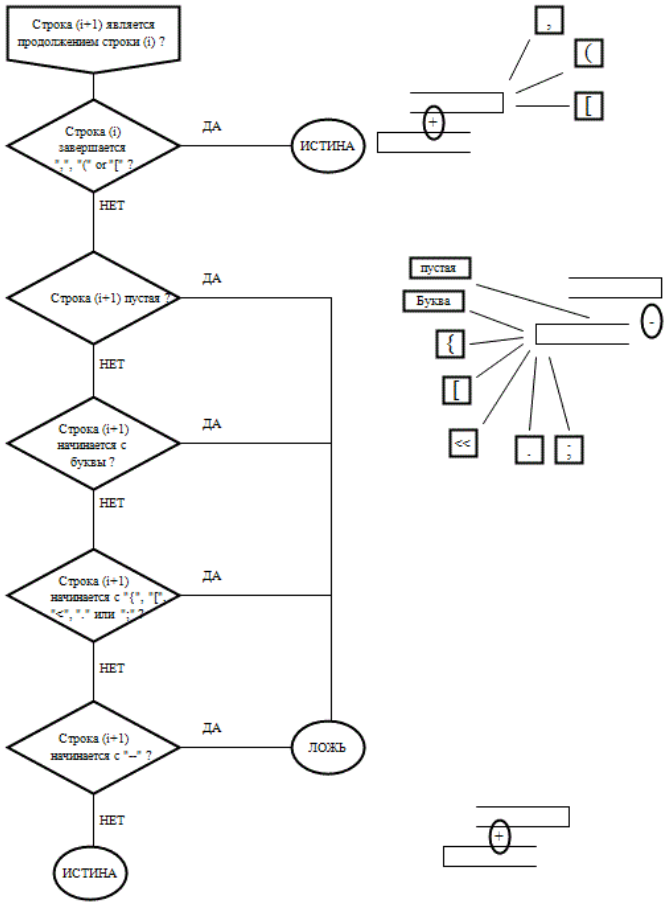
- строка **заканчивается символами** ',', '(', '[';
- следующая строка начинается символами:
  - '"' (двойная кавычка- начинает строчную константу)
  - '+'
  - '-' (одинарный знак минус)
  - '*'
  - '/'
  - '%'
  - '|'
  - '&'
  - '>'

- '<' (но не '<<')
- '='
- '#'

Другой способ запомнить правило переноса:
строка считается продолжающейся на следующую только если

- или **предыдущая строка завершается символами '(', '[', ','**
- или **следующая строка не пуста и не может начинать новый оператор**, т.е.:
  - **не начинается с буквы,**
  - **с символа '{'** (не начинает декларацию переменной),
  - **с символа '['** (не начинает условие для ветви CASE),
  - **с символа '<<'** (не начинает оператор вывода в консоль),
  - **с символов '.' или ';'** (не является отдельно стоящим символом для завершения блока кода или функции)
  - **с символа '--'** (не начинает блочный комментарий).

Или использовать иллюстрацию ниже:



Для упрощения чтения кода, содержащего длинные строки с переносами, рекомендуется отделять такие строки от соседних в блоке пустыми строками. (Компилятор выдает такую рекомендацию в виде предупреждения).

I. 1. b. Блочные операторы

Блочные операторы создают вложенный уровень операторов.
Блок вложенных операторов завершается символом ';'.
Функции и другие блоки уровня класса завершаются символом '.' (точка). А именно: заголовок класса, список импорта, определение типа перечисления, определение блока констант.
Строка не может завершаться двумя и более подряд одинаковыми символами завершения ';' или '.'. Но сочетание ';.' для завершения последней блочной конструкции в функции и самой функции допускается.
Если завершения требуют более двух блоков операторов, то символ завершения для внешнего блока будет находиться на отдельной строке. Например:

```
FOR i  IN [1  TO  100] :
     CASE i % 3 == 0 ? <<  "Fizz"  ;
     CASE i % 5 == 0 ? <<  "Buzz"  ;
     CASE i % 3 != 0 && i % 5 != 0 ?
         << i.Str  ;
;
```

Имеется только одно ключевое слово для условного оператора (**CASE**), одно для оператора цикла (**FOR**), один оператор сохранения и гарантированного восстановления (**PUSH**). Имеется так же оператор бесконечного цикла **FOR, INFINITE**, но его применение ограничено. Для отладки кода может использоваться блочный оператор **DEBUG**. Для подавления предупреждений компилятора на участке кода (например, о наличии не удаленных операторов **DEBUG**) имеется блочный оператор **SILENT**. Для ограничения времени выполнения кода на некотором участке имеется блок **LIMIT**.

Все прочие операторы являются простыми: **BREAK**, **CONTINUE**, **STOP**, **LIKE**, **REVERT**, вызовы функций, декларация и модификация переменных, ввод и вывод (**>>**, **<<**), возврат из функции (**==>**). Оператор **BREAK** может использоваться только для прекращения цикла и не используется для других целей.

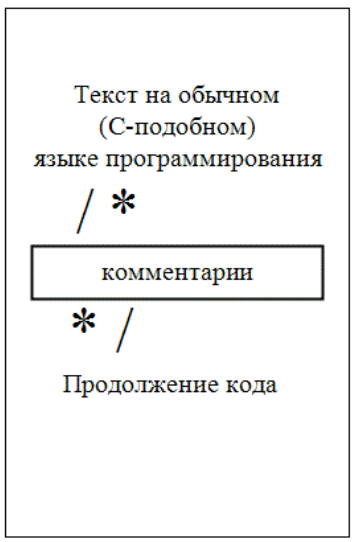
Компилятор требует помечать маркером **BAD** классы, в которых не соблюдаются правила структурирования кода:

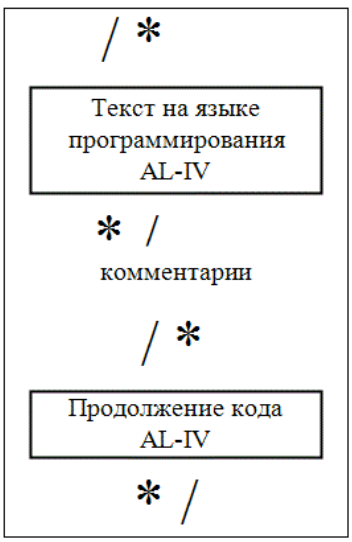
- Количество вложенных уровней операторов **CASE** и **FOR не должно превышать четырех** (блоки **PUSH** / **DEBUG** не учитываются). Четвертый уровень вложенности может содержать только один простой оператор (или конкатенацию простого оператора и знака оператора возврата из функции **==>**).
- Количество простых операторов в блоке не должно превышать семи.
- Количество вложенных блочных операторов в блоке так же не должно превышать семи.
- Специальный оператор-комментарий вида
  --------- 'текст комментария'
  может разделять группы операторов(число знаков '-' произвольно, но не менее двух), в этом случае подсчёт ведётся только внутри группы.

I. 1. c. Комментарии

- Многострочные комментарии начинаются в начале строки (не считая начальных пробелов) с символов
  /
  и заканчиваются строкой,
  завершающейся символами /
- Текстовой файл, содержащий исходный код, считается всегда начинающимся с многострочных комментариев. Признаком завершения таких комментариев является строка, завершающаяся символами **/***. Т.е., код AL-IV всегда должен находиться между строками **/*** и ***/** (как если бы он был комментарием для С-подобного языка программирования).

Иллюстрация:





- Комментарии до конца строки начинаются с двойной дроби (**//**). Не завершают текущий оператор.

- Комментарии, начинающиеся двумя символами --, имеют форму
  `-------------- 'текст'`
  и служат для разбиения блоков на разделы. Так же, используются в декларациях классов и в оглавлении модуля.
  Такой блочный комментарий может начинать переиспользуемый блок кода (операторы **LIKE**, **REVERT**). В этом случае он должен иметь модификатор **REUSED** (записывается через запятую после метки). Например:
  `---------------- 'start extracting ', REUSED`

I. 1. d. Регистрозависимость для идентификаторов

- ***КЛЮЧЕВЫЕ И ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА*** записываются только в **ВЕРХНЕМ** регистре.
  CASE являются ключевым словом, но **Case** , **case** - нет.
- ***{Имена классов}*** *(а так же перечислений и структур) заключаются в фигурные скобки.*
- ***имена простых типов*** данных являются зарезервированными: **BOOL**, **BYTE, INT** , **REAL**, **STR** (и других простых типов, кроме этих пяти, нет).
- ***{Имена классов}*** начинаются с прописных букв: **{My_class}**, **{Car|_mobile}**.
- ***{имена_перечислений_и_структур}*** начинаются со строчной буквы: **{color|s_fill}**.

Имена типов образуют особое пространство имен, не пересекающееся с прочими именами (переменных, констант, функций, процедур, модулей).

- Рекомендуется использовать КАПИТАЛИЗИРОВАННЫЕ имена для констант.
- ***Имена 'КОНСТАНТ'***, являющихся элементами перечислений, всегда заключаются в апострофы.
- ***Все имена*** регистро-зависимы.

Например, **A** и **a** - это разные переменные (или константы).
Имена **Публичных** полей, функций, параметров функций начинаются с Прописной буквы.
Члены класса, имена которых начинаются со строчной буквы, локальны в классе (но доступны в его наследниках).
Локальные переменные функций рекомендуется именовать со строчной буквы, параметры функции - с прописной.

- **Символ подчеркивания** в конце имени_ является неотъемлемой частью имени (и служит атрибутом слабой ссылки для объектов).

I. 1. e. Именование

При декларации переменной, функции, типа данных, элемента перечисления, таблицы - ее длинное имя может быть разделено на части символом '|'. Первая часть - это краткое имя. Так же, символом '||' от первой цепочки именования может быть отделена вторая цепочка, и так далее. Единственное требование: все цепочки должны начинаться с той же буквы (без учета регистра).

При этом:

- Длина хотя бы одного полного имени должна быть не менее 8 знаков:
  S|ample|_val (переменная доступна в коде по именам S, Sample и Sample_val);
- Исключение: переменным цикла необязательно давать длинное имя, допускается краткое имя короче 8 символов:
  `FOR x IN M[] : ... ;`
- Также: если при декларации переменной ей придан атрибут **INDEXING**, достаточно иметь краткое имя:
  `INT k INDEXING {record}`
- Если имя должно заканчиваться символом подчёркивания (признак слабой ссылки), то подчёркиванием должны заканчиваться обе части имени, например:
  `List_|of_items_ || L_items_ []`
- Для имён классов, записей и перечислений действует то же правило, и при подсчёте длины имени - окружающие фигурные скобки и разделители '|' не учитываются:
  {My|_class}, итого 8 символов.

Имена функций, переменных, полей классов и структур, таблиц, и именованных констант - это идентификаторы, начинающиеся с буквы.
Имена {Классов} - идентификаторы, заключаются в фигурные скобки и начинаются с прописной буквы.
Имена {структур} и {перечислений} так же идентификаторы в фигурных скобках, но именуются со строчной буквы.
Имена элементов перечислений записываются в апострофах. Так как это константы, то рекомендуется для них использовать КАПИТАЛИЗИРОВАННЫЕ имена.

I. 1. f. Модификаторы

Многие декларации языка (заголовок модуля, оператор, декларация переменной или типа данных) могут иметь модификаторы.

- записываются в форме **, ИМЯ** в конце оператора (или декларации), к которому относятся.
- Если модификаторов несколько, то они перечисляются через запятые.

Например, в заголовке класса:

```
CLASS    {My|_class} ,    BITWISE, RECURSIVE :
```

Модификаторы не воздействуют на основную семантику кода.
Они лишь помогают компилятору в обеспечении большей надежности / оптимальности по быстродействию / размеру кода и т.п.
Если в готовой программе, не содержащей ошибок, убрать все модификаторы (полагая, что в языке они не требуются), это (обычно) не изменит результат работы программы (но может повлиять на размер кода и скорость работы в сторону увеличения или уменьшения).

В некоторых случаях в качестве модификаторов используется наличие некоторого особого идентификатора в составе имени переменной (в любом регистре). Например, слово dummy в составе имени предотвращает выдачу компилятором предупреждений о том, что данная переменная или параметр не используется в коде. Слово temp в имени сообщает, что локальная переменная временная, и нет необходимости предупреждать программиста о том, что присвоенный ей создаваемый объект не будет существовать по окончании работы функции.

I. 2. Оператор присваивания

I. 2. a. Простой оператор присваивания

```
x = y
```
где **x** - переменная или поле структуры или объекта, **y** - выражение.

Для сравнения на равенство двух операндов используется си-подобный оператор **==**. (В SQL-подобных выражениях может использоваться одинарный знак '=' для сравнения на равенство).

I. 2. b. Операторы присваивания в комбинации с арифметической, логической или поразрядной логической операцией

Дополнительные операции **+=**, **-=**, ***=**, **/=**, **%=**, **|=**, **&=**, **^=**, **||=**, **&&=**, могут использоваться как эквивалент присваивания результата соответствующей операции **+**, **-**, *****,... между переменной, которой присваивается результат, и выражением справа от знака операции.

I. 2. c. Операторы отправки данных

Операция **<<** используется для добавления строки к строке, элемента к массиву и вывода текста в текущий главный поток вывода.

- Для строковой переменной **A** и строкового выражения **B** запись **A << B** означает эквивалентна оператору присваивания вида **A = A B**
- в случае массива **A[]** и выражения **B** запись **A[] << B** означает добавление элемента в конец массива и эквивалентна вызову функции **A[].Add(B)**
- Если операнд слева - объект класса, имеющего метод **Write**, то вызывается этот метод, с передачей выражения справа (строкового типа) в качестве результата;
- Если слева структура, а справа - объект класса, имеющего метод **Read** (возвращающий структуру), то вызывается этот метод, далее как в предыдущем пункте;
- Если левый операнд не указан, то это оператор записи строки в выходной поток (в консоль - для консольной программы).

Аналогичная операция **>>** используется *для добавления нового объекта в массив объектов, см. в разделе, посвященном классам и объектам.*

Справа от оператора >> может находиться декларация новой переменной.

- Операторы ввода-вывода **<<** и **>>** могут комбинироваться в одном операторе в виде
  << строка [>> переменная] [>> переменная] ...
  Например:
  ```
  <<  "Введите число:"  >> REAL Number|_entered
  ```

I. 2. d. Операторы повторного присваивания и отправки данных

Если оператор начинается с символа '..' (без кавычек), то это ссылка на левую часть предыдущего оператора присваивания(или на зафиксированную левую часть присваивания). Соответственно, три точки подряд означают такую ссылку с последующей операцией обращения к полю или методу объекта, или обращения к функции с указанным операндом в качестве первого параметра.
Для такого повторного присваивания (или отправки данных):

- не ведется подсчет операторов (нет ограничения на количество таких повторных присваиваний);
- выражение фиксируется в качестве левой части для последующих "повторных" присваиваний:
  - если употреблено в левой части обычного оператора присваивания(a = b, a += b, a *= b, и т.п.);
    Например:
    ```
    Text = New_memo( THIS,  "TEXT",  "VH")
    ...Set_width(200) // эквивалентно Text.Set_width(200)
    ...Set_anchor_bottom(TRUE ) // то же что Text.Set_anchor_bottom(TRUE)
    ```
  - является левой частью обычного оператора отправки данных (a << b, a [] << b, в том числе в случае, когда оператор '<<' заменяет вызов метода Write);
    Например:
    ```
    Lines[] <<  "#!/bin/bash"
    ..[] <<  "echo run WinE"
    ..[] <<  "wine"
    ```
  - если в операторе в цепочке разыменования полей/методов использован псевдо-оператор '...' в позиции одинарной точки (например:
    ```
    A.Field1[indexA]...Do_something(params)
    ...Color = RED
    ...Foo(params)
    ```
    В приведенном примере в двух последних строчках символ '..' замещается компилятором выражением `A.Field1[indexA]`

I. 3. Выражения

Приоритет	Операция	Расшифровка	Группа
1	-	Взятие обратного знака	Арифметика (чисел), на выходе число
2	* / %	Умножение, деление, остаток от деления	
3	+ -	Сложение, вычитание	
4	IN !IN	Проверка на вхождение значения в диапазон или массив	Сравнение чисел, строк, классов, результат булевское
5	LIKE !LIKE	Сравнение строк без учета регистра букв и с учетом шаблонных символов '?' и '%' во втором операнде	
6	< <= > >= == != <>	Сравнения	
7	~	Поразрядное отрицание	Поразрядные операции с целыми, результат - целое
8	& ^	И, Исключающее или (xor) - поразрядные	
9	|	ИЛИ поразрядное	
10	!	Логическое НЕ	Логические операции (с булевскими, результат - булевское)
11	&&	Логическое И	
12	||	Логическое ИЛИ	

- *Операция конкатенации строк не имеет специального знака. Операнды записываются рядом.*
- При наличии поразрядных логических операций в модуле в список модификаторов модуля добавляется **BITWISE**.
- *смешение поразрядных и арифметических операций в одном выражении недопустимо.*
- *побитовые операции сдвигов реализуются с помощью встроенных функций **ShiftL|eft**, **ShiftR|ight**, **RotateL|eft**, **RotateR|ight** - отдельные знаки бинарных операций для этого отсутствуют.*
- *допускается использовать цепочки сравнений.*
  ***a < b <= c == 1 > d** эквивалентно:*
  a < b && b <= c && c == 1 && 1 > d

I. 3. a. Операции проверки наличия элемента

Операторы **IN** и **!IN** могут использоваться в выражениях:

- для проверки наличия скалярного значения в массиве (при этом массив может быть переменной или конструкцией вида
  [значение1, значение2, ..., значение3])
- для проверки наличия подстроки в строке
- так же в SQL-выражениях может использоваться **IN** (но для противоположной проверки используется синтаксис SQL: **NOT IN**)

В случае конструируемого массива, список значений должен содержать только константы. В этом случае, когда тип проверяемого значения - строка, не допускается конструировать массив из единственного элемента. Либо справа должна быть строка, а не массив.
Операция **IN** / **!IN** не применима к паре вещественное число - вещественный массив (так же как запрещено сравнение на равенство значений типа **REAL**).

I. 4. Прочие простые операторы

- Для возврата из функции до окончания ее кода используется оператор возврата **==>** .
  - Такой оператор может записываться отдельно либо следом за другим простым оператором в той же строке, образуя простой оператор с выходом из функции.
    Например:

```
CASE R > 0 ?  RESULT = R  ==> ;
```

- Для принудительного завершения и продолжения цикла **FOR** могут использоваться операторы **BREAK** и **CONTINUE**, для которых обязательно указывается переменная цикла, например:

```
FOR x  IN  [1 TO 100] :
     do_something
     CASE  condition ? BREAK x;
;
```

- В цикле **FOR, INFINITE** нельзя использовать операторы **CONTINUE** и **BREAK**. Завершить такой цикл можно только выходом из функции (оператор **==>**) или в результате срабатывания исключительной ситуации.
- Наличие метки в операторах **BREAK** / **CONTINUE** позволяет продолжить или остановить внешний цикл из тела вложенного напрямую (однако необходимо записать **BREAK** для всех вложенных циклов через запятую:
  BREAK k, BREAK j, CONTINUE i).

I. 5. Условный оператор CASE

- Выражение в операторе **CASE** должно иметь значение булевского, целочисленного, перечислимого ***или строкового*** типа.
- Выражения других типов (вещественных, структурных или объектных) не допускаются.
- Заголовок всегда завершается знаком '**?**'.
- В случае булевского типа условия, немедленно начинается блок кода, который выполняется при выполнении условия. И за ним может следовать блок **ELSE** (см. ниже).
- В остальных случаях, каждая ветвь начинается с массива или диапазона значений в квадратных скобках
  [массив значений]: код
- В качестве значений должны использоваться целочисленные, строчные или перечислимые ***константные*** значения (литеральные или именованные константы), тип значений которых соответствует типу выражения в заголовке.
- Значение в списке значений может быть одиночным или представлять диапазон значений в виде **X TO Y**, где X и Y - константы (только для целочисленного условия).
- Повторение значений (и пересечение диапазонов) не допускается.
- В случае перечислимого типа условия, должны быть перечислены все значения соответствующего перечислимого типа, и не может использоваться ветвь **ELSE**.
- Общая ветвь **ELSE** выполняется (при её наличии), если ни одна из констант не совпала, и ни одна из ветвей не была выполнена.
- Обычно **ELSE** (при его наличии) располагается начиная с новой строки. Но допускается размещать **ELSE** в той же строке, что и **CASE**, если весь оператор вместе с завершающей ';' умещается в одну строку (80 символов). В случае однострочного **CASE ... ELSE ... ;** оператора, ключевое слово **ELSE** должно отделяться от предыдущего оператора (ветви "then") как минимум одним пробелом или табуляцией;
- Символ ';' завершает весь оператор **CASE**.

Оператор **BREAK** **не используется** для завершения ветви.
По исполнении любой ветви всегда происходит выход из всего оператора **CASE**.

Классический пример - решение квадратного уравнения в области вещественных чисел:

```
STRUCTURE {roots_sq|uare_equation} :
     INT N|umber_solutions
     REAL X1|_solution
     REAL X2|_solution  .

FUNCTION Square_eq|uation(
          REAL A|_coefficient,
          REAL B|_coefficient,
          REAL c|_coefficient)  ==> {roots_sq}
         :
    CASE A.Near(0) ?  ==>   ;
    REAL  d|escriminant = B * B - 4 * A * C
    CASE d.Sign ? [0]: RESULT.N = 1
                       RESULT.X1 = -B / (2 * A)
                       RESULT.X2 =  RESULT.X1
                 [1]: d = d.Sqrt
                      RESULT.X1 = (-B - d) / (2 * A)
                      RESULT.X2 = (-B + d) / (2 * A)
                      RESULT.N = 2  ; .
```

Особый вариант оператора **CASE ?** - без условия вообще, может использоваться для последовательного перебора вариантов, вычисляющихся последовательно, до первого выражения в квадратных скобках, получившего значение **TRUE**. После чего либо выполняется соответствующая ветвь, либо, если истинное условие не обнаружено, то срабатывает блок **ELSE** (при его наличии).

Синтаксически такой вариант оператора **CASE** отличается так же тем, что после каждого выражения в квадратных скобках записывается знак '**?**' (а не двоеточие, как в случаях выше).

```
FUNCTION  Square_eq2|uation(
REAL  A|_coefficient,
REAL  B|_coefficient,
REAL  c|_coefficient)  ==>   {roots_sq}
    :
    ----------------------  'sequentional version'
    REAL  d|escriminant = B * B - 4 * A * C
    CASE ?
    [A.Near(0)]?  ==>
    [d.Near(0)]? RESULT.N = 1
                 RESULT.X1 = -B / (2 * A)
                 RESULT.X2 = RESULT.X1
    [d > 0] ? d = d.Sqrt
              RESULT.X1 = (-B - d) / (2 * A)
              RESULT.X2 = (-B + d) / (2 * A)
              RESULT.N = 2  ;  .
```

Так же, имеется особый вариант оператора **CASE ?**, предназначенный для использования в обобщенных функциях. Подробнее см. в соответствующем разделе.

I. 6. Операторы цикла FOR

Оператор цикла **FOR** с перечислением элементов массива или диапазона значений - это единственный вариант контролируемого цикла:

```
    FOR  переменная IN  массив[диапазон] : тело  ;
```
или
```
    FOR  переменная IN [диапазон]: тело  ;
```
или
```
    FOR  переменная  ENUM|ERATE  метод: тело  ;
```

- Переменная такого цикла всегда имеет тип, совпадающий с типом элементов перечисляемого массива.
- В случае диапазона значений без массива, тип переменной - целое число;
- Такая переменная:
  - Не требует декларации.
  - Не требует длинного синонима.
  - Не может использоваться вне цикла, кроме как в другом цикле **FOR** .
  - Используется как метка операторов **BREAK имя** и **CONTINUE имя**.

- Переменная цикла **FOR** с диапазоном значений (не на основе массива) не может быть изменена в теле цикла.
- В случае массива с пустым диапазоном в цикле перечисляются все элементы с индекса 0 до индекса ***** (индекс последнего элемента в массиве).
- Для изменения границ перечисления и направления перечисления следует явно указать диапазон либо в индексной части массива, либо использовать конструируемый с помощью диапазона массив целых в форме **[X TO Y]** или **[Y DOWNTO X]**, где **X** и **Y** - целочисленные выражения.
- В случае цикла по массиву, к переменной цикла применима псевдо-функция **INDEX**, возвращающая целочисленный индекс элемента в массиве.
- Границы перечисления индексов вычисляются до начала цикла, и изменение размера динамического массива в процессе работы цикла не влияют на порядок просмотра индексов. Это может приводить к выходу индекса за границы массива, в результате чего переменная цикла будет получать значение **NONE**.
- Для варианта **FOR x ENUM y : ... ;** y должен быть вызовом любого метода, возвращающего булевское значение и не имеющего параметров. Этот метод должен вернуть **TRUE** для прекращения цикла, и вызывается перед каждой итерацией. Однако, переменная цикла при этом всегда отсчитывает значения от 0 до максимального значения, и если эта максимальная величина превышена, цикл прекращается. Предельное значение по умолчанию равно 1_000_000. Но может быть изменено явно:
  FOR x ENUM(MAX 1000) method : ... ;

Для организации неконтролируемого бесконечного цикла используется блочный оператор

```
FOR, INFIINITE  : тело цикла  ;
```

- Цикл **FOR, INFINITE** может быть завершён только оператором выхода из функции **==>** .
- Операторы **BREAK** и **CONTINUE** не используются.
- Класс, в котором встречается цикл **FOR, INFINITE**, должен быть отмечен маркером **INFINITIVE**.

Переменная цикла, если она целочисленная, может (как и любая целочисленная переменная) получать атрибут **INDEXING тип** или **INDEXING (список, типов)**. См. подробнее в главе, посвященной контролю типов индексируемых массивов.

В случае завершения блока оператора **FOR** оператором выхода из блока (**BREAK**), финальная последовательность операторов (после последнего оператора **CONTINUE** в этом цикле) может быть отделена специальным оператором **DONE**. Например:

```
FOR a IN A[] :
    CASE !a.satisfying_conditions ? CONTINUE a ;
    DONE // искомый элемент найден
         a.do_something
         do_something_more
         BREAK a ;
```

Оператор **DONE** уменьшает уровень вложенности блочных операторов (как если бы блок оператора **FOR** уже завершился).

I. 6. a. О возможности присваивания значений переменным цикла:

- Если переменная цикла изменяется в диапазоне значения (массив не перечисляется), присваивание другого значения переменной запрещено в цикле;
- в случае, когда переменная цикла пробегает все или часть элементов массива (например, строкового, целочисленного и других типов, кроме структур) - присваивание нового значения изменяет эту переменную, но не изменяет соответствующий элемент массива. Например:

```
FOR s IN Strings_array[]:
    s = s.Trim
    ... work with trimmed String_array[s.INDEX] ...
;
```

- В случае массива структур, переменная цикла является макросом, обеспечивающим доступ к текущему элементу массива. Соответственно, изменение полей, или присваивание нового значения этой переменной меняет соответствующие (или все) поля текущего элемента массива;
- В любом случае, изменение переменной цикла не влияет на обращение к псевдо-функции **переменная.INDEX** - она всегда возвращает целочисленный индекс текущего (пробегаемого в цикле) элемента в массива.

I. 7. Блок операторов PUSH

Предназначен для гарантированного сохранения некоторой переменной или поля с одновременным присваиванием ему нового значения на время выполнения блока, с последующим гарантированным восстановлением значения этой переменной или поля по окончании исполнения блока. Либо для вызова специального PUSH-метода, с обязательным вызовом соответствующего ему POP-метода при выходе из блока PUSH.

Восстанавливающие действия гарантируются в том числе:

- При выполнении принудительного выхода из блока из-за срабатывания операторов **BREAK**, **CONTINUE**
- При выходе из функции оператором возврата **==>**
- При возникновении исключительной ситуации (например, по прерыванию длительной операции или операции, вызвавшей слишком глубокую рекурсию вызовов функций)

Синтаксис:

```
PUSH   переменная = выражение : тело  ;
```

или

```
PUSH  переменная : тело  ;
```

или

```
PUSH  метод(параметры) : тело  ;
```

- На входе в такой блок значение переменной сохраняется (например, в локальном стеке, или в локальном динамическом массиве).
- После чего ей присваивается новое значение(если указано).
- При достижении завершающей скобки блока (любым способом, даже аварийным), восстанавливается прежнее значение переменной.
- В качестве переменной может быть скалярная переменная любого простого или объектного типа, или элемент массива.

В операторе **PUSH** может быть вызван специальный метод, имеющий модификатор **POP(метод2)**. По окончании такого блока **PUSH**, гарантированно вызывается указанный в скобках модификатора POP закрывающий метод2 (он не должен иметь параметров, и не может быть вызван другим способом). Метод с модификатором **POP** (открывающий) так же не может вызываться напрямую, кроме использования его в операторе **PUSH**.
Например:

```
PUSH db.Transaction :
      // some database operations
      db.Commit
;
```

I. 8. Блок операторов DEBUG

Предназначен для временной инъекции кода, который требуется только для целей отладки. Внутри этого блока не ведётся подсчёт и контроль уровней вложенности блоков. Компилятор всегда выдаёт предупреждение о наличии таких блоков с тем, чтобы программист не забыл удалить их из кода (или закомментировать) по окончании отладки.
Внутри отладочных блоков могут быть декларированы локальные переменные, но использовать их можно только опять же внутри отладочных блоков (хотя и не обязательно только в том блоке, в котором они декларируются).

```
DEBUG :
    тело ;
```

Если для правильной компиляции блоков **DEBUG** требуются дополнительные классы, отсутствующие в основном списке импорта, рекомендуется использовать оператор **IMPORT** со специальным модификатором **DEBUG**.

Для оператора **DEBUG**:

- не учитываются уровни вложенности кода: компилятор не выдает предупреждений или ошибок по поводу чрезмерно большой вложенности самого оператора **DEBUG** или вложенных в него блочных операторов;
- разрешается опускать двоеточие:
  DEBUG count_passes += 1 ;
- если первый оператор в блоке **DEBUG** - это вывод в консоль строки, и первое выводимое значение - строчный литерал, то можно опустить символ **<<** :
  DEBUG: "test!"#NL ;
  DEBUG "another test" ;

I. 9. Блок операторов SILENT

Предназначен для предотвращения выдачи компилятором предупреждений в некотором участке кода. Например, предупреждений о наличии отладочных блоков DEBUG:

```
SILENT:
     DEBUG :
         тело ;
;
```

I. 10. Блок операторов LIMIT

Предназначен для ограничения времени выполнения некоторого кода:

```
LIMIT TIME(n) MSEC:
       тело блока
;
```

Варианты заголовка:

- **LIMIT TIME(n) units: ... ;**
  ограничение по времени выполнения. Здесь units - это идентификатор, задающий единицы измерения времени в скобках (n - дробное числовое выражение):
  - **MSEC || MILLISECONDS** - миллисекунды,
  - **SEC|ONDS** - секунды,
  - **MIN|UTES** - минуты,
  - **HOURS** - часы,
  - **DAYS** - дни;
- **LIMIT FUN|CTIONS(n) : ... ;**
  ограничение по числу вызванных функций;
- **LIMIT LOOPS(n) : ... ;**
  ограничение по числу итераций циклов;

Проверки на возможное превышение заданных ограничений производятся обычно 1 раз на каждой из 65535 итераций циклов, выполненных в коде AL-IV. Поэтому финальное превышение времени или другого заданного параметра может превышать заданное на эти самые 64К итераций циклов, выполнившихся между проверками.
В случае превышения заданного ограничения выполнение кода в пределах блока прекращается, и выполнение продолжается со следующего за блоком оператора. При этом, если выполнялись какие-то блоки PUSH, то они гарантированного завершаются соответствующей им операцией POP (например, восстановлением значения переменной).

I. 11. Оператор LIKE

(Не путать с операцией LIKE в выражении!)

Предназначен для вставки ранее написанного кода в новой позиции. Вставляемый код должен быть расположен между двумя блочными комментариями (на одном уровне вложенности). Метка первого блочного комментария (в совокупности с именем функции) используется для указания того, какой именно блок кода следует повторить.
Имеются правила:

- Используемый блок должен быть помечен модификатором **REUSED**.
- Если вставляется блок из другой функции, после **LIKE** записывается имя этой функции.
- Далее следует мульти-точие (множество точек, количество которых не ограничивается, но не менее трёх), после чего записывается метка вставляемого блока кода (в апострофах - как и в коде).
- Возможна вставка **только кода из своего класса**.
- Вставляемый код должен быть расположен выше по тексту.
- Сам вставляемый код не должен содержать операторов **LIKE**.
- Вставка происходит "как есть", контроль синтаксиса и семантики происходит только при компиляции оператора **LIKE**, поэтому следует не допускать дублирования деклараций, имён переменных, переменных циклов (во избежание ошибок компиляции).
- Количество повторных использований в операторах **LIKE** одного используемого блока кода никак не ограничивается.
- На время вставки кода счётчик вложенности блоков сбрасывается в ноль (т.е. проблем со слишком большой вложенностью блоков **CASE**/**FOR** не будет, даже если суммарный уровень вложенности превысит допустимые три уровня).

Пример:

```
...
---------------------------  'find a dot'
pos = s.Find(".")
CASE pos < 0 ? pos = s.Len ;
------------------------------  'end of find'
...
...
LIKE ............. 'find a dot'  // place in the same function
or
LIKE Method1 ..... 'find a dot'  // place in another function
```

Дополнительно, при использовании оператора **LIKE**, можно выполнить замены некоторых фрагментов кода:

```
LIKE [method].......'label', BUT (образец1 ==> замена1) [*N1] [, (образец2 ==> замена2) [*N2] ]
```

Списки замен заключаются в скобки и перечисляются через запятую. Замена должна иметь множитель в форме *** число** после скобки, если таких замен несколько (число замен должно соответствовать количеству указанных образцов для замены, множитель можно опустить, только если такой образец единственный).
Все замены выполняются непосредственно перед компиляцией оператора LIKE, на участке кода, который подставляется вместо LIKE.
И образец, и строка, на которую он заменяется, может быть заключен (или не заключен) в апострофы. Только строка справа от знака ==> может быть пустой, образец должен содержать хотя бы один символ.

И, пожалуйста, помните:

- Лучше использовать **LIKE .... 'имя'**, чем просто копировать код.
- **НО** Лучше выносить общий код в отдельную функцию, если это возможно (нежели использовать **LIKE**).

Поэтому не злоупотребляйте использованием **LIKE** без необходимости.

I. 12. Оператор REVERT

Предназначен для вставки ранее написанного кода в новой позиции с переворачиванием направлений присваивания. Переворачиваемый код должен быть расположен между двумя блочными комментариями (на одном уровне вложенности). Метка первого блочного комментария (в совокупности с именем функции) используется для указания того, какой именно блок кода следует повторить.
Имеются правила:

- Используемый блок должен быть помечен модификатором **REUSED**.
- Если вставляется блок из другой функции, после **REVERT** записывается имя этой функции.
- Далее следует мульти-точие (множество точек, количество которых не ограничивается- но не менее трёх), после чего записывается метка вставляемого блока кода.
- Возможна вставка только кода из своего класса.
- Вставляемый код должен быть расположен выше по тексту.
- Сам вставляемый код может содержать только операторы присваивания, причём левая и правая сторона должны допускать переворачивание (хотя бы переходом к использованию сеттеров и геттеров).
- Если вставляется код из другого метода, он не должен использовать обращений к локальным переменным (и параметрам).
- Количество операторов **REVERT** для одного используемого блока кода никак не ограничивается.

Пример:

```
----------------  'save form coordinates '
config.Set_i("Left", Left)
config.Set_i("Top", Top)
config.Set_i("Width", Width)
config.Set_i("Left", Height)
------------------------------  'end of save'
...
...
REVERT .............  'save form coordinates'
```

II. Переменные, константы и типы данных

II. 1. Простые типы данных

II. 1. a. Встроенные простые типы данных

Простые (встроенные) типы данных (которые не могут быть переопределены):

- **BOOL**- булевский тип, значения **TRUE** и **FALSE**.
- **BYTE** - байтовый тип, значения от 0 до 255 (использование байтового типа в классе требует добавления модификатора **BYTES** в заголовке класса).
- **INT|EGER** - целочисленный тип (обычной точности).
- **REAL** - вещественный тип.
- **STR|ING** -строковый тип.

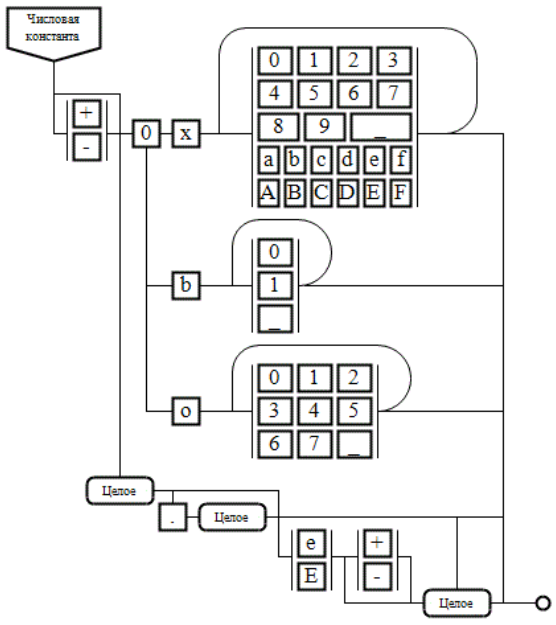
Для типов данных **разрядность не задаётся**.

- Для переменных базовых типов всегда используется разрядность, которую определяет разработчик компилятора (или разработчик задает ее на уровне проекта, если компилятор допускает такую возможность).

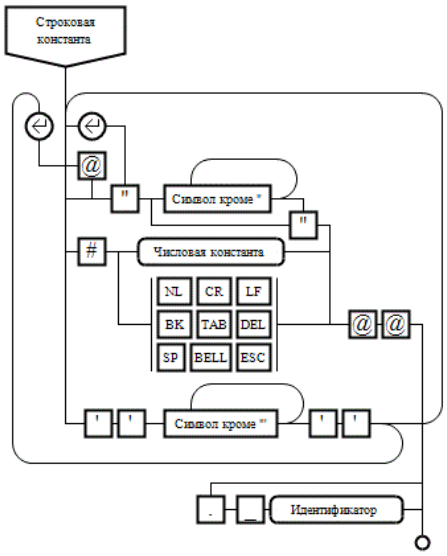
При использовании байтового типа в классе, он должен быть помечен в заголовке модификатором **BYTES**, и не может маркироваться как **SAFE**.

II. 1. b. Запись констант базовых типов данных

- Целые константы в десятичной системе: **[-|+]<0...9_>...**
  1_345 = 1345
  (***подчеркивания игнорируются*** и могут использоваться для разделения групп разрядов).
- В двоичной системе:
  0b_0100_1010_0011_1001
  (Регистр буквы, определяющей основание системы счисления, не учитывается: B=b, X=x, кроме случая восьмеричной системы)
- В шестнадцатеричной системе(x=X):
  0xF2EA_d44a
- В восьмеричной системе(только малая буква 'o'):
  0o_7_342_001
- Вещественные константы: **{0...9}.{0...9}[e[+|-]{0...9}]**
  (e=E). Например:
  -2.73e3
  11_E-04



- Строковые константы: **"{символ кроме "}"**
  где символ - любой печатный знак или пробел,
  двойные кавычки в строке не записываются, для их записи необходимо использовать символьную константу **#QU**). Например:
  #QU ***"Этот текст заключен в двойные кавычки"*** **#QU**
- Имеется несколько именованных символьных констант, которые записываются в форме #ИМЯ:
  - **#NL** - новая строка (обычно это два символа **#CR #LF**, с кодами 13 и 10, соответственно);
  - **#CR** - возврат каретки (код 13);
  - **#LF** - перевод каретки (код 10);
  - **#BK** - забой одного символа (код 8);
  - **#TAB** - табуляция (код 9);
  - **#SP** - пробел (код 32);
  - **#BELL** - гудок (код 7);
  - **#QU** - двойная кавычка (код 34);
  - **#ESC** - специальный символ ESCAPE (код 27);
- Так же, всегда символ может быть закодирован в форме #целое, например, **#32** или **#0x20** эквивалентны одиночному пробелу (**" "**, или **#SP**);
- Для конкатенации символьных и строчных констант и переменных какой-либо символ операции не используется.
  Например:
  "Этот текст завершается возвратом каретки" **#CR**
- Если строка начинается символом двойной кавычки, то эта строка считается продолжением предыдущей. Т.е. для записи длинных строчных констант, не вмещающихся в одну строку, достаточно разбить строку в любом месте и записать продолжение в следующей строке. Пример:
  "Это очень длинная строка. Настолько длинная, что "
  "она не может быть записана в одной строке кода."
- Если перед началом строковой константы записан символ @, то для всех последующих конкатенаций с последующими строковыми константами и переменными, при переносе выражения на следующую строку, все символы переноса строки **#NL** становятся частью константы. Например:
  @"Эта константа состоит из трёх строк: первая,"
  "вторая,"
  "и третья"
- Символ @@ перед строчной константой отменяет действие символа @ - до конца конкатенации или до следующего символа @.
- В качестве продолжения символьной константы в отдельной строке кода может использоваться литерал **''**текст**''** (ограниченный удвоенным символом апострофа). Такой литерал может содержать любые символы, кроме удвоенного апострофа (в том числе двойные кавычки и одинарные апострофы) .



II. 1. c. Перечисления

Перечислимый тип данных определяет **набор именованных констант**. Сам набор так же именуется для того, чтобы на него можно было ссылаться.

- В декларации перечисления элементы перечисления записываются через запятую. Форма декларации:

```
ENUM {names|_of_enumeration} : values .
```

(точка завершает декларацию).

- Имя типа перечисления заключается в фигурные скобки и начинается со строчной буквы.
- Имена элементов перечисления заключаются в апострофы.
- Имён для элемента может быть несколько, краткое имя и остаток полного имени разделяются символом '|'.
- Имена должны быть уникальны в пределах класса (среди всех перечислений класса).

```
ENUM {color|s_of_traffic_light} :
     'R|ED_COLOR',
     'Y|ELLOW_COLOR',
     'G|REEN_COLOR' .
```

Операции над значениями перечислимых типов данных:

- Переменной с типом, созданным оператором **ENUMERATION**, нельзя присвоить числовое значение или значение из чужого перечисления.
- NONE-значением перечислимого типа считается первое значение (соответствует физическому нулю во внутреннем представлении).
- Значение переменной (или константы, или выражения) такого типа может быть преобразовано в целочисленное: встроенная функция **Int**.
- Обратное преобразование числового значения в тип перечисления не предусмотрено (кроме как с помощью собственной функции, в которой выполняется такое преобразование, например, оператором **CASE**).
- Встроенная функция **Name** возвращает имя элемента по его значению. В качестве имени возвращается первое имя константы (с апострофами).

Перечисление может использоваться как диапазон индексов фиксированного массива при его декларации:

```
STR LName|s_of_lights[{color}]
```

В этом случае в качестве индексов этого массива могут использоваться переменные и константы соответствующего перечисления:

```
LName['R'] = "Red color"
```

Элементы перечисления считаются неупорядоченными, и не могут сравниваться операциями <, <=, >, >= (применимы сравнения == и !=). Из элементов перечисления может быть сформирован константный массив, и для пары элемент - массив элементов допустима операция **IN** (и **!IN**):

```
  CASE A IN [ 'R', 'Y'] ? ... ;
```

В случае конфликта имён перечислимых элементов, они обязаны квалифицироваться именем перечисления в форме:
{enumeration_name}.'ENUMERATION_ITEM'.
В случае совпадения имён самих перечислений, в квалификацию должен включаться класс, из которого это перечисление родом:
{Class_name}.{enumeration_name}.'ENUMERATION_ITEM'.

При использовании в операторе **CASE** выражения типа перечисления:

- запрещается использовать ветвь **ELSE**,
- должны быть использованы (перечислены) все константы,
- в одной ветви может использоваться более одной константы:

```
CASE Greek ?
     ['ALPHA', 'BETHA']: ...
     ['GAMMA', 'DELTA']: ...
     ['EPSILON']: ...
     ['DZETHA', 'ETHA', 'TETHA', 'IOTHA', 'KAPPA']: ...
     ... ;
```

II. 2. Переменные, массивы

II. 2. а. Декларация переменных

Декларация переменной начинается с указания ее типа данных в **{фигурных}** скобках (или одного из зарезервированных имён типов **BOOL**, **BYTE**, **INT**, **REAL**, **STR**).

- При объявлении глобальной или локальной переменной её полное имя должно быть длиной не менее 8 символов.
  - *ИСКЛЮЧЕНИЯ:*
    1. *Переменные циклов FOR могут иметь только краткий вариант имени;*
    2. *Целочисленные переменные с явно заданным атрибутом INDEXING;*
- Имя может быть коротким, но тогда вслед за короткой частью после разделителя '|' размещается остальная часть имени.

- Для массивов далее задаётся размерность в [квадратных] скобках.
    - для динамических массивов - пустые квадратные скобки,
    - для фиксированных массивов с целочисленными индексами указывается константа, задающая размер фиксированного массива (индекс последнего элемента + 1),
    - для фиксированных массивов с индексами из перечисления указывается имя перечислимого типа, в фигурных скобках:
      `BOOL My_arr|ay_of_flags[{traffic}]`
- Далее (только для целочисленных переменных и массивов) может быть указан атрибут **INDEXING тип** или **INDEXING (список,типов)**. Присваивание такого атрибута переменной означает, что указанные типы будут сверяться с типами элементов массивов, при индексировании их такой переменной. Несовпадение типов будет вызывать ошибку компиляции. Аналогичная проверка производится при прямом присваивании значения переменной (если справа простое выражение вида X+Y или X-Y - так же проверяются соответствующие атрибуты X и Y). А так же при передаче переменной (или простого выражения) в качестве параметра, имеющего такой атрибут.
- Затем, при необходимости, **МОДИФИКАТОРЫ** переменной/поля (**READ**, **INIT**, **MAXLEN[n]**).
- Далее может следовать присваивание начального значения (в качестве начального значения допускается использовать константы).

Переменной всегда присваивается начальное значение. Если такое значение не присвоено явно, присваивается значение по умолчанию.

- Для объекта - это **NONE**-значение соответствующего класса (см. "Классы").
- Для строк - пустая строка.
- Для динамических массивов - пустой массив.
- Для перечислений, коллекций, чисел - арифметический ноль (для перечислений это всегда первый элемент перечисления).
- Структура получает в качестве значения структуру с полями, имеющими ноль-значение.

Правила именования переменных:

- Краткое имя - это идентификатор от 1 символа,
- Длинное имя - идентификатор от 8 до 80 символов,
- Далее в коде употребляется краткое имя наравне с длинным.

Регистр букв имён переменных:

- Имена **Публичных** полей классов и **Параметров** функций начинаются с прописной буквы (в том числе с атрибутом **READ** - только для чтения).
- **КОНСТАНТЫ** (обычно) содержит только прописные буквы.
- Со строчной буквы начинается имя **приватных** полей класса и **локальных** переменных.

II. 2. b. Модификаторы полей

Если переменная требует модификаторов, они записываются в декларации вслед за именами переменной и её размерностями.

- **READ|ONLY**- используется для переменных уровня модуля, полей классов(а так же параметров функций).
    - Для поля класса разрешает доступ только для чтения, кроме самого модуля и из методов класса-наследника.
    - Поле должно быть публичным (именоваться с **П**рописной буквы).
- **INIT|IALIZE -** для поля класса, требует инициализации поля при конструировании объекта (см. "Классы").
- **DEPRECATED('текст')** - данное поле "осуждается", но ещё поддерживается. В тексте указывается, что предлагается в качестве замены. Рекомендуется переходить к замене по мере возможности, в следующей версии поле может уже не поддерживаться.
- **ABANDONED('текст')** - поле "отменено". В тексте указывается, что есть взамен (если есть). Использование такого поля в коде уже невозможно, и вызывает ошибку компиляции.

Кроме явно указываемых выше модификаторов существуют так же неявная маркировка переменных и параметров функции:

- Параметр, объявленный в заголовке функции, но не использующийся в теле функции, должен содержать в имени строку "**dummy**" (в любом регистре).

- Переменная, объявленная в теле функции (и даже если ей присвоили значение), но не использованная в выражениях, должна иметь строку "**dummy**" (в любом регистре) в своём имени.
- Объектная локальная переменная, которой присвоен новый объект, который не добавляется в массив сильных ссылок и для которого не указывается владелец, должна иметь в своём имени строку "**temp**" (в любом регистре). Например,
  load|er_Temp = {Text_file}(Path = path)

II. 2. c. Декларация именованных констант

Константы декларируются только на уровне класса, и только встроенных простых типов (**BYTE**, **INT**, **BOOL**, **REAL**, **STR**).

- Декларация одиночной константы в одной строке:
  CONST имя_типа {Имя_блока_констант} : ИМЯ_КОНСТАНТЫ = выражение .
- Декларация нескольких однотипных констант:
```
CONST имя типа :
      ИМЯ_КОНСТАНТЫ_1 = выражение
     ИМЯ_КОНСТАНТЫ_2 = выражение
     ...
      ИМЯ_КОНСТАНТЫ_N = выражение .
```

Рекомендуется использовать капитализированные имена констант (все буквы в верхнем регистре).
Блок констант может иметь имя (задается в фигурных скобках после типа констант). Имя блока констант может использоваться при задании ограничений на значения параметров функций.

II. 2. d. Декларация массивов

После имен переменной, являющейся массивом, в квадратных скобках записывается размерность.

- Массивы могут быть только одномерными (при необходимости, может использоваться механизм методов-индексаторов для создания многомерных массивов и других индексируемых наборов данных).
- Массив может быть фиксированным или динамическим.
- В квадратных скобках динамического массива ничего не пишется при его декларации:
  STR Lines|_of_text[]
- Для фиксированного массива при его декларации в квадратных скобках записывается либо целочисленное константное выражение, либо имя типа перечисления. Для целочисленного, значение выражения - размер массива (на 1 больше последнего индекса по данному измерению)..
  Пример(массив 16 элементов):
  REAL A|rray_of_values[16]
- При использовании массива даже как цельного объекта (в отличие от его элементов), всегда указываются квадратные скобки.
  Например: **A[]**

Начальное значение для массивов обеспечивается обязательно, как и для всех остальных переменных:

- Массив переменной размерности первоначально пуст.
- Фиксированный массив объектов класса первоначально заполнен **NONE**-объектами соответствующего класса.
- Фиксированный массив строк заполнен пустыми строками.
- Фиксированный массив чисел, перечислений, коллекций заполнен арифметическими нулями.
- Фиксированный массив структур заполнен NONE-экземплярами структур соответствующего типа (целочисленные поля равны нулю, строчные - пустым строкам, и т.д. - рекурсивно по полям структурных типов).

II. 2. e. Конструктор массива

Конструктор массива - это перечень элементов массива в квадратных скобках в виде:
[значение, значение, ... значение]
В качестве значений попускаются только константы.

Конструктор массива может использоваться как второй операнд операции проверки на вхождение **A IN B** (и противоположной **A !IN B**).

II. 2. f. Использование массивов

Основные операции с массивами:

- Чтение и запись элемента массива. В левой части оператора присваивания или в качестве операнда выражения указывается имя массива и в квадратных скобках записывается выражение, вычисляющее целочисленный индекс элемента по этой размерности.
  Например: `A[i] = B`
- Добавление элемента в динамический массив оператором `<<`, пример:
  `X[] << Value`
- Вызов одной из встроенных функций над динамическими массивами:
  - Добавление элемента в конец массива:
    `X[] << s`
  - Вставка элемента в позиции индекса i:
    `X[].Insert(i, Value)`
  - Удаление элемента в позиции i:
    `X[].Delete(i)`
  - Удаление нескольких (Count) элементов, начиная с позиции i:
    `X[].Delete_range(i, Count)`
  - Удаление всех элементов, равных R:
    `X[].Remove(R)`
  - Поиск индекса первого элемента, равного R:
    `X[].Find(v)`
  - Проверка наличия элемента R в массиве X[]:
    `present = R IN X[]`
  - Аналогично предыдущему (если элемента R нет в массиве, возвращается -1):
    `CASE X[].Find(R) < 0 ? not_found ;`
  - Получение количества элементов в массиве:
    `count = X[].Count`
  - Очистка динамического массива:
    `X[].Clear`
  - Установка нового размера массива (дополнение NONE-объектами или обрезка до указанного размера, если указан размер, меньший текущего):
    `X[].Allocate(new_size)`
- При обращении за пределами массива возвращается `NONE`-объект соответствующего типа.
- Присваивание значения за пределами массива игнорируется.

Передача массива как параметра в функцию:

- Указываются квадратные скобки для массива.
- На место динамического массива может быть передан только динамический массив.
- На место фиксированного массива с целочисленными индексами может быть передан фиксированный массив или часть динамического или фиксированного массива, с указанием диапазона индексов в форме:
  `A[first TO last]`
- Для указания того, что формальный параметр функции является массивом фиксированного размера с целочисленными индексами, в квадратных скобках записывается символ '*'.
- В качестве параметра - фиксированного массива с размером, заданным перечислением, может быть передан только фиксированный массив с размером, заданным тем же перечислением.

Не допускается:

- Присваивание целого массива как единого значения.
- Возврат массива в качестве результата функции (Следует декларировать параметр-массив требуемого типа, и возвращать результаты через него).

Особое индексирование элементов массивов (и символов строк) значением '*':

- Символ '*' в квадратных скобках после имени массива в позиции, в которой ожидается операнд выражения (например, переменная) означает индекс последнего элемента в массиве (равный Count-1);
- Так, A[*] - последний элемент массива, а A[*-1] означает обращение к предпоследнему элементу (в позиции Count-2).

II. 3. Функции

II. 3. a. Заголовок функции

FUNCTION имена (параметры) ==> тип_результата , модификаторы :
операторы.

FUNCTION имена (параметры)==> тип_результата, модификаторы:

- Статическую функцию начинает функцию слово **FUNCTION** или **FUN**.
- Метод класса начинает ключевое слово **METHOD**.
- Если метод переопределенный, используется ключевое слово **OVERRIDE**.
- Для конструктора и деструктора класса используются ключевые слова **CONSTRUCT** и **DESTRUCT**, соответственно. И в этом случае уже нет имен функций, параметров, результатов, модификаторов - просто двоеточие, которое завершает заголовок.
- Для переопределения стандартных операций +, -, *, / используется особая форма функции: начинается ключевым словом **OPERATOR**.
- Функции тестирования начинаются с ключевого слова **TEST**.

FUNCTION имена (параметры)==> тип_результата, модификаторы:

- Функции могут иметь несколько имен (как минимум одно длинное имя не менее 8 символов)..
- В случае двойного именования, сначала записывается короткое имя, далее, после разделителя '|' - остаток имени. При определении замещения метода класса-предшественника указывается только первое имя замещаемого метода.
- Имя публичной функции начинается с прописной буквы. Если требуется сделать публичной функцию или метод, начинающийся со строчной буквы, может использоваться модификатор **PUBLIC**.
- Имя функции, доступной только в самом классе (и его наследниках) - начинается только со строчной буквы.
- **OPERATOR** не имеет имен, и его параметры задаются в форме **ТИП ОПЕРАЦИЯ ТИП** (или для унарного оператора "-": **ОПЕРАЦИЯ ТИП**), при этом являясь его уникальным именем (допускается переопределение несколькими операторами одной и той же операции - с разными типами данных).
- В классе может быть объявлен один метод без имени (вместо имени в заголовке указывается точка), параметры для такого метода записываются в квадратных скобках. Если для такого метода объявлен сеттер, то такой метод может использоваться в левой части оператора присваивания.

FUNCTION имена (параметры) ==> тип_результата, модификаторы:

- Параметры в круглых скобках в заголовке указываются при их наличии..
- В случае их отсутствия параметры не записываются, в том числе скобки опускаются.
- Число параметров не ограничено, но если количество явных параметров(не считая **THIS**) превышает три хотя бы у одной функции, то при вызове такой функции в коде, все параметры, начиная с четвертого (или ранее), должны передаваться в форме присваивания
  Имя_параметра= выражение

Параметры перечисляются через запятую.

- Для параметра сначала записывается тип параметра, затем имена параметра, далее размерность (необязательно).
- Параметр может иметь более одного имени(краткое имя отделяется от остальной части имени символом '|', как обычно).
- Полное имя параметра должно быть не менее 8 символов.
- Имена параметров публичных функций должны начинаться с прописной буквы.

Для параметра-массива после имени указывается размерность в квадратных скобках.

- Для динамического массива, в квадратных скобках ничего не записывается.
- Для фиксированных массивов с целочисленными индексами записывается символ '*'.
- Для фиксированных массив с перечислимыми индексами указывается имя типа перечисления (в фигурных скобках), например, **BOOL A|rray_of_color[{color}]**

Целочисленные параметры могут иметь атрибут **INDEXING тип** или **INDEXING (список, типов)**. Если такая переменная используется для индексации массивов, то тип элементов массива сверяется с заданными типами. При вызове функции, если на место параметра передается переменная (или простое выражение наподобие X+Y, один из аргументов которого - такая переменная), то их атрибуты сравниваются.

Параметры не могут иметь модификаторы.
Параметры, которые не используются, должны иметь слово dummy (в любом регистре) в качестве части своего имени.

Скалярные параметры всегда принимаются по значению, и не могут изменяться в теле функции.

Параметры могут иметь ограничения на значения или участвовать в ограничениях, которые задают возможные списки и диапазоны константных значение параметров - безусловно, либо в зависимости от значений других констант. (См. модификаторы **RESTRICT**, **IF**, **THEN** - для функции).

Для операторов, параметрами являются типы данных. Как минимум, один из типов данных должен быть классом или структурой. В теле оператора (представляющим собой выражение) для ссылки на параметры используются буквы A и B, хотя в заголовке оператора эти буквы не пишутся. Буква A соответствует первому операнду, B - второму. В случае переопределения унарной операции "-" для ссылки на единственный операнд используется имя A.

В случае обобщенных функций в позиции типа параметра размещается перечисление из нескольких типов, заключенное в фигурные скобки, например: **{INT, REAL, STR, {color}}**. Смотрите подробнее в дополнительном разделе.

FUNCTION имена (параметры) ==> тип_результата, модификаторы:

Если тип результата не указывается (вместе со стрелочкой '==>'), то функция не возвращает результат.

- Для присваивания значения результату функции, используется присваивание значения псевдо-переменной **RESULT**.
- Как и все прочие локальные переменные, псевдо-переменная **RESULT** изначально инициализируется **NONE**-значением:
  - **NONE** - для объектов,
  - **0** - для чисел,
  - **""** - для строк,
  - первый элемент - для перечислений,
  - **FALSE** - для типа **BOOL**.
- Функция не может возвращать массив в качестве результата, только скаляр.
- Специальный символ возврата **==>** используется и как операция выхода из функции в теле функции (он может пристыковываться к концу любого простого оператора).

Например, подсчет символов точки в строке:

```
FUNCTION Ndots|_count (STR S) ==> INT :
    FOR i IN [0 TO S.Len-1] :
        CASE S[i] == "." ? RESULT+=1 ;
    ; .
```

В случае обобщенных функций тип результата так же может быть списком типов, заключенных в фигурные скобки: **{INT, BOOL}**. См. подробнее в дополнительном разделе.

Допускается после типа возвращаемого результата запись вида:
RESULT|дополение_к_описанию_возвращаемого_значения.

FUNCTION имена (параметры)==> тип_результата, модификаторы:

Модификаторы функции перечисляются через запятую. Возможны модификаторы:

- **RECURSIVE -для рекурсивной функции.** Если функция уже использует некоторую рекурсивную функцию, и сама не вызывает себя непосредственно, то этот модификатор необязателен.
  - Если в процессе выполнения обнаруживается превышение некоторого порога рекурсивных вызовов (обычно - 128, но может быть меньше или больше), то происходит возврат к самому верхнему рекурсивному уровню, этот вызов функции игнорируется, а в качестве результата, если он требуется, возвращается **NONE**-значение.
- **NEW** - функция **возвращает новый объект** (экземпляр класса).
  - Как минимум один оператор в функции представляет собой присваивание переменной **RESULT** конструктора нового объекта или результата вызова функции, имеющей модификатор **NEW**.
  - Функцию с модификатором **NEW** можно вызвать только особо (не в составе выражения), присвоив результат какому-либо внешнему владельцу, своей переменной **RESULT**, или временной локальной переменной (с подстрокой temp в имени).
  - Если в функции есть хотя бы одно присваивание нового объекта напрямую переменной **RESULT**, то она обязана иметь модификатор **NEW**.
- **REPLACE** - для переопределенного виртуального метода, **не возвращающего результат**, сообщает, что виртуальный метод класса не использует вызов функции-предшественника, т.е. прототип функции полностью переопределен. Если такой модификатор отсутствует, то в теле требуется хотя бы один вызов метода предка (оператором **BASE**).

- **NATIVE** - в качестве тела функции записывается строчная константа, которая обычно вставляется непосредственно в тело сгенерированной функции в откомпилированного коде. (Либо тело функции завершается оператором **NATIVE "строка"**).
    - Класс, в котором используются **NATIVE**-функции, должен иметь модификатор **NATIVE**.
- **SETTER FOR идентификатор** - функция является методом-присваивателем для поля или получатель-метода, заданного идентификатором. Это поле или метод должно быть объявлено непосредственно перед методом-присваивателем. В случае поля параметр должен быть единственным, того же типа, что и поле. В случае метода параметров должно быть на один больше, чем у метода-получателя, и тип последнего параметра должен совпадать с типом результата получателя (а прочие параметры - идентичны по типам). Наличие такого метода для поля не означает, что полю можно присваивать значения обычным оператором присваивания, кроме особых случаев (см. **REVERT**, **/debug**).
- **CALLBACK** - функция предназначена для вызова из кода самого класса и кода нативных функций, её не следует вызывать из пользовательского кода, в том числе из кода классов-наследников данного класса. Например, событие мыши на форме **mouse_down**: следует переопределять данный метод в своём наследнике формы, но не вызывать напрямую метод класса **{Form}**.
  Модификатор **CALLBACK** следует использовать в случаях, когда требуется предотвратить удаление кода функции оптимизатором.
- **POP(Method2)** - функция может быть вызвана только в операторе **PUSH**. По окончании блока **PUSH**, гарантированно вызывается **Method2** (он не должен иметь параметров, и не может вызываться напрямую вообще никаким способом).
- **DEPRECATED('text')** - функция "осуждается" и теперь необходимо использовать альтернативу, указанную в тексте. В будущих версиях функция, возможно, перестанет поддерживаться. Либо, на некоторых поддерживаемых платформах функция не может быть реализована (в этом случае в апострофах следует записывать фиксированную фразу: **'Platform dependent'**).
- **ABANDONED('text')** - функция более не поддерживается. Используйте альтернативу, указанную в тексте в апострофах. Попытка вызова функции вызывает ошибку компиляции. Такой модификатор может использоваться в унаследованном классе, чтобы указать неприменимость метода, определенного в базовом классе. Но отслеживаться компилятором будут только прямые вызовы этого метода в финальном классе.
- **STORE(Parameter=value)** - Специальный "невидимый" параметр целого типа, создающий на стороне вызывающего класса для каждого отдельного вызова функции скрытую целочисленную переменную, которая и передаётся в качестве параметра, причем по ссылке. См. подробнее в **Дополнительной секции**.
- **FORGET** - при вызове такого метола, все сохранённые невидимые значения на стороне вызывающего объекта сбрасываются в их начальное состояние. См. подробности в **Дополнительной секции**.
- **TRAP** - метод предназначен для использования в качестве отладочной ловушки, срабатывающей на то или иное событие по изменению поля класса. Для одного поля может быть установлено до трех методов-ловушек (на операции чтения, записи, и добавления/удаления элементов в поле, являющееся массивом).
  Список ловушек для поля задается в его модификаторе **TRAP (ловушка1, ловушка2, ловушка3)** . Тип ловушки определяется параметрами:
    - для скалярного поля, метод без параметров вызывается при чтении значения поля, метод с параметром (тип которого должен совпадать с типом поля) вызывается перед присваиванием нового значения (которое передается в качестве параметра ловушке);
    - для поля-массива, добавляется параметр типа **INT** (получает индекс изменившегося или читаемого элемента); в случае первого параметра типа **BOOL** ловушка срабатывает на операции добавления / вставки / удаления элемента, при этом в случае добавления и вставки в этом поле передается **TRUE**, при удалении - **FALSE**; такая ловушка вызывается перед удалением либо после добавления элемента.
- **RESTRICT имя IN [список]** или
  RESTRICT имя IN {имя_блока_констант} или
  RESTRICT имя IS CONST - ограничение на параметр с указанным именем: он может быть только константой (в первом варианте - только из указанного списка).
- **IF имя IN [список]** - параметр функции, заданный именем, участвует в ограничении значения следующего за ним (модификатор **THEN**) параметра. Вместо списка может быть указано имя набора констант в фигурных скобках (см. описание оператора **CONSTANT**).
- **IF имя IS CONST** - если параметр функции, заданный именем, получает константу в качестве значения, то работает следующее ограничение значения параметра (модификатор **THEN**).
- **THEN имя IN [список]** - условное ограничение заданного именем параметра только указанными значениями (при условии, указанном предыдущим модификатором **IF**).

В случае, если в теле функции используются переопределенные операторы, то (независимо от вида функции - **FUNCTION**, **METHOD**, **CONSTRUCTOR**, **OPERATOR** и т.п.), должен присутствовать модификатор **OPERATORS(список классов)** . В скобках должен

задаваться список классов, в которых отыскиваются переопределенные операторы (именно в этом порядке).

II. 3. b. Тело функции

Тело функции состоит из операторов, и завершается символом '.' (точка).

- Тело функции на любом уровне блока не должно содержать более 7 простых операторов и 7 блочных операторов.
- Блок может быть разбит разделяющими комментариями вида
  `---------- 'метка'`
  (не более, чем 7 таких комментариев на блок), в этом случае подсчёт начинается после комментария заново.

Если обозначенные выше ограничения не соблюдены, в заголовке класса требуется указывать модификатор **BAD**.

В случае единственного оператора присваивания значения выражения псевдо-переменной **RESULT**, имя переменной может быть опущено и тело имеет форму
: = выражение
(пробел между ':' и '=' - не обязателен).

II. 3. c. Вызов функций

Круглые скобки требуется использовать только при наличии параметров.

- Параметры перечисляются в круглых скобках через запятую в порядке, соответствующем декларации параметров.
- Параметр-объект передаётся в функцию неявно. Он может быть явно использован (**THIS**).
- Если функция возвращает результат, её нельзя вызывать как самостоятельный оператор без присваивания её результата какой-либо переменной. Если результат не требуется, результат должен быть "присвоен" переменной **NONE**, например, в форме:
  NONE = объект.Функция(параметры)
- Любая статическая функция может быть вызвана в префиксной форме, когда её первый параметр выносится в префикс, например,
  x.Sin - вместо **Sin(x)**
  s.Lower.Ending(".a14") - вместо **Ending(Lower(s), ".a14")**

III. Классы и объекты

III. 1. Классы

III. 1. a. Декларация класса

Синтаксис.

```
/*
CLASS {имена} , модификаторы :
    IMPORT: {Class1}...;
    BASE CLASS {базовый_класс}
    поле1
    поле2
    ...
    метод1
    метод2
    ...
    ---------- 'метка секции 1 '
    ...
    ---------- 'метка секции 2 '
    ...
END

HISTORY
    ...
*/
```

В одном файле располагается декларация одного класса.
Все прочие декларации (перечислимые типы, функции, константы, поля) существуют только в пределах классов.

Глобальных (статических) полей классов не бывает, все поля являются членами каких-либо экземпляров классов.

- Код AL-IV всегда располагается между строками с символами конца и начала многострочного комментария/* и */.
- **Имена классов** заключаются в фигурные скобки и начинаются с прописной буквы.
- Имя класса может разделяться на части символом '|'. До этого символа находится краткое имя класса, после - длинное имя класса.
  Например: {My|_class} определяет два имени - краткое {My} и длинное {My_class} для одного класса.
- Длинное имя должно быть не менее 8 символов (не считая фигурные скобки).
- Имя файла класса (без расширения класса) должно совпадать с длинным именем класса (без фигурных скобок). Но нарушение этого правила не вызывает ошибку, только предупреждение.

III. 1. b. Модификаторы класса

Модификаторы для класса записываются после имен класса, через запятые. Имеются модификаторы:

- **ABSTRACT** - класс предназначен только для наследования новых классов на его основе, создавать объекты этого класса нельзя.
- **BAD** - класс не выдерживает требований по оформлению кода (не более 3 параметров, не более 3 уровней вложенности блочных операторов, не более 7 полей в секции класса, не более 7 простых + 7 блочных операторов в секции кода).
- **BITWISE** - в классе используется поразрядная логика.
- **BYTES** - в классе используются байты.
- **DESTRUCTORS** - имеется деструктор.
- **NATIVE** - имеются "нативные" (низкоуровневые) функции, зависящие от целевой платформы.
- **OPERATORS** - переопределяются операторы.
- **RECURSIVE** - имеются рекурсивные функции.
- **STOPPING** - используется оператор **STOP** .
- **INFINITE** - есть бесконечные циклы.
- **UNTESTED** - есть не оттестированный код.
- **TESTED(nn)** - класс оттестирован на nn процентов .
- **SAFE** - всё хорошо, класс оттестирован, не содержит никаких замечаний по безопасности или оформлению.
- **DEPRECATED('text')** - класс поддерживается, но его использование осуждается. В скобках и апострофах указано, что использовать в качестве альтернативы. Желательно как можно быстрее отказаться от использования этого класса, т.к. в скором времени он может более не поддерживаться.
- **ABANDONED('text')** - использование этой версии класса запрещено. В скобках и кавычках указано, что использовать в качестве альтернативы (не обязательно это класс, возможно, технический приём, или поле другого класса). Попытка объявить переменную с таким классом или функцию, возвращающую объект такого класса (и любая другая ссылка на класс) вызывает ошибку компиляции.
- **INT64 REQUIRED** - требуется 64 бит на все целые значения (переменные, поля, параметры функций - кроме некоторых исключений, таких, как счетчики циклов и индексирующие переменные) ; Если в проекте используется ключ /int32, требующий использования 32-битных целых, это противоречие вызывает ошибку компилятора.
- **INT64 DESIRED** - желательно использование 64-битных целых в проекте. При наличии в проекте опции /int32, компилятор выдает предупреждение о наличии несоответствия настроек проекта пожеланиям разработчика класса;

III. 1. c. Секция импорта

Первыми в классе после заголовка должны быть указаны оператор **IMPORT**, если они требуется. В операторе импорта перечисляются используемые классы (в том числе родительский класс, если класс образован не от базового класса **{Object}**).
IMPORT: {Класс1}, {Класс2}, ...;
(В качестве завершения блока IMPORT может быть точка или точка с запятой).

Операторы **IMPORT** могут иметь модификаторы:

- **TEST** - перечисленные модули подключаются только на этапе тестирования;
- **DEBUG** - только при наличии ключа **/debug** в опциях проекта (т.е. при сборке в режиме "отладка");

- **FRIENDS** - модули объявлены "дружественными", и получают доступ к скрытым полям и методам данного класса (как если бы они были публичными) .

III. 1. d. Наследование

Для указания предка класса, первым в классе после операторов **IMPORT** записывается:

```
BASE CLASS {Base_class}
```

- Если конструкция **BASE CLASS...** отсутствует, то класс не имеет предков, кроме общего для всех классов предка - безымянного класса**{}**.
- Если в классе переопределяются методы предка, то все такие переопределения начинаются словом **OVERRIDE**:

```
OVERRIDE имя_метода(параметры) ==> тип_результата :
                ...
.
```

- Для выполнения действий при создании каждого экземпляра класса, после присваивания значений полей, используется процедура **CONSTRUCT|ION**

```
CONSTRUCT :
                ...
.
```

- Для выполнения действий при уничтожении объекта, используется специальная процедура

```
DESTRUCT :
                ...
.
```

- Если классы использует деструктор, то он обязан иметь модификатор **DESTRUCTORS** в заголовке.

При переопределении метода в классе:

- Используется префикс **OVERRIDE** вместо **METHOD**.
- Указывается только одно имя метода.
- Перечисляются (в скобках, через запятую) все определения параметров переопределяемой функции (должны совпадать первые буквы имен соответствующих параметров). Типы должны быть совместимы, размерности для массивов - совпадать.
- Для методов, возвращающих значение:
  - указывается тип возвращаемого результата.
  - Пример:

```
OVERRIDE enabled ==> BOOL :
      RESULT =o.Edit1.Text!="" .
```

Переопределённый метод может вызывать базовый метод (одноимённый метод родительского класса).

- Для явного вызова базового метода в теле переопределённого метода используется запись **BASE**.
- При этом все параметры передаются обычным способом, в том числе явно передаётся сам объект как неявный параметр:
  BASE(параметры) .
- Метод, не возвращающий значение, либо должен хотя бы раз вызвать базовый метод, либо ему следует добавить модификатор **REPLACE**.

III. 1. e. Объекты. Сильные и слабые ссылки

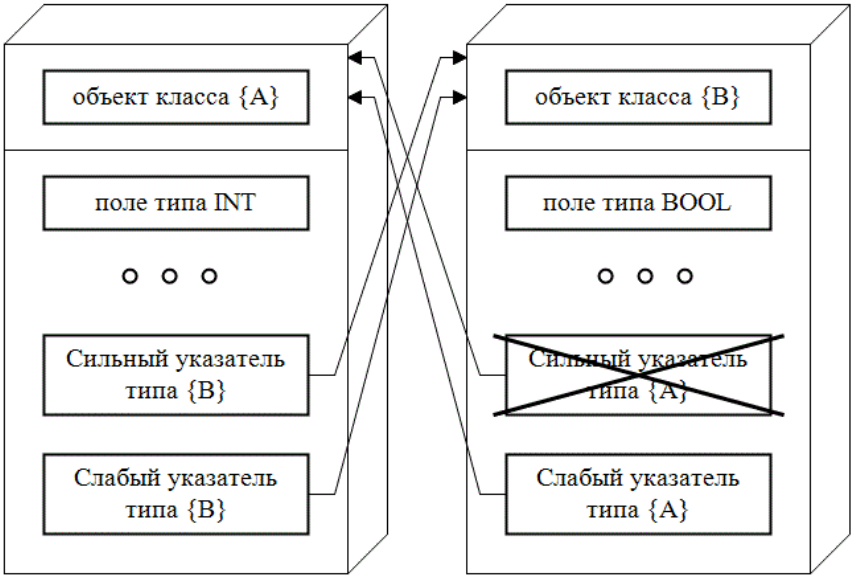
Переменная класса (экземпляр класса) является **объектом** класса.

- Объекты класса декларируются как переменные с именем класса в качестве типа данных.
  Например:
  {My_class} M|y_object={My_class}
- Поле или переменная, не имеющая в именах завершающего символа подчеркивания, является **сильной (удерживающей)** ссылкой объекта.
- **Переменные_** с завершающим подчеркиванием в имени являются слабыми ссылками и **лишь ссылаются на существующие объекты**.
- Слабая ссылка действительна лишь до тех пор, пока объект существует (на него ссылаются переменные-сильные указатели, или жив владелец объекта).
- Все локальные переменные функций являются сильными ссылками. Слабыми ссылками могут быть только поля класса.
- При обнулении счетчика использования сильными ссылками объект уничтожается. Все использующие переменные немедленно начинают ссылаться на **NONE**.

- Если в момент создания объекта для него явно указан владелец (**OWNED BY**), то наличие сильных ссылок на него уже никак не влияет на время жизни объекта: он будет уничтожен вместе со своим владельцем, при этом все ссылки на него, сильные или слабые, перенаправляются на **NONE** - объект.
- **NONE**-объект - это специальный объект класса, имеющий все поля равными **NONE** (и 0 - для чисел, пустым строкам - для строк, и т.д.). Запись в поля этого псевдо-объекта игнорируется, чтение из его полей возвращает **NONE**-значения, вызов методов не выполняет никаких действий.
- Все объектные переменные (как сильные, так и слабые ссылки) изначально имеют значение **NONE**, чтение за пределами массива объектов так же возвращает **NONE** .

ПРАВИЛО

- Поле F_A класса **A**, являющееся удерживающей ссылкой на объект, не может иметь тип класса **B**, являющегося предком по отношению к классу **A**.
- Если класс **A** имеет удерживающее поле класса **B**, то класс **B** не может иметь удерживающих полей класса **A** или классов, родственных **A** по прямой линии (как предков, так и потомков).
- Если класс **A1** содержит удерживающее поле класса **A2**, класс **A2** содержит удерживающее поле класса **A3**, и т.д., и класс **A(n-1)** содержит удерживающее поле класса **A(n)** , то класс **A(n)** не может содержать удерживающих полей классов **A1**, **A2**, ..., **A(n-1)** .
- В частности, объект не может ссылаться на объекты своего класса.
- В результате, **невозможно создание цепочек взаимного владения**.
- Одноранговые сети, графы, и т.п. организуются с помощью **слабых_** ссылок.



- Вновь созданные объект должен быть
  - либо присвоен внешней (по отношению к функции) объектной переменной без подчёркивания на конце имени (сильной ссылке),
  - либо добавлен во внешний массив сильных ссылок (операция**, >> массив []**),
  - либо через запятую должен быть указан объект-владелец вновь созданного объекта, например, **OWNED BY var_ref**,
  - либо присвоен переменной, в имени которой присутствует строка temp (в любом регистре, например, Example|_TempVar) .
- Переменная является внешней по отношению к функции, если она:
  - либо является полем самого класса,
  - либо является полем внешней переменной,
  - либо является псевдо-переменной **RESULT**,
  - либо является полем переменной **RESULT**,
  - либо является полем объектного параметра.
- Если вновь созданный объект хотя бы раз в теле функции присваивается переменной **RESULT**, то функция должны быть помечена маркером **NEW**.
- Функция с маркером **NEW** может вызываться только в отдельном операторе присваивания, как и для оператора создания нового объекта. При этом вновь

созданный такой функцией объект так же должен быть присвоен внешней сильной ссылке или temp-переменной.

- Для результата NEW-функции не может быть применена конструкция **OWNED BY**, в отличие от прямого создания объекта.

III. 1. f. Поля

Поля декларируются в классе так же, как и локальные переменные в функции, но на том же уровне, что и функции, методы и другие декларации класса.

Декларации поля (схема):
ТИП Имя|уточнение [размер], модификатор = инициализация

- **ТИП** - это либо один из базовых типов (**BOOL**, **BYTE**, **INT|EGER**, **REAL**, **STR|ING**), либо имя перечисления, структуры или класса в фигурных скобках.
- Полное имя поля должно быть не менее 8 символов (не считая символа '|'). Если имя начинается с прописной буквы, поле публичное.
  STR Public_s|tring_variable

  Со строчной буквы обычно начинаются закрытые поля.
  INT local_i|nteger

  Если имя поля (только для объектов классов) завершается символом подчеркивания, то это слабая ссылка на объект.
  {Matrix} Transform_|ation_matrix_
  В этом случае подчеркиванием должна завершаться любая часть имени, отделенная символами '|'.
- **[размер]** - необязательный размер массива (не применим к скаляру). Если квадратные скобки пустые, то массив динамический.
  STR List|_of_strings[]

  Размер фиксированного массива может быть задан числовой константой,
  INT Ten_num|bers[10]

  именем перечисления в фигурных скобках,
  BOOL Traffic_l|ights[{colors|_of_traffic}]

  или символом '*'
  BYTE Pixels|_array[*]
  (особый случай, позволяет менять размер массива из нативного кода, в то время как со стороны AL-IV массив видится как обычный фиксированный массив).

Поля класса могут иметь модификаторы, перечисляемые после декларации поля через запятые (но до инициализации константой, если она присутствует).

- **READ** или **READONLY**- модификатор "только для чтения". Изменять такое поле смогут методы самого класса, его наследников и классов-друзей;
- **INIT** или **INITIALIZE**- поле требует инициализации при создании экземпляра объекта. Если для создания экземпляра используется явный конструктор вида **destination={Class_name}(Field1=value1, Field2=value2, ...)** и данное поле не перечислено, это вызывает ошибку компиляции. Если **INIT**-поле является еще и полем только для чтения, то явный конструктор можно будет использовать только в теле самого класса, его наследников и друзей;
- **TRAP(ловушка, ...)** - задает список отладочных методов-ловушек для поля (до 3 ловушек на поле); в качестве ловушки может быть указан метод своего класса, не возвращающий значений в качестве результатов, при этом:
  - если поле скалярное, то в качестве допускаются либо методы без параметров (ловушка на чтение поля), либо методы с единственным параметров (тип которого совпадает с типом поля - ловушка на запись нового значения, вызывается перед записью);
  - если поле является массивом, то допускаются следующие сочетания параметров:
    - **(INT)** - ловушка на чтение элемента с указанным (параметром) индексом;
    - **(INT, {тип поля})** - ловушка на изменение элемента массива с индексом, заданным первым параметром, при этом новое значение передается во втором параметре;
    - **(BOOL, INT)** - ловушка на изменение размера динамического массива: в булевом первом параметре передается **TRUE** в случае вставки нового значения (такая ловушка срабатывает после операции вставки), **FALSE** передается для операции удаления (в этом случае метод-ловушка вызывается перед удалением элемента).
  - ловушки не предназначены для изменения логики работы программы, и с их помощью невозможно изменить присваиваемое или возвращаемое значение, или отменить операцию (хотя и возможно добавить или

удалить другие элементы в массив - но не нужно так делать). Задача ловушек - позволить упростить поиск проблемных операций, с целью отладки алгоритма;

- **CLAMP**-поле-массив фиксированного размера (кратного 2^n), при индексировании которого применяется "заворачивание" по модулю Count (в случае массива из 256 элементов обращение к элементу с индексом 256 дает элемент с индексом 0, 257 - соответственно, 1, и т.д.) ;
- **DEPRECATED(' текст ')** - поле является осуждаемым, не рекомендуется к использованию, и вызывает предупреждение компилятора;
- **ABANDONED (' текст ')** - поле отменено, его использование будет вызывать ошибку компиляции;

III. 1. g. Методы

Декларацию методов, в отличие от статических функций, начинает ключевое слово **METHOD**. Все методы класса являются виртуальными и всегда могут быть переопределены в классе-наследнике (используется слово **OVERRIDE** вместо **METHOD**).

- Ключевая особенность метода: в качестве первого параметра ему передается объект его класса. В списке параметров этот параметр не перечисляется. При необходимости явного обращения к этому параметру используется зарезервированное слово **THIS**. Обращение к полям и методам своего класса (а так же к полям класса-предка или методам предка или наследника) осуществляется неявно, без указания слова THIS.

```
CLASS {Example|_using_of_fields_in_method}:
STR Field|_for_example="Hello, this is Field!"
METHOD Print|_field: << Field.
END
```

- Имеет место полиморфизм: вызов метода приведёт к вызову метода, определённого для экземпляра класса или наследника класса, представляющего объект.
- Всегда можно вызвать из переопределенного метода соответствующий метод класса-предка: **BASE** или **BASE(параметры)** . В том числе несколько раз, и в разных частях кода этого метода (но только этого).
- Метод без параметров должен либо вызывать метод **BASE** хотя бы один раз, либо иметь модификатор **REPLACE** (сообщающий компилятору, что новый метод полностью замещает предыдущий, и его вызывать не нужно) .
- Конструктор всегда вызывает конструктор класса-предка до начала работы, и делает это неявно. Вызывать **BASE** в конструкторе явным образом не требуется (и нельзя). Т.к. конструктор параметров не имеет никогда, с передачей параметров проблем нет.
- Деструктор всегда вызывает деструктор класса-предка по окончании своей работы, так же неявно.

III. 2. Работа с объектами классов

III. 2. a. Оператор создания экземпляра

Конструкция объекта:

```
variable={Class_name}(
 Field1=expression, Field2[] << expression, ...)
```

- Если значение присваивается переменной в операторе декларации этой переменной (который начинается с указания типа переменной в фигурных скобках), и оператором создаётся объект именно этого типа, то в декларации может быть опущено имя типа декларируемой переменной:

```
stream|_read_temp={File_stream}(
       Path=source_path, Mode= 'READ')
```

- Если при создании объекта не требуется инициализировать поля, то круглые скобки опускаются.
- Если за именем типа и списком инициализации следует через запятую запись **OWNED BY выражение**, то тем самым созданному объекту назначается исключительный владелец.
    - Для такого объекта, счётчики использования строгими указателями не влияют на время жизни объекта: он существует, пока жив его владелец.
    - Если на момент создания объекта значение выражения в части **OWNED BY** равно **NONE**, то владелец не назначается, и при отсутствии других строгих ссылок на него, проживёт не долее, чем выполняется функция, в которой он создан.
- Либо далее может располагаться (через запятую) операция добавления созданного объекта в массив объектов: **, >> переменная []**.
- Создаваемый объект должен быть присвоен либо внешнему для функции объекту (параметру или его полю, полю класса или его полю), или добавлен во внешний по отношению к функции массив строгих ссылок на объекты, либо ему должен быть назначен исключительный владелец (**, OWNED BY**). В противном

случае, переменная, которой присваивается созданный объект, должен иметь в своём имени подстроку 'temp'.

III. 3. Другие операторы уровня класса

Если класс имеет предка в иерархии, он должен быть указан оператором **BASE CLASS {Имя}**, но после секции импорта, в которой указывается класс предка.

Завершается класс оператором **END** в отдельной строке.

До оператора **END** могут встречаться:

- операторы определения перечислений
```
ENUM {имя | уточнение}:
    'НАИМЕН|ОВАНИЕ1',
    'НАИМЕН|ОВАНИЕ2',
     ...  .
```
- операторы описания структур
```
STRUCTURE {имя|уточнение} :
    поле1
    поле2
    ....
```
- операторы описания таблиц
```
TABLE Имя | уточнение: {структура}
    NAME " имя "
    COUNTER(список)
    NOTNULL(список)
    NAMES(поле=" имя ", ...) .
```
- декларации полей
```
ТИП Имя | уточнение [размер], модификатор
 =инициализация
```
- декларация конструктора
```
CONSTRUCT:
    ТЕЛО.
```
- декларация деструктора
```
DESTRUCT:
    ТЕЛО.
```
- декларации статических функций
```
FUNCTION Имя|уточнение(параметры) ==> ТИП_РЕЗУЛЬТАТА,
    модификатор1, модификатор2, ... :
    ТЕЛО.
```
- декларации методов
```
METHOD Имя|уточнение(параметры) ==> ТИП_РЕЗУЛЬТАТА,
    модификатор1, модификатор2, ... :
    ТЕЛО.
```
- декларации переопределённых функций
```
OVERRIDE Имя(параметры) ==> ТИП, модификатор1, ...:
    ТЕЛО.
```
- функции тестирования
```
TEST Имя | уточнение (параметры) ==> ТИП_РЕЗУЛЬТАТА :
     ТЕЛО.
```
- переопределения операторов
```
OPERATOR ТИП ОПЕРАЦИЯ ТИП==> ТИП_РЕЗУЛЬТАТА:
     ВЫРАЖЕНИЕ.
```
- блочные комментарии
```
-------------------------- 'текст' текст '
'
```
- директивы NATIVE
```
NATIVE: " код специфичный для платформы " .
```
- операторы TODO
```
TODO: " строка " .
```

III. 4. Завершение класса. История изменений. Массив DATA[].

Класс завершается оператором **END**. Если после строки с оператором **END** первая непустая строка начинается ключевым словом **HISTORY**, то далее размещается блок истории изменений класса. Независимо от наличия истории изменений, при наличии непустого текста после директивы END, все эти строки доступны в коде класса через псевдо-массив строк **DATA[]**.

Если директива **HISTORY** имеется, то история должна иметь достаточно строгий формат. История состоит из блоков **CREATED/UPDATED** (блок **CREATED** может быть только первым, и только единственным, если присутствует). Формат заголовка блока:

(' CREATED' | ' UPDATED ') '(' YYYY-MM[-DD] ')' [',' ' VERSION' '"' text '"' [' BY' '"'text'"'] ':'
Содержимым блока являются сообщения в форме:
(' ADDED' | ' CHANGED' | ' FIXED') ':' ('"'text'"' | identifier) [',' ('"'text'"' | identifier)]... (';' | '.')
Последнее сообщение блока изменений должно завершаться точкой, или состоять из одной только точки.

История изменений может завершаться вместе с концом текста. Но если кроме истории изменений текст после директивы **END** содержит другую информацию, то история должна завершаться директивой
HISTORY ENDED
Например:

```
HISTORY:

UPDATED(2018-08-15), VER "1.0a":
    ADDED: "This history added";
    CHANGED: "No other changes yet".
```

IV. Структуры (STRUCTURE)

IV. 1. Декларация структур

'

Синтаксис.

```
STRUCTURE {name|_of_structure}:
    поле1
    поле2
    ...
    поле N.
```

Структуры декларируются внутри классов.
Имя структуры всегда начинается со строчной буквы.
Имя структуры может делиться значком '|' на части: часть до первого знака является кратким именем. Полное имя не должно быть короче 8 символов.
Все декларации структур в классе всегда доступны для использования в любых классах, использующих этот класс.

Структура состоит из:

- полей простых типов фиксированного размера (**BOOL**, **BYTE**, **INT**, **REAL**, {перечисления}, **STR**)
- полей типа структуры (любые другие структуры в области видимости - рекурсивное вложение не допускается)
- ссылок на экземпляры классов
- массивов из выше перечисленных элементов
- ссылок на другие структуры (для включения ее в свое тело на том же уровне вложения) в виде **LIKE {имя_типа}** (или **LIKE {Класс}.{структура}**, в случае ссылки на структурный тип из другого класса).

Декларация структуры завершается точкой.

Структура, как и класс, может содержать поля любого типа. Но имеются ограничения на объекты классов:
поля структуры, ссылающиеся на объекты классов (т.е. имеющие тип класса), должны быть слабыми ссылками (и их имена должны завершаться подчеркиванием).
Если же поле класса имеет тип структуры, то в этом случае оно само не может быть слабой ссылкой (т.к. любая переменная или поле типа структуры является ее единственным владельцем).

Все структурные переменные и поля автоматически инициализируются присваиванием полям NONE-значений (для чисел - это нули, для строк - пустые строки, для объектов классов - NONE). NONE-значение так же может быть получено для структуры при чтении из массива за пределами доступного диапазона индексов.

IV. 2. Работа со структурами

Структура - это класс без методов, имеющий всегда только одну ссылку на него. Поэтому при создании экземпляра структуры нельзя использовать модификатор **OWNED BY** или правую запись в массив **>> Array[]**, как для объектов классов.

Если одной структурной переменной присваивается значение другой структурной переменной, то при этом к переменной в правой части присваивания должна быть применена встроенная функция-метод **CLONE** или **Dismiss**. В случае использования **Clone** значение исходной структуры дублируется, в случае **Dismiss** - исходная переменная может быть обнулена (получает значение **NONE**). Это касается всех видов присваивания или добавления в массив (элемент массива структур так же считается переменной) .

```
Destination=Source.Clone
Destination=Source.Dismiss
Destination=Source_array[i].CLONE
Destination_array[j]= Source_array[i].Clone
Destination_array[] << Source.Dismiss
```

Например:

```
STRUCTURE   {person|_info} :
    STR F|irst_name
    STR L|ast_name
    INT Y|ear_birthday
    {education} E|ducation
    STR S|tate_province.

...

{person} Smith|_John={person} (F="John", L= "Smith",
               Y=1950, E='High', S="NY")
```

Такой же конструктор может использоваться при передаче фактического параметра функции в позиции параметра типа структуры.
При присваивании структуры ее содержимое просто копируется.

При передаче структуры в качестве параметра, она может использоваться в вызванной функции только для чтения, в том числе нельзя изменять значения ее полей. Это отличается от правил работы с объектами, для которых поля могут быть изменены .
Соответственно, при присваивании значения всей структуры-параметра, может (и должен) использоваться только псевдо-метод **Clone**, и не может быть применен метод **Dismiss**.

Структура может быть возвращена как результат функции. При работе с псевдо-переменной **RESULT** типа структуры необязательно ее инициализировать, т.к. она уже проинициализирована NONE-значением по умолчанию.

Структуры не могут передаваться в качестве параметра или результата нативных (низкоуровневых) функций.

Структура может быть создана обычным для объектов оператором создания структуры:
{имя-структуры}(поле1=выражение, поле2=выражение, ...)
Например,

```
{complex} c|omplex_var={complex}(Im=5)
```

Если цикл **FOR** организован как перечислитель массива структур, то переменная цикла является макросом, позволяющим обратиться к очередному элементу массива как на чтение, так и на запись. Т.е., если выполнить присваивание какому-либо полю этой переменной, то фактически при этом изменится поле структуры, хранящейся в соответствующем элементе массива:

```
FOR x IN A[]:
    x.Some_field=Some_value.CLONE;
    //все структуры массива A[] изменяют значение поля Some_field
```

Существует возможность присваивания структур разных типов, имеющих одноименные (и однотипные, или совместимые по типам, поля) :

```
x, BUT (FieldX1, FieldX2, ...) =y, BUT (FieldY1, FieldY2, ...)
```

В левом списке **BUT** должны быть перечислены все поля из x, которых нет в переменной y. В правом - все поля из y, которых нет в x. Также можно указать дополнительные поля, не участвующие в присваивании, как слева, так и справа. Левый список BUT нужно указывать всегда, даже если нет необходимости опускать поля из x для присваивания. В этом случае скобки опускаются. В минимальном варианте такое присваивание выглядит так:

```
x, BUT = y
```

При присваивании разнородных структур псевдо-функции **CLONE**/**DISMISS** не вызываются.

Структурные переменные могут использоваться при работе с базами данных, в сочетании с операторами SQL (**SELECT**, **INSERT**, **UPDATE**, **DELETE**). См. подробнее в разделе о встроенной поддержке SQL.

IV. 3. О реализации структур в конечной программе

Не должно быть никаких допущений насчет того, как поля структур располагаются в оперативной памяти во время работы программы, или об их порядке или выравнивании (на границы машинных слов). То же касается размеров структур в

памяти, и способах размещения их в памяти. Так же, предположение о большей (или меньшей) эффективности структур могут оказаться полностью неверны.
Структуры могут быть фактически реализованы как объекты, или для простых структур, не содержащих строк, динамических массивов и ссылок на объекты, могут быть реализованы именно как структуры в целевом языке.
Все, что действительно нужно знать о структурах - это то, что структура хранится как простая переменная, при присваивании копируется как переменная (например, поле за полем). На них нет возможности сослаться указателем (хотя при передаче в качестве параметра используются именно указатели - но поля структуры предназначены только для чтения).
Требование использовать функции CLONE / DISMISS введено для того, чтобы программист чаще обращал внимание на то, что данные структуры при присваивании именно копируются, и оригинальная структура не будет изменена при изменении полей принимающей структуры. Это уменьшает количество ошибок в программе.

V. Тестирование

V. 1. Основные положения

Класс может содержать особые функции, которые начинаются ключевым словом **TEST** вместо **FUNCTION**. Они предназначены для тестирования и выполняются всегда на этапе компиляции.

- Если тесты не выполняются для класса успешно, класс должен быть помечен атрибутом **UNTESTED**.
- Если на этапе тестирования выполнялись не все строки кода, подлежащего тестированию, класс так же должен быть помечен атрибутом **UNTESTED**.
- Если хотя бы один оператор **ASSERT** не выполняется в тестах класса, класс так же д алее не компилируется (так же, как при ошибке компиляции) .
- Если класс должен быть помечен как **UNTESTED**, но он не имеет этого атрибута, это считается ошибкой, и компиляция завершается на этапе тестирования.
- Класс может быть помечен как частично оттестированный, с указанием численного значения степени оттестированности: **TESTED(n)** . В этом случае достаточным считается иметь заданный числом n процент оттестированных строк кода.

Тестированию подлежат:

- Все строки кода.
- Все условные ветви кода.

Тест считается покрывающим, только если все подлежащие тестированию строки кода выполнялись при тестировании класса хотя бы один раз. Если это условие не выполнено, класс должен быть помечен как **UNTESTED**.

Все секции функции **TEST** должны иметь операторы **ASSERT**. Компилятор не может проконтролировать качество использования этих операторов, поэтому отсутствие таких операторов в секции вызывает только предупреждение.

Тестированию не подлежат:

- Нативные функции.
- Абстрактные классы.
- Классы, не содержащие методов.
- Методы, не содержащие операторов.

Однако, при тестировании класса, унаследованного от абстрактного, все не переопределённые методы его абстрактного предка должны быть протестированы, чтобы сам класс считался протестированным.
Допускается в абстрактном классе поместить тестирующие функции с параметрами, предназначенные для тестирования всех или части функций абстрактного класса.
Тесты с параметрами не выполняются автоматически, но могут быть вызваны из других тестирующих функций. В частности, если создать тест(ы) с параметрами внутри абстрактного класса для тестирования его методов, то эти тесты могут быть вызваны в унаследованных классах для упрощения тестирования кода абстрактного предка.

Тестирование выполняется каждый раз при компиляции проекта (кроме случаев, когда результаты предыдущего успешного тестирования сохранены, и не изменился

код, от которого они зависят, либо в опциях проекта для компилятора тестирование запрещено ключом /$NOTESTS - но это зависит от компилятора).

При кроссплатформенной компиляции в некоторые целевые платформы (Java/Android) тестирование не выполняется на этапе компиляции, но со специальным ключом может быть отдельно собрано (консольное, или - псевдо-консольное) приложение для целевой платформы, специально для целей тестирования, работа которого как раз состоит в выполнении тестов. Интеграция результатов тестирования в цикл работы по компиляции приложения в этом случае невозможна.

В случае кросс-компиляции для платформы Linux, тестирование кода возможно на исходной платформы. Но следует учесть, что достоверность тестов в таком случае так же не 100% (платформы все-таки отличаются, и одинаковый код на финальной платформе может в некоторых случаях работать по-разному). Впрочем, кросс-платформенная компиляция для Linux выполняется (обычно) только для самого компилятора, после чего должен работать откомпилированный и собранный компилятор на платформе Linux.

V. 2. Синтаксис

Функция тестирования начинается заголовком, в котором слово **TEST** используется вместо слова **FUN|CTION**.

Тестирующие функции могут иметь параметры, но такие тесты не вызываются автоматически: они могут быть вызваны только из других тестирующих функций.

Если для функции тестирования требуются классы, не перечисленные в директивах **IMPORT**, то следует добавить такие директивы с модификаторами **TEST**, например:

```
IMPORT, TEST: {String_functions}.
```

Как и другие функции, тестирующая функция разбивается на секции блочными комментариями

```
-------------- 'текст'
```

Но для тестирующих функций имеется дополнительное требование: секция должна содержать как минимум один оператор **ASSERT** .

Оператор **ASSERT** предназначен для проверки булевского выражения. Если значение выражения в операторе

```
ASSERT выражение
```

ложно, утверждение не выполняется. При наличии хотя бы одного проваленного утверждения класс считается не оттестированным.

Оператор **ASSERT** может иметь второй (строковый) аргумент

```
ASSERT x, y
```

- он вычисляется и отображается в сообщении о неудачном утверждении, для упрощения понимания, что именно пошло не так.

VI. Встроенная поддержка SQL- запросов

VI. 1. Кодирование SQL запросов

Если в классе определён метод **Write(STR)** , то для объектов такого класса разрешена операция **объект << строка**, фактически вызывающая этот метод.

В случае класса **{DB}**, предоставляющего интерфейс для работы с базами данных, операция << может (и должна) использоваться для установки текста SQL-запроса (в поле SQL, которое доступно только по чтению).

```
db << SELECT (*), FROM Students a,
      JOIN Students b ON(a.Exam1=b.Exam1),
      ORDER BY (Surname, Firstname, Middlename)
db.Open
```

VI. 2. Структура таблиц БД, оператор TABLE

При установке текста SQL, если текст строкового выражения начинается с одного из ключевых слов **INSERT** / **DELETE** / **UPDATE** / **SELECT**, то вся строка представляет из себя SQL-подобный оператор, который формирует текстовую строку с SQL-запросом. При этом если не все, то многие параметры SQL-утверждения могут быть проконтролированы на этапе компиляции кода, что существенно снижает вероятность возникновения ошибок на этапе выполнения программы.

Такие SQL-запросы ссылаются на таблицы, декларируемые утверждениями **TABLE** (ссылающимися на декларации структур **STRUCTURE**). Например:

```
STRUCTURE {abiturient}:
        REAL ID|entity_counter_64bit
        STR Surname|_last_name, MAXLEN[40]
        STR FirstName, MAXLEN[40]
        STR MiddleName, MAXLEN[40]
        INT Exam1|_scores
        INT Exam2|_scores
        INT Exam3|_scores
        {date_time} D|ate_in_documents
        BOOL OriginalDocuments
        BOOL AgreeToEnroll.


TABLE Students|_want_to_be:
        {abiturient}
        NAME "Students"
        COUNTER (ID)
        NOTNULL (Surname, FirstName, D)
        NAMES (D="DateJoin") .
```

VI. 3. SQL- подобный синтаксис

Запросы SQL-подобны, но их синтаксис несколько изменён по сравнению со стандартом SQL:

- Все ключевые слова записываются в верхнем регистре: **SELECT**, **UPDATE**, **INSERT**, **DELETE**, **DISTINCT**, **TOP**, **FROM**, **AS**, **INTO**, **WHERE**, **GROUP**, **ORDER**, **BY**, **IN**, **NOT**, **IS**, **NULL**, **JOIN**, **ON**, **LEFT**, **OUTER** ;
- все части SQL-подобного утверждения разделяются запятыми (что упрощает перенос на другую строку, в соответствии с правилами AL-IV) . Например:
  SELECT DISTINCT, FROM Students, (*), WHERE Graduate_date BETWEEN {d1}, AND {d2}
- Список выбираемых значений, а так же полей для сортировки и группировки, всегда заключается в круглые скобки;
  Список значений может размещаться либо сразу вслед за словом **SELECT** (или **SELECT TOP**(n), **DISTINCT**), либо после списка используемых таблиц (часть **FROM-JOIN-JOIN** -...) ;
- Идентификаторы по возможности считаются ссылками на таблицы, алиасы таблиц, имена полей, и только при несовпадении со всеми ожидаемыми идентификаторами из этих списков, далее рассматриваются компилятором как части AL-IV-выражения (имена переменных, функций и т.п.) . Например:
  UPDATE Students, SET Exam3=64,
  WHERE ID={Ident_to_update}
- При необходимости явно указать, что часть кода относится не к синтаксису SQL, а к синтаксису AL-IV, **выражение AL-IV заключается в фигурные скобки**. Все такие выражения должны иметь один из типов данных **INT** / **REAL** / **BOOL** /**STR** / **{date_time}** / {перечислимый} (и соответствовать типам полей). Другие типы не допускаются (как исключение: тип **BYTE** может рассматриваться как целочисленный);
- Для представления счётчиков (автоинкрементных полей) следует использовать тип **{id}**, т.к. разрядности обычного целого может быть недостаточно для представления длинного целого, использующегося в счетчиках;
- Выражения AL-IV автоматически преобразуются в строку, воспринимаемую БД. Нет необходимости (и нежелательно) самому преобразовывать параметрические значения в строки с использованием функция Bool_sql, Int_sql, Str_sql, Date_sql, Real_Sql;
- В операторе **INSERT**, в отличие от T-SQL и подобно **UPDATE**, список полей и присваиваемых значений задаётся парами
  ИМЯПОЛЯ=ЗНАЧЕНИЕ. Например:

  ```
  db << INSERT INTO Students,
        Surname= {"Origatsu"},
        FirstName={"Pei"},
        MiddleName={"Q"},
        Exam1={84},
        Exam2={81},
        Exam3={92},
        D={Date(2017, 7, 14)},
        OriginalDocuments= { TRUE }
  db.Exec
  ```

- В операторах **INSERT** и **UPDATE**, возможно указать переменную (или выражение) типа структуры, соответствующей таблице, для занесения содержимого этой записи в таблицу, вместо перечисления большого числа полей:
  db << INSERT INTO Students, BY Rec
  db.Exec
- В предыдущем варианте запроса возможно дополнительно указать в форме **BUT имя_массива[]** строковый массив, содержащий имена полей, для которых не будут устанавливаться (или изменяться) значения:
  db << UPDATE Students, BY Rec, BUT array[]
- Если запрос использует более одной таблицы, то единственный способ подключить дополнительные таблицы - это утверждения **JOIN**, следующие за **FROM** (или за первой таблицей - в **UPDATE**). При этом все таблицы, включая основную, должны иметь уникальные алиасы, и далее обращение происходит к полям в виде **алиас.поле**, например:

  ```
  db << SELECT FROM Students a,
        JOIN Students b ON (a.Exam1=b.Exam1),
  ```

```
      (  a.ID         AS a_ID,
         a.FirstName  AS a_F,
         a.MiddleName AS a_M,
         a.Surname    AS a_S,
         COUNT (b.*)  AS CNT_b),
      GROUP BY (a_ID)
db.Open
```

- Разрешены вложенные запросы в операциях `X IN (SELECT...), X NOT IN (SELECT...), EXISTS (SELECT...), NOT EXIST(SELECT...)`
- Иногда необходимо в качестве операнда предоставить некий текст, который будет просто вставлен в соответствующую позицию в SQL- запросе, без анализа его синтаксиса. В этом случае используется псевдо-функция `SQL(...)` , где вместо... вписывается выражение AL-IV, возвращающее строку (или просто строковая константа). Например:
  ```
  ...WHERE x IN SQL("(1,2,3)"),
        AND SQL("GetDate()") BETWEEN b.D1 AND b.D2
  ```
  При этом вставленный текст всегда заключен в круглые скобки. Т.е. можно записать:
  `... WHERE Id IN SQL(List_id[0 TO *].Merge(","))`
  (не добавляя строчные литералы "(" и ")" вокруг списка значений.
- Строки запроса могут разделяться блочными комментариями вида
  `------------ ' комментарий '`
  Части запроса между такими комментариями могут быть удалены перед выполнением запроса методом `Remove_after`(" текст "). Это можно использовать для формирования запросов с переменным содержимым, при том, что компилятор проверит запрос на корректность, пока все строки на месте. Для случаев, когда таких опциональных частей в запросе несколько, и удобнее сформулировать не условия удаления таких частей, а условия их оставления в запросе, следует такие комментарии завершать символом '?' и использовать вызовы метода `Allow`(" текст "). При этом, если даже метод `Remove_not_allowed` не вызывался, перед выполнением запроса в любом случае будут удалены части SQL-кода, находящиеся после комментариев вида
  `------ ' текст?'` (до следующего блочного комментария или до конца запроса).

VI. 4. Выполнение запросов INSERT, UPDATE, DELETE

Запросы `INSERT`, `UPDATE` и `DELETE` не требуют результатов. Оператором `<<` запрос устанавливается, и методом `Exec|ute` выполняется:

```
db << INSERT INTO Students,
      Surname= {stud.Last_name},
      FirstName={stud.First_name},
      MiddleName={stud.Mid_name},
      Exam1={exam[0]},
      Exam2={exam[1]},
      Exam3={exam[2]},
      D={date_exam},
      OriginalDocuments={ TRUE}
db.Exec
```

Для получения идентификатора вставленной записи следует использовать метод `Identity(STR Table_name, STR Identity_field) ==> {id}`, так как корректный способ получения этого значения отличается от базы к базе.

Аналогично, для получения количества измененных/вставленных строк, следует вызывать метод `Row_count`. Но он может быть не реализуем для каких-то типов баз данных, поэтому имеет маркер `DEPRECATED('Depends on database kind')` .

При вставке записей оператором `INSERT`, или изменении записи таблицы оператором `UPDATE`, код можно значительно сократить, если использовать возможность вставки данных из структуры, соответствующей таблице базы данных:

```
db << INSERT INTO Table_name, BY structure_var
db << UPDATE Table_name, BY structure_var
```

При этом возможно пропустить определенные поля, имена которых перечислены в списке строк:

```
db << INSERT INTO Table_name, BY structure_var, BUT null_fields[]
db << UPDATE Table_name, BY structure_var, BUT skip_fields[]
```

Если в базе настроена вставка значений NULL по умолчанию в поля, не перечисленные в операторе INSERT, тот в пропущенные поля будут помещены значения NULL.

VI. 5. Получение результатов SELECT

Для получения результатов запроса SELECT:

- рекомендуется использовать цикл `FOR x IN ENUM db.Results:...;`
  В методе `Results` класса `{DB}` последовательно выполняется просмотр записей от первой к последней:
  ```
  db.Open
  FOR j|dummy ENUM db.Results:
      id=db.CInt("ID")
      name= db.CStr("Name")
      STR r|esults_array[] << "id=" id " name=" name ;
  db.Close
  ```
  Этот вариант не требует вручную прописывать код проверки исчерпания результатов и перехода к следующей записи, все это делается в методе `Results`.

- но возможно использовать и классический цикл с предварительной проверкой того, что есть данные для считывания:

```
db.Open
CASE !db.Ended?
     FOR i|dummy IN [1 TO 999_999] :
         {results_tab} r|esults1 << db
         {results_tab} all|_results[] << r
         CASE !db.Next? BREAK i ;
     ;
;
db.Close
```

Либо цикл с проверкой на каждой итерации:

```
db.Open
FOR j|dummy IN [0 TO 999_999]:
    CASE db.Ended ? BREAK j ;
    id=db.CInt("ID")
    name= db.CStr("Name")
    STR r|esults_array[] << "id=" id " name=" name
    db.Forward ;
db.Close
```

- Если требуется получить единственное значение, для этого есть набор функций Open_VInt_close, Open_VStr_close и т.п. Например:

```
REAL x|_some_value=db.Open_VReal_close
```

Для приема результатов:

- имеется возможность использовать особую форму оператора **<<** , в котором в качестве приемника используется переменная структурного типа, а справа указывается переменная типа **{DB}** (или унаследованного типа), с дополнительным указанием таблицы, в форме:

```
db.Open
FOR i|dummy ENUM db.Results:
    {results_tab} r|esults1 << db, BY TABLE Students
    {results_tab} all|_results[] << r;
db.Close
```

При этом структура слева не обязана быть точно такой же, как и та структура, на которой основана таблица - но для всех ее полей должно быть однозначное соответствие по именам с базовой структурой такой таблицы;

- Либо читать поля по мере надобности используя соответствующие методы класса **{DB}**: **VBool(n)**, **VInt(n)**, **VReal(n)**, **VStr(n)**, **VDate(n)**, **VId(n)** , передавая индекс поля в списке выбираемых значений;
- Либо читать поля методами **CBool(name)** , **CInt(name)** , **CReal(name)** , **CStr(name)** , **CDate(name)** , **CId(name)** , передавая в качестве параметра имя поля (или алиас значения из списка выбора). Причем, это должна быть константная строка, и при этом не происходит никаких потерь эффективности по сравнению с предыдущим вариантом (методы Cxxx имеют "скрытый" параметр, который позволяет кэшировать найденные индексы полей, и фактически после первой итерации далее обращение к выбранным идет по числовым индексам, без повторения процедуры поиска поля).
- Так же, имеются методы **VNull(n)** и **CNull(name)** , возвращающие логическую "истину", если получено значение **NULL** для указанного поля.

VI. 6. Транзакции

Транзакции реализованы через PUSH-методом Transaction. Метод Transaction должен быть вызван в блочном операторе **PUSH**:

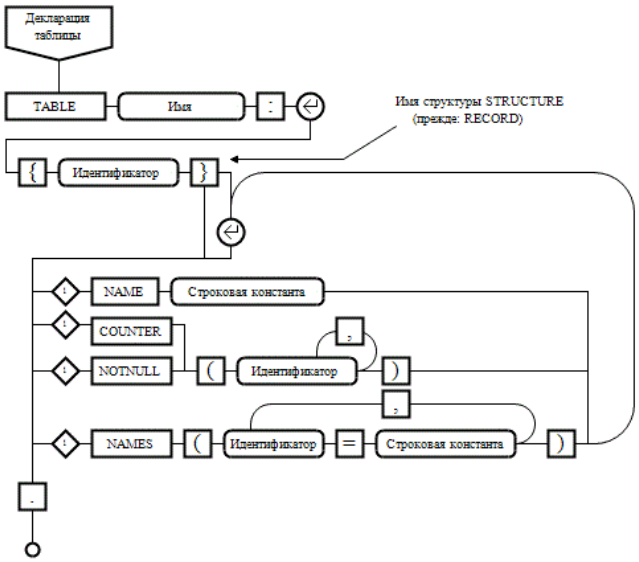
```
PUSH  DB.Transaction :
      FOR sel IN Containers.Selection[] :
          DB << UPDATE Package,
                SET Barcode = {E_Barcode.Text},
            WHERE  Packid =  {Containers.[sel,
                    .Columns[].Count-1].Id_from}

         DB.Exec
      ;
      CASE !DB.Commit ?
          Main_.Handle_error(
          "{Containers_param}.value_change(E_BARCODE)")
      ;
;
```

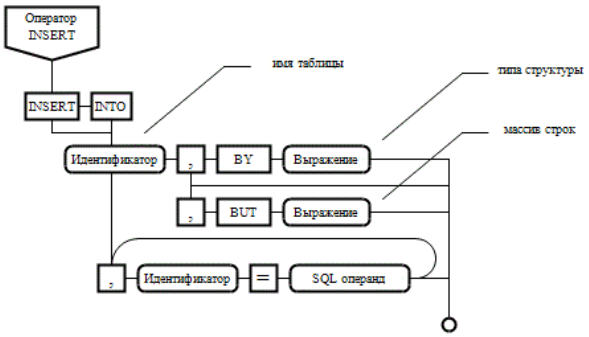
Если выполнение успешно дошло до ';', завершающей блок, то далее, в зависимости от того, был вызван метод **Commit**, или нет (или последним был вызван **Rollback**), произойдет либо логическое успешное завершение транзакции (операция "commit"), либо откат сделанных изменений (операция "rollback").

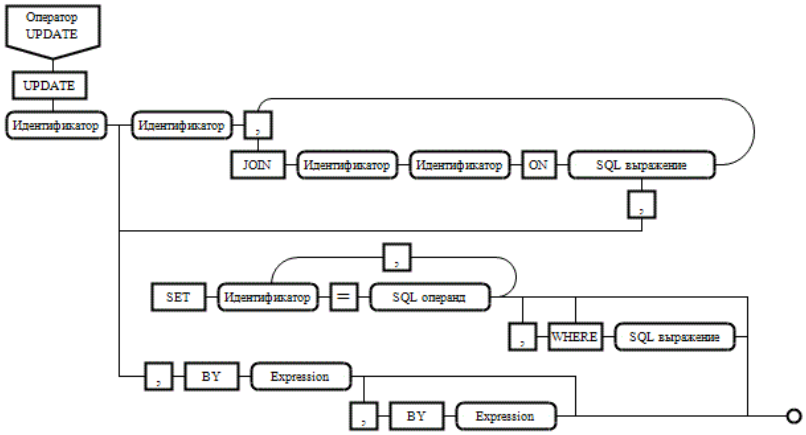
Чтобы программист "не забыл" вызвать метод Commit в теле блока транзакции, метод **Transaction** "снабжен" модификатором **WAIT(Commit, Rollback)**. Т.е., если ни тот, ни другой метод не встретится в блоке, программист получил сообщение об ошибке.

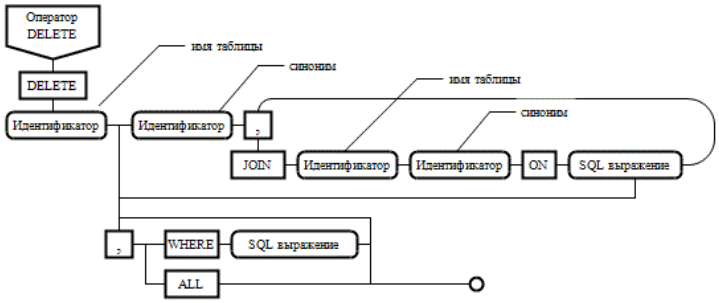
VI. 7. Синтаксические диаграммы

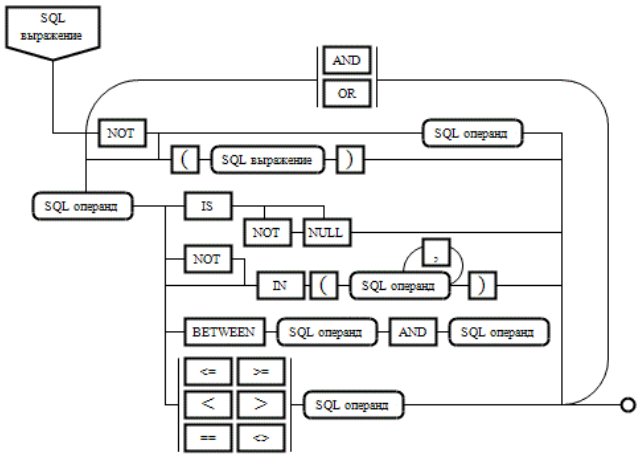
Ниже приводятся синтаксические диаграммы для операторов, формирующих SQL-запросы:

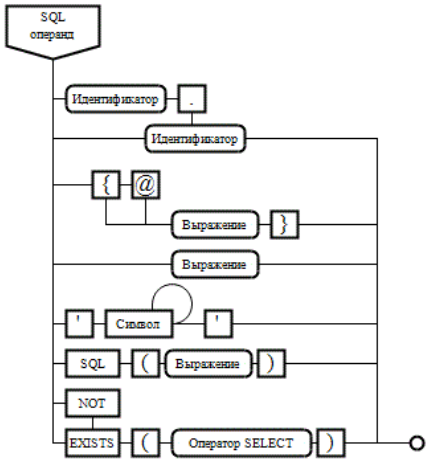





Оператор UPDATE
UPDATE
Идентификатор
Идентификатор
,
JOIN
Идентификатор
Идентификатор
ON
SQL выражение
,
SET
Идентификатор
=
SQL операнд
,
WHERE
SQL выражение
BY
Expression
BY
Expression


Оператор DELETE
DELETE
имя таблицы
синоним
имя таблицы
синоним
Идентификатор
Идентификатор
JOIN
Идентификатор
Идентификатор
ON
SQL выражение
WHERE
SQL выражение
ALL


SQL выражение
AND
OR
NOT
SQL операнд
(
SQL выражение
)
SQL операнд
IS
NOT
NULL
NOT
IN
(
SQL операнд
)
BETWEEN
SQL операнд
AND
SQL операнд
<=
>=
<
>
==
<>
SQL операнд


SQL операнд
Идентификатор
.
Идентификатор
{
@
Выражение
}
Выражение
'
Символ
'
SQL
(
Выражение
)
NOT
EXISTS
(
Оператор SELECT
)

VII. Дополнительно

VII. 1. Локализация строковых ресурсов

AL-IV поддерживает механизм локализации строк с помощью псевдо-функций вида _идентификатор. Для строки это выглядит как вызов функции, например, "Red square"._Rsq или _Rsq("Red square")

Но вызовом функции данная конструкция не является. Она лишь сообщает компилятору о том, что строку "Red square" следует поместить в массив локализуемых строк, индекс строки в этом массиве запомнить и использовать для извлечения строки из этого массива. Фактически, при этом может быть извлечена другая строка, записанная в этот массив вместо исходной строки "Red square" в результате работы функций локализации. Например, это может быть строка " Красный квадрат ".

Все имена таких строковых ресурсов должны быть уникальными в пределах класса.

В качестве стандартного API для управления локализацией предлагается класс {Localize_str} (хотя это не обязательно, и всегда может быть изготовлен другой класс, работающий по другим алгоритмам).

При использовании класса {Localize_str} следует как минимум один раз (при инициализации программы) вызвать его статическую функцию Localize, указав краткое имя языка локализации. Предполагается, что имя языка приложение сохраняет само, в доступном ему месте (например, используя класс {Configuration}). Для упрощения настройки приложения в плане выбора одного из доступных для локализации языков, имеется метод List_languages, возвращающий список полных и кратких имён языков, извлечённых из названий языковых файлов (предполагая, что они именуются в едином стиле English_EN.lng).

Метод Localize имеет дополнительный строковый параметр Prefix, который позволяет переводить в программе только языковые ресурсы с именами, начинающимися с этого префикса (префикс задаётся без лидирующего подчёркивания). Пустая строка в качестве префикса означает трансляцию всех строк, независимо от имён ресурсов.

При обращении к методу Localize первым делом вызывается метод Save_untranslated, если до этого он не вызывался из программы. Пользователь-переводчик всегда может обнаружить первичный языковый файл, сохранённый этим методом, сделать из него копию, поименовав её в форме Имя-языка_ИЯ.lng и отредактировав в текстовом редакторе. Строки, подлежащие переводу, имеют форму NAME=Value. Заменять следует только значения Value, не трогая имя и знак равенства. В строке могут быть и другие знаки '=', они могут изменяться в соответствии с требованиями языка. Отредактированная версия языкового файла должна быть сохранена в формате UTF-8.

Местом хранения языковых файлов по умолчанию являются либо директория, в которой запускается приложение, либо (если запись туда невозможна) - специальная директория для данных приложения. Либо, имеется возможность явно указать такую директорию (метод Set_directory_lang).

Важно, что если в процессе разработки были добавлены новые транслируемые строки, то они распространяются не только на исходный файл, сохраняемый из приложения при вызове Save_untranslated, но и на все уже переведённые языковые файлы. После чего достаточно отредактировать их, переводя только новые строки.

VII. 2. Локализация ключевых слов языка

AL-IV поддерживает возможность трансляции ключевых слов самого языка с использованием любого другого письменного языка. При этом частично используется техника локализации строк (см. выше).

Все ключевые слова канонической (английской) версии языка размещаются в текстовом файле Default_.lng в директории с исходными текстами компилятора. Достаточно скопировать этот файл и переименовать, например, в Klingon_KL.lng, после чего заменить строки на свои, и становится возможно использовать ключевые слова на соответствующем языке. Ключевые слова в языковом файле имеют имена, начинающиеся буквой 'K', и в основном сосредоточены в секции [{Translation_to_canonical_keyword}].

Для использования в классе национальных ключевых слов, класс должен начинаться со спецификации языка вида ['KL'] или [Language= 'KL'], после которой первое же ключевое слово должно быть записано уже на указанном языке. Например:

```
['RU'] КЛАСС {Привет_мир}, НЕТЕСТИРОВАН:

ФУНКЦИЯ Main|_ главная : << "Привет, мир!" >> .

КОНЕЦ
```

В связи с тем, что язык перевода может содержать ряд морфологических особенностей (падежи, спряжения, связки, предлоги и т.д.), специально для перевода ключевых слов разработаны правила, позволяющие более гибко сопоставлять более одного национального варианта для каждого ключевого слова, и даже использовать два отделенных пробелами идентификатора вместо одного ключевого слова.

Для этого в качестве перевода может использоваться более одного словосочетания, при этом словосочетания разделяются запятыми. Внутри словосочетания (которое может быть одним словом) вертикальный разделитель разделяет несколько возможных окончаний, которые могут продолжить основной корень. Например,
Kthis=ЭТОТ| ОБЪЕКТ,ЭТОМУ|_ОБЪЕКТУ| ОБЪЕКТУ, ЭТОГО_ОБ|ЪЕКТА, ЭТОГО ОБЪЕКТА
- позволяет использовать вместо THIS слова и словосочетания:

- ЭТОТ
- ЭТОТ ОБЪЕКТ
- ЭТОМУ
- ЭТОМУ_ОБЪЕКТУ
- ЭТОМУ ОБЪЕКТУ
- ЭТОГО_ОБ
- ЭТОГО_ОБЪЕКТА
- ЭТОГО ОБЪЕКТА

Примечание: в языке AL-IV ключевое слово **THIS** бывает необходимо в основном в двух случаях:

- когда сам объект должен быть передан в качестве параметра,
- или когда создаётся новый объект, и объект **THIS** указывается в качестве его владельца:
  x={Тип}(...), ПРИНАДЛЕЖАЩИЙ ЭТОМУ ОБЪЕКТУ - прямой перевод
  x={Type}(...), OWNED BY THIS

Вместе с ключевыми словами локализуются кодированные символы (**#NL**, **#TAB** и т.п.), а так же встроенные псевдо-функции (.Len, .Str, .Index и др.) Но в языке имеется так же возможность упростить перевод любых ранее изготовленных классов на национальные языки. Для этого для транслируемого класса создаётся класс-зеркало на соответствующем языке, с единственным модификатором в заголовке TRANSLATION OF {Class_name}. Он должен содержать только переводы наименований для полей, функций (и их параметров), перечислений, структур, констант. Переименования помещаются в соответствующие секции. Например:

```
['RU']
КЛАСС {Текст_файл}, ПЕРЕВОД {Text_file}:

ПОЛЯ:
   Path=Путь|_к_текстовому_файлу
   Encoding=Кодировка
.

МЕТОДЫ:
   Load=Загруз|ить(Строки|_массив[])
   Save=Сохран|ить(Строки|_массив[])
.

ФУНКЦИИ:
   Text_load=Текст_загруз|ить(
      Путь|_к_текстовому_файлу, Строки|_массив[])
   Text_save=Текст_сохран|ить(
      Путь|_к_текстовому_файлу, Строки|_массив[])
.

КОНЕЦ
```

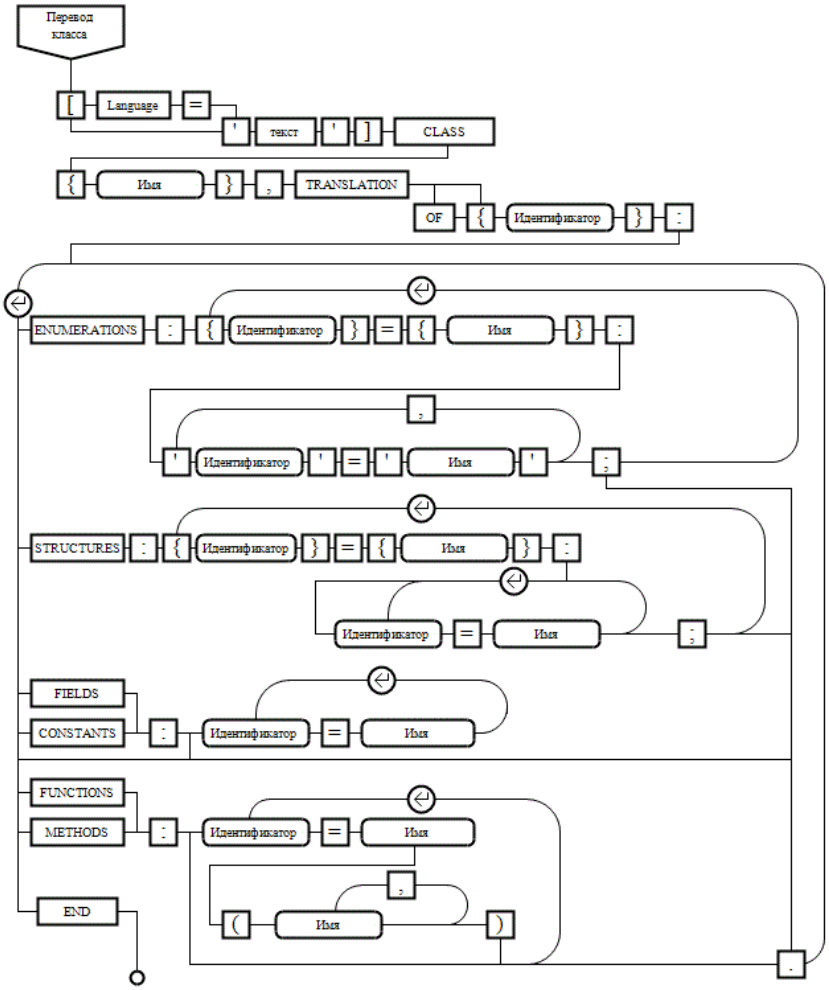
Теперь достаточно включить такой класс в список импорта, и использовать переведённые наименования его функций, методов, полей и т.д. - с использованием национальных букв/ иероглифов/ рун/ литер/ клиньев/ каракулей/ узелков и т.п.

Примечание для русскоязычных читателей. Возможность по русификации (или точнее, произвольной локализации) предоставляется как механизм, позволяющий развивать навыки программирования в любом возрасте, без необходимости начинать с изучения английского языка. Несмотря на довольно большой список слов в словаре, основа языка АЛФОР требует значительно меньше важных ключевых слов для того, чтобы можно было начать писать свой код или понимать уже написанный. Например, класс для вычисления вещественных корней квадратного уравнения:

```
['RU'] КЛАСС {Квадратное_уравнение}, НЕТЕСТИРОВАН:
ИМПОРТ: {Математика}.
ВЕЩЕСТВЕННОЕ А\_коэффициент_при_х_в_квадрате_не_должен_быть_0
ВЕЩЕСТВЕННОЕ В\_коэффициент_при_Х
ВЕЩЕСТВЕННОЕ С\_коэффициент_свободный

МЕТОД Решение\_квадратного_уравнения(ВЕЩ Ответ\ы_в_массиве [])
      :
      Ответ []. Стереть
      --------- 'А==0 - неверное условие '
      ЕСЛИ Абс(А) < 0.000_000_001 ? ==> ;
      ВЕЩЕСТВЕННЫЙ д\искриминант=В * В - 4 * А * С
      ЕСЛИ ?
      [д==0]? Ответ[] << -В/ (2 * А)
      [д > 0]? ВЕЩЕСТВЕННЫЙ к\орень_из_д = д.Корень
                Ответ[] << (-В - к) / (2 * А)
                 Ответ[] << (-В + к) / (2 * А) ; .
```

КОНЕЦ

Формат класса для перевода на другой язык, в форме диаграммы:



VII. 3. STORE - скрытые параметры

Модификатор **STORE** для метода создаёт скрытый (для вызывающей стороны) целочисленный параметр, значение которого сохраняется на стороне вызывающего объекта.

В модификаторе задаётся имя параметра, как его видит функция, и может быть задано первоначальное значение:

```
, STORE(Имя=значение)
```

Для каждого вызова метода в вызывающем классе создаётся скрытое целочисленное поле для параметра, и это поле передаётся в метод по ссылке неявным образом. Передача по ссылке означает, что если метод изменяет значение **STORE**-параметра, то по завершении метода поле получает это новое значение, вычисленное в методе. И при следующем вызове этого метода в этой же строке кода, уже это новое значение будет передано в качестве дополнительного параметра.

Для того, чтобы "забыть" сохранённые значения параметров, может использоваться вызов метода, имеющего модификатор **FORGET**. При этом будут сброшены в начальное состояние (указанное в модификаторе **STORE**) значения всех полей, которые создавались для передачи в скрытые параметры методов того же класса, что и метод с модификатором **FORGET**.

Сочетание модификаторов **STORE** и **FORGET** позволяет значительно ускорить доступ к массивам данных, индексируемым константными строками. Например, при получении значений полей записей в выборке **SELECT** из базы данных, или при доступе к колонкам элемента в listview по их заголовкам.

Пример реализации доступа к колонкам listview по именам:

```
FUN Cells|_by_constant_names(
     INT  Row|_index,
     STR  Name|_of_cell)  ==>   STR , STORE(I|ndex_of_name = -1),
                                RESTRICT  Name IS  CONST
```

```
    :
CASE  I < 0 ? I = Column_index(Name)  ;
RESULT  = Subitems(Row, I)  .
```

В данном примере, модификатор **FORGET** должны иметь все методы, меняющие состав колонок и их названия. Преимущество в быстродействии варианта с использованием STORE в противопоставлении обычному способу может отличаться на порядки, так как поиск индекса по имени происходит только при первом обращении к функции.

VII. 4. Отменённые и устаревшие классы, структуры, перечисления, поля, функции

По мере развития классов и библиотек классов, авторы могут приходить к необходимости переименовывать какие-либо вещи, какие-то начинают устаревать, и их поддержка начинает дорого обходиться, какие-то замещаются новыми, какие-то просто прекращают существовать.

Чтобы иметь возможность управлять на уровне языка отмиранием ненужных старых возможностей, и обеспечивать плавный переход к использованию в новых версиях продукта только новых возможностей, в язык введены модификаторы **DEPRECATED('текст')** и **ABANDONED('текст')** .

- Эти модификаторы применимы к классам, структурам (**STRUCTURE**), полям классов и структур, перечислениям, функциям.
- Текст должен содержать имя альтернативы (если такая имеется) или хотя бы краткое пояснение, почему возможность отменяется или устаревает. Содержимое текста компилятором, разумеется, не проверяется.
- Модификаторы **DEPRECATED** и **ABANDONED** взаимоисключающие.
- Предлагается сначала для устаревающих понятий добавить модификатор **DEPRECATED**, и сохранять его до тех пор, пока устаревшая (но как-то работающая) старая возможность ещё поддерживается.
- При использовании устаревающей возможности в использующем коде, компилятор будет выдавать предупреждение, содержащее указанный в модификаторе текст.
- Если добавлен модификатор **ABANDONED**, соответствующий элемент уже нельзя использовать: компилятор будет сообщать об ошибке. При этом для функций можно убрать тело, для класса - тело всех функций.
- Смысл сохранять память об удалённой декларации заключается в том, чтобы предоставить использующему класс программисту информацию об альтернативе (сообщение 'текст' может содержать имя альтернативной функции, класса, или несколько таких имён, может быть, ссылку на Web- ресурс и т.п.).

Второе применение модификаторов **DEPRECATED** и **ABANDONED** - это модификация функциональности в классе-наследнике путем отказа от использования части методов, полей. Или по той причине, что метод требует совершенно иного набора параметра, или его работа существенно отличается от работы аналогичного метода предка.

Например, при создании класса **{Dialog}** как наследника класса **{Form}**, метод **Show** декларирован как **ABANDONED('Show_modal')**. Это сделано для того, чтобы при использовании форм класса **{Dialog}** программист получал от компилятора сообщение об ошибке при попытке вызова метода Show.

К сожалению, сообщение об ошибке от компилятора не будет получено в том случае, если программист присвоил свою форму класса **{Dialog}** переменной класса **{Form}**. В этом случае будет вызван отмененный метод (и если он не содержит код, этот вызов будет проигнорирован).

VII. 5. Ограничения на значения параметров

Функция может иметь модификаторы **RESTRICT** (безусловное ограничение на параметр) и **IF/THEN** (условные ограничения на значения одних параметров в зависимости от значений других параметров). При этом:

- Все такие ограничения на параметры записываются после всех прочих модификаторов функции.
- Количество ограничивающих модификаторов не ограничено.
- Если устанавливается ограничение на параметр, или параметр участвует в части **IF** условного ограничения, то в качестве значения параметра можно передать только константу. Исключением является случай условия **IF параметр IS CONSTANT, THEN...**
- Ограничения на параметры контролируются компилятором на этапе компиляции кода. Если условия, указанные в ограничениях, не выполняются, компилятор выдаёт сообщение об ошибке.
- Параметр может быть ограничен или участвовать в условии ограничения, если он имеет простой тип (**BOOL**, **BYTE**, **INT**, **REAL**, **STR**, или перечисление - **ENUM**), и не является массивом.

Безусловное ограничение имеет форму:
RESTRICT имя_параметра IN [список_значений]

или (для вещественных типов):
RESTRICT имя_параметра IN [N1 TO N2]

или (без указания значений) :
RESTRICT имя_параметра IS CONSTANT

- Второй вариант (с диапазоном) используется для вещественных параметров. Проверка идёт по условию
  N1 <=X <=N2,
  где:
  - X - переданное функции значение,
  - N1 и N2 - границы отрезка [N1 TO N2].
- Третий вариант не определяет проверку значений, но обязывает передавать в функцию только константное значение.

Условное ограничение имеет форму:
IF параметр1 IN [список или диапазон],
THEN параметр2 IN [список или диапазон]

или
любая часть **IN [...]**может заменена на **IS CONSTANT** .

- Оба параметра могут принимать только константы (кроме случая **IF p1 IS CONSTANT**)
- При выполнении условия в части **IF**, обязано выполняться условие в части **THEN**, чтобы компилятор не вызывал ошибку.
- Если условие в части **IF** не выполнено, часть **THEN** просто игнорируется (но параметр всё равно может принимать только константу).

зачем это может понадобиться?

Например, для реализации набора переходников к библиотеке OpenGL. Можно упростить работу, просто собрав переходники к соответствующим функциям, и передавать так же целочисленные константы, но добавить ограничивающие модификаторы, чтобы обеспечить проверку на недопустимые наборы значений параметров. Это намного проще, чем строить системы классов или использовать для "чистоты" перечисления, и обеспечивает тот же уровень защиты (если не лучший). И при этом работают классические примеры (того же Nene), практически без изменений (достаточно убрать символы ';' в конце операторов).

VII. 6. Контроль зацикливания

a. На каждой итерации любого цикла уменьшается внутренний глобальный счётчик. По достижении значения 0 вызывается внутренняя процедура, которая снова устанавливает значение глобального счётчика (обычно это 65535, и может вызывать дополнительные действия.

b. Для визуальных приложений, имеющих графическую оболочку, после достаточно длительного выполнения начатой длинной операции (обычно более 2 секунд), может отображаться текущий прогресс и краткое описание начатой операции и её стадии выполнения. А так же кнопка для принудительной отмены операции.

c. В случае, если программист не предусмотрел начало длинной операции, тем не менее нажатие на кнопку, пункт меню, и т.п., так же считается точкой начало длительной операции "по умолчанию". Для такой операции так же может отображаться "прогресс", и такая операция так же может быть отменена автоматически.

VII. 7. Оптимизация кода. INLINE- вставки

a. Функции могут иметь модификатор **INLINE**, что является указанием для компилятора выполнять для данной функции непосредственную вставку кода вместо генерации вызова этой функции.

b. При включенной оптимизации, компилятор самостоятельно может выполнять автоматическую **INLINE** вставку для достаточно коротких или однократно используемых функций. Опция компилятора **/!autoinline** запрещает автоматические **INLINE**-вставки (но не отменяет **INLINE**-вставки функций, имеющих модификатор **INLINE**).

c. Функции в некоторых случаях не могут быть вставлены, вместо вызова их. Например, запрещена **INLINE**-вставка метода другого класса, или вставка функции (из другого класса), использующей декларации, не импортированные в данном классе. Так же, нельзя вставлять код, содержащий операторы **==>** , **BREAK** и **CONTINUE**.

d. В некоторых случаях, непосредственная вставка кода функций, вместо вызова их обычным способом, может ухудшать производительность. Например, если вставляется несколько функций, содержащих большое число локальных переменных, то суммарное число локальных переменных в функции-рецепиенте может оказаться слишком большим. Так как в AL-IV все переменные должны быть

проинициализированы до начала выполнения функции, то в случае частого вызова этой функции (в цикле) будет получено существенное снижение быстродействия.

VII. 8. Оптимизация кода. UNROLL (раскрутка) для циклов FOR

a. Циклы **FOR**, имеющие в качестве границ диапазон, заданный константами, может быть полностью развернут модификатором **UNROLL** без параметров (количество итераций не должно превышать некоторого предела, заданного компилятором, в текущей версии это 1024 итерации). Пример:
FOR i IN [0 TO 19], UNROLL:...;

b. Любые циклы **FOR** могут быть развернуты с фактором развертывания, заданным в скобках. Фактор должен быть целой константой со значением не меньше 2.

c. При развертывании циклы увеличивается размер кода, но может сокращаться время выполнения. Но в некоторых случаях, производительность может заметно деградировать (в случае C#: возможно, это связано с особенностями работы его оптимизатора кода).

d. Опция компиляции/**!unroll** запрещает все развертывания циклов.

VII. 9. Синтаксический сахар

a. Если имеется несколько подряд присваиваний (или добавлений в строку или в массив) одному и тому же получателю, то получатель в очередных операторах может быть заменен символом.. (две точки). При этом, в случае массива, квадратные скобки вслед за точками опускать нельзя. Например:

```
A[] << item1 << item2 << item3
..[] << item4 << item5
```

Строки с такими "продолжениями" присваиваний не учитываются при подсчете операторов, и количество продолжений не ограничено. Между полным присваиванием/добавлениям и продолжениями могут находиться другие операторы, не являющиеся присваиваниями/добавлениями (но дополнительные присваивания тому же получателю должны находиться в пределах одной функции). Например:

```
A[] << item1
CASE condition? ..[] << item2;
```

b. Если целью был объект, то дальнейшие обращения к его полям могут заменяться тремя точками:

```
B|utton_ok=New_button("OK", "OK")
...Set_width(50)
```

c. Имеется возможность в случае присваивания полю, указанию которого предшествует цепочка разыменований (например, A.B.C(params).D[index].E), также указать, какое поле будет считаться целеприемником при последующих продолженных присваиваниях/добавлениях. Для этого вместо символа точки в соответствующей позиции (после интересующего поля, заменяемого далее удвоенной точкой) употребляется троеточие. Например:

```
A...B.C[] << item1
...Calculate(param) // здесь выполняется A.Calculate(param)
```

d. Если в пределах одного выражения встречается несколько подвыражений вида A.B[index].C(params).D, и далее цепочка повторяется, и только финальное поле отличается, то второе и последующее подвыражения могут быть заменены на .E, .F и т.п. При этом лидирующие знаки -, !, ~ не относятся к выражению, и их не следует опускать. Например:

```
P=pt1.Offset(-PBox.Bounds.Loc.X, -.Y)
```

e. В операторах **DEBUG** допускается опускать двоеточие вслед за ключевым словом **DEBUG**, а так же опускать символы **<<**, если первым идет оператор вывода в консоль, и первым операндом является строковый литерал:

```
DEBUG "test" ;
```

f. В операторах вывода в консоль (**<<**) автоматически выполняется преобразование конкатенируемых операндов в строки, если компилятор успешно обнаруживает соответствующие функции преобразования в строки **S|tr|ing** (это практически всегда верно для целочисленных и вещественных значений;

g. Функция, возвращаемый результат которой определяется единственным выражением, может записываться в краткой форме:

```
FUN name|_of_fun(...) ==> {тип} :=выражение.
```

(Пробел между ':' и '=' не обязателен, допускается ':=') .
Но если такая функция возвращает новый объект класса, унаследованного от класса результата, то требуется полная запись **RESULT={Класс}(инициализаторы)**

VII. 10. Краткая справка по "встроенным" функциям

Причина, по которой существуют такие функции: невозможность для них в рамках языка строго описать параметры/ результаты. Например, на уровне синтаксиса языка запрещено задавать параметр как "массив из элементов любого типа" . В то же время, удобно иметь набор одинаково поименованных функций для работы с любыми массивами.

Примечание. *С введением "перегрузки" функций такая причина часто может быть нивелирована, хотя по-прежнему есть проблема с описанием параметров вида "структура любого типа" или "любое перечисление": AL-IV не предусматривает таких обобщений на уровне языка.*

Поскольку все такие функции являются статическими, они могут вызываться как в классическом варианте foo(x), так и в префиксном x.foo.

Функция	Расшифровка	Группа
Index(Variable) ==> INT	Индекс текущего элемента в цикле, когда переменная пробегает массив или часть массива	циклы
Name({enum} Item) ==> STR	Имя элемента перечисления Например, "'RED'"	перечисления
First({enum} Item) ==> {enum}	Первый элемент перечисления	
Last({enum} Item) ==> {enum}	Последний элемент перечисления	
Prev|ious({enum} Item) ==> {enum}	Предыдущий элемент перечисления	
Next({enum} Item) ==> {enum}	Следующий элемент перечисления	
Int({enum} Item) ==> INT	Преобразовать в целое (индекс элемента в перечислении начиная с 0)	
Int(STR S) ==> INT	Целочисленное значение из строки	строки
Real(STR S) ==> REAL	Вещественное значение из строки	
Len(STR S) ==> INT	Длина строки	
Find(STR S, STR W) ==> INT	Поиск индекса первой подстроки W в строке S (0... Len-1 или -1, если такой подстроки нет или W=="")	
Str({INT,REAL,BOOL}) ==> STR или S ({INT,REAL,BOOL}) ==> STR	Преобразование в строку целого, вещественного, булевского значения	
Count(A[]) ==> INT	Количество элементов в массиве	массивы
Allocate(A[], INT Size)	Выделение необходимого количества элементов в массиве, чтобы размер был в итоге равен Size (если размер уже превышает Size, он не изменяется)	
Find(A[], W) ==> INT	Поиск индекса первого элемента в массиве (кроме массива структур). Результат 0... Count-1 или -1, если не найден.	
Insert(A[], INT I, V)	Вставка элемента V в позицию I в массиве A[]	
Clear(A[])	Чистка динамического массива	
Delete(A[], INT I)	Удаление элемента с индексом I	
Remove(A[], V)	Удаление всех элементов V	
Swap(A[], INT I, INT J)	Обмен элементов с индексами I и J	
Int(REAL X) ==> INT	Отбрасывание дробной части (Truncate)	вещественные
ShiftL(INT N, INT K) ==> INT	Логический сдвиг N на K бит влево (если K<0, выполняется сдвиг на -K бит вправо). Выдвигаемые биты теряются, вдвигаются нули.	побитовые операции сдвига
ShiftR(INT N, INT K) ==> INT	Логический сдвиг вправо.	
RotateL(INT N, INT M) ==> INT	Циклический сдвиг влево на 1 бит младших M бит. Выдвигаемые биты вдвигаются справа.	
RotateR(INT N, INT M) ==> INT	Циклический сдвиг вправо	
Clone({structure} structure) ==> {structure}	Создание копии структуры	структуры
Dismiss({structure} structure) ==> {structure}	Отвязывание структуры от места хранения (перемещение структуры)	

Все встроенные функции могут идентифицироваться полностью капитализированными именами, например: A[].COUNT

VII. 11. Обобщенные функции

Обобщенные (или иначе - перегруженные) функции - это функции с переменным набором типов данных параметров (и, возможно, результата).

Тип данных параметра (и, возможно, результата) может быть заменен списком типов параметров в фигурных скобках. Например:

```
FUNCTION  Min|imum_of_two_values(
      { INT, REAL, STR}  A|_operand,
      { INT, REAL, STR}  B|_operand )    ==>    {INT, REAL, STR}
      :
      CASE  A < B ?  RESULT = A  ==>   ;
      RESULT = B  .
```

Правила:

1. Только статические функции могут быть обобщенными.

2. Только результат не может иметь вариантный тип - как минимум один параметр должен быть вариантного типа.

3. Все параметры (и если результат вариантного типа, то и результат) должны иметь одинаковое число вариантов. Порядок вариантов важен, и все типы с одинаковым индексом образуют один набор параметров (и результата, если результат вариантного типа).

4. Первый параметр с вариантным набором типов должен иметь все различные типы в списке. С каким именно набором параметром обобщенная функция вызывается, определяет тип первого параметра с вариантным типом.

В теле обобщенных функций могут использоваться вариантные операторы **CASE ?**, ветви которых определяются не условиями, а типами входных параметров (точнее, первого параметра с вариантным набором типов):

```
FUNCTION  S|tr|ing_from_rect_or_point(
{STR,{rect},{point}}  V|alue_to_convert)  ==>   STR
    :
     CASE  ?
     {{ 1/STR }}:  RESULT = V
     {{ 2/{rect} }} :  RESULT = S_rect(V)
     {{ 3/{point} }}:  RESULT = S_point(V)  ;   .
```

Список вариантов, задающих ветвь такого условного оператора, состоит из элементов вида N/ тип, где N - это порядковый номер варианта типа в списке вариантов, начиная с 1 (а тип - соответствующий тип первого вариантного параметра).

Так же, как и в случае перечислений, ветвь **ELSE** использоваться не может, и все варианты типа параметра должны быть использованы.

В отличие от обычного кода, при компиляции каждого варианта "обобщенной" функции, все ветви, не соответствующие текущей компилируемой сигнатуре, просто пропускаются. Поэтому декларации переменных в таких ветвях недоступны другим ветвям.

При использовании таких перегруженных функций, отыскиваются функции по именам и сигнатурам "обобщенных" параметров.

VII. 12. Методы для индексации.[] - организация многомерных массивов

В классе может быть декларирована единственный метод, вместо имени которого указывается точка, а параметры заключены в квадратные скобки:.[] - метод. Обращение к такому методу выглядит для объекта класса X почти как к массиву:
X.[индексы]

Кроме этих особенностей, если для такой функции декларируется "сеттер"-метод, то такая функция может использоваться в левой части оператора присваивания:
X.[индексы] = значение

С помощью методов индексации организуются многомерные массивы, например, как в классе **{Matrix|_of_REAL}**:

```
--------------------------------  'access Items[] via .[i, j] method'

METHOD .[ INT  I|ndex_from0,  INT J|ndex_from0]  ==>  REAL
    :
     CASE  J < 0 || J >= NColumns ?  ==>   ;
     INT  i_j INDEXING  REAL  = I * NColumns + J
     RESULT  = Items[i_j]  .

METHOD  set_items(
INT I|ndex_of_row_from0,  INT  J|ndex_of_column_from0,  REAL Value|_to_set),
SETTER  FOR .[]
    :
     CASE  J < 0 || J >= NColumns ?  ==>   ;
```

```
  INT  i_j INDEXING  REAL  = I * NColumns + J
Items[i_j] = Value  .
```

VII. 13. Операторы

В главе о синтаксисе декларации функци й кратко описывался синтаксис декларации операторов. Здесь информация об операторах и их применении в коде собрана вместе.

Оператор - это статическая функция, которая предназначена для использования в выражениях вместо встроенных операций +, -, *, /. Ее декларация отличается от декларации обычной функции:
OPERATOR {тип1} операция {тип2}==> {тип3}:....
Например:
OPERATOR {complex} + REAL==> {complex}:....

Кроме операторов с двумя операндами, может так же декларироваться оператор для операции '-' с единственным операндом:
OPERATOR - {тип}==> {тип}:...;

Как минимум, один из типов параметров должен отличаться от базовых типов **BOOL**, **BYTE**, **INT**, **REAL**, **STR**. Это может быть структура, класс, или перечисление.

Чтобы операторы могли использоваться в функции, она должна иметь модификатор **OPERATORS({Имя_класса})** , где {Имя_класса} - задает класс, в котором отыскиваются операторы. Если таких классов несколько, они перечисляются через запятую:
OPERATORS({Класс1}, {Класс2}, ...) .

Имена параметров для оператора в декларации отсутствуют (указывается только тип), но подразумевается, что параметров два, и первый имеет имя **A**, и второй **B** . Именно так на них и следует ссылаться в коде тела оператора. В случае одно-операндного оператора для операции '-', ее единственный параметр называется **A** . Возвращаемое значение всегда имеет имя **RESULT**, как для любой функции.

Класс, содержащий операторы, должен содержать модификатор **OPERATORS**.

Пример кода оператора:

```
OPERATOR  {Matrix}  *  REAL   ==>   {Matrix}, NEW
 :
  RESULT  = New_matrix(A.NRows, A.NColumns)
  FOR  i  IN  [0  TO  RESULT .NRows-1] :
      FOR j  IN  [0  TO  RESULT.NColumns-1] :
          RESULT .[i, j] = B * A.[i, j]  ;
  ; .
```

VII. 14. Многопоточность

Многопоточность в AL-IV реализуется классом **{Thread}**, но при этом обеспечивает полную изоляцию данных одного потока от данных других потоков. Любые объекты, передаваемые потокам на обработку, "исчезают" из адресного пространства передающего потока, и не его стороне все ссылки на этот объект (как сильные, так и слабые) теперь ссылаются на NONE- объект. До тех пор, пока объект не будет возвращен передавшему потоку. Собственно, это один из вариантов ожидания работы потока: периодически проверять наличие объекта, который был передан этому потоку на выполнение.
Потоки исполнения всегда образуют иерархию: если поток A запускает поток B, то B становится подчиненным по отношению к A. Если поток A завершается, то поток B может продолжать работу, но коммуницировать с запустившим потоком он уже не сможет. Тем не менее, операция **yield** работать будет, хотя "возвращенные" в исходный поток объекты будут просто уничтожены.
Сам объект потока, когда он запускается, на самом деле тоже исчезает из адресного пространства запустившего потока как объект. Но на стороне запустившего потока формируется "фантомный" объект **{Thread}** (контроллер), на который перенаправляются ссылки с прежнего объекта. У контроллера хозяйский поток может получать некоторую информацию о состоянии запущенного потока (**Running**, **Stopping**, **Terminating**), передавать объекты в очередь на обработку (**Take**), отдавать команды (**Stop**, **Wait**).
Для реализации некоторого процесса, выполняющегося в отдельном потоке, следует унаследовать свой класс от **{Thread}**, переопределить в нем метод **execute**. При этом, если поток берет на обработку объекты, то он должен использовать методы **receive** (для получения объектов) и yield (для отправки готовых объектов). Метод **yield** может отправлять в поток-хозяин и новые объекты, а не только те, что были получены через **receive**.

Иногда желательно получать дополнительно информацию о ходе выполнения задачи, переданной потоку. Например, для индикации процентов исполнения. Помимо штатной возможности регулярно создавать на стороне запущенного потока объекты, и передавать их контролирующему потоку (**yield**), можно использовать класс **{Global_var}**. Он зависит от платформы, но позволяет чрезвычайно просто

создавать и изменять именованные глобальные целочисленные переменные, к которым любой поток может обратиться в любой момент без особых усилий и блокировок.

VII. 15. Нативные (низкоуровневые) функции

Нативные функции могут быть только статическими (не могут быть методами). Они должны иметь модификатор **NATIVE** . Для них в качестве параметров и возвращаемого результата не допускаются структуры, хотя объекты классов разрешены.

Есть два основных варианта нативных функций: содержащие только нативный код, и имеющие обычный код AL-IV, и завершающий нативный код (что позволяет подготовить часть параметров в локальных переменных, или проверить некоторые условия, и предотвратить выполнение нативного кода, если необходимо) .

В первом случае после двоеточия сразу располагается либо строковая константа-литерал (может быть многострочной, с префиксным символом '@', который автоматически добавляет символы завершения строки в константу после каждой строки).

```
FUNCTION Writeln_number|_only_positive(INT N|umber_to_writeln), NATIVE: @
"if X_N < 0 then Exit;"
"writeln(IntToStr(X_N));" .
```

В качестве константы может быть указана так же именованная строчная константа, но тогда ей должно предшествовать ключевое слово **NATIVE**:

```
FUNCTION Some_native_fun, NATIVE: NATIVE NATIVE_string_constant.
```

Во втором случае, сначала идет код AL-IV:

```
FUNCTION Writeln_number|_only_positive(INT N|umber_to_writeln), NATIVE:
    CASE N < 0?==> ;
    STR s|tring_to_writeln=N.S
    NATIVE @
    "writeln(X_s);" .
```

В обеих реализациях приведенной выше функции Writeln_number, имеется проверка на неотрицательность параметра. Т.е., оба приведенных примера эквивалентны.

Содержимое строчной константы, указанной в качестве тела нативной функции, практически без изменений вставляется в код результирующей функции, и это должен быть код на целевом языке - том, в который выполняется компиляция проекта. Но могут быть особенности, в зависимости от языка.

Например, для языка Паскаль первые строки, начинающиеся с 'var', рассматриваются как декларации локальных переменных, и вставляются до слова begin, начинающего результирующую нативную функцию. (На самом деле, если первая строка начинается с var, то в секцию декларации переменных до **begin** попадают все строки, до последней строки, начинающейся с var. В том числе, все промежуточные строки, не начинающиеся с var - это позволяет в секции декларации использовать, в том числе, директивы условной компиляции).

Во всех случаях, для нативных функций, возвращающих результат, переменная **RESULT** в точке входа в нативный код, уже проинициализирована **NONE**-значением (для чисел - нулем, для строк - пустой строкой) . Поэтому возврат из нативной функции без присваивания в переданном в строке коде значения переменной **RESULT** возвратит это значение.

Для доступа к своим параметрам и локальным переменным нативная функция в теле должна добавлять префикс 'X_' к их именам. Для работы с объектами и их полями и методами, надо точно знать во что превращается код после работы компилятора AL-IV - для конкретного целевого языка. Обычно, поля получают префикс 'F_', методы и функции 'M_'. Следует учитывать, что не все языки программирования учитывают регистр букв в именах, и в Паскале имена a и A тождественны.

Можно ознакомиться с примерами написания нативного кода для C#, Pascal, Java - в библиотеке функций.

Класс, содержащий нативные функции, должен иметь модификатор **NATIVE**. Следует избегать написания и использования таких функций, кроме крайних случаев. Например, такими случаями могут быть: острая необходимость оптимизации скорости выполнения некоторого участка кода, или необходимость использования системной функции, доступной только из нативного кода.

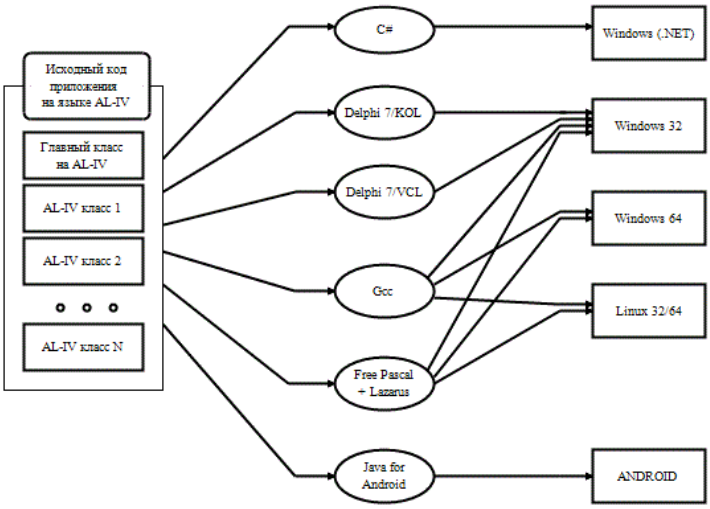
VIII. Зачем нужен еще один язык?

VIII. 1. Настоящая многоплатформенность

Да, существует возможность создавать приложения, которые действительно будут многоплатформенными. Для этого нужно выбрать язык (C++, C#, Java, Pascal) и использовать какой-нибудь фреймворк (библиотека классов). На этом пути вас ждет масса приключений и разочарований.

Приключений - потому что авторы языков и библиотек, которые вы выбрали, очень часто имеют свое представление о том, что важно, что не нужно вам. Идут путями, о существовании которых вам раньше даже не приходилось задумываться. Будьте готовы к тому, что какую-нибудь простенькую проблемку вы будете решать неделями, перекапывая весь Интернет, роясь в тоннах чужих исходных кодов, и перечитывая сотни (ладно, десятки) страниц различных форумов.
Разочарований, потому что иногда придется пересматривать свои потребности, исходя из ограничений выбранного вами инструмента. Или прекращать использовать выбранную связку, и искать другую, переписывая на новый лад все ранее написанное.

В чем же фундаментальное отличие AL-IV в плане многоплатформенности, если это всего лишь надстройка над одним из существующих языков (над C #, Delphi, Free Pascal или Java) ? Особенность AL-IV в том, что он изначально не ориентирован ни на какую платформу. Вы пишите код только один раз. Для запуска программы на другой платформе достаточно поправить конфигурационные файлы и вызвать компилятор.
Требуется лишь обеспечить поддержку необходимой платформы. К счастью, платформ (операционных систем) не так много. Существует вероятность, что со временем, у нас будет поддержка всех основных платформ.
На данный момент (июнь 2019) дерево поддерживаемых платформ выглядит примерно так:



Что, если какая-либо из поддерживаемых ветвей отпадет (например, перестанет поддерживаться желаемая целевая платформа, или средство разработки неожиданно подорожает) ?
На этот вопрос ответ есть. Он заключается в простоте языка AL-IV. Компилятор для него (компилирующий в промежуточный целевой язык) изготавливается за сравнительно короткое время. Всегда можно вернуться к генерации кода на языках C++/Java/Python/... или сделать новый генератор. Задача построения набора переходных нативных классов, реализующих функциональность базовых библиотек, может оказаться сложнее (но и этот вопрос решаем), т.к. базовая библиотека изначально невелика и спроектирована так, чтобы по возможности упростить такую работу.

VIII. 2. Надежность

Практически все современные языки программирования выросли из древних примитивных "ассемблеров", допускающих небезопасные операции, адресную арифметику. Нет на данный момент языков высокого уровня, в которых было бы невозможно обратиться к несуществующему объекту, получить во время работы исключение в результате ошибочного доступа по нулевому адресу, а в некоторых тяжелых случаях напороться и на порчу произвольной памяти.
Вы можете использовать т.н. "управляемую" (managed) память в C++ (в C # почти все объекты хранятся в такой памяти), или использовать none- объекты в objective-c. Но часть кода все равно будет написана в старом небезопасном стиле, и изменить эту часть вам, скорее всего, не удастся (даже если это ваш собственный код). Даже в случае современных C #/ Java вы не будете избавлены от необходимости обеспечивать в своем коде проверки на равенство указателя значению null или предусматривать обработку исключений.

В случае AL-IV ситуация отличается кардинально. На уровне компилятора (и самого языка) контролируется выход за пределы массивов (возвращая фиктивный NONE элемент или предотвращая операцию записи), обращение к неприсвоенным объектам, гарантируется инициализация всех переменных, предотвращается зацикливание и бесконечная рекурсия. Имеются встроенные средства тестирования кода, с контролем степени "покрытия" кода тестами. Программирование с AL-IV становится спокойным и монотонным занятием, без экстрима и авантюризма.

Исключения в коде AL-IV невозможны. При разработке нативных методов/ функций рекомендуется следовать этой парадигме, обеспечивая в случае отказа в финальном

коде безболезненную обработку ошибок в стиле пост-обработки списка ранее произошедших ошибок, и только с целью выяснить, успешно ли была выполнена последняя группа операций. В нормальных ситуациях достаточно сообщить о возникших проблемах (вывести их в лог, отобразить на экране и т.п.)

V III. 3. Зачем нужен новый синтаксис?

В действительности, всегда существует возможность остаться в рамках существующего синтаксиса, но при этом изменить семантику.

При разработке нового синтаксиса основной целью было упростить работу с исходным кодом в произвольном текстовом редакторе (с поддержкой UTF-8 - это единственное требование к такому текстовому редактору) .

Именно отсюда - требование на запись ключевых слов только В ВЕРХНЕМ РЕГИСТРЕ. В этом случае исходный код проще читать (и ненамного сложнее редактировать).

Что касается отсутствия начальной скобки блочного оператора, и специального символа завершения каждого оператора. Их отсутствие делает текст чище, и упрощает модификацию текста (такие операции, как рефакторинг). И при этом, не затрудняет чтение.
А для целей перевода ключевых слов на национальные языки - еще и сокращает количество слов, требующих перевода. Как минимум, исключает слово END - КОНЕЦ из словаря.

Об ограничениях на количество вложенных операторов, на количество операторов между блочными комментариями. Эти ограничения мало влияют на возможности программиста. Слишком много уровней вложенности существенно затрудняют понимание кода. При использовании нескольких отступов, когда уровень вложенности становится слишком глубок, ширина строки оказывается слишком узкой для текста, и строки кода все чаще приходится делить на части. Поэтому, вынесение глубоко вложенных блоков в отдельные методы/функции - весьма корректное требование.
Разместить блочный оператор для разбиения слишком длинной последовательности операторов - вообще не является чрезвычайно сложным пожеланием.

Остается вопрос только насчет ограничения на три параметра на функцию/метод. Первоначально, при вводе такого ограничения, предполагалось, что если оно окажется слишком сложным для постоянного следования, то будет снято или ослаблено.
Но в процессе разработке довольно сложного и объемного программного кода (компилятора языка, редактора исходного кода, и других приложений) выяснилось, что данное требование более чем легко исполнимо, и не должно представлять трудностей. При этом вполне очевидно, что ограничение на максимальное число параметров существенно упрощает работу с библиотеками классов/функций, так как разработчику не приходится запоминать большие списки параметров.
Ограничение было снято в итоге, и заменено на требование явно указывать имя параметра в форме присваивания, начиная, как минимум, с четвертого параметра. Но это только для того, чтобы еще более упростить возможный рефакторинг кода путем вынесения кода из вложенных блоков в отдельные функции, или для адаптации существующего кода, ранее написанного на других языках программирования.

VIII. 4. Где цикл "while" ?

Циклы вида while/ repeat... until небезопасны, так как могут приводить к зацикливанию программного кода без какого-то шанса завершить этот цикл.
Циклы типа FOR i IN [...] безопасны в этом плане, т.к. рано или поздно завершаются (возможно, очень много придется ждать, но не вечность).

При программировании в Алфор следует для каждого цикла, для которого в другом языке был бы использован while, использовать оператор FOR, задавая в качестве диапазона, например, [0 TO N], где N - верхнее допустимое число итераций. И первым оператором внутри цикла ставить выход по анти-условию продолжения цикла:

```
CASE !continue_cond ? BREAK i;
```

Такой подход гарантирует прекращение цикла хотя бы по условию исчерпания заданного диапазона.

VIII. 5. Зачем нужен новый способ управления памятью?

С тех пор, как была изобретена куча динамических данных, на самом деле, в плане управления памятью было сделано только одно нововведение: автоматическое уничтожение объектов при достижении счетчиком использования значения 0.
Правда, при этом выяснилось, что могут образоваться взаимные ссылки объектов друг на друга (а в более сложных случаях - кольцевые замкнутые маршруты взаимного использования), в результате чего автоматически удалить объект не получается. Для устранения возникшей проблемы была придумана т.н. "чистка мусора". К сожалению, данная процедура переводит абсолютно все системы, использующие эту методику для освобождения циклически связанных объектов, в разряд медленных и непредсказуемых. А это значит, что их становится нельзя использовать в системах реального времени. Или даже нежелательно использовать в системах массового обслуживания.

Да, всегда есть возможность для критичных подсистем отказаться от управляемой памяти, и вести разработку в "старом стиле", напрямую управляя выделением памяти.

Но при этом исчезают преимущества автоматического освобождения памяти. И это, на самом деле сложный путь, так как давно уже программы не пишутся с нуля. Программисты постоянно используют уже существующие библиотеки функций, классов. Если нет возможности использовать управляемую память, то нет и возможности использовать классы/функции, использующие управляемую память. Возможности программиста резко сокращаются, разработка замедляется.

Совсем другое дело, если у вас есть возможность работать с автоматически освобождаемыми объектами, но при этом нет необходимости отказываться от разработки приложений реального времени. Чистка мусора не нужна, объекты освобождаются немедленно в тот момент, когда это требуется - это ли не мечта программиста?

Конечно, в случае нового способа приходится несколько менять стратегию создания объектов. Как минимум, приходится думать о времени жизни объекта в точке программного кода, где он создается. И, конечно же, появляется необходимость держать массивы указателей на дочерние объекты. Но это небольшая потеря по сравнению с приобретением возможности полностью отказаться от чистки мусора (и от самого мусора в динамической куче).

VIII. 6. Зачем встроенная поддержка операторов SQL?

Действительно, как сочетается стремление сделать язык максимально простым и встраивание прямой поддержки SQL- выражений?
Ответ: такая поддержка позволяет проверять значительную часть семантики SQL- запросов на этапе компиляции программы.
А именно: соответствия имен полей действительным именам полей (объявленным в декларации таблиц TABLE), корректность их использования (учет того, что поле может быть null, или является автоинкрементным и, соответственно, ему нельзя присвоить значение в утверждении UPDATE, например). И проконтролировать собственно синтаксис SQL -запросов. Эти проверки выполняются на этапе компиляции, позволяя уменьшить вероятность того, что программа, предназначенная для работы с БД, будет запущена с явными ошибками в своем коде.

Этот принцип - возможность статического анализа корректности операций на этапе компиляции - значительно упрощает разработку. К сожалению, в отношении SQL в настоящее время этот принцип обычно не действует, хотя работа с БД является одной из ключевых в современной практике программирования. В AL-IV сделана попытка исправить эту тенденцию.

В отличие от обобщенного программирования, встраивание некоторых специализированных возможностей в язык, например, операторов SQL, или комплексных чисел, практически не увеличивает порог вхождения.
При написании кода, вы не используете эти возможности, и можете даже не знать об их существовании. При чтении чужого кода, придется ознакомиться с новой для вас возможностью, если ее использование встретилось в этом коде.
Но ознакомиться со специализированным расширением языка на порядок проще, нежели:

- ознакомиться с правилами написания обобщенных функций/ методов/ классов;
- изучить конкретное описание встретившейся декларации переменной, операций с ее типом, запрограммированным в данном конкретном дженерике, включающих:
  - методы,
  - переопределенные операторы,
  - преобразования типов данных,
  - и часто - с учетом наследования от других обобщенных типов.

VIII. 7. Почему нет возможности управлять разрядностью переменных?

Потому что это делает ваш код проще (и уменьшает число возможных ошибок при одновременном использовании якобы однотипных значений с различной разрядностью).
Во многих современных языках имеется возможность двойственного определения типов переменных: для значений, для которых разрядность не очень важна, используется базовый тип (например, Integer), а в случаях, когда его явно недостаточно, или наоборот, требуется экономия разрядности, применяются типы с уточнением разрядности (Int64, Smallint, Shortint).

В итоге, образуется несметное количество различных типов данных и их сочетаний. В результате в процессе разработки, приходится предусматривать ситуации, когда разрядности одной переменной может не хватать для хранения результата, полученного в результате операций с переменными с другой разрядностью. (*На самом деле, никто ничего не предусматривает, и вместо этого, есть возможность просто получить неправильные результаты, без какого-либо разумного объяснения, что же, собственно, произошло, и без возможности что-либо исправить в коде*).
Зачем все эти тонкие спецификации разрядности нужны, если используются в действительности только для экономии памяти? Проще отказаться от них навсегда,

существенно упростив для программиста выработку решений о том, какие типы данных использовать.

По этой причине в AL-IV нет беззнаковых типов данных (кстати, их нет, например, и в Java). И нельзя задавать разрядность каждой переменной отдельно. Только для всех целочисленных или вещественных переменных сразу, задавая опции компилятора (/int32, /$REAL=EXTENDED/DOUBLE/SINGLE).

VIII. 8. Где тип данных CHAR?

Этот тип данных устарел, и не отражает современных реалий. В случае использования кодировки utf-8, один символ может кодироваться последовательностью байтов, от одного до шести. Т.е., по сути, для хранения одного символа, нужна строка. Что и сделано в AL-IV.

В AL-IV каждый символ строки, в свою очередь, является строкой. Таким образом, извлекая его операцией S[n], мы фактически вызываем функцию S.Substring(n, 1).

VIII. 9. Почему нет указателей на структуры?

Потому что структуры введены в язык прежде всего как средство иерархического управления наборами данных. Они позволяют агрегировать данные (поля), и при этом гарантируют, что структуры всегда хранится только в одном экземпляре, и при выходе из области видимости гарантированно освобождает память.

Структуры - это промежуточный тип данных между простыми типами и классами. С одной стороны, они содержат отдельные поля (и передаются в функции фактически по ссылке), с другой - они присваиваются практически как простые переменные (путем простого копирования), и не могут модифицироваться в функциях, если являются параметрами (т.е. всегда передаются только для чтения, в нотации C++/Java/Pascal языков - как константные параметры - const).

В AL-IV отсутствует возможность работать с указателями на структуры или на их части.

Отсутствие указателей и адресной арифметики существенно повышают надежность кода. Индексирование элементов массива значительно безопаснее, т.к. у компилятора в этом случае есть информация о том, к какому объекту (массиву) производится доступ в коде, и он может обеспечить контроль выхода за его границы, хотя бы динамический (во время выполнения).

VIII. 10. Краткость кода

a. Краткие имена

При написании кода на AL-IV, вы можете использовать краткие имена переменных, и код становится кратким. Как на заре программирования, когда программисты часто использовали сокращенные имена переменных, ограничиваясь одной-двумя-тремя буквами, и вообще не заботились о том, чтобы в будущем самим понять, что они там по-быстрому накалякали. С той разницей, что компилятор потребует предоставить при декларации переменной/функции/типа данных и более длинный вариант имени (не менее 8 символов в сумме), и при желании понять, для чего предназначена переменная, всегда можно найти ее декларацию. В случае использования специализированного IDE, достаточно кликнуть по имени переменной, чтобы ее декларация была продублирована в окне подсказки по текущему символу.

b. Префиксный вызов функций

Вы можете существенно уменьшить количество вложенных скобок в выражениях, заменяя классический вызов функции с параметром в скобках на префиксную форму. Например:

```
s.Replace_all(",", ".").Trim.Remove_ending(".").TrimR.Find_last("_")
```

Сравните:

```
Find_lastTrimR(Remove_ending(Trim(Replace_all(s, ",", ".")), ".")), "_")
```

В первом случае (префиксный вариант) код и короче, и намного понятней (второй вариант синтаксически так же верен для AL-IV, впрочем).

c. Удаление избыточных проверок

Если вы пишете на AL-IV, вам не требуется постоянно проверять на null (в случае AL-IV - на NONE, но в любом случае - не требуется). Компилятор сам выполнит необходимые проверки, и в случае, если объект является NONE, то обращение к его полям/методам не приведет к исключению/падению приложения/синему экрану смерти. Что произойдет? Чаще всего, ничего не произойдет. Будет возвращено значение NONE для методов/функций, возвращающих объект, и для объектов-полей. А для чисел/строк/перечислений будет возвращен 0.

Таким образом, проверка на NULL (в нашем случае - NONE) часто оказывается избыточной, т.к. поля NONE-объекта существуют физически, но содержат NONE-значения. Следовательно, вместо проверки

```
CASE sender.{Paint_table}!=NONE
     && sender.{Paint_table}.Count > 0
     ?...
```

можно выполнить проверку

```
CASE sender.{Paint_table}.Count > 0 ?
```

, что намного читабельнее.

Вам так же не требуется заботиться о делении на ноль, или вычислении операций с операндом NaN. Будет возвращено значение NaN. Падение приложения в этом случае не предусматривается. Если полученное значение вас не устраивает, вы можете найти причину неудачи, и исправить ее. Возможно, речь идет о том самом делении на ноль, или об обращении к математической функции с недопустимыми параметрами. Но это точно не причина падать для всего приложения.

d. Операторы LIKE

Вам не требуется постоянно выполнять рефакторинг при написании кода, только для того, чтобы повторно использовать уже написанный выше кусок кода. То есть, да, его можно оформить как отдельно стоящую функцию, придумать для нее имя, оформить заголовок, придумать, как передавать ей параметры, и все это - чтобы один-два раза использовать повторно. А можно просто ограничить повторно используемый код "скобками"

```
--------------- ' не хочется переписывать ', REUSED
...
--------------- 'end'
```

и далее в коде обратиться к этому фрагменту:

```
LIKE.......... ' не хочется переписывать '
```

Это как бы макрос (что было бы очень плохо - см. на C/C++ с их макросами), но без возможности вложенного вызова других операторов **LIKE**, или выхода за пределы текущего класса.

e. Оформление блочных операторов

В языке AL-IV нет скобок begin/end или {...}. Вам не придется тратить пару лишних строк кода на это оформление вложенных блоков (у вас появится время на то, чтобы придумать, как уложиться в допустимые три уровня вложенности операторов FOR/CASE). Код становится практически таким же кратким, как в Python (но в случае, если у вас собьются ведущие пробелы/табуляции, код не будет испорчен - вложенность блоков определяется завершающим блоки символом ';' .

При переносе на новую строку не используются специальные символы, засоряющие текст - достаточно соблюсти пару простых правил (завершить предыдущую строку запятой, открывающей скобкой '(', '[', или начать строку продолжения с последовательности символов, которые не могут начинать новый оператор. Например, начать с кавычки или знака операции '+', '-', '&&' и т.п.).

f. Декларация локальных переменных

В AL-IV локальная переменная декларируется в том месте, где она впервые нужна (например, получает первое значение). И (это важно, и это действительно отличается от большинства других языков) **действует до конца функции**. Фактически, это означает, что локальные переменные были объявлены как бы в самом начале функции, и проинициализированы значением по умолчанию (нулем, пустой строкой, объектом **NONE**, а в случае динамического массива изначально пусты). Если получится так, что код не вошел в ту ветку, в которой вы декларировали эту переменную, она все равно является декларированной и имеет значение по умолчанию.

Это может выглядеть странно по сравнению с большинством других языков программирования. Но в реальности, не имеет (почти) никаких недостатков, и позволяет дополнительно сократить код. Например:

```
CASE x?
[0]:... some code here
[1]: BOOL res|ult_has_1=TRUE
     ... some code here
[2]:... some code here
[3]:... some code here;
[10]: res=TRUE
```

```
    ... some code here
CASE res?
    << "We have 1 in the number !!!"#NL;
```

(Недостаток, на самом деле, есть: если вы декларируете первый раз накопительную переменную/массив и т.п. - во вложенном цикле, и не очищается ее до повторного входа в этот цикл, то накопление продолжится - но этот косяк будет полностью на вас, извините, хотя и сломает только ваш алгоритм, но не станет ронять всю программу).

g. Автоматическое освобождение объектных переменных

Да, для большинства современных языков это давно не новость. Но есть отличия: программе на языке AL-IV не требуется сборщик мусора. Это, как минимум, позволяет компилировать код для языков/систем, в которых нет автоматического освобождения объектов при обнулении счетчика использования (например, Delphi 32/ Free Pascal). И при этом сохранять свойство автоматического освобождения ресурсов при исчерпании ссылок на них.

Для краткости кода это означает отсутствие дополнительного кода, занимающегося высвобождением переменной по окончании ее использования. (Причем, если в том же C# рекомендуется вызывать Dispose для тех же битмапов или MemoryStream, то для AL-IV этот Dispose вызывается автоматически, при завершении существования соответствующих экземпляров этих классов).

Содержание

В начало