Язык Клаус

справочное руководство

Памяти Никлауса Вирта.
С благодарностью Дмитрию Тарасевичу
и Анне Михеевой.

Алфавит и кодировка

Язык Клаус поддерживает только кодировку UTF-8. Эта кодировка используется для исходных файлов, для хранения строковых данных в оперативной памяти, для работы функций файлового ввода-вывода стандартной библиотеки, а также во всех остальных случаях.

Алфавит языка включает в себя:

- пробельные символы: пробел, табуляция, возврат каретки, перевод строки;
- буквы латиницы: **`A-Z`** и **`a-z`**;
- буквы кириллицы: **`А-Я`** и **`а-я`**;
- арабские цифры: **`0-9`**;
- символы: **`_ + - * \ / % ^ = < > ! & | ~ : ; . , ( ) [ ] { } $ # ` ' " №`**

Строковые и символьные литералы в исходном коде могут содержать любые символы, допустимые в кодировке UTF-8, за исключением символов возврата каретки, перевода строки и нулевого символа.

Язык Клаус не регистрочувствительный — т.е., прописные буквы в ключевых словах и идентификаторах считаются эквивалентными строчным.

Лексика

Лексемы языка Клаус всегда можно отделять друг от друга пробельными символами — пробелами, табуляциями, символами перевода строки и возврата каретки. Перевод строки и возврат каретки равнозначны пробелу и не имеют специального синтаксического значения, с той оговоркой, что:

- перевод строки и возврат каретки недопустимы внутри строковых и символьных литералов;
- перевод строки и возврат каретки считаются окончанием однострочного комментария.

В остальном, разбиение исходного текста на строки остаётся полностью на усмотрение программиста.

В тех случаях, когда Клаус может распознать лексемы однозначно, отделять их друг от друга пробельными символами не требуется.

В языке Клаус есть следующие виды лексических единиц:

- **Ключевое слово** (зарезервированное слово) — слово, состоящее только из букв кириллицы и имеющее специальное значение в синтаксисе языка. Некоторые ключевые слова могут иметь несколько разных форм-синонимов, имеющих одинаковое значение, например, **`любой любая любое`** или **`целое целые целых`**, и т.п. Перечень ключевых слов и их синонимов приведён в соответствующем разделе.
- **Идентификатор** (имя) — слово, состоящее из букв латиницы или кириллицы, арабских цифр и символов нижнего подчёркивания; первым символом идентификатора не может быть цифра.

Идентификаторы используются в качестве имён программ и их параметров, подпрограмм и их параметров, типов данных, констант, переменных, исключений и их параметров, полей структур.

- **Знак языка** — один не буквенный и не цифровой символ, либо комбинация из двух таких символов. К знакам языка относятся знаки арифметических и логических операций, круглые и квадратные скобки, точка, двоеточие, а также другие знаки, имеющие специальное синтаксическое значение.
- **Литерал** — буквальное значение того или иного типа данных в исходном коде программы. Форматы литералов описаны для каждого из типов данных в соответствующем разделе.
- **Однострочный комментарий** — любой текст, начиная от удвоенной косой черты «**//**» и до конца строки — т.е., до ближайшего символа перевода строки или возврата каретки.
- **Многострочный комментарий** — любой текст, начиная от комбинации символов «**/***» и до ближайшей комбинации символов «***/**». Вложенные многострочные комментарии не допускаются.

Ключевые слова языка

В нижеследующем списке ключевые слова, перечисленные в одной строке, являются синонимами — т.е., в исходном тексте программист может использовать любую из перечисленных словоформ по своему усмотрению. Это позволяет заметно, хотя и не полностью, приблизить текст программы к нормам русского языка.

Пример:

```
типы
    с: строка;
    ц1, ц2: целые;
    тПервый = массив строк;
    тВторой = словарь целых ключ символ;
    тТретий = массив словарей массивов структур
        поле1: дробное;
        поле2: логическое;
    окончание ключ строка;
```

Использование тех или иных форм ключевых слов остаётся на усмотрение программиста: конструкции «**словарей массивы строка**» и «**словарь массивов строк**» с точки зрения синтаксиса языка Клаус полностью эквивалентны. Авторы, однако, надеются, что из соображений читаемости кода большинство программистов всегда предпочтут второй вариант.

Перечень ключевых слов

```
бросить
вв
вернуть вернуться
вх
выбор
вых
да
дальше
для
до
дробное дробные дробных
если
есть
завершить
задача
и
```

```
из
или
иначе
исключение исключения
используется используются
каждый каждого каждой
ключ
когда
константа константы
курс
либо
логическое логические логических
любой любая любое
массив массивы массивов
момент моменты моментов
напоследок
начало
не
нет
нету
ничего
обратный
объект объекты объектов
окончание конец
от
ошибка
переменная переменные
повторить
пока
практикум
прервать
программа
продолжить
процедура
пусто
раз раза
символ символы символов
словарь словари словарей
сообщение
строка строки строк
структура структуры структур
тип типы
то
тогда
точность
функция
целое целые целых
цикл
```

Структура исходного кода

Программа на языке Клаус, а также любая подпрограмма — процедура или функция — состоит из заголовка, раздела определений и составного блока инструкций, представляющего собой тело программы или подпрограммы.

Определение программы

Заголовок программы начинается ключевым словом **`программа`**, за которым следует идентификатор — имя программы. Далее, если программа принимает аргументы командной строки, в круглых скобках следует

объявление единственного параметра, принимаемого программой, который всегда должен быть определён как входной и иметь тип **`массив строк`**.

Заголовок программы завершается знаком «`;`».

```
программа Пустая;
начало
    ничего;
окончание.

программа Эхо(арг: массив строк);
переменные
    к: целое;
    рзд: строка = "";
начало
    для к от 0 до длина(арг)-1 цикл начало
        вывести(рзд, арг[к]);
        рзд := " ";
    конец;
    вывести(НС);
окончание.
```

Между заголовком и телом программы могут располагаться определения переменных, констант, типов данных и подпрограмм. При этом каждое из определений может обращаться к именам, объявленным выше него по тексту исходного кода.

Тело программы представляет собой составной блок инструкций, который должен содержать одну или более инструкций Клаус, а также может содержать секции обработки исключений.

После окончания тела программы должен следовать знак «`.`».

Задачи из Практикума

Если программа является решением задачи из учебного курса Практикума, то её заголовок начинается ключевым словом **`задача`**, за которым следует имя решаемой задачи, далее слово **`практикум`** и имя учебного курса, содержащего решаемую задачу. После этого может следовать определение аргументов командной строки в круглых скобках, а затем знак «`;`».

```
задача Урок1Задание1 практикум ВводныйКурс;
начало
    вывести("Привет, Мироздание!");
окончание.
```

В нынешней версии Клауса такая форма заголовка никак не влияет на выполнение программы и служит лишь для того, чтобы организовать работу с учебными курсами в отладочной среде. В следующих версиях имя задачи может быть использовано для определения контекста выполнения программы, указания сценария автопроверки и в других подобных целях.

Подключение модулей

Модулем в языке Клаус называется именованная библиотека, содержащая определения типов данных, констант, переменных, исключений, процедур и функций. Для получения доступа к этим определениям необходимые модули должны быть подключены к программе с помощью конструкции **`используется`**.

Перечень модулей, используемых программой, должен располагаться сразу после заголовка программы; несколько подобных списков не допускаются. Перечень начинается с ключевого слова **`используется`** (или

используются), вслед за ним через запятую должны быть перечислены имена используемых модулей, в конце списка требуется знак «;».

```
программа Пример;
используются
    Терминал, Графика, Файлы;
начало
    ничего;
окончание.
```

Модуль **Клаус**, содержащий стандартную библиотеку языка, неявно подключается к каждой программе и считается первым в списке используемых модулей.

В нынешней версии языка нет возможности создавать собственные модули. Однако, программисты могут использовать несколько встроенных модулей, описанных в разделе «Встроенная библиотека» настоящего руководства.

Определение подпрограммы

В языке Клаус существует две разновидности подпрограмм: процедуры и функции. Они полностью идентичны во всём, за тем исключением, что функция возвращает значение, а процедура не возвращает.

Подпрограммы могут быть определены в разделе определений основной программы, либо в разделах определений других подпрограмм с произвольным уровнем вложенности.

Заголовок процедуры начинается ключевым словом **процедура**, заголовок функции — словом **функция**.

Далее следует имя подпрограммы и, если подпрограмма принимает параметры, перечень определений параметров в круглых скобках. Для подпрограмм, не принимающих параметры, обязательно должны быть указаны пустые круглые скобки.

Затем, только для функций, должен следовать знак «:» и имя типа данных, возвращаемого функцией — ключевое слово или идентификатор. Безымянное определение типа данных в этом случае не допускается.

Заголовок подпрограммы завершается знаком «;».

Раздел определений подпрограммы и её тело устроены так же, как у основной программы. После окончания тела подпрограммы должен следовать знак «;».

Параметры подпрограмм

В Клаусе определено три режима передачи параметров в подпрограммы: входные, выходные и проходные параметры.

Значение входного параметра непосредственно передаётся в подпрограмму, для чего в подпрограмме создаётся локальная переменная. Подпрограмма может произвольным образом менять значение входного параметра, но эти изменения не выходят за пределы подпрограммы. Даже если в качестве входного параметра была указана переменная, значение этой переменной после выхода из подпрограммы всегда останется таким же, каким было перед вызовом.

В качестве входных параметров можно указывать переменные, константы, буквальные значения и выражения, включая вызовы функций.

Выходным или проходным параметром может служить только переменная или часть переменной — например, поле структуры или элемент массива. В этом случае в подпрограмму передаётся не значение, но ссылка на переменную или её часть. Если подпрограмма вносит изменение в значение такого параметра, то это новое значение будет присвоено той переменной из внешней области видимости, что была передана в подпрограмму в качестве параметра — т.е., такое присваивание будет иметь эффект после выхода из подпрограммы.

Различие между выходными и проходными параметрами состоит в том, что значения проходных параметров передаются в подпрограмму без изменений, а значения выходных обнуляются перед вызовом подпрограммы.

Определение параметра начинается ключевым словом **`вх`**, **`вых`** или **`рв`** для входных, выходных и проходных параметров, соответственно. Ключевое слово **`вх`** можно опустить, в таком случае параметр будет считаться входным.

Далее следует имя параметра или несколько имён через запятую, а затем знак «`:`» и имя типа данных. Если указано несколько имён параметров, то все они будут иметь одинаковый режим и тип. Для объявления параметров разных режимов или типов их определения нужно разделить знаком «`;`». Для каждого из параметров можно указать несколько синонимичных имён, разделив их знаком «`/`».

В качестве типов параметров можно использовать ключевые слова, соответствующие простым типам Клауса, а также идентификаторы ранее объявленных типов данных. Безымянные определения типов в данном случае не допускаются. Единственным исключением из этого правила является определение входного параметра основной программы, который всегда должен быть объявлен как **`массив строк`**.

Предварительное определение

Предварительное определение подпрограммы представляет собой её заголовок, после которого вместо тела подпрограммы указано ключевое слово **`дальше`** и знак «`;`».

```
функция мояФункция(а, б: целое): дробное; дальше;
```

Если в исходном коде дано такое предварительное определение, то в той же области видимости ниже по тексту должно располагаться полное определение этой подпрограммы, включая её тело, причём заголовок должен точно совпадать с заголовком в предварительном определении.

Предварительное определение используется в тех случаях, когда несколько подпрограмм должны вызывать друг друга рекурсивно.

Примеры

```
процедура пустая();
начало
    ничего;
окончание;

функция синус(х: дробное): дробное;
начало
    вернуть sin(x);
окончание;

функция корень(вх х: дробное; вых рез: дробное): логическое;
начало
    если х < 0 то вернуть нет;
    рез := х^(1/2);
    вернуть да;
окончание;

тип Точка = структура
    г, в: целое;
конец;

функция тчк(х, у: целое): Точка;
переменная
    т: Точка;
начало
```

```
    т.г := х;
    т.в := у;
    вернуть т;
окончание;

процедура подвинуть(вв т: Точка; вх дх, ду: целое);
начало
    т.г += дх;
    т.в += ду;
окончание;

процедура первая(); дальше;

процедура вторая();
начало
    первая();
окончание;

процедура первая();
начало
    вторая(); // здесь будет бесконечная рекурсия
окончание;
```

Области видимости

Каждой подпрограмме, а также основной программе, соответствует своя область видимости идентификаторов. В пределах одной области видимости имена должны быть уникальны — т.е., ни одно имя не может совпадать с другим именем, объявленным в той же области видимости. Регистр букв в именах не учитывается — т.е., слова **`имя`**, **`ИМЯ`** и **`Имя`** считаются совпадающими.

Область видимости подпрограммы является вложенной по отношению к основной программе или к объемлющей подпрограмме, в которой она определена. Имена, объявленные во вложенной области видимости, могут совпадать с именами, объявленными в объемлющих областях видимости, но не друг с другом.

К области видимости основной программы относятся:

- Имя программы;
- Имя входного параметра программы, если он объявлен;
- Все имена, объявленные в области определений программы.

По аналогии, к области видимости подпрограммы относятся имя подпрограммы, имена её параметров и все локально объявленные имена.

Таким образом, имя подпрограммы принадлежит сразу двум областям видимости — той, в которой подпрограмма объявлена, и своей собственной. Следовательно, имя подпрограммы должно быть уникально в обеих этих областях.

При обращении к идентификатору в исходном коде программы имя разрешается по следующим правилам:

- Выполняется поиск имени в локальной области видимости, среди тех имён, что объявлены выше по тексту исходного кода;
- Если имя не найдено, то выполняется поиск выше по тексту во всех объемлющих областях видимости, по порядку изнутри наружу;

Иными словами, в каждой области видимости видны её собственные имена, а также все имена из объемлющих областей видимости, если они не были перекрыты собственными. Если во вложенной области видимости объявлено имя, совпадающее с именем из внешней области видимости, то обращение к последнему будет невозможно, т.к. локальный идентификатор скрывает за собой внешний.

Имена, объявленные во вложенных областях видимости, никогда не видны из объемлющих областей.

Модули, подключаемые с помощью конструкции **используется**, становятся внешними областями видимости по отношению к основной программе. Если идентификатор не найден в пределах программы, то выполняется поиск в каждом из подключённых модулей по порядку от конца к началу списка.

Модуль **Клаус**, содержащий стандартную библиотеку языка, неявно подключается к любой программе и является первым в списке подключённых модулей — т.е., поиск имён выполняется в нём в последнюю очередь, если имя не было найдено ни в одном из явно подключённых модулей.

Определения

Глобальные определения типов данных, констант и переменных размещаются разделе определений основной программы — между её заголовком и началом составного блока инструкций. Такие определения принадлежат глобальной области видимости, т.е., они видны в теле основной программы, а также во всех вложенных областях видимости — в процедурах и функциях, — если не перекрыты в них локальными определениями.

Любая подпрограмма имеет собственный раздел определений между её заголовком и телом, в котором также могут быть определены типы, константы и переменные. Эти определения являются локальными по отношению к подпрограмме — т.е., они видны только в ней, а также во вложенных в неё подпрограммах, но не видны из объемлющих областей видимости.

Во всех случаях видны только те определения, которые были даны выше по тексту исходного кода. Глобальная переменная или константа, объявленная после подпрограммы, не будет видна внутри этой подпрограммы.

```
программа Пример;
    тип мойТип = массив целых;
    константа Пи = 3.1415926;
    переменная мт: мойТип;
начало
    ничего;
окончание.
```

Определения типов

Секция определений типов должна располагаться в разделе определений программы или подпрограммы. Секция начинается ключевым словом **тип** или **типы**, за которым следует одно или несколько объявлений типов. Таких секций в разделе определений может быть сколько угодно, они могут быть расположены подряд или вперемежку с другими секциями и подпрограммами.

Объявление типа начинается с идентификатора (или нескольких идентификаторов, разделённых знаком «**/**»), за которыми следует знак «**=**», а за ним — определение типа. Определением может служить ключевое слово, обозначающее простой тип данных, идентификатор ранее объявленного типа, либо определение составного типа — массива, словаря или структуры. После каждого определения должен следовать знак «**;**».

```
программа Пример;
типы
    ЦЧ = целое;
    ЦЧ2 = ЦЧ;
    МЦЧ = массив целых;
    МСтр = массив структур
        п1, п2: целое;
        п3: строка;
    окончание;
```

```
начало
    ничего;
окончание.
```

Такие определения типов можно использовать при объявлении переменных, элементов массивов и словарей, полей структур, параметров подпрограмм и возвращаемых значений функций. Параметры исключений могут иметь только простой тип, обозначаемый ключевым словом языка.

Объявление типов, имеющих различные имена, но эквивалентные определения, зачастую используется для улучшения читаемости исходного кода программы.

Если в объявлении типа данных указано несколько идентификаторов через «**/**», то все эти идентификаторы являются синонимами, т.е. в исходном коде для улучшения читаемости может использоваться любой из них, на усмотрение программиста.

```
программа Пример;
тип
    Точка/Точек = структура
        x: дробное;
        y: дробное;
    окончание;
переменные
    т: Точка;
    мт: массив Точек;
начало
    ничего;
окончание.
```

Синтаксис определений составных типов подробно описан в разделе **Типы данных**.

Определения констант

Константа представляет собой именованное значение, которое определяется при компиляции и не может меняться в процессе выполнения программы. Константы не имеют другого смысла, кроме улучшения читаемости исходного кода программы.

Секция определений констант должна располагаться в разделе определений программы или подпрограммы. Секция начинается ключевым словом **`константа`** или **`константы`**, за которым следует одно или несколько объявлений констант. Таких секций в разделе определений может быть сколько угодно, они могут быть расположены подряд или вперемежку с другими секциями и подпрограммами.

Объявление константы начинается с идентификатора (или нескольких идентификаторов, разделённых знаком «**/**»), за которым следует знак «**=**», а за ним — буквальное значение, имя ранее определённой константы или выражение. После каждого определения должен следовать знак «**;**».

Если для константы указано несколько имён через «**/**», то все эти имена являются синонимами, т.е. в исходном коде для улучшения читаемости может использоваться любое из них, на усмотрение программиста.

Тип константы может быть указан явно — для этого после имени константы и перед знаком «**=**» должен следовать знак «**:**», а за ним — имя типа или его определение. Для констант составных типов явное указание типа является обязательным.

Если тип константы не указан явно, он определяется исходя из типа указанного буквального значения или выражения. В этом случае, например, если требуется объявить константу — дробное число со значением, равным единице, то буквальное значение должно быть специально указано как дробное: **`1.0`** или **`1э0`**.

Если определением константы служит выражение, то это выражение должно быть вычислимо на этапе компиляции — т.е., оно не может содержать иных операндов, кроме буквальных значений и имён ранее объявленных констант. Обращения к переменным и вызовы функций в таком выражении недопустимы.

```
программа Пример;
тип
    Точка = структура
        г, в: целое;
    окончание;
константы
    целыйНуль = 0;
    дробныйНуль = 0.0;
    нечисло = 0 / 0;
    бесконечность = 1 / 0;
    Пи = 3.1415926;
    прямойУгол = Пи / 2;
    т: Точка = {г = 10, в = 20};
начало
    ничего;
окончание.
```

Определения переменных

Переменная представляет собой именованное значение простого или составного типа, которое может меняться в процессе выполнения программы. Начальное значение переменной может быть указано при объявлении. Если оно не указано, то при входе в область видимости переменная получает нулевое начальное значение, как это описано в разделе **Типы данных**.

Для переменных составных типов начальное значение не может быть указано.

Секция определений переменных должна располагаться в разделе определений программы или подпрограммы. Секция начинается ключевым словом **`переменная`** или **`переменные`**, за которым следует одно или несколько объявлений переменных. Таких секций в разделе определений может быть сколько угодно, они могут быть расположены подряд или вперемежку с другими секциями и подпрограммами.

Объявление переменной начинается с идентификатора или списка идентификаторов, перечисленных через запятую — так можно объявить несколько однотипных переменных. Каждый из идентификаторов в списке может иметь несколько синонимов, перечисленных через «**`/`**». Затем следует знак «**`:`**», а за ним — определение типа переменной. После этого может быть указан знак «**`=`**» и начальное значение. Каждое объявление завершается знаком «**`;`**».

Если для переменной указано несколько имён через «**`/`**», то все эти имена являются синонимами, т.е. в исходном коде для улучшения читаемости может использоваться любое из них, на усмотрение программиста. Следует иметь в виду, что имена, перечисленные через «**`/`**», являются именами одной и той же переменной, а при перечислении имён через запятую создаются несколько переменных, каждая из которых может иметь несколько синонимичных имён.

В качестве определения типа допускается одно из ключевых слов простых типов, идентификатор ранее объявленного типа, либо безымянное определение составного типа.

Начальное значение должно быть либо буквальным, либо выражением, вычислимым на этапе компиляции — т. е., в этом выражении можно использовать только буквальные значения и константы, а переменные и вызовы функций не допускаются. Если одновременно объявляются несколько переменных через запятую, то все они получат одинаковые начальные значения.

Особым случаем является объявление переменной в инструкции перехвата исключения **`когда...тогда`**. В этом случае считается, что переменная относится к ближайшей объемлющей области видимости — т.е., к разделу определений той подпрограммы, которая содержит инструкцию перехвата. Однако, обращение к такой переменной возможно только в пределах инструкции перехвата исключения, в которой она объявлена.

```
программа Пример;
тип
    комплексное = структура
        д: дробное;
        м: дробное;
    окончание;
переменные
    стр: строка = "Новая строка";
    н, м: целое = 1;
    к: комплексное;
    точки/точек: словарь структур
        х, у: дробное;
    окончание ключ строка;
начало
    ничего;
окончание.
```

Определения исключений

Всплывающее исключение представляет собой специальный именованный объект, который создаётся для того, чтобы прервать последовательность выполнения программы в случае возникновения нештатной ситуации и задействовать средства обработки ошибок.

Секция определений исключений должна располагаться в разделе определений программы или подпрограммы. Секция начинается ключевым словом **`исключение`** или **`исключения`**, за которым следует одно или несколько объявлений исключений. Таких секций в разделе определений может быть сколько угодно, они могут быть расположены подряд или вперемежку с другими секциями и подпрограммами.

Объявление исключения начинается с идентификатора. Далее, если исключение принимает параметры, то их определения должны быть указаны в круглых скобках. Затем может следовать ключевое слово **`сообщение`**, а за ним текст сообщения об ошибке — буквальное значение или выражение строкового типа. Каждое объявление завершается знаком «`;`».

```
программа Пример;
исключения
    Ошибка1;
    Ошибка2 сообщение "Вторая ошибка";
    ОшибкаВТексте(стр, симв: целое; чтоНеТак: строка)
        сообщение "Ошибка в строке %(стр), символ %(симв). %(чтоНеТак)";
начало
    ошибка ОшибкаВТексте(5, 10, "Пропущена запятая.");
окончание.
```

Параметры исключений объявляются по тем же правилам, что и параметры подпрограмм, с двумя оговорками:

- они могут быть только простых типов;
- они могут быть только входными.

Если текст сообщения об ошибке не указан в определении исключения, его можно указать при создании исключения. Если он не указан и там, то в качестве сообщения об ошибке используется имя исключения.

Если исключение имеет параметры, то текст сообщения может содержать спецификаторы вида **`%(имя)`**, где имя — имя параметра исключения. При создании исключения такие спецификаторы будут заменены на строковые представления значений параметров.

Типы данных

Приведение типов

Значения простых типов можно преобразовывать в другие простые типы. В большинстве случаев такое преобразование должно быть выполнено явно. Для этого используется конструкция приведения типа, которая записывается как ключевое слово — имя типа данных, за которым в круглых скобках указывается исходное значение или выражение. Такая конструкция конвертирует исходное значение в указанный тип, а если это невозможно, создаётся исключение.

```
переменные
    д: дробное = 123.45;
    ц: целое = 10;
    с: строка = "yes";
    л: логическое;
начало
    л := логическое(с); // л = да
    с := строка(ц);     // с = "10"
    ц := целое(д);      // ц = 123
    д := ц;             // д = 123.0 (неявное преобразование)
    д := дробное("Ку"); // исключение ОшибкаКонвертации
окончание;
```

Преобразование типов данных выполняется по правилам, описанным ниже для каждого из простых типов.

Неявное преобразование допускается в двух случаях:

- при преобразовании символа в строку, но не наоборот;
- при преобразовании целого числа в дробное число, но не наоборот.

В нынешней версии языка для приведения типа не допускается использовать имена ранее объявленных типов данных — допустимы только ключевые слова простых типов. Кроме того, не допускается явное приведение составных типов, хотя в некоторых случаях возможно неявное преобразование.

Простые типы

Символ

Переменная символьного типа содержит кодпойнт Юникод в формате 32-битного целого числа баз знака. Нулевые значения допускаются.

Синтаксис

```
переменные
    п1: символ;
    п2: массив символов;
```

Ключевые слова **`символ`** и **`символов`** полностью эквивалентны и взаимозаменяемы.

Формат литерала

```
'X'   // символ в кодировке UTF-8, заключённый в одиночные апострофы
№2A   // код символа в формате 16-ричного целого числа
#2A   // код символа в формате 16-ричного целого числа
```

Начальное значение

Символ с нулевым кодом.

Преобразование типов

При приведении к целочисленному типу возвращает численное значение кода символа.

При приведении к строке возвращает строку, содержащую один символ в кодировке UTF-8, за исключением нулевого значения, которое приводится к пустой строке.

Строка

Переменная строкового типа содержит последовательность символов в кодировке UTF-8. Строка может быть пустой, т.е. не содержать ни одного символа. Нулевые символы в строке не допускаются.

Синтаксис

```
переменные
    п1: строка;
    п2: массив строк;
начало
    п1 := "АБВГ";
    вывести(п1[2]); // будет выведено "Б", т.к. буквы кириллицы занимают 2 байта
окончание.
```

Ключевые слова **строка** и **строк** полностью эквивалентны и взаимозаменяемы.

В исходном коде программы для обращения к символу в строке нужно указать целочисленный индекс в квадратных скобках после имени строковой переменной. Этот индекс означает номер байта в строке, соответствующий началу требуемого символа. Первый символ в строке начинается с нулевого байта.

Такое обращение к символу в строке допустимо только для чтения. Для модификации символов в строке можно использовать функции стандартной библиотеки.

Формат литерала

```
""               // пустая строка
"123абв"         // последовательность символов, заключённая в кавычки
"123""абв"       // для добавления в строку символа " его нужно удвоить
""#41№20№42#0A   // последовательность 16-ричных кодов символов с кавычками в начале
"123"#20"абв"№0A // символы в кавычках и коды можно комбинировать
```

Начальное значение

Пустая строка.

Преобразование типов

При приведении к целому или дробному числу выполняется преобразование из текстового представления числа, при ошибках возникает исключение **ОшибкаКонвертации**.

При приведении к моменту выполняется преобразование из стандартного текстового представления даты и/или времени, при ошибках возникает исключение **ОшибкаКонвертации**.

При приведении к логическому значению строка должна иметь одно из следующих значений, без учёта регистра:

```
"1", "y", "yes", "t", "true", "д", "да", "и", "истина" // истина
"0", "n", "no", "f", "false", "н", "нет", "л", "ложь"  // ложь
```

Любые значения простых типов приводятся к строковому типу без ошибок.

Целое

Переменная целого типа содержит 64-битное целочисленное значение со знаком.

Синтаксис

```
переменные
    п1: целое;
    п2: массив целых;
```

Ключевые слова **целое** и **целых** полностью эквивалентны и взаимозаменяемы.

Формат литерала

```
-0123456789        // десятичные цифры
-$0123456789abcdef // 16-ричные цифры (латиница)
-$0123456789абцдеф // 16-ричные цифры (кириллица)
```

Для отрицательных чисел применяется знак «-» в начале литерала; знак «+» для положительных чисел не допускается. Регистр символов 16-ричных цифр не учитывается, латинские и кириллические цифры можно смешивать.

Начальное значение

Нуль.

Преобразование типов

Целочисленные значения приводятся к любым простым типам без ошибок, за исключением типа **объект**.

При приведении к символьному типу целое число означает кодпойнт Юникод (старшие 32 бита при преобразовании отбрасываются).

При приведении к строке выполняется преобразование к десятичному текстовому представлению числа.

При приведении к моменту целое число означает количество полных суток, считая от «нулевого» момента.

При приведении к логическому истинным считается любое значение, отличное от нуля.

Дробное

Переменная дробного типа содержит 64-битное вещественное значение с двойной точностью.

Дробное число может принимать особые значения: **нечисло**, положительная бесконечность и отрицательная бесконечность. Такие значения нельзя указать буквально в исходном коде, но можно получить с помощью деления на нуль.

Синтаксис

```
переменные
    п1: дробное;
    п2: массив дробных;
```

Ключевые слова **дробное** и **дробных** полностью эквивалентны и взаимозаменяемы.

Формат литерала

```
-0123456.789      // представление с десятичной точкой
-0123456.789e+123 // экспоненциальное представление

д1 :=  0/0;       // нечисло
д2 :=  1/0;       // +бесконечность
д3 := -1/0;       // -бесконечность
```

В качестве десятичного разделителя всегда используется точка. В качестве символа экспоненциальной части может использоваться латинская «E» и кириллическая «Е» или «Э» в любом регистре. Если десятичная точка и экспоненциальная часть отсутствуют, литерал считается целочисленным.

Начальное значение

Нуль.

Преобразование типов

При приведении к строке число конвертируется в строковое представление в десятичном или экспоненциальном формате, в зависимости от того, какой из них короче. Для небольших чисел, если дробная часть числа равна нулю, полученное строковое представление будет целочисленным.

При приведении к строке **нечисло** преобразуется в текст "Nan", а бесконечность — в "+Inf" или "-Inf". Эти строковые значения (в любом регистре) можно использовать для приведения строки к дробному числу, но не в качестве литералов в исходном коде.

При приведении дробного числа к целому дробная часть отбрасывается без округления. Значение **нечисло** и бесконечные значения при приведении к целому вызывают исключение **НеверноеЧисло**.

При приведении к моменту число означает количество суток, считая от «нулевого» момента.

Момент

Переменная типа **момент** содержит значение даты и времени — 64-битное вещественное число с двойной точностью, означающее число суток, прошедшее от «нулевого» момента. Целая часть числа соответствует количеству полных суток, а дробная — суточному времени, выраженному в долях суток.

Нулевым моментом считается полночь 1899-12-30 00:00:00. Более поздние даты представлены положительными числами, более ранние — отрицательными.

Синтаксис

```
переменные
    п1: момент;
    п2: массив моментов;
```

Ключевые слова **момент** и **моментов** полностью эквивалентны и взаимозаменяемы.

Формат литерала

```
`2023-09-15`       // дата в обратных апострофах
`12:45:50.25`      // время в обратных апострофах
`2023-09-15 12:45` // дата и время в обратных апострофах
`09-15 12:00`      // полдень 15 сентября текущего года
```

```
`15 12:00`          // полдень 15 числа текущего месяца
`15`                // полночь 15 числа текущего месяца
```

Дата и время указываются в стандартном формате: **YYYY-MM-DD hh:mm:ss**.

Может быть указана только дата, только время или обе части вместе — в последнем случае дата и время разделяются пробелом. Если дата или время не указаны, они считаются равными нулю.

При указании даты можно опустить год, либо год и месяц, вместе со следующим за ними символом «-» — по умолчанию подставляются, соответственно, текущие год и месяц.

При указании времени можно опустить секунды вместе с предшествующим символом «:» — по умолчанию подставляется 0 секунд.

В качестве значения секунд можно указать дробное число.

Примечание: при указании литерала даты в исходном коде, если опущены год или месяц, то их текущие значения вычисляются в момент компиляции программы и в дальнейшем остаются постоянными.

Начальное значение

Полночь 1899-12-30.

Преобразование типов

Моменты взаимно-однозначно приводятся к дробным числам и обратно.

При приведении момента к целому числу дробная часть (т. е., значение суточного времени) отбрасывается без округления. Значение **нечисло** и бесконечные значения при приведении к целому вызывают исключение **`НеверноеЧисло`**.

Приведение момента к строке выполняется по следующим правилам:

- Если момент имеет значение **нечисло** или бесконечное значение, преобразование вернёт строку "Nan", "+Inf" или "-Inf".
- Моменты с ненулевой целой и дробной частью возвращают строковые представления даты и времени в стандартном формате, разделённые пробелом.
- Моменты с ненулевой целой и нулевой дробной частью возвращают строковое представление даты.
- В остальных случаях возвращается строковое представление времени.

При приведении строки к моменту допускаются такие же строковые значения, как в литералах, а также значения "nan", "inf", "+inf" или "-inf" в любом регистре. При ошибках преобразования возникает исключение **`ОшибкаКонвертации`**.

Логическое

Переменная логического типа имеет одно из двух значений: истинное или ложное. Внутренним представлением такого значения является 8-битное целое без знака, которое принимает значения 0 или 1.

Синтаксис

```
переменные
    п1: логическое;
    п2: массив логических;
```

Ключевые слова **`логическое`** и **`логических`** полностью эквивалентны и взаимозаменяемы.

Формат литерала

В качестве литералов в исходном коде применяются ключевые слова **да** и **нет**.

Начальное значение

Ложь.

Преобразование типов

При приведении логического значения к целому истина соответствует значению 1, а ложь — значению 0. При приведении целого к логическому истинным считается любое ненулевое значение.

При приведении к строке истина приводится к значению "да", а ложь — к значению "нет". При приведении строки к логическому допустимы следующие строковые значения, в любом регистре:

```
"1", "y", "yes", "t", "true", "д", "да", "и", "истина" // истина
"0", "n", "no", "f", "false", "н", "нет", "л", "ложь"  // ложь
```

Любые другие строковые значения вызовут исключение **ОшибкаКонвертации**.

Объект

Переменная типа **объект** содержит дескриптор встроенного объекта Клауса — например, открытого файла, окна и др. подобных объектов. Такие объекты создаются и уничтожаются вызовами функций встроенной библиотеки. Переменной типа **объект** можно присвоить значение **пусто**, либо непустое значение, возвращаемое функцией. Кроме того, переменным типа **объект** можно присваивать значения других переменных этого же типа.

Значение типа **объект** имеет смысл только в пределах сеанса выполнения программы. Поэтому такие значения нельзя прочесть из файла, ввести с клавиатуры или получить любым другим способом, кроме вызова процедуры или функции, создающей объект.

Объекты, созданные при выполнении программы, должны быть уничтожены перед её завершением, иначе при завершении программы возникнет исключение **ОшибкаВыполнения**. Например, каждый открытый файл должен быть закрыт, каждое созданное окно уничтожено, и т. п.

Внимание! Присваивание переменной типа **объект** значения **пусто** не уничтожает объект. Для уничтожения объектов нужно вызывать соответствующие функции встроенной библиотеки.

Физически объект представляет собой 32-битное целое число со знаком, которое является индексом во внутреннем списке объектов Клауса. Значение **пусто** соответствует нулевому целочисленному значению, отрицательные значения не применяются.

При уничтожении объектов их целочисленные дескрипторы высвобождаются и могут быть (и, скорее всего, сразу же будут) использованы повторно при создании других объектов. Если программист присвоил значение одного и того же объекта нескольким переменным, то его ответственностью является обнуление всех этих переменных при уничтожении объекта.

Синтаксис

```
переменные
    п1: объект;
    п2: массив объектов;
```

Ключевые слова **объект** и **объектов** полностью эквивалентны и взаимозаменяемы.

Формат литерала

В качестве литерала в исходном коде применяется ключевое слово **пусто**. Иные значения дескриптора объекта не могут быть указаны буквально.

Начальное значение

Пусто.

Преобразование типов

Приведение других типов к типу **объект** не допускается.

Приведение значения типа **объект** к целому возвращает физическое значение дескриптора — 32-битное целое число. Значение **пусто** приводится к нулю.

При приведении объекта к строке возвращается строковое представление численного значения дескриптора в 16-ричном формате.

Составные типы

Массив

Переменная типа **массив** содержит несколько однотипных значений, каждому из которых соответствует целочисленный индекс. Первый элемент массива имеет индекс 0, последний элемент — индекс на единицу меньший, чем длина массива. Индексы в массиве следуют один за другим, без пропусков. Элементы массива могут иметь любой тип — простой или составной.

Синтаксис

Определение массивного типа состоит из ключевого слова **массив** и следующего за ним определения типа элемента массива. Для определения многомерных массивов можно использовать разные формы ключевого слова: **массив массивов**, **массив массивов массивов** и т. п.

```
тип
    тМЦ = массив целых;
    тМСДкС = массив словарей дробных ключ символ;
переменная
    мойМассив: массив массивов моментов;
    ещёМассив: массив словарей структур
        поле1: символ;
        поле2: строка;
    окончание ключ строка;
```

Ключевые слова **массив** и **массивов** полностью эквивалентны и взаимозаменяемы.

В исходном коде программы для обращения к элементу массива нужно указать индекс элемента в квадратных скобках после имени переменной-массива. Для работы с массивами и их элементами используются функции **длина()**, **добавить()**, **вставить()**, **удалить()** и конструкции **есть()**, **нету()**.

```
переменная
    м1: массив целых;
    м2: массив массивов моментов;
начало
    длина(м1, 1);
    м1[0] := 10;
    длина(м2, 2);
```

```
    длина(м2[1], 11);
    м2[1][10] := `2023-07-06 12:00`;
окончание.
```

Формат литерала

Литерал массива начинается со знака «**[**», за которым следует список элементов массива через запятую и завершающий знак «**]**». Список элементов содержит выражения, определяющие значения элементов массива.

```
переменная
    м: массив строк;
начало
    м := ["Иванов", "Петров", "Сидоров"];
окончание.
```

Кроме того, в качестве литерала можно использовать ключевое слово **пусто** — это означает пустой массив.

Начальное значение

Пустой массив.

Преобразование типов

Явное приведение массивов к другим типам данных запрещено.

Переменной типа **массив** можно присвоить значение — массив, состоящий из элементов такого же типа или типа, совместимого по присваиванию. Например, массиву дробных можно присвоить массив целых, а массиву строк можно присвоить массив символов.

Словарь

Переменная типа **словарь** содержит несколько однотипных значений, каждому из которых соответствует значение ключа, уникальное в пределах словаря. Ключи в словаре отсортированы в порядке возрастания. Элементы словаря могут иметь любой тип — простой или составной, а ключи — любой из простых типов.

Синтаксис

Определение словарного типа состоит из ключевого слова **словарь** и следующего за ним определения типа элемента словаря; после этого следует слово **ключ** и ключевое слово, указывающее тип ключа. Затем, если ключ словаря имеет тип **дробное** или **момент**, может следовать ключевое слово **точность**, а за ним константное выражение дробного типа, определяющее точность сравнения ключей.

Определением типа элемента словаря может служить ключевое слово, идентификатор ранее объявленного типа, либо безымянное определение типа. Определением типа ключа может быть только ключевое слово простого типа данных.

```
тип
    тСЦ = словарь целых ключ строка;
    тСМДкС = словарь массивов дробных ключ символ;
переменная
    мойСловарь: словарь моментов ключ объект;
    ещёСловарь: словарь структур
        поле1: символ;
        поле2: строка;
    окончание ключ строка;
```

Ключевые слова **словарь** и **словарей** полностью эквивалентны и взаимозаменяемы.

В исходном коде программы для обращения к элементу словаря нужно указать ключ элемента в квадратных скобках после имени переменной-словаря. Для добавления в словарь пары ключ-значение можно использовать инструкцию присваивания. Кроме того, для работы со словарями и их элементами используются функции **длина()**, **добавить()**, **вставить()**, **удалить()** и конструкции **есть()**, **нету()**.

```
переменная
    сл1: словарь целых ключ символ;
    сл2: словарь словарей моментов ключ целое ключ строка;
начало
    сл1['0'] := 10;
    сл2["первый"][10] := `2023-07-06 12:00`;
окончание.
```

Для словарей с ключом типа **дробное** или **момент** можно указать точность сравнения ключей. Если указана ненулевая точность, то два ключевых значения будут считаться равными, если модуль их разности не превышает указанного значения точности.

```
переменная
    сл: словарь строк ключ дробное точность 0.01;
начало
    сл[0] := "Нуль";
    сл[0.001] := "Снова нуль";
    сл[0.02] := "Две сотых";
окончание.
```

В приведённом примере в словарь будут добавлены два значения с ключами 0 и 0.02, т. к. ключевое значение 0.001 совпадёт с нулём в пределах указанной точности, и строковое значение "Снова нуль" будет присвоено ранее добавленному нулевому ключу.

Формат литерала

Литерал словаря начинается со знака «**{**», за которым следует список пар ключ-значение через запятую и завершающий знак «**}**». Каждый элемент списка начинается с выражения, определяющего значение ключа, за которым следует знак «**:**» и выражение, определяющее словарное значение, сопоставленное этому ключу.

Если несколько элементов списка соответствуют одному и тому же значению ключа, то этому ключу присваивается словарное значение из последнего по порядку элемента.

```
переменная
    сл: словарь целых ключ строка;
начало
    сл := {
        "Иванов": 150,
        "Петров": 200,
        "Сидоров": 250};
окончание.
```

Кроме того, в качестве литерала можно использовать ключевое слово **пусто** — это означает пустой словарь.

Начальное значение

Пустой словарь.

Преобразование типов

Явное приведение словарей к другим типам данных запрещено.

Переменной типа **словарь** можно присвоить значение — словарь, состоящий из элементов такого же типа или типа, совместимого по присваиванию, при условии точного совпадения типов ключей. Например, словарю дробных можно присвоить словарь целых, а словарю строк можно присвоить словарь символов.

Структура

Переменная типа **структура** содержит одно или несколько именованных значений различных типов, называемых полями структуры; значением структурной переменной является совокупность значений всех её полей. Имена полей должны быть уникальны в пределах структуры. Определение структуры является статическим, т. е. состав её полей не может быть изменён в процессе выполнения программы.

Синтаксис

Определение структурного типа начинается ключевым словом **структура** и заканчивается ключевым словом **окончание**, между которыми располагаются определения полей структуры. Определения полей синтаксически идентичны определениям переменных, с той оговоркой, что для полей структур не могут быть указаны начальные значения.

Типы полей структур могут быть простыми или составными; при объявлении полей допускаются безымянные определения типов, в т.ч. определения вложенных структур. Имена полей вложенной структуры должны быть уникальны в её пределах, но могут совпадать с именами полей объемлющей структуры.

Ключевые слова **структура** и **структур** полностью эквивалентны и взаимозаменяемы.

```
тип
    тСтр = структура
        п1: целое;
        п2, п3: дробное;
        п4: массив строк;
        п5: структура
            п1: символ;
            п2: словарь объектов ключ строка;
        конец;
    окончание;
переменная
    точка: структура
        г, в: целое;
    конец;
    люди: массив структур
        фамилия, имя, отчество: строка;
        ДР: момент;
    конец;
```

В исходном коде программы для обращения к полю структуры нужно указать имя поля, отделив его от имени переменной знаком «.». Аналогичным образом можно обращаться к полям вложенных структур.

```
переменная
    стр: структура
        п1: строка;
        п2: структура
            п1, п2: целое;
        конец;
    окончание;
начало
    стр.п1 := "значение";
    стр.п2.п1 := 10;
окончание.
```

Формат литерала

Литерал структуры начинается со знака «**{**», за которым следует список значений полей через запятую и завершающий знак «**}**». Каждый элемент списка начинается с имени поля, за которым следует знак «**=**» и выражение, определяющее значение этого поля.

Литерал не обязательно должен содержать все поля структуры; если поле отсутствует в списке, оно получит нулевое начальное значение. Однако, каждое поле должно должно быть упомянуто в списке не более одного раза.

```
программа Пример;
типы
    Точка = структура
        г, в: целое;
    окончание;
    Размер = Точка;
переменная
    фигура: структура
        вид: строка;
        цвет: целое;
        поз: Точка;
        разм: Размер;
    окончание;
начало
    фигура := {
        вид = "эллипс",
        цвет = $FFFFFF,
        поз = {г = 10, в = 20},
        разм = {г = 50, в = 50}
    };
окончание.
```

Кроме того, в качестве литерала структуры можно использовать ключевое слово **`пусто`** — в этом случае все поля структуры будут иметь нулевые начальные значения.

Начальное значение

Все поля структуры имеют начальные значения, соответствующие их типам.

Преобразование типов

Явное приведение структур к другим типам данных запрещено.

Переменной типа **`структура`** можно присвоить значение — структуру, имеющую точно такое же количество полей с точно такими же именами (без учёта регистра) и типами, совместимыми по присваиванию. Например, структуре, имеющей одно строковое поле, можно присвоить структуру, имеющую одно символьное поле с таким же именем.

Выражения и операции

Выражения в Клаусе записываются в инфиксной форме, для группировки операций используются круглые скобки.

Операции в выражениях выполняются в порядке возрастания приоритета (чем меньше приоритет, тем раньше — см. табл. ниже). Равноприоритетные операции выполняются по порядку слева направо. Для бинарных операций сначала вычисляется левый, а затем правый операнд. Все операнды выражений обязательно вычисляются и все операции над ними выполняются, даже если это излишне для вычисления итогового значения выражения.

Операции сложения, вычитания и умножения возвращают целый результат, если оба операнда целые. Если хотя бы один операнд дробный, то операция вернёт дробный результат. Операция деления всегда возвращает дробный результат.

Операции целочисленного деления и остатка от деления, а также побитовые операции определены только для целочисленных операндов и возвращают целочисленный результат.

Операции сложения и вычитания определены для целых и дробных чисел, а также моментов. Вычитание момента из числа запрещено; в остальных случаях сложение и вычитание двух моментов, либо момента и числа возвращают момент.

Операция соединения определена для символов и строк в любых сочетаниях и возвращает строковый результат. При этом нулевые символы приводятся к пустой строке.

Операции сравнения определены для всех простых типов, сравнимых между собой:

- символы можно сравнивать с символами, но не со строками;
- строки можно сравнивать со строками, но не с символами;
- целые и дробные числа можно сравнивать между собой в любых сочетаниях;
- моменты можно сравнивать с моментами;
- логические значения можно сравнивать между собой: **`да`** больше, чем **`нет`**.
- объекты можно сравнивать между собой, но это не имеет иного смысла, кроме возможности создать словарь с ключом типа **`объект`**. При сравнении объектов значение **`пусто`** считается наименьшим.

Операции сравнения запрещены для дробных чисел и моментов, имеющих значение **нечисло**. Если хотя бы один из аргументов операции сравнения является нечислом, возникает исключение **`НеверноеЧисло`**.

Арифметические операции для нечисел разрешены, но если хотя бы один из аргументов является нечислом, то результатом операции также будет **нечисло**.

Значения составных типов — массивы, словари, структуры — не могут быть операндами в выражениях.

Таблица операций

Знак	Операция	Операнды	Результат	Приоритет
`-`	Унарный минус	целое, дробное, момент	целое, дробное, момент	0
`! не`	Унарное отрицание	логическое, целое	логическое, целое	0
`^`	Возведение в степень	целое, дробное	целое, дробное	1
`*`	Умножение	целое, дробное	целое, дробное	2
`/`	Деление	числа	число	2
`\`	Целочисленное деление	целое	целое	2
`%`	Остаток от деления	целые	целое	2
`+`	Сложение	числа, моменты	число, момент	3
`-`	Вычитание	числа, моменты	число, момент	3
`&`	Побитовое И	целые	целое	4
`|`	Побитовое ИЛИ	целые	целое	5
`~`	Побитовое ЛИБО	целые	целое	5
`++`	Соединение	строки, символы	строка	6
`=`	Равно	простые	логический	7
`<>`	Не равно	простые	логический	7
`>`	Больше	простые	логический	7
`<`	Меньше	простые	логический	7
`>=`	Больше или равно	простые	логический	7
`<=`	Меньше или равно	простые	логический	7
`&& и`	Логическое И	логические	логический	8
`|| или`	Логическое ИЛИ	логические	логический	9
`~~ либо`	Логическое ЛИБО	логические	логический	9

Инструкции

Знак «;»

После каждой инструкции в языке Клаус требуется знак «`;`» (точка с запятой). Этот знак является ни частью инструкции, ни разделителем между инструкциями, но признаком окончания инструкции.

Знак «`;`» не допускается:

- в условном ветвлении перед ключевым словом **`иначе`**, если оно есть;
- в цикле с постусловием перед ключевым словом **`пока`**.

Во всех остальных случаях знак «`;`» требуется после каждой инструкции, за исключением окончания тела основной программы, где требуется знак «`.`» (точка).

Инструкция присваивания

Инструкция присваивания предназначена для установки значения переменной или части переменной — элемента массива, элемента словаря, поля структуры.

В Клаусе определены две разновидности инструкций присваивания: простая и вычисляющие.

Простое присваивание

Простое присваивание записывается в следующей форме:

```
<переменная> := <выражение>;
```

где **переменная** — путь к переменной, а **выражение** должно возвращать значение совместимого типа, которое будет присвоено указанной переменной.

Путь к переменной состоит из имени переменной и последовательности квалификаторов — индексов массивов, ключей словарей и имён полей структур.

Для присваивания значения полю структуры имя поля указывается через точку после пути к структуре. Для присваивания значения элементу массива (словаря) индекс массива (ключ словаря) указывается в квадратных скобках после пути к массиву/словарю, например:

```
программа Пример;
типы
    тСтр1 = структура
        с1п1: строка;
        с1п2: дробное;
    окончание;
    тСтр2 = структура
        с2п1: массив массивов тСтр1;
        с2п2: строка;
    окончание;
переменные
    а: целое;
    б: дробное;
    в: массив строк;
    г: словарь строк ключ дробное;
    сл: словарь тСтр2 ключ строка;
    м: массив тСтр1;
начало
    а := 10;
```

```
    б := а + 0.01;
    длина(в, 1);
    в[0] := "значение";
    г[3.1415] := "Пи";
    добавить(сл, "Ключ");
    длина(сл["Ключ"].с2п1, 2);
    сл["Ключ"].с2п1[0] := м;
    длина(сл["Ключ"].с2п1[1], 1);
    сл["Ключ"].с2п1[1][0].с1п1 := "Новое значение";
окончание.
```

Вычисляющее присваивание

В Клаусе определены следующие инструкции вычисляющего присваивания:

+=	Прибавить и присвоить
-=	Вычесть и присвоить
*=	Умножить и присвоить
/=	Разделить и присвоить
\=	Разделить нацело и присвоить
%=	Разделить и присвоить остаток от деления
^=	Возвести в степень и присвоить
&=	Вычислить побитовое «И» и присвоить
|=	Вычислить побитовое «ИЛИ» и присвоить
~=	Вычислить побитовое «ЛИБО» и присвоить

Каждая из этих инструкций выполняет арифметическую или побитовую операцию, используя в качестве левого операнда значение переменной, указанной слева, а в качестве правого — значение выражения, указанного справа. Результат операции становится новым значением переменной, например:

```
начало
    А += 1; // прибавляет к значению переменной А единицу
    Б *= 2; // удваивает значение переменной Б
    В ^= 3; // возводит значение переменной В в третью степень
    Г |= 8; // устанавливает единицу в 3-м (считая от 0) бите переменной Г
окончание;
```

Совместимость типов при присваивании

Тип значения, возвращаемый выражением в правой части инструкции, должен быть совместим по присваиванию с переменной в левой части.

Для переменных простых типов:

- Строковой переменной может быть присвоено символьное значение, но не наоборот;
- Дробной переменной может быть присвоено целое значение, но не наоборот;
- В остальных случаях требуется точное соответствие типов.

Для переменных составных типов:

- Массиву может быть присвоен массив, состоящий из элементов совместимого типа;

- Словарю может быть присвоен словарь, имеющий ключ точно такого же типа и состоящий из элементов совместимого типа.
- Структуре может быть присвоена структура, имеющая ровно столько же полей с точно такими же именами (без учёта регистра) и значениями совместимых типов. Порядок следования полей не имеет значения.

Таким образом, например, массиву дробных чисел можно присвоить массив целых чисел, а структуре, имеющей строковое поле, можно присвоить структуру, имеющую одноимённое символьное поле.

Копирование данных при присваивании

С логической точки зрения, каждая переменная языка Клаус содержит свою собственную копию данных. При этом гарантируется, что присваивание значения одной переменной никогда не повлечет за собой изменение значения какой-нибудь другой переменной.

Физически, при присваивании значений переменных, а также при передаче параметров в процедуры и функции, используется механизм копирования-по-требованию. А именно, при присваивании значения переменная-приёмник получает ссылку на ту же копию данных, что хранится в переменной-источнике, так что в оперативной памяти имеется лишь одна копия данных, на которую ссылаются две (или более) переменных.

Далее, если значение одной из переменных модифицируется, то перед этим для неё создаётся уникальная копия данных, и все изменения вносятся в эту вновь созданную копию. Таким образом, копирование данных в оперативной памяти выполняется только тогда, когда оно в самом деле необходимо.

Нужно учитывать, что копирование-по-требованию работает только для переменных, имеющих полностью идентичные типы данных. Например, присваивание словаря целых словарю дробных вызовет немедленное копирование всех ключей и значений из одного словаря в другой.

Кроме того, копирование-по-требованию выполняется лишь для всего значения переменной как целого и не работает ни для отдельных элементов массивов и словарей, ни для отдельных полей структур. Например, если переменная А содержит ссылку на массив, полученную от переменной Б, и первому элементу этого массива присваивается новое значение, то перед выполнением присваивания для переменной А будет создана новая копия всего массива.

Инструкция вызова подпрограммы

Для вызова подпрограммы — процедуры или функции — в исходном коде записывается её имя, за которым должны следовать круглые скобки. Если для подпрограммы определены параметры, то перечень значений параметров указывается в скобках через запятую в том количестве и в том порядке, как это предусмотрено определением подпрограммы. Если параметры отсутствуют, круглые скобки должны быть пустыми.

```
начало
    мояПроцедура();
    моя‌ВтораяПроцедура(10, 20, "тридцать");
окончание;
```

Таким образом можно вызывать как процедуры, так и функции — в последнем случае значение, возвращаемое функцией, теряется. Если возвращаемое функцией значение нужно использовать в дальнейшем по ходу программы, то его необходимо присвоить некоторой переменной, например:

```
начало
    моя‌Переменная := моя‌Функция('А', "Б");
окончание;
```

Функции, возвращающие значения простых типов можно, кроме того, использовать в качестве аргументов в выражениях:

```
начало
    результат := вычислитьОдно(1, 2) + вычислитьДругое(3, 4, 5) + 1;
    если вычислитьТретье() > 0 то результат := 0;
окончание;
```

Ничего — пустая инструкция

Пустая инструкция записывается ключевым словом **ничего**:

```
начало
    ничего;
окончание;
```

Эта инструкция не выполняет никаких действий. Она может использоваться в тех случаях, когда правила синтаксиса требуют наличия инструкции, которая фактически не нужна, например:

```
начало
    если ВсёХорошо то ничего
    иначе сообщить("Что-то пошло не так!");
окончание;
```

Примечание: Эта инструкция выглядит бесполезной, однако это не так: без неё невозможно обойтись, если требуется полностью подавить всплывающее исключение. Кроме того, она бывает удобна, чтобы установить точку останова в отладчике перед выходом из блока, но после выполнения всех инструкций в блоке.

Завершить — завершение программы

Для завершения работы программы используется инструкция **завершить**:

```
завершить;
завершить 0;
завершить кодОшибки;
```

где **кодОшибки** — целое число (или выражение целочисленного типа), которое будет выдано операционной системе в качестве кода завершения процесса. Если программа была выполнена успешно, она должна вернуть код завершения равный нулю. Ненулевой код завершения свидетельствует об ошибке, возникшей при выполнении программы.

Так, если при выполнении программы возникают необработанные исключения, программа будет по умолчанию завершена с кодом -1. Если все инструкции, составляющие тело программы, были выполнены успешно, и программа не была явно завершена инструкцией **завершить**, то код завершения по умолчанию будет равен нулю. Аналогично, инструкция **завершить** вызывает успешное завершение программы, если код не указан явно.

Инструкция **завершить** допустима внутри подпрограмм любого уровня вложенности. Если инструкция **завершить** выполняется внутри составного блока, содержащего секцию **напоследок**, то инструкции в этой секции будут выполнены перед выходом из блока — так, как если бы инструкция **завершить** вызвала исключение.

Вернуть — возврат из подпрограммы

Инструкция возврата используется для возврата из процедуры или функции с указанием значения, возвращаемого функцией.

Инструкция записывается ключевым словом **вернуть** или **вернуться**, после которого может быть указано возвращаемое значение функции.

```
функция мояФункция(): строка;
начало
    вернуть "результат";
окончание;

процедура мояПроцедура();
начало
    вернуться;
окончание;
```

Ключевые слова **вернуть** и **вернуться** полностью эквивалентны и взаимозаменяемы.

Если инструкция используется в теле функции, то после ключевого слова должно быть указано значение (или выражение), возвращаемое функцией. Если все инструкции в теле функции были выполнены, и функция не была явно завершена инструкцией **вернуть**, то она вернёт нулевое начальное значение соответствующего типа.

Для процедур указывать возвращаемое значение не допускается.

Если инструкция **вернуть** выполняется в теле основной программы, то она работает в точности так же, как инструкция **завершить**. В этом случае возвращаемое значение должно быть целым числом и означает код завершения процесса.

Если инструкция **вернуть** выполняется внутри составного блока, содержащего секцию **напоследок**, то инструкции в этой секции будут выполнены перед выходом из блока — так, как если бы инструкция **вернуть** вызвала исключение.

Начало-конец — составной блок инструкций

Составной блок представляет собой инструкцию, которая может содержать одну или несколько других инструкций и предоставляет средства для обработки исключений.

Составной блок начинается ключевым словом **начало** и заканчивается словом **конец** или **окончание**. Ключевые слова **конец** и **окончание** полностью эквивалентны и взаимозаменяемы. Между началом и окончанием блока может находиться одна или несколько других инструкций (в т.ч., вложенные составные блоки).

```
начало
    выполнитьОдно();
    выполнитьДругое();
конец;
```

Составной блок может применяться во всех случаях, когда синтаксис языка требует наличия ровно одной инструкции, а фактически их требуется несколько — в условных ветвлениях, в циклах, в инструкции выбора и др. Тело основной программы и любой процедуры или функции также представляет собой составной блок.

```
если А > Б то чтоТоОдно()
иначе чтоТоДругое();

если Ы < Ъ то начало
    чтоТоОдно();
    второе();
    третье();
конец иначе начало
```

```
    чтоТоДругое();
    четвёртое();
    пятое();
конец;
```

Инструкции в теле составного блока выполняются по порядку одна за другой. Если при выполнении некоторой инструкции возникло исключение, то следующие за ней инструкции не выполняются, и управление немедленно передаётся в секцию обработки исключений.

Секция обработки исключений, если она есть в блоке, располагается после всех инструкций, составляющих блок, и начинается ключевым словом **исключение** или **исключения**.

```
начало
    выполнитьОдно();
    выполнитьВторое();
    выполнитьТретье();
исключение
    сообщить("Возникла какая-то ошибка!");
окончание;
```

Если блок не имеет секции обработки исключений, то возникшее исключение всплывает в объемлющий блок, имеющий такую секцию — в т.ч., возможно, за пределы выполняемой подпрограммы. Если такой объемлющий блок вообще отсутствует, то выполнение программы завершается с кодом -1.

Секция обработки исключений может быть простой или детализированной.

Простая секция содержит одну или несколько любых инструкций. Если блок имеет простую секцию обработки исключений, то при возникновении в блоке любого исключения управление передаётся в эту секцию, имеющиеся в ней инструкции выполняются, и любое исключение считается подавленным, если его всплытие не было возобновлено.

Чтобы возобновить всплытие исключения, в секции обработки исключений (и только в ней) можно использовать инструкцию **бросить**.

Детализированная секция обработки исключений содержит одну или несколько инструкций перехвата исключений **когда-тогда** и не может содержать никаких других инструкций. Исключение, попавшее в такую секцию, считается подавленным в том случае, если оно соответствует какой-либо из инструкций перехвата; все остальные исключения всплывают в объемлющий блок.

```
начало
    ч := целое("Не число");
исключение
    когда ОшибкаКонвертации тогда ч := 0;
    когда любое тогда сообщить("Какая-то неожиданная ошибка!");
конец;
```

Составной блок может заканчиваться секцией гарантированного выполнения.

Если в блоке есть секция гарантированного выполнения, то перед выходом из блока — вне зависимости от того, возникло ли в нём исключение, или все инструкции были выполнены успешно — управление передаётся в эту секцию. Также управление передаётся в эту секцию при выходе из блока инструкциями **прервать**, **продолжить**, **вернуть** и **завершить**.

Секция гарантированного выполнения начинается ключевым словом **напоследок**. Эта секция должна располагаться в конце блока, после основной секции и секции обработки исключений.

```
скрытьКурсор();
начало
    чтоТоВыполнить();
    ещёЧтоТоВыполнить();
    вернуть 0;
исключение
    сообщить(текст_исключения());
напоследок
    показатьКурсор();
конец;
```

Для-начало — доступ к полям структур

Составной блок инструкций может начинаться с ключевого слова **для**, за которым указан путь к переменной или константе структурного типа, а вслед за ним — слово **начало**, тело составного блока и слово **конец** или **окончание**.

В таком составном блоке создаётся локальная область видимости, позволяющая напрямую обращаться к полям указанной структуры в пределах блока, например:

```
программа Пример;
тип
    Точка = структура
        г, в: целое;
    окончание;
переменная
    тчк: Точка;
для тчк начало
    г := 10; // присвоить значение полю тчк.г
    в := 20; // присвоить значение полю тчк.в
окончание.
```

Между словами **для** и **начало** могут быть указаны несколько переменных, перечисленных через запятую. В этом случае, если указанные структуры имеют поля с одинаковыми именами, то поля той структуры, что указана позже, скроют за собой одноимённые поля структур, перечисленных раньше.

Следует иметь в виду, что конструкция **для** не вычисляет никаких значений перед входом в блок. Вместо этого полный путь к значению структуры вычисляется каждый раз при обращении к её полям, как если бы при обращении к каждому полю был указан полный путь к нему. Поэтому, например, следующая программа будет работать правильно, хотя сомнительно выглядит с точки зрения структурного программирования:

```
программа ТакНеНадо;
типы
    Точка = структура
        г, в: целое;
    окончание;
    ЦветныеТочки = массив структур
        поз: Точка;
        цвет: целое;
    окончание;
переменная
    тчк: ЦветныеТочки;
    к: целое;
для тчк[к], тчк[к].поз начало
    длина(тчк, 10);
    для каждого к из тчк цикл /*для лучше написать здесь*/ начало
        г := к * 5;      // тчк[к].поз.г
```

```
        в := 10;         // тчк[к].поз.в
        цвет := $909090; // тчк[к].цвет
    конец;
окончание.
```

Когда-тогда — перехват исключений

Инструкция перехвата исключения может использоваться только в секции обработки исключений составного блока (т.е., после ключевого слова **исключение)** и не может соседствовать в ней с какими-то ещё инструкциями, кроме других инструкций перехвата.

Инструкция начинается ключевым словом **когда**, за которым следует одно из трёх:

- одно или несколько имён исключений через запятую;
- определение переменной исключения;
- ключевое слово **любое** (или **любой**, **любая**).

Далее следует ключевое слово **тогда**, а за ним тело инструкции перехвата — ровно одна инструкция (в т.ч., возможно, составной блок), которая будет выполнена в том случае, если подходящее исключение возникло при выполнении основной секции составного блока.

Чтобы просто подавить исключение, телом инструкции перехвата может быть инструкция **ничего**. Чтобы возобновить всплытие перехваченного исключения, используется инструкция **бросить**.

```
начало
    выполнитьОдно();
    выполнитьДругое();
исключения
    когда ОшибкаКонвертации тогда сообщить("Неправильный формат данных.");
    когда НеверныйИндекс, НеверныйКлюч тогда сообщить("Неправильный программист.");
    когда ош: МояОшибка тогда начало
        ош.текст := ош.текст ++ " Не волнуйтесь, так и было задумано! ;)";
        бросить;
    конец;
    когда любое тогда завершить 100;
конец;
```

Запрещено упоминать одно и то же исключение в нескольких инструкциях перехвата в пределах одной секции. Инструкция **когда любое** перехватывает любое возникшее исключение, если оно не упомянуто в других инструкциях в той же секции. Такая инструкция должна быть последней в секции.

Если в инструкции перехвата не объявлена переменная, а перечислены одно или несколько имён исключений, то тело инструкции выполняется при возникновении любого из них. При этом можно узнать, какое именно из исключений возникло, и получить текст сообщения об ошибке вызовом функций **имя_исключения()** и **текст_исключения()**, но нет возможности модифицировать текст или параметры исключения.

Для доступа к параметрам исключения нужно объявить переменную — указать её имя после ключевого слова **когда** и отделить двоеточием от имени исключения. Такая инструкция может перехватывать только один вид исключения.

Переменная, объявленная таким образом, представляет собой структуру, которая содержит:

- Поле **текст** строкового типа, содержащее текст сообщения об ошибке;
- Поля простых типов, соответствующие параметрам в определении исключения.

Эти поля получают свои значения при создании исключения и могут быть изменены при его перехвате. Следует иметь в виду, что если текст сообщения об ошибке содержал спецификаторы вида **%(имя)**, то в момент

перехвата поле **текст** содержит уже отформатированный текст, где все спецификаторы заменены на значения параметров. Изменение значений параметров исключения при его перехвате не приводит к повторному формированию текста сообщения об ошибке; для того, чтобы изменить текст сообщения, нужно присвоить новое значение полю **текст**.

Модификация текста и параметров перехваченного исключения имеет смысл только в том случае, если его всплытие будет возобновлено инструкцией **бросить**.

Ошибка — создание исключения

Эта инструкция применяется для того, чтобы прервать последовательность выполнения и создать всплывающее исключение.

Инструкция начинается ключевым словом **ошибка**, за которым следует имя исключения. Если для исключения определены параметры, то их значения должны быть перечислены через запятую в круглых скобках после имени исключения. Далее может следовать ключевое слово **сообщение**, а за ним выражение строкового типа — текст сообщения об ошибке.

Инструкции, следующие после инструкции **ошибка**, не выполняются. Вместо этого управление передаётся в секцию обработки исключений ближайшего составного блока. Если такой блок отсутствует, выполнение программы будет завершено с кодом -1.

```
исключение
    МояОшибка(п1, п2: целое; п3: строка) сообщение "Тут какая-то ошибка";
    ЕщёОшибка(стр, симв: целое);
начало
    ошибка ОшибкаВыполнения;
    // всё, что написано ниже, никогда не будет выполнено
    ошибка МояОшибка(10, 20, "тридцать");
    ошибка ЕщёОшибка(1, 2) сообщение "Ошибка в строке %(стр), символ %(симв).";
конец;
```

Если текст сообщения об ошибке был указан и в определении исключения, и при его создании, то последний имеет приоритет. Если текст не указан ни в определении, ни при создании исключения, то в качестве текста сообщения используется имя исключения.

Если для исключения определены параметры, то текст сообщения об ошибке может содержать спецификаторы вида **%(имя)**, где **имя** — имя параметра исключения. При создании исключения такие спецификаторы будут заменены на строковые представления значений соответствующих параметров.

Кроме того, значения параметров исключения могут быть получены в секции перехвата исключений составного блока, как это описано в разделе **«Когда-тогда — перехват исключений»**.

Бросить — воссоздание исключения

Эта инструкция записывается ключевым словом **бросить**. Её можно применять только в секции обработки исключений составного блока: она прерывает последовательность выполнения и воссоздаёт то же самое исключение, которое привело к передаче управления в данную секцию обработки исключений.

Инструкции, следующие после инструкции **бросить**, не выполняются. Вместо этого управление передаётся в секцию обработки исключений, расположенную в ближайшем объемлющем составном блоке. Если такой блок отсутствует, выполнение программы будет завершено с кодом -1.

Если параметры исключения или текст сообщения об ошибке были модифицированы в секции обработки исключений, то исключение будет воссоздано с новыми значениями.

Если-то-иначе — условное ветвление

Эта инструкция применяется для выполнения тех или иных действий в зависимости от некоторого условия.

Инструкция начинается ключевым словом **если**, после которого должно быть указано условие — значение или выражение логического типа. Далее следует ключевое слово **то**, а за ним — ровно одна инструкция, которая будет выполнена в том случае, если указанное условие имеет истинное значение.

Далее может следовать знак «;», либо ключевое слово **иначе**, а за ним ещё ровно одна инструкция и знак «;». Эта последняя инструкция будет выполнена в том случае, если указанное условие имеет ложное значение.

В обоих случаях ровно одна инструкция может представлять собой составной блок, содержащий сколько угодно инструкций.

```
начало
    если А < Б то
        вывести("А меньше Б.");
    если Н < М то
        вывести("Н меньше М.")
    иначе начало
        вывести("Н больше или равно М.");
        если Н > М то вывести("Всё-таки больше.");
    конец;
конец;
```

Если требуется перебрать несколько условий подряд, то инструкции ветвления можно располагать друг за другом:

```
начало
    если имя = "Петя" то вывести("Узнаю брата Петю!")
    иначе если имя = "Вася" то вывести("Узнаю брата Васю!")
    иначе если имя = "Коля" то вывести("Узнаю брата Колю!")
    иначе вывести("Что-то не узнаю...");
конец;
```

Здесь действует правило: каждое **иначе** относится к ближайшему **если**, расположенному выше по тексту исходного кода.

Во многих подобных случаях будет удобнее воспользоваться инструкцией многовариантного выбора.

Выбор-из — многовариантный выбор

Эта инструкция применяется для выполнения тех или иных действий в зависимости от выбора значения из нескольких предусмотренных вариантов.

Инструкция начинается ключевым словом **выбор**, за которым следует выбираемое значение — буквальное значение или выражение простого типа. Затем, если выбираемое значение имеет тип **дробное** или **момент**, может следовать ключевое слово **точность**, а за ним константное выражение дробного типа, определяющее точность сравнения. Далее следует слово **из**, а за ним тело инструкции и слово **окончание**.

Тело инструкции состоит из одного или нескольких описателей варианта выбора. Каждый описатель начинается с перечня подходящих значений, перечисленных через запятую, далее следует знак «:», а за ним ровно одна инструкция и знак «;». Эта инструкция будет выполнена в том случае, если данный вариант выбора окажется подходящим. Инструкция может представлять собой составной блок, содержащий сколько угодно вложенных инструкций.

Перечень подходящих значений может состоять из буквальных значений или константных выражений. Вариант выбора считается подходящим, если выбираемое значение точно совпадает с одним из значений, перечисленных

в описателе варианта (дробные значения и моменты должны совпадать с указанной точностью). Тело инструкции выбора не может содержать нескольких описателей с совпадающими значениями.

В последнем из описателей вместо перечня подходящих значений можно указать ключевое слово **иначе** (без двоеточия). Такой описатель будет считаться подходящим в любых случаях, если выше не нашлось никакого другого подходящего описателя.

```
программа Пример;
переменная
    имя: строка;
начало
    сообщить("Введите имя: ");
    ввести(имя);
    выбор имя из
        "Петя": вывести("Узнаю брата Петю!");
        "Вася": вывести("Узнаю брата Васю!");
        "Коля": вывести("Узнаю брата Колю!");
        "Оля", "Даша", "Вика": вывести("Узнаю сестрёнку!");
        "Таня": вывести("Привет, Таня :)");
        иначе вывести("Что-то не узнаю...");
    конец;
окончание.

программа ЕщёПример;
переменная
    N: дробное = 0.345;
начало
    выбор N точность 0.09 из
        0: вывести("Нуль");
        0.1: вывести("Одна десятая");
        0.2: вывести("Две десятых");
        0.3: вывести("Три десятых"); // совпадёт с указанной точностью
        0.4: вывести("Четыре десятых");
    конец;
окончание.
```

Для выбора из значений типа **дробное** или **момент** можно указать точность сравнения. Если указана ненулевая точность, то вариант выбора будет считаться подходящим, если модуль разности указанной константы и выбираемого значения не превышает указанного значения точности.

Примечание: следует быть осторожным при использовании этой инструкции с дробными числами и моментами без указания точности. Сравнение вещественных чисел на точное равенство может давать непредсказуемый результат в зависимости от операционной системы и аппаратного обеспечения.

Повторить-раз — простой цикл

Эта инструкция предназначена для выполнения одних и тех же действий несколько раз. Количество повторений (итераций) цикла вычисляется заранее.

Инструкция начинается ключевым словом **повторить**, за которым следует целочисленное выражение, определяющее количество итераций цикла. После этого следует слово **раз** (или **раза**) и ровно одна инструкция, составляющая тело цикла. Тело цикла может быть составным блоком, содержащим сколько угодно инструкций.

```
программа Пример;
начало
    повторить 10 раз вывести("А"); // будет выведено ААААААААА
окончание.
```

Значение, определяющее количество итераций, вычисляется один раз перед началом цикла. Это значение должно быть положительным, иначе тело цикла не будет выполнено ни разу. Изменение в теле цикла каких-либо данных, от которых зависит это значение, не приведёт к изменению количества итераций.

Выполнение цикла можно прервать инструкцией **прервать**. Инструкция **продолжить** прерывает текущую итерацию цикла и немедленно начинает следующую — при том условии, что количество итераций ещё не достигло максимального значения.

Для-от-до-цикл — цикл со счётчиком

Эта инструкция предназначена для выполнения одних и тех же действий несколько раз. Количество повторений (итераций) цикла вычисляется заранее. В теле цикла доступна целочисленная переменная — счётчик, значение которой увеличивается (или уменьшается) на 1 после каждой итерации цикла.

Инструкция начинается ключевым словом **для**, за которым следует имя переменной — счётчика цикла. Далее следует слово **от**, за ним выражение, определяющее начальное значение счётчика, а затем слово **до** и выражение, определяющее конечное значение счётчика. После этого следует слово **цикл** и ровно одна инструкция, составляющая тело цикла. Тело цикла может быть составным блоком, содержащим сколько угодно инструкций. Перед словом **цикл** допускается слово **обратный**.

Счётчиком цикла может быть только локальная переменная целого типа: она должна быть объявлена в разделе определений той программы или подпрограммы, в теле которой выполняется цикл. Выражения начального и конечного значения счётчика должны возвращать целое. Значения этих выражений вычисляются один раз перед первой итерацией цикла. Изменение в теле цикла каких-либо данных, влияющих на значения этих выражений, не приведёт к изменению количества итераций.

```
программа Пример;
переменная
    цвета: массив целых;
    ц: целое;
начало
    длина(цвета, 16);
    для ц от 0 до 15 цикл цвета[ц] :=  ц;
    для ц от длина(цвета)-1 до 0 обратный цикл начало
        цветФона(цвета[ц]);
        вывести("Цв", строка(ц));
    конец;
окончание.
```

Перед первой итерацией счётчику цикла присваивается начальное значение. Затем выполняется тело цикла. После этого значение счётчика увеличивается на 1, и тело цикла выполняется снова. Так происходит до тех пор, пока значение счётчика меньше или равно конечному значению. После этого цикл завершается, и начинают выполняться инструкции, следующие после цикла.

Значение переменной-счётчика цикла можно менять в теле цикла, но это не приведёт изменению количества итераций. Переменная-счётчик сохранит присвоенное ей значение до конца текущей итерации, после чего ей будет присвоено следующее заранее вычисленное значение, так что на входе в каждую следующую итерацию счётчик будет иметь значение на 1 большее, чем на входе в предыдущую.

Если начальное значение счётчика больше конечного, тело цикла не будет выполнено ни разу. Если начальное и конечное значения равны, тело цикла будет выполнено один раз.

Если перед ключевым словом **цикл** указано слово **обратный**, то цикл выполняется в обратном порядке: при каждой итерации значение счётчика уменьшается на единицу. В этом случае начальное значение счётчика должно быть больше или равно конечному.

Выполнение цикла можно прервать инструкцией **прервать**. Инструкция **продолжить** прерывает текущую итерацию цикла и немедленно начинает следующую — при том условии, что счётчик ещё не равен конечному значению.

После выхода из цикла переменная-счётчик сохраняет значение, присвоенное ей на последней выполненной итерации.

Для-каждого-цикл — цикл по набору данных

Эта инструкция предназначена для выполнения одних и тех же действий для каждого элемента из набора данных. Набором данных может служить массив, словарь или строка — в последнем случае элементом набора является каждый символ в строке.

Инструкция начинается ключевыми словами **для каждого**, за которыми следует имя переменной-итератора. Переменная-итератор должна быть объявлена в локальной области видимости — т.е., в разделе определений той программы или подпрограммы, в теле которой выполняется цикл.

Далее следует ключевое слово **из** и имя переменной, содержащей набор данных — строку, массив или словарь. Перед словом **из** допустимо слово **от**, а за ним выражение, определяющее начальную позицию итератора в наборе данных.

Затем следует слово **цикл** и ровно одна инструкция, составляющая тело цикла. Тело цикла может быть составным блоком, содержащим сколько угодно инструкций. Перед словом **цикл** допускается слово **обратный**.

Для массивов и строк переменная-итератор должна быть целочисленного типа. Для массива она означает индекс элемента в массиве, а для строки — номер байта в строке, с которого начинается очередной символ. Байты в строке и элементы в массиве нумеруются начиная с нуля.

Для словарей тип переменной-итератора должен точно соответствовать типу ключа словаря — при каждой итерации цикла этой переменной присваивается очередное значение ключа.

```
программа ЦиклПоСтроке;
переменные
    Б: целое;
    С: строка;
    смв: символ;
начало
    сообщить("Введите строку: ");
    ввести(С);
    для каждого Б из С цикл начало
        смв := С[Б];
        вывести(Б, ": ", смв, НС);
    конец;
окончание.

программа ЦиклПоМассиву;
переменные
    М: массив строк;
    кво, К: целые;
начало
    сообщить("Введите количество: ");
    ввести(кво);
    длина(М, кво);
    для каждого К из М цикл начало
        сообщить("Введите элемент ", К, ": ");
        ввести(М[К]);
    конец;
```

```
    сообщить("Значения в обратном порядке:", НС);
    для каждого К из М обратный цикл вывести(М[К], НС);
окончание.

программа ЦиклПоСловарю;
переменные
    с: строка;
    сл: словарь строк ключ строка = {
        "Иванов"   : "Иванов Иван Иванович",
        "Петров"   : "Петров Пётр Петрович",
        "Сидоров"  : "Сидоров Сидор Сидорович",
        "Николаев" : "Николаев Николай Николаевич",
        "Захаров"  : "Захаров Захар Захарович"};
начало
    для каждой с из сл от "Н" цикл вывести(сл[с], НС);
окончание.
```

Выражение, следующее за ключевым словом **от**, определяет тот элемент набора данных, с которого начнётся цикл. Тип значения этого выражения должен соответствовать типу переменной-итератора.

- Для строки — выражение должно возвращать целое число, означающее номер байта в строке, соответствующего началу символа, с которого начнётся цикл.
- Для массива — выражение должно возвращать целое число, означающее индекс элемента массива, с которого начнётся цикл.
- Для словаря — если в словаре есть ключ, равный значению выражения, то цикл начнётся с этого ключа. Если такого ключа нет, то цикл начнётся с ближайшего следующего значения ключа: прямой цикл с ближайшего большего значения, а обратный цикл с ближайшего меньшего значения.

Если перед ключевым словом **цикл** указано слово **обратный**, то цикл выполняется в обратном порядке:

- Для строки — символы перебираются от конца строки к началу.
- Для массива — элементы перебираются от большего индекса к меньшему.
- Для словаря — ключи перебираются в порядке убывания.

Выполнение цикла можно прервать инструкцией **прервать**. Инструкция **продолжить** прерывает текущую итерацию цикла и немедленно начинает следующую — при том условии, что цикл ещё не дошёл до конца набора данных.

Значение переменной-итератора можно менять в теле цикла, но это не приведёт изменению количества итераций. Переменная-итератор сохранит присвоенное ей значение до конца текущей итерации, после чего ей будет присвоено следующее заранее вычисленное значение.

После выхода из цикла переменная-итератор сохраняет значение, присвоенное ей на последней выполненной итерации.

Примечание: модификация набора данных в теле цикла — добавление или удаление элементов массивов и словарей, вставка и удаление символов строки — может приводить к нежелательным последствиям: к возникновению исключений, неполному обходу набора данных и другим неприятностям.

Пока-цикл, цикл-пока — цикл с условием

В Клаусе есть две разновидности условного цикла: цикл с предусловием и цикл с постусловием.

Цикл с предусловием начинается ключевым словом **пока**, за которым следует выражение логического типа, слово **цикл**, тело цикла и знак «**;**».

Цикл с постусловием начинается ключевым словом **цикл**, за которым следует тело цикла, а после него слово **пока**, выражение логического типа и знак «**;**».

Телом цикла в обоих случаях должна быть ровно одна инструкция, в т.ч. это может быть составной блок, содержащий сколько угодно инструкций.

```
переменная
    д: дробное = 0;
начало
    пока д < 10 цикл д += 1.5;
    цикл д -= 1.2 пока д >= 0;
окончание.
```

Для цикла с предусловием значение логического выражения вычисляется перед первой итерацией. Если условие ложно, тело цикла не выполняется ни разу, а управление передаётся инструкции, следующей после цикла. Если условие истинно, тело цикла выполняется, после чего снова вычисляется значение логического выражения, и так происходит до тех пор, пока значение выражения не станет ложным.

Тело цикла с постусловием выполняется по меньшей мере один раз. После этого вычисляется значение логического выражения, и если оно истинно, тело цикла выполняется снова. Так происходит до тех пор, пока значение выражения не станет ложным, после чего управление передаётся инструкции, следующей после цикла.

Выполнение цикла можно прервать инструкцией **прервать**. Инструкция **продолжить** прерывает текущую итерацию цикла и немедленно переходит к следующей проверке условия, после которой цикл либо продолжается, либо завершается.

Прервать — прерывание цикла

Эта инструкция записывается ключевым словом **прервать**. Она прерывает выполнение цикла: инструкции в теле цикла, следующие после неё, не выполняются, и управление немедленно передаётся к ближайшей следующей инструкции после цикла.

Если в теле цикла есть составной блок, имеющий секцию **напоследок**, и в этом блоке была выполнена инструкция **прервать**, то инструкции в секции **напоследок** будут выполнены перед выходом из цикла.

Продолжить — прерывание итерации цикла

Эта инструкция записывается ключевым словом **продолжить**. Она прерывает выполнение текущей итерации цикла и немедленно начинает следующую. Инструкции в теле цикла, следующие после инструкции **продолжить**, не выполняются. Далее, в зависимости от разновидности цикла:

- **повторить-раз** и **для-от-до**: новая итерация начинается в том случае, если прерванная итерация не была последней.
- **пока-цикл** (с предусловием): перед началом новой итерации вычисляется значение условного выражения, и новая итерация начинается, если оно истинно.
- **цикл-пока** (с постусловием): новая итерация начинается безусловно. Значение условного выражения не вычисляется.
- **для-каждого**: новая итерация начинается для следующего элемента в наборе данных, если ещё остались необработанные элементы.

Если в теле цикла есть составной блок, имеющий секцию **напоследок**, и в этом блоке была выполнена инструкция **продолжить**, то инструкции в секции **напоследок** будут выполнены перед завершением итерации.

Встроенная библиотека

Клаус — стандартная библиотека

Модуль **Клаус**, содержащий определения стандартной библиотеки языка, подключается к каждой программе автоматически. Определения и подпрограммы этого модуля по умолчанию доступны любой программе.

Константы

```
константы
    НС = ""№0Д№0А; // новая строка
    Таб = №09;     // символ табуляции
    ВК = №0Д;      // символ возврата каретки
    ПС = №0А;      // символ перевода строки
    КФ = №1А;      // символ конца файла

константы
    МинЦелое = -9223372036854775808;    // Минимальное значение целого числа
    максЦелое = 9223372036854775807;    // Максимальное значение целого числа
    минДробное = 2.2250738585072E-308;  // Минимальное значение дробного числа
    максДробное = 1.79769313486232E308; // Максимальное значение дробного числа
    Pi = 3.14159265358979;              // Значение числа Пи
    Пи = 3.14159265358979;              // Значение числа Пи
```

Исключения

В стандартной библиотеке Клауса определены следующие исключения:

ОшибкаВводаВывода - возникает при ошибках чтения и записи данных в стандартные потоки ввода-вывода или файлы, при неуспешном завершении функций управления файлами и в других подобных случаях.

ОшибкаКонвертации - возникает при ошибках преобразования значений из текстового формата в двоичный и обратно.

НеверныйИндекс - возникает при обращении к несуществующему индексу в массиве.

НеверныйКлюч - возникает при обращении к несуществующему ключу в словаре.

НеверныйСимвол - возникает при ошибках, связанных с обращением к неверному байту в строке.

НеверноеИмя - возникает при обращении к несуществующему идентификатору.

НеверноеЧисло - возникает при ошибках, связанных с обработкой дробных чисел.

НеверныйТип - возникает при ошибках, связанных с несовместимостью типов данных.

НеверныйСинтаксис - возникает при синтаксических ошибках.

ОшибкаВыполнения - прочие ошибки программы или операционной системы.

ВнутренняяОшибка - ошибки, допущенные разработчиками Клауса. При возникновении такой ошибки авторы будут благодарны за добавление сообщения о проблеме в репозиторий проекта. Заранее спасибо!

Все предопределённые исключения не принимают параметров. Программист может создать любое из перечисленных исключений по своему усмотрению.

Базовый ввод-вывод

Особенности поведения функций ввода-вывода могут различаться в зависимости от реализации терминала, используемого для запуска программы.

Процедура вывести()

```
процедура вывести(вх арг0, арг1, арг2, ...);
```

Выводит значения, переданные в параметрах, в стандартный поток вывода. По умолчанию поток вывода направлен на экран терминала, но он может быть переназначен средствами операционной системы при запуске программы.

Допускает любое количество параметров простых типов. Перед записью в поток значения параметров преобразуются в строковые прямым приведением типа.

При ошибках может вызывать исключение **`ОшибкаВводаВывода`**.

Примечание: для перевода строки нужно передать в параметре символ LF (**`№0А`**) или константу **`НС`**.

Процедура сообщить()

```
процедура сообщить(вх арг0, арг1, арг2, ...);
```

Выводит значения, переданные в параметрах, в стандартный поток сообщений об ошибках. По умолчанию этот поток направлен на экран терминала, но он может быть переназначен средствами операционной системы при запуске программы.

Допускает любое количество параметров простых типов. Перед записью в поток значения параметров преобразуются в строковые прямым приведением типа.

При ошибках может вызывать исключение **`ОшибкаВводаВывода`**.

Примечание: для перевода строки нужно передать в параметре символ LF (**`№0А`**) или константу **`НС`**.

Функция ввести()

```
функция ввести(вв арг0, арг1, арг2, ...): целое;
```

Читает значения переданных параметров из стандартного потока ввода и возвращает количество прочитанных значений. Если значения всех параметров прочитаны успешно, то функция вернёт число, равное количеству переданных параметров.

По умолчанию поток ввода принимает пользовательский ввод с клавиатуры. В этом случае, если в потоке не хватает данных для чтения всех переданных параметров, то терминал будет переведён в режим ввода, чтобы пользователь мог ввести необходимые данные и нажать клавишу Enter.

Поток ввода может быть переназначен на файл при запуске программы. В этом случае функция будет последовательно читать данные из файла, а если в файле не хватает данных, то недостающие параметры останутся непрочитанными, и функция вернёт количество фактически прочитанных значений.

При вызове функции ей можно передать любое количество параметров простых типов, за исключением типа **`объект`**. Для каждого из переданных параметров, в зависимости от его типа, функция выполняет следующие действия:

- **Символ:** считывает из потока ровно один символ и присваивает его переданному параметру. Если в потоке есть несколько символов, то считывается только первый из них, а все остальные остаются в

потоке для последующего чтения. В этом случае из потока считываются в т.ч. символы окончания строки, которые терминал записывает в поток при нажатии клавиши Enter.

- **Строка:** считывает из потока все символы вплоть до окончания строки или конца потока и присваивает полученную строку переданному параметру. Окончанием строки считается символ **#0D**, символ **#0A** или комбинация **#0D#0A**. Символы **#0D** и **#0A** удаляются из потока, но не попадают в возвращаемое строковое значение.
- **Целое, Дробное, Момент, Логическое:** пропускает идущие друг за другом символы окончания строки, пробелы и табуляции (**#0D, #0A, #20, #09**), если они есть в начале потока. Затем считывает из потока все символы вплоть до ближайшего разделителя — пробела, табуляции, окончания строки или конца потока — и приводит прочитанную строку к требуемому типу данных. При этом пробелы и табуляции, расположенные в потоке после прочитанного значения, будут удалены из потока, вплоть до ближайшего непробельного символа или до начала следующей строки.

Символы **#04** и **#1A** считаются концом потока. Встретив в потоке любой из этих символов, функция прекращает работу, так что значения всех оставшихся параметров прочитаны не будут.

Функция считывает из потока ровно столько символов, сколько требуется для заполнения всех переданных параметров. Если пользователь ввёл больше данных, чем нужно, то лишние символы остаются в потоке и будут прочитаны при следующих вызовах функции. Если функция вызвана без параметров, она считывает из потока одно строковое значение.

При ошибках преобразования значений из их текстовых представлений создаётся исключение **ОшибкаКонвертации**, при ошибках чтения из файла — исключение **ОшибкаВводаВывода**.

Примечание 1: Чтение и запись значений моментов может быть взаимно неоднозначна, т.к. при приведении момента к строке текстовые представления даты и времени разделяются пробелом. Это может приводить к необходимости считывать момент в два параметра, расположенных друг за другом — сначала дату, затем время — и суммировать полученные значения.

Примечание 2: Не рекомендуется использовать эту функцию в сквозном режиме терминала, т.к. её поведение в этом режиме может быть неочевидным.

Примечание 3: Следует с осторожностью комбинировать посимвольное чтение потока ввода с чтением значений других типов. В тех случаях, когда программа должна предложить пользователю ввести с клавиатуры один символ и нажать Enter, рекомендуется считывать из потока строковое значение.

Массивы

Конструкции есть() и нету()

```
переменная
    м: массив целых;
начало
    длина(м, 10);
    если есть(м[10]) то вывести("да");
    если нету(м[10]) то вывести("нет"); // будет выведено "нет"
окончание;
```

Эти конструкции выглядят как функции, но фактически ими не являются. Конструкция **есть()** имеет истинное значение, если существует указанный в скобках элемент массива — т.е., если указанный индекс принадлежит диапазону от 0 до длины массива минус 1. Конструкция **нету()** имеет истинное значение в противоположном случае.

Функция длина()

```
функция длина(вх м: массив): целое;
функция длина(вв м: массив; вх размер: целое): целое;
```

Возвращает количество элементов в переданном массиве.

Если передан параметр **размер**, то функция установит для переданного массива указанную длину и вернёт новое значение длины. В этом случае в первом параметре должна быть передана переменная.

Если новая длина больше старой, то прежнее содержимое массива сохраняется, а новые элементы добавляются в конец массива и инициализируются нулевыми начальными значениями. Если новая длина меньше старой, то сохраняются прежние значения элементов от нуля до новой длины, а остальные элементы уничтожаются.

Процедура добавить()

```
процедура добавить(вв м: массив; вх элемент);
```

Добавляет переданный элемент в конец массива. Длина массива увеличивается на 1.

Процедура вставить()

```
процедура вставить(вв м: массив; вх индекс: целое; вх элемент);
```

Вставляет в массив переданный элемент. Если переданный **индекс** равен длине массива, то элемент добавляется в конец массива; если меньше, то элемент вставляется в переданную позицию, а все элементы, начиная от переданного индекса, включительно, сдвигаются в сторону увеличения индекса. Длина массива увеличивается на 1.

Если передан недопустимый индекс, создаётся исключение **НеверныйИндекс**.

Процедура удалить()

```
процедура удалить(вв м: массив; вх индекс: целое);
процедура удалить(вв м: массив; вх индекс, кво: целое);
```

Удаляет элементы из массива.

Удаляет переданное количество элементов, начиная от указанного индекса, включительно. Если параметр **кво** не передан, удаляет один элемент. Если после удалённых элементов остались ещё элементы, то они сдвигаются в сторону уменьшения индекса.

Если передан недопустимый индекс или слишком большое количество, создаётся исключение **НеверныйИндекс**.

Процедура очистить()

```
процедура очистить(вв м: массив);
```

Удаляет все элементы из массива.

Словари

Конструкции есть() и нету()

```
переменная
    сл: словарь целых ключ строка;
начало
    сл["первый"] := 10;
    если есть(сл["первый"]) то вывести("да");  // будет выведено "да"
    если нету(сл["первый"]) то вывести("нет");
окончание;
```

Эти конструкции выглядят как функции, но фактически ими не являются. Конструкция **`есть()`** имеет истинное значение, если существует указанный в скобках элемент словаря — т.е., если в словаре есть указанный ключ. Конструкция **`нету()`** имеет истинное значение в противоположном случае.

Функция длина()

```
функция длина(вх сл: словарь): целое;
```

Возвращает количество элементов в переданном словаре.

Процедура добавить()

```
процедура добавить(вв сл: словарь; вх ключ);
процедура добавить(вв сл: словарь; вх ключ, значение);
```

Добавляет в словарь переданный **`ключ`**. Если передано **`значение`**, то оно присваивается добавленному ключу; если нет, то новый ключ получает нулевое начальное значение.

Если указанный ключ уже существует в словаре, то новое значение присваивается существующему ключу, либо не меняется, если новое значение не передано.

Процедура вставить()

```
процедура вставить(вв сл: словарь; вх ключ, значение);
```

Добавляет в словарь переданную пару ключ-значение или присваивает новое значение переданному ключу, если тот уже существует в словаре. Эквивалентным вызову этой функции является присваивание:

```
сл[ключ] := значение;
```

Процедура удалить()

```
процедура удалить(вв сл: словарь; вх ключ);
```

Удаляет из словаря переданный ключ и соответствующее ему значение. Если указанный ключ отсутствует в словаре, создаётся исключение **`НеверныйКлюч`**.

Процедура очистить()

```
процедура очистить(вв сл: словарь);
```

Удаляет из словаря все ключи и соответствующие им значения.

Числа

Функция нечисло()

```
функция нечисло(вх число: дробное): логическое;
```

Возвращает истину, если переданный параметр имеет значение **нечисло**.

Функция конечно()

```
функция конечно(вх число: дробное): логическое;
```

Возвращает истину, если переданный параметр имеет конечное численное значение. Возвращает ложь, если значение переданного параметра — **нечисло** или бесконечность с любым знаком.

Функция округл()

```
функция округл(вх число: дробное): целое;
```

Округляет переданное значение до ближайшего целого и возвращает полученное целое число. Если передано **нечисло** или бесконечность, возникает исключение **`НеверноеЧисло`**.

Функция окр()

```
функция окр(вх число: дробное; вх знаков: целое): дробное;
```

Возвращает дробное число, округлённое до указанного количества **`знаков`** после запятой. Количество **`знаков`** может быть отрицательным — в этом случае число будет округлено до десятков, сотен и т.п.

Функция цел()

```
функция цел(вх число: дробное): дробное;
```

Возвращает целую часть переданного числа.

Функция дроб()

```
функция дроб(вх число: дробное): дробное;
```

Возвращает дробную часть переданного числа.

Функция sin()

```
функция sin(вх число: дробное): дробное;
```

Возвращает синус переданного числа.

Функция cos()

```
функция cos(вх число: дробное): дробное;
```

Возвращает косинус переданного числа.

Функция tg()

```
функция tg(вх число: дробное): дробное;
```

Возвращает тангенс переданного числа.

Функция arcsin()

```
функция arcsin(вх число: дробное): дробное;
```

Возвращает арксинус переданного числа.

Функция arccos()

```
функция arccos(вх число: дробное): дробное;
```

Возвращает арккосинус переданного числа.

Функция arctg()

```
функция arctg(вх число: дробное): дробное;
```

Возвращает арктангенс переданного числа.

Функция ln()

```
функция ln(вх число: дробное): дробное;
```

Возвращает натуральный логарифм переданного числа.

Функция exp()

```
функция exp(вх число: дробное): дробное;
```

Возвращает экспоненту переданного числа — т. е., число, возведённое в степень *e*.

Функция случайное()

```
функция случайное(вх макс: целое): целое;
```

Возвращает случайное число в диапазоне от 0 включительно до переданного значения **`макс`**, но не включая **`макс`**. Генератор псевдослучайных чисел инициализируется при первом обращении к функции.

Моменты

Функция дата()

```
функция дата(): момент;
функция дата(вх м: момент): момент;
```

Возвращает значение переданного момента с обнулённым временем — т. е., только дату. Если параметр не передан, возвращает текущую дату.

Функция время()

```
функция время(): момент;
функция время(вх м: момент): момент;
```

Возвращает значение переданного момента с обнулённой датой — т. е., только время. Если параметр не передан, возвращает текущее время.

Функция сейчас()

```
функция сейчас(): момент;
```

Возвращает значение текущего момента — дату и время.

Строки

Во всех строковых функциях стандартной библиотеки Клауса, если иное не оговорено явно, целочисленные значения индексов и размеров имеют семантику байтов, а не символов — как если бы строка представляла собой не последовательность символов переменной длины, а массив однобайтовых целых чисел.

Байты в строке нумеруются начиная с нуля: первый байт имеет индекс 0, а последний имеет индекс на единицу меньший, чем длина строки.

Библиотечные функции поддерживают внутреннюю целостность строки, не допуская появления в ней некорректных символов, и вызывают исключение **`НеверныйСимвол`** в случае ошибок. Например, такое исключение возникнет при вставке подстроки в позицию, соответствующую середине многобайтового символа, при удалении из строки части многобайтового символа, а также в других аналогичных ситуациях.

Конструкции есть() и нету()

```
переменная
    с: строка;
начало
    с := "АБВГ";
    если нету(с[10]) то вывести("нет"); // индекс за пределами длины строки
    если нету(с[3]) то вывести("нет");  // индекс не указывает на начало символа
    если есть(с[4]) то вывести("да");   // будет выведено "да"
окончание;
```

Эти конструкции выглядят как функции, но фактически ими не являются. Конструкция **`есть()`** имеет истинное значение, если существует указанный в скобках символ в строке — т.е., если указанный индекс принадлежит диапазону от 0 до длины строки минус 1 и указывает на начало символа. Конструкция **`нету()`** имеет истинное значение в противоположном случае.

Функция длина()

```
функция длина(вх стр: строка): целое;
функция длина(вв стр: строка; вх размер: целое): целое;
```

Возвращает длину переданной строки в байтах. Следует иметь в виду, что количество байтов в строке не равняется количеству символов, т. к. в кодировке UTF-8 символ может занимать от 1 до 4 байтов.

Если передан параметр **размер**, то функция установит для переданной строки указанную длину в байтах и вернёт новое значение длины. В этом случае в первом параметре должна быть передана строковая переменная.

Если новая длина больше старой, то прежнее содержимое строки сохраняется, и строка дополняется справа пробелами (**№20**) до указанной длины.

Если новая длина меньше старой, то прежнее содержимое строки обрезается справа до указанной длины. В этом случае должна быть указана такая новая длина, чтобы последний символ в строке, если он занимает несколько байтов, не оказался разбит на части — иначе возникнет ошибка **НеверныйСимвол**.

Пример

```
переменная
    стр: строка;
начало
    // Буквы кириллицы занимают в кодировке UTF-8
    // по два байта, а символ пробела - один байт.
    стр := "АБВГ";
    длина(стр, 10); // стр = "АБВГ  "
    длина(стр, 6);  // стр = "АБВ"
    длина(стр, 5);  // ошибка
окончание.
```

Функция симв()

```
функция симв(вх стр: строка): символ;
функция симв(вх стр: строка; вх индекс: целое): символ;
```

Возвращает символ, расположенный в переданной позиции строки **стр**. Строка не должна быть пустой; переданный индекс должен быть в пределах строки и должен соответствовать началу символа в строке, иначе возникнет исключение **НеверныйСимвол**.

Эквивалентным вызову этой функции является обращение к символу с указанием индекса в квадратных скобках: **стр[индекс]**.

Если параметр **индекс** не указан, он считается равным 0, и функция возвращает первый символ в строке.

Функция след()

```
функция след(вх стр: строка; вх индекс: целое): целое;
функция след(вх стр: строка; вх индекс, кво: целое): целое;
функция след(вх стр: строка; вх индекс, кво: целое; вых симв: целое): целое;
```

Возвращает позицию в байтах, соответствующую началу следующего символа в строке, считая от символа, переданного в параметре **индекс**. Значение переданного индекса должно соответствовать началу символа в строке, иначе возникнет исключение **НеверныйСимвол**.

В параметре **кво** можно передать количество символов, на которое необходимо сместиться: 1 — следующий символ, 2 — через один символ, и т. п. Если переданное количество превышает число оставшихся (справа от **индекс**) символов в строке, то функция вернёт длину строки в байтах.

Если передан параметр **симв**, то функция запишет в него фактически пройденное количество символов. Таким образом, чтобы посчитать общее количество символов в строке, можно передать 0 в параметре **индекс** и **максЦелое** в параметре **кво**.

Функция пред()

```
функция пред(вх стр: строка; вх индекс: целое): целое;
функция пред(вх стр: строка; вх индекс, кво: целое): целое;
функция пред(вх стр: строка; вх индекс, кво: целое; вых симв: целое): целое;
```

Возвращает позицию в байтах, соответствующую началу предыдущего символа в строке, считая от символа, переданного в параметре **индекс**. Для получения позиции последнего символа в строке в параметре **индекс** нужно передать значение, большее или равное длине строки в байтах. Если переданный индекс находится в пределах строки, то он должен соответствовать началу символа в строке, иначе возникнет исключение **НеверныйСимвол**.

В параметре **кво** можно передать количество символов, на которое необходимо сместиться: 1 — предыдущий символ, 2 — через один символ, и т. п. Если переданное количество превышает число оставшихся (слева от **индекс**) символов до начала строки, то функция вернёт 0.

Если передан параметр **симв**, то функция запишет в него фактически пройденное количество символов. Таким образом, чтобы посчитать общее количество символов в строке, можно передать **максЦелое** в параметрах **индекс** и **кво**.

Примечание: для строки из многобайтовых символов функция **след()** работает быстрее, чем **пред()**.

Функция часть()

```
функция часть(вх стр: строка; вх индекс: целое): строка;
функция часть(вх стр: строка; вх индекс, размер: целое): строка;
```

Возвращает часть строки указанного размера, начиная с указанного индекса. Значение переданного индекса должно соответствовать началу символа в строке, иначе возникнет исключение **НеверныйСимвол**.

Если параметр **размер** не передан или выходит за пределы строки, то функция вернёт часть строки от указанного индекса до конца строки. В противном случае, **размер** должен иметь такое значение, чтобы последний символ в возвращаемой подстроке, если он занимает несколько байтов, не оказался разбит на части — иначе возникнет исключение **НеверныйСимвол**.

Процедура добавить()

```
процедура добавить(вв стр1: строка; вх стр2: строка);
```

Добавляет значение **стр2** в конец строки **стр1**. Эквивалентным вызову этой функции является присваивание:

```
стр1 := стр1 ++ стр2;
```

Процедура вставить()

```
процедура вставить(вв стр: строка; вх индекс: целое; вх подстр: строка);
```

Вставляет строку **подстр** в указанную позицию строки **стр**, так что символы из строки **подстр** окажутся в строке **стр** непосредственно перед тем символом, на который указывает **индекс**. При этом длина строки **стр** увеличивается на длину строки **подстр**.

Если переданный **индекс** больше или равен длине **стр**, то подстрока будет добавлена к строке справа. В противном случае значение индекса должно соответствовать началу символа в строке, иначе возникнет исключение **НеверныйСимвол**.

Процедура удалить()

```
процедура удалить(вв стр: строка; вх индекс, число: целое);
```

Удаляет из строки указанное число байтов, начиная с переданного индекса, включительно.

Значение индекса должно соответствовать началу символа в строке, иначе возникнет исключение **НеверныйСимвол**.

Если строка содержит меньше байтов справа от индекса, чем передано в параметре **число**, то будут удалены все байты до конца строки. В противном случае, переданное число должно быть таким, чтобы последний удаляемый символ, если он занимает несколько байтов, не оказался разбит на части — иначе возникнет исключение **НеверныйСимвол**.

Процедура вписать()

```
процедура вписать(
  вв стр1: строка; вх индекс: целое; вх стр2: строка);
процедура вписать(
  вв стр1: строка; вх индекс: целое;
  вх стр2: строка; вх индекс2: целое);
процедура вписать(
  вв стр1: строка; вх индекс: целое;
  вх стр2: строка; вх индекс2, число: целое);
```

Копирует в **стр1** байты из **стр2**.

Из **стр2** копируется указанное **число** байтов, начиная от **индекс2**, включительно; эти байты размещаются в **стр1**, начиная от **индекс**, и заменяют собой байты, которые раньше находились в **стр1** в этих позициях. Если длины **стр1** не хватает, чтобы разместить все копируемые байты, то её длина увеличивается до необходимой.

Если не передано **число**, то копируются все байты от **индекс2** до конца **стр2**. Если не передан **индекс2**, то копируется вся **стр2** от начала до конца.

Все индексы и размеры должны иметь такие значения, чтобы никакой многобайтовый символ в обеих строках не оказался при копировании разбит на части, иначе возникнет ошибка **НеверныйСимвол**.

Пример

```
переменная
    стр1, стр2: строка;
начало
    // Буквы кириллицы занимают в кодировке UTF-8
    // по два байта, а арабские цифры - один байт.
```

```
    стр1 := "1234АБВГ";
    стр2 := "0000";
    вписать(стр1, 2, стр2);               // стр1 = 120000БВГ
    вписать(стр1, максЦелое, стр2, 0, 2); // стр1 = 120000БВГ00
    вписать(стр1, 7, стр2);               // ошибка
    вписать(стр1, 6, стр2, 0, 3);         // ошибка
окончание.
```

Функция найти()

```
функция найти(
    вх что, где: строка): целое;
функция найти(
    вх что, где: строка; вх начИдкс: целое): целое;
функция найти(
    вх что, где: строка; вх начИдкс: целое;
    вх учтРегистр: логическое): целое;
```

Ищет подстроку **что** в строке **где** и возвращает индекс первого байта найденной подстроки. Если подстрока не найдена, возвращает -1. Если передан **начИдкс**, поиск начинается с переданного индекса, включительно.

Если в параметре **учтРегистр** передано **нет**, то регистр символов при поиске не учитывается — т. е., прописные буквы считаются равными строчным. Если этот параметр опущен, функция выполняет поиск с учётом регистра.

Процедура заменить()

```
процедура заменить(вв стр: строка; вх индекс, число: целое; вх чем: строка);
```

Заменяет в переданной строке указанное **число** байтов, начиная с переданного индекса, на байты из строки **чем**. Длина строки изменяется соответственно.

Значения **индекс** и **число** должны быть такими, чтобы в результате замены никакой многобайтовый символ не оказался разбит на части, иначе возникнет ошибка **НеверныйСимвол**.

Функция формат()

```
функция формат(вх шаблон: строка; вх арг0, арг1, арг2, ...): строка;
```

Заменяет все спецификаторы в переданном шаблоне на значения переданных аргументов, преобразованные к строке в соответствии с указанным форматом, и возвращает полученную строку. В качестве аргументов допустимы значения простых типов или одномерные массивы простых типов. Для каждого из простых аргументов шаблон должен содержать один спецификатор, для каждого массива — столько спецификаторов, сколько элементов содержится в массиве.

Спецификаторы имеют следующий синтаксис:

```
'%'[индекс':']['-'][ширина]['.'точность]Тип
```

Символы в апострофах должны быть указаны в шаблоне буквально, без апострофов.

Элементы в квадратных скобках необязательны.

`'%'`	Начало спецификатора. Чтобы добавить в шаблон просто символ '%', его нужно удвоить: "%%".

`индекс':'`	Целое. Взять из списка параметров аргумент с указанным индексом. Если этот элемент не указан, берётся следующий по порядку аргумент.
`'-'`	Выравнивание вставляемого текста по левому краю поля. По умолчанию текст выравнивается по правому краю. Этот элемент работает только при условии, что указана ширина.
`ширина`	Целое. Ширина поля — т. е., минимальная длина вставляемой строки. Если строка короче, то она дополняется до указанной длины пробелами слева (по умолчанию) или справа, если указан «-».
`'.'точность`	Целое. Значение этого элемента зависит от типа спецификатора.
`Тип`	Тип спецификатора — буква в любом регистре. Перечень допустимых типов приведён ниже.

В случае ошибок функция создаёт исключение **`ОшибкаКонвертации`**. Это может происходить при ошибках в синтаксисе спецификаторов, при несоответствии типов спецификаторов типам аргументов, а также если передано недостаточно аргументов для всех спецификаторов.

Типы спецификаторов

«I» или «Ц» — целое число. Аргумент должен быть целым. Значение конвертируется в строковое представление целого числа. Если указана точность, то строка будет содержать не менее указанного количества цифр — если нужно, слева будут добавлены нули.

«E» (лат.) или «Э» — экспоненциальный формат. Аргумент может быть целым или дробным. Число приводится к строке в экспоненциальном (научном) формате. Точность определяет общее количество цифр в мантиссе числа, порядок записывается тремя цифрами.

«F» или «Ф» — формат с фиксированной точкой. Аргумент может быть целым или дробным. Точность определяет количество цифр после десятичной точки. Для слишком больших чисел используется экспоненциальный формат.

«G» или «О» (кир.) — общий числовой формат. Аргумент может быть целым или дробным. Число приводится к строке в фиксированном или экспоненциальном формате в зависимости от того, что получится короче. Точность определяет общее количество цифр в строковом представлении числа.

«N» или «Ч» — такой же формат числа, как с фиксированной точкой, но с добавлением разделителей групп разрядов.

«D» или «Д» — дата в стандартном формате: YYYY-MM-DD. Аргумент должен быть моментом. Точность не применяется.

«T» (лат.) или «В» (кир.) — время в стандартном формате: hh:mm:ss. Аргумент должен быть моментом. Точность не применяется.

«M» (лат.) или «М» (кир.) — дата и время в стандартном формате, разделённые пробелом. Аргумент должен быть моментом. Точность не применяется.

«S» или «С» (кир.) — строка. Аргумент может быть любого простого типа — строковые значения копируются, а значения других типов конвертируются в строку прямым приведением типа. Точность не применяется.

«X» (лат.) или «Ш» — шестнадцатеричное представление целого числа. Аргумент может быть целым, символом или объектом. Точность определяет минимальное количество цифр — если нужно, слева будут добавлены нули.

Пример

```
стр := формат("Меня зовут %s %s. Мне %i лет.", "Пётр", "Иванов", 15);
// стр = "Меня зовут Пётр Иванов. Мне 15 лет."
```

```
стр := формат("Меня зовут %1:s %0:s. Мне %2:i лет.", "Пётр", "Иванов", 15);
// стр = "Меня зовут Иванов Пётр. Мне 15 лет."

стр := формат("[%i]", 10);                  // стр = "[10]"
стр := формат("[%%]", 10);                  // стр = "[%]"
стр := формат("[%10i]", 10);                // стр = "[        10]"
стр := формат("[%.4i]", -10);               // стр = "[-0010]"
стр := формат("[%10.4ц]", -10);             // стр = "[     -0010]"
стр := формат("[%0:-10i]", 10);             // стр = "[10        ]"
стр := формат("[%x]", 1234567);             // стр = "[12D687]"
стр := формат("[%x]", -1234567);            // стр = "[FFFFFFFFFFED2979]"
стр := формат("[%0:10x]", 'Ё');             // стр = "[       401]"
стр := формат("[%0:10.4Ш]", 'Ё');           // стр = "[      0401]"
стр := формат("[%12.4e]", -1.234);          // стр = "[ -1.234E+000]"
стр := формат("[%12.4f]", -1.234);          // стр = "[     -1.2340]"
стр := формат("[%g]", 1.234);               // стр = "[1.234]"
стр := формат("[%12g]", -1.234);            // стр = "[      -1.234]"
стр := формат("[%0:-12.4G]", 1.234);        // стр = "[1.234       ]"
стр := формат("[%14d]", момент(44567.123)); // стр = "[    2022-01-06]"
стр := формат("[%-14д]", момент(44567.123)); // стр = "[2022-01-06    ]"
стр := формат("[%14t]", момент(44567.123)); // стр = "[      02:57:07]"
стр := формат("[%-14в]", момент(44567.123)); // стр = "[02:57:07      ]"
стр := формат("[%s]", "Привет!");           // стр = "[Привет!]"
стр := формат("[%14c]", "Привет!");         // стр = "[       Привет!]"
стр := формат("[%-14c]", "Привет!");        // стр = "[Привет!       ]"
```

Функция загл()

```
функция загл(вх стр: строка): строка;
```

Возвращает строку, в которой все строчные буквы заменены на прописные.

Функция строч()

```
функция строч(вх стр: строка): строка;
```

Возвращает строку, в которой все прописные буквы заменены на строчные.

Исключения

Функция имя_исключения()

```
функция имя_исключения(): строка;
```

Возвращает имя активного исключения. Эту функцию можно вызывать только в секции обработки исключений, иначе она вернёт пустую строку.

Функция текст_исключения()

```
функция текст_исключения(): строка;
```

Возвращает текст сообщения об ошибке активного исключения с уже добавленными в него значениями параметров исключения. Эту функцию можно вызывать только в секции обработки исключений, иначе она вернёт пустую строку.

Разное

Процедура уничтожить()

```
процедура уничтожить(вв о: объект);
```

Уничтожает переданный объект, выполняя все необходимые действия, связанные с его уничтожением, и присваивает переданному параметру значение **пусто**.

Если переданный объект не существует — т.е., он был ранее уничтожен, никогда не был создан, либо в параметре было передано значение **пусто**, — процедура будет выполнена без ошибок.

Процедура пауза()

```
процедура пауза(вх мсек: целое);
```

Приостанавливает выполнение программы на переданное число миллисекунд.

Функция имя_программы()

```
функция имя_программы(): строка;
```

Возвращает имя выполняемой программы. Имя программы указывается в заголовке программы после ключевого слова **программа** или **задача**.

Функция имя_практикума()

```
функция имя_практикума(): строка;
```

Возвращает имя учебного курса, если программа представляет собой решение задачи из Практикума. Имя учебного курса указывается в заголовке программы после ключевого слова **практикум**.

Терминал — управление терминалом

Модуль **Терминал** содержит библиотеку управления текстовым терминалом.

Функции управления терминалом реализованы путём выдачи ESC-последовательностей в стандартный поток вывода или поток сообщений об ошибках, в соответствии со стандартом ECMA-48. Эти функции могут не работать или особенности их работы могут различаться в зависимости от реализации терминала, используемого для запуска программы.

Константы

```
константы
    // Идентификаторы стандартных потоков
    идСтдВывод = 1;
    идСтдСообщ = 2;
    // Режимы терминала
    трКанон = нет;
    трСквозной = да;
константы
    // Стили шрифта
    стшЖирный  = 1; // жирный шрифт
```

```
стшКурсив  = 2; // курсив
стшПодчерк = 4; // одинарное подчёркивание
стшЗачерк  = 8; // зачёркнутый текст
```

Процедура режимТерминала()

```
процедура режимТерминала(вх поток: целое; вх сквозной: логическое);
```

Выбирает один из двух стандартных потоков, с которым будут работать все последующие вызовы функций управления терминалом, и устанавливает для него режим работы.

Параметр **поток** может иметь одно из двух значений: **идСтдВывод** — стандартный поток вывода, **идСтдСообщ** — стандартный поток сообщений об ошибках.

Если параметр **сквозной** имеет значение **трСквозной**, то терминал переводится в режим сквозного ввода — все нажатия клавиш немедленно помещаются в поток ввода и отключается эхо-печать введённых символов.

Процедура размерЭкрана()

```
процедура размерЭкрана(вх горз, верт: целое);
```

Устанавливает размеры окна терминала. Параметр **горз** означает количество символов по горизонтали, а параметр **верт** — количество строк по вертикали.

Процедура очиститьЭкран()

```
процедура очиститьЭкран();
```

Очищает экран, используя установленный цвет шрифта и фона.

Процедура очиститьСтроку()

```
процедура очиститьСтроку(вх где: целое);
```

Очищает указанную часть строки от позиции курсора, используя установленный цвет шрифта и фона. Если переданное значение **где** отрицательно, очищает строку слева от курсора, включая позицию курсора. Если **где** положительно, очищает строку справа от курсора, включая позицию курсора. Нулевое значение очищает всю строку целиком.

Процедура курсор()

```
процедура курсор(вх горз, верт: целое);
```

Устанавливает курсор в переданную позицию. Нумерация строк на экране и символов в строке начинается с нуля.

Процедура курсорГорз()

```
процедура курсорГорз(вх горз: целое);
```

Устанавливает курсор в позицию переданного символа в текущей строке, вертикальная координата курсора не меняется. Нумерация символов в строке начинается с нуля.

Процедура курсорВерт()

```
процедура курсорВерт(вх верт: целое);
```

Устанавливает курсор в переданную строку на экране, горизонтальная координата курсора не меняется. Нумерация строк начинается с нуля.

Процедура подвинутьКурсор()

```
процедура подвинутьКурсор(вх горз, верт: целое);
```

Сдвигает курсор на указанное количество строк и/или столбцов относительно текущей позиции. Положительные значения параметров **`горз`** и **`верт`** сдвигают курсор вправо и вниз, а отрицательные — влево и вверх.

Процедура запомнитьКурсор()

```
процедура запомнитьКурсор();
```

Запоминает текущую позицию курсора. Для возврата курсора в сохранённую позицию используется процедура **`вернутьКурсор()`**.

Процедура вернутьКурсор()

```
процедура вернутьКурсор();
```

Возвращает курсор в позицию, которая была ранее сохранена вызовом процедуры **`запомнитьКурсор()`**.

Процедура скрытьКурсор()

```
процедура скрытьКурсор();
```

Скрывает мигающий прямоугольник курсора. Скрытый курсор продолжает выполнять свои функции как указатель текущей позиции для вывода текста.

Процедура показатьКурсор()

```
процедура показатьКурсор();
```

Делает видимым мигающий прямоугольник курсора, если он был ранее скрыт.

Процедура цветФона()

```
процедура цветФона(вх цвет: целое);
```

Устанавливает цвет фона, который будет использоваться при следующих вызовах функций вывода текста, а также очистки строк.

Параметр **цвет** может принимать значения от 0 до 255 и обозначает номер цвета в 256-цветной палитре xTerm.

Процедура цветШрифта()

```
процедура цветШрифта(вх цвет: целое);
```

Устанавливает цвет шрифта, который будет использоваться при следующих вызовах функций вывода текста.

Параметр **цвет** может принимать значения от 0 до 255 и обозначает номер цвета в 256-цветной палитре xTerm.

Процедура стильШрифта()

```
процедура стильШрифта(вх стиль: целое);
```

Устанавливает стиль шрифта, который будет использоваться при следующих вызовах функций вывода текста.

В параметре **стиль** можно передать нуль, либо одно или несколько значений констант **стшXXXX**, скомбинированных с помощью побитового ИЛИ. Например, выражение **стшЖирный | стшКурсив** означает жирный курсив, а выражение **стшЗачерк | стшПодчерк | стшКурсив** означает зачёркнутый и подчёркнутый курсив.

Процедура сброситьАтрибуты()

```
процедура сброситьАтрибуты();
```

Сбрасывает ранее установленные цвет фона, цвет шрифта и стиль шрифта к значениям по умолчанию.

Функция цвет256()

```
функция цвет256(вх кр, зел, син: целое): целое;
```

Возвращает ближайший подходящий цвет из 256-цветной палитры xTerm, наиболее точно соответствующий переданным значениям цветовых составляющих в модели RGB.

Параметры **кр**, **зел** и **син** соответствуют красной, зелёной и синей составляющим и могут принимать значения от 0 до 255.

Функция прочестьСимвол()

```
функция прочестьСимвол(): символ;
```

Читает один символ из стандартного потока ввода и возвращает прочитанный символ.

В каноническом режиме терминала, если поток ввода пуст, функция ожидает, пока пользователь введёт с клавиатуры один или несколько символов и нажмёт **Enter**. После этого функция возвращает первый из введённых символов, а остальные символы остаются в потоке, так что следующие вызовы **прочестьСимвол()** выполняются немедленно. В том числе функция читает из потока и возвращает символ **№0A**, соответствующий нажатию клавиши **Enter**, а также любые другие управляющие символы (если терминал отправляет их в поток).

В сквозном режиме терминала, если поток ввода пуст, функция ожидает от пользователя нажатия любой клавиши и немедленно возвращает полученный символ. В этом режиме при нажатии **Enter** функция вернёт символ **№0D**.

При нажатии функциональных клавиш и клавиатурных комбинаций терминал может записывать в поток ввода управляющие последовательности, состоящие из нескольких символов. Для чтения таких последовательностей нужно вызвать функцию несколько раз подряд.

Если стандартный поток ввода перенаправлен на файл, эта функция последовательно возвращает символы из файла, а по достижении конца файла всегда возвращает символ **№1А**. При этом функция **естьСимвол()** всегда будет возвращать истинное значение.

Функция естьСимвол()

```
функция естьСимвол(): логическое;
```

Возвращает истинное значение, если в потоке ввода есть непрочитанные данные. В этом случае функция **прочестьСимвол()** не ожидает ввода от пользователя, а немедленно возвращает следующий символ из потока.

Если стандартный поток ввода перенаправлен на файл, эта функция всегда возвращает истинное значение.

Файлы — файловый ввод-вывод

Модуль **Файлы** содержит библиотеку файлового ввода-вывода.

Типы

```
тип
    // Структура информации о файле, заполняемая
    // функциями файлПервый() и файлСледующий()
    ФайлИнфо = структура
        имя: строка;     // имя файла
        размер: целое;   // размер в байтах
        атрибуты: целое; // атрибуты, объединённые побитовым ИЛИ
        возраст: момент; // дата и время последней модификации
    конец;
```

Константы

```
константы
    // типы файлов
    фтТекст    = 0; // текстовый файл
    фтДвоичный = 1; // файл с произвольным форматом данных

константы
    // режимы открытия файла
    фрдЧтение    = $01; // разрешить чтение файла
    фрдЗапись    = $02; // разрешить запись в файл

константы
    // режимы совместного доступа к файлу
    фсдИсключить = $10; // закрыть другим процессам доступ к файлу
    фсдЧтение    = $20; // разрешить другим процессам чтение файла
    фсдЛюбой     = $40; // разрешить другим процессам любые действия с файлом
```

```
константы
    // позиционирвание в файле
    фпОтНачала     = 0; // считая от начала файла
    фпОтКонца      = 1; // считая от конца файла
    фпОтносительно = 2; // считая от текущей позиции в файле

константы
    // атрибуты файлов
    фаТолькоЧтение = $00000001;
    фаСкрытый      = $00000002;
    фаСистемный    = $00000004;
    фаМеткаТома    = $00000008;
    фаКаталог      = $00000010;
    фаАрхивный     = $00000020;
    фаНормальный   = $00000080;
    фаВременный    = $00000100;
    фаСимвСсылка   = $00000400;
    фаСжатый       = $00000800;
    фаШифрованный  = $00004000;
    фаВиртуальный  = $00010000;
```

Функция файлСоздать()

```
функция файлСоздать(вх типФайла: целое; вх путь: строка; вх режим: целое): объект;
```

Создаёт новый файл с указанным именем в режиме чтения и записи. В параметре **путь** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, файл будет создан в текущем каталоге.

Если файл с таким именем уже существует, прежние данные из него будут удалены. При ошибках создаётся исключение **ОшибкаВводаВывода**.

Параметр **типФайла** означает тип данных создаваемого файла. Значение **фтТекст** означает текстовый файл в кодировке UTF-8, значение **фтДвоичный** — данные в произвольном формате. Функции **файлЗаписать()** и **файлПрочесть()** обращаются к файлу по-разному, в зависимости от значения этого параметра.

Параметр **режим** используется для указания режима совместного доступа к созданному файлу со стороны других процессов и может принимать одно из значений констант **фсдХХХХ**.

Возвращаемое значение представляет собой объект, который нужно передавать первым параметром во все остальные функции файлового ввода-вывода.

Успешно созданный файл должен быть закрыт вызовом процедуры **файлЗакрыть()**.

Функция файлОткрыть()

```
функция файлОткрыть(вх типФайла: целое; вх путь: строка; вх режим: целое): объект;
```

Открывает существующий файл с указанным именем и устанавливает позицию чтения-записи в начало файла. В параметре **путь** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, будет открыт файл в текущем каталоге.

Если файл с таким именем не существует, а также при других ошибках, создаётся исключение **ОшибкаВводаВывода**.

Параметр **типФайла** означает тип данных открываемого файла. Значение **фтТекст** означает текстовый файл в кодировке UTF-8, значение **фтДвоичный** — данные в произвольном формате. Функции **файлЗаписать()** и **файлПрочесть()** обращаются к файлу по-разному, в зависимости от значения этого параметра.

Параметр **режим** используется для указания режима доступа к файлу — только на чтение, только на запись, либо на чтение и запись, — а также режима совместного доступа к открытому файлу со стороны других процессов. Для этого используются константы **фрдXXXX** и **фсдXXXX**, которые можно комбинировать с помощью операции побитового «ИЛИ». Например, чтобы открыть файл для чтения и записи, запретив другим процессам запись в файл, но разрешив чтение, нужно передать в параметре **режим** выражение **фрдЧтение | фрдЗапись | фсдЧтение**.

Возвращаемое значение представляет собой объект, который нужно передавать первым параметром во все остальные функции файлового ввода-вывода.

Успешно открытый файл должен быть закрыт вызовом процедуры **файлЗакрыть()**.

Процедура файлЗакрыть()

```
процедура файлЗакрыть(вв файл: объект);
```

Закрывает файл, открытый вызовом функции **файлСоздать()** или **файлОткрыть()**, и присваивает значение **пусто** переданному объекту.

Эта функция обязательно должна быть вызвана после успешного создания или открытия файла, иначе при завершении программы возникнет исключение **ОшибкаВыполнения**. Поэтому вызов этой функции желательно помещать в секцию **напоследок** составного блока, например:

```
переменная
    ф: объект;
начало
    ф := файлОткрыть(фтТекст, "мой-файл.txt", фрдЗапись | фсдИсключить);
    начало
        файлРазмер(ф, 0);
        файлЗаписать(ф, "Вот такие числа:", НС);
        файлЗаписать(ф, 10, ' ', 20, ' ', 30, НС);
    исключение
        когда любое тогда сообщить("Ошибка при записи в файл.");
    напоследок
        файлЗакрыть(ф);
    конец;
исключение
    когда любое тогда сообщить("Ошибка при открытии файла.");
окончание.
```

Функция файлРазмер()

```
функция файлРазмер(вх файл: объект): целое;
функция файлРазмер(вх файл: объект; вх размер: целое): целое;
```

Возвращает размер переданного файла в байтах.

Если передан параметр **размер**, то устанавливает для файла переданный размер и возвращает новый размер файла. При этом позиция чтения-записи будет установлена в конец файла.

Функция файлПоз()

```
функция файлПоз(вх файл: объект): целое;
функция файлПоз(вх файл: объект; вх поз: целое): целое;
функция файлПоз(вх файл: объект; вх поз: целое; вх откуда: целое): целое;
```

Возвращает текущую позицию чтения-записи для переданного файла — номер байта, считая от начала файла. Первый байт в файле имеет номер 0.

Если передан параметр **поз**, то устанавливает новую позицию чтения-записи и возвращает установленное значение. Для записи в конец файла позиция должна указывать на байт, следующий за последним байтом в файле — т.е., быть равна размеру файла в байтах. Для текстовых файлов программист должен следить за тем, чтобы переданная позиция указывала на начало символа, в противном случае при попытке чтения возникнет исключение **ОшибкаВводаВывода**.

Параметр **откуда** определяет точку отсчёта для переданной позиции и может иметь одно из следующих значений:

- **фпОтНачала** — в **поз** передан номер байта, считая от начала файла. Это значение используется по умолчанию.
- **фпОтКонца** — в **поз** передан номер байта от конца файла. Значение 0 означает байт, следующий за концом файла, значение 1 — последний байт в файле, и т.п.
- **фпОтносительно** — в **поз** передано смещение относительно текущей позиции чтения-записи. Положительные значения перемещают позицию ближе к концу файла, отрицательные — к началу.

Процедура файлПрочесть()

```
процедура файлПрочесть(вх файл: объект; вых арг1, арг2, арг3, ...);
```

Читает данные из переданного файла, начиная с текущей позиции чтения-записи, и по порядку присваивает прочитанные значения переданным параметрам. Позиция чтения-записи сдвигается к концу файла на число прочитанных байтов. Если в файле недостаточно данных, чтобы прочесть все переданные параметры, возникает исключение **ОшибкаВводаВывода**.

Переданный файл должен быть открыт вызовом функции **файлОткрыть()** или **файлСоздать()**. Если файл был открыт как текстовый, то он должен содержать корректные символы в кодировке UTF-8, и текущая позиция чтения записи должна указывать на начало символа, иначе возникнет исключение **ОшибкаВводаВывода**.

Двоичный файл

При чтении данных из двоичного файла, в зависимости от типа каждого переданного параметра, процедура выполняет следующие действия:

- **строка** — читает из файла 64-битное целое со знаком, представляющее собой длину строки в байтах. Затем, если прочитанная длина больше нуля, читает соответствующее число байтов и заполняет ими содержимое строки. При этом корректность прочитанных символов UTF-8 не проверяется, что может приводить к различным ошибкам при дальнейших обращениях к прочитанной строке.
- **объект** — значение этого типа не может быть прочитано из файла. При попытке чтения возникнет исключение **НеверныйТип**.
- **другие простые типы** — читает из файла один или несколько байтов, в соответствии с размером внутреннего представления значения данного типа, как это описано в разделе **Типы данных**.
- **составные типы** — значения составных типов не могут быть прочитаны из файла.

Текстовый файл

При чтении данных из текстового файла, в зависимости от типа каждого переданного параметра, процедура выполняет следующие действия:

- **`символ`** — читает из файла один символ в кодировке UTF-8 и приводит его к 32-битному целому, означающему кодпойнт Юникод.
- **`строка`** — читает из файла все символы до ближайшего символа окончания строки (**`№0Д`**, **`№0А`** или их комбинация), либо до конца файла. Символы окончания строки будут прочитаны из файла, но не попадут в возвращаемое строковое значение. Если текущая позиция чтения-записи указывает на символ окончания строки, прочитанное строковое значение будет пустым.
- **`объект`** — значение этого типа не может быть прочитано из файла. При попытке чтения возникнет исключение **`НеверныйТип`**.
- **другие простые типы** — читает из файла все символы до ближайшего символа-разделителя: пробела, табуляции, возврата каретки, перевода строки, либо до конца файла. Если текущая позиция чтения-записи указывает на символ-разделитель, то он и следующие за ним разделители будут пропущены. Полученное строковое значение будет приведено к соответствующему типу данных; если это невозможно, возникнет исключение **`ОшибкаКонвертации`**.
- **составные типы** — значения составных типов не могут быть прочитаны из файла.

Процедура файлЗаписать()

```
процедура файлЗаписать(вх файл: объект; вх арг1, арг2, арг3, ...);
```

Записывает в файл значения переданных параметров, начиная с текущей позиции чтения-записи. Если текущая позиция чтения-записи меньше размера файла в байтах, то новые данные будут записаны поверх старых; в противном случае размер файла будет увеличен до необходимого. Текущая позиция чтения-записи смещается к концу файла на число записанных байтов. При ошибках записи возникает исключение **`ОшибкаВводаВывода`**.

Переданный файл должен быть открыт вызовом функции **`файлОткрыть()`** или **`файлСоздать()`**.

Двоичный файл

При записи данных в двоичный файл, в зависимости от типа каждого переданного параметра, процедура выполняет следующие действия:

- **`строка`** — записывает в файл 64-битное целое со знаком, представляющее собой длину строки в байтах. Затем, если строка непустая, записывает один или несколько байтов — содержимое строки в кодировке UTF-8.
- **другие простые типы** — записывает в файл один или несколько байтов, в соответствии с размером внутреннего представления значения данного типа, как это описано в разделе **Типы данных**.
- **составные типы** — значения составных типов не могут быть записаны в файл.

Текстовый файл

При записи данных в текстовый файл, в зависимости от типа каждого переданного параметра, процедура выполняет следующие действия:

- **`символ`** — записывает в файл один символ в кодировке UTF-8.
- **`строка`** — записывает в файл символы в кодировке UTF-8, составляющие содержимое строки. Никакие дополнительные символы, свидетельствующие об окончании строки, в файл не записываются.

- **другие простые типы** — записывает в файл строковое представление переданного значения, полученное прямым приведением типа. Никакие дополнительные символы-разделители в файл не записываются.
- **составные типы** — значения составных типов не могут быть записаны в файл.

Функция файлЕсть()

```
функция файлЕсть(вх путь: строка): логическое;
```

Возвращает истинное значение, если существует файл, но не каталог, с указанным именем. В параметре **путь** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, функция проверяет наличие файла в текущем каталоге.

В случае ошибок создаётся исключение **ОшибкаВводаВывода**.

Функция файлЕстьКат()

```
функция файлЕстьКат(вх путь: строка): логическое;
```

Возвращает истинное значение, если существует каталог с указанным именем. В параметре **путь** должно быть передано имя каталога и, возможно, абсолютный или относительный путь к нему. Если путь не указан, функция проверяет наличие подкаталога в текущем каталоге.

В случае ошибок создаётся исключение **ОшибкаВводаВывода**.

Процедура файлСоздКат()

```
процедура файлСоздКат(вх путь: строка);
```

Создаёт каталог с указанным именем.

В параметре **путь** может быть передано только имя создаваемого каталога, либо полный путь к нему; относительный путь не допускается. Если передано только имя, то каталог будет создан в текущем каталоге; если передан полный путь, то каталог будет создан по указанному пути.

В случае ошибок создаётся исключение **ОшибкаВводаВывода**.

Процедура файлУдалКат()

```
процедура файлУдалКат(вх путь: строка);
```

Удаляет каталог с указанным именем.

В параметре **путь** может быть передано только имя удаляемого каталога, либо полный путь к нему; относительный путь не допускается. Если передано только имя, то удаляемый каталог должен располагаться в текущем каталоге; если передан полный путь, то будет удалён каталог по указанному пути.

В случае ошибок создаётся исключение **ОшибкаВводаВывода**.

Функция файлТекКат()

```
функция файлТекКат(): строка;
функция файлТекКат(вх путь: строка): строка;
```

Возвращает путь к текущему каталогу.

В параметре **путь** может быть передан относительный или абсолютный путь к каталогу, который будет назначен текущим. В этом случае функция вернёт полный путь к новому текущему каталогу.

В случае ошибок создаётся исключение **ОшибкаВводаВывода**.

Функция файлДомКат()

```
функция файлДомКат(): строка;
```

Возвращает путь к домашнему каталогу текущего пользователя операционной системы.

Функция файлВрмКат()

```
функция файлВрмКат(вх общий: логическое): строка;
```

Возвращает путь к каталогу для размещения временных файлов. Если в параметре **общий** передано истинное значение, функция вернёт глобальный временный каталог, в противном случае — временный каталог текущего пользователя, если это поддерживается операционной системой.

Функция файлВрмИмя()

```
функция файлВрмИмя(вх общий: логическое; вх префикс: строка): строка;
```

Возвращает путь и имя файла, которые можно использовать для создания временного файла. Если в параметре **общий** передано истинное значение, функция вернёт путь к файлу в глобальном временном каталоге, в противном случае — во временном каталоге текущего пользователя, если это поддерживается операционной системой.

Имя файла состоит из переданного префикса и уникального суффикса из нескольких цифр. Если в параметре **префикс** передана пустая строка, будет использован префикс **"TMP"**.

Функция файлПолныйПуть()

```
функция файлПолныйПуть(вх путь: строка): строка;
```

Дополняет переданный относительный путь к файлу или каталогу до абсолютного, с учётом текущего каталога выполнения программы, и заменяет символы-разделители пути на принятые в данной операционной системе.

Если в параметре **путь** передано пустое значение, возвращает полный путь к текущему каталогу.

Функция файлВыполняемый()

```
функция файлВыполняемый(): строка;
```

Возвращает имя файла выполняемой программы.

Функция файлПуть()

```
функция файлПуть(вх путь: строка): строка;
```

Выделяет из переданной строки часть, соответствующую пути к каталогу, включая последний символ-разделитель пути.

Функция файлИмя()

```
функция файлИмя(вх путь: строка): строка;
```

Выделяет из переданной строки часть, соответствующую имени файла, включая расширение.

Функция файлРасширение()

```
функция файлРасширение(вх путь: строка): строка;
```

Выделяет из переданной строки часть, соответствующую расширению имени файла, включая ведущую точку.

Функция файлАтрибуты()

```
функция файлАтрибуты(вх путь: строка): целое;
```

Возвращает атрибуты файла с указанным именем — целочисленное значение, представляющее собой комбинацию всех установленных атрибутов файла, объединённых операцией побитового ИЛИ.

В параметре **путь** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, функция вернёт атрибуты файла в текущем каталоге.

При ошибках создаётся исключение **ОшибкаВводаВывода**.

Возможным значениям атрибутов соответствуют константы **фаXXXX**. Некоторые атрибуты могут быть неприменимы, в зависимости от операционной системы.

Функция файлВозраст()

```
функция файлВозраст(вх путь: строка): момент;
```

Возвращает дату и время последней модификации файла.

В параметре **путь** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, функция вернёт возраст файла в текущем каталоге.

При ошибках создаётся исключение **ОшибкаВводаВывода**.

Функция файлПереместить()

```
процедура файлПереместить(вх откуда, куда: строка);
```

Переименовывает и/или перемещает файл с указанным именем.

В параметре **откуда** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, файл должен располагаться в текущем каталоге.

В параметре **куда** должно быть передано новое имя файла и, возможно, новый абсолютный или относительный путь к нему. Если путь не указан, функция переместит файл в текущий каталог и присвоит ему указанное имя.

Если новый путь отличается от старого, файл будет перемещён; если новое имя отличается от старого, файл будет переименован.

Для переименования или перемещения каталога в параметре **откуда** нужно передать путь к каталогу, завершающийся символом-разделителем пути — «\» или «/», в зависимости от операционной системы.

При ошибках создаётся исключение **ОшибкаВводаВывода**.

Функция файлУдалить()

```
процедура файлУдалить(вх путь: строка);
```

Удаляет файл с переданными именем.

В параметре **путь** должно быть передано имя файла и, возможно, абсолютный или относительный путь к нему. Если путь не указан, функция удалит файл в текущем каталоге.

При ошибках создаётся исключение **ОшибкаВводаВывода**.

Функция файлПервый()

```
функция файлПервый(вх шаблон: строка; вых инфо: ФайлИнфо): объект;
```

Создаёт объект — список файлов, подходящих под переданный **шаблон**, и заполняет переданную структуру данными о первом из найденных файлов.

Шаблон может содержать абсолютный или относительный путь к каталогу, а также маску имени файла с символами «*» и «?».

Если функция нашла один или несколько подходящих файлов, то она возвращает непустой объект, который можно передавать в функцию **файлСледующий()** для получения информации о следующих найденных файлах. Этот объект должен быть в дальнейшем уничтожен вызовом процедуры **уничтожить()**. Если подходящие файлы не найдены, функция вернёт значение **пусто**.

Пример

```
// Выводит список файлов и каталогов в домашнем каталоге пользователя,
// за исключением скрытых файлов и каталогов.
программа СписокФайлов;
переменные
    шаблон: строка;
    поиск: объект;
    инфо: ФайлИнфо;
    найдено: логическое;
начало
    шаблон := файлПолныйПуть(файлДомКат() ++ "*");
    вывести(шаблон, НС);
    поиск := файлПервый(шаблон, инфо);
    если поиск <> пусто то начало
        цикл начало
            если инфо.атрибуты & фаСкрытый = 0 то начало
                если инфо.атрибуты & фаКаталог <> 0 то
                    вывести(формат(
                        "%20с %8ш %19м %с",
                        "<каталог>", инфо.атрибуты, инфо.возраст, инфо.имя), НС)
                иначе
                    вывести(формат(
                        "%20ц %8ш %19м %с",
                        инфо.размер, инфо.атрибуты, инфо.возраст, инфо.имя), НС);
            конец;
```

```
            найдено := файлСледующий(поиск, инфо);
        конец пока найдено;
    напоследок
        уничтожить(поиск);
    конец;
окончание.
```

Функция файлСледующий()

```
функция файлСледующий(вх список: объект; вых инфо: ФайлИнфо): логическое;
```

Заполняет переданную структуру данными о следующем найденном файле из переданного списка и возвращает истинное значение. Если в списке больше нет файлов, возвращает значение **нет**; в этом случае структура **инфо** остаётся незаполненной.

Объект **список** должен быть создан вызовом функции **файлПервый()** и в дальнейшем уничтожен вызовом процедуры **уничтожить()**.

Пример использования см. в описании функции **файлПервый()**.

Графика — графический вывод

Модуль **Графика** содержит библиотеку графических примитивов, а также базовые функции для работы с изображениями.

Все функции этой библиотеки работают только в том случае, если программа запущена в отладочной среде. При запуске программы из системного терминала модуль **Графика** недоступен и вызывает ошибку при компиляции программы.

Все функции графического вывода используют систему координат, где точкой начала отсчёта считается левый верхний угол холста — графического окна или изображения.

Типы

```
типы
    Точка/Точки/Точек = структура
        г, в: целое;
    окончание;

    МассивТочек/МассивыТочек/МассивовТочек = массив Точек;

    Размер/Размеры/Размеров = Точка;
```

Константы

```
константы
    // стили пера
    грспПусто      = 0;
    грспЛиния      = 1;
    грспТочка      = 2;
    грспТире       = 3;
    грспТочкаТире  = 4;
    грсп2ТочкиТире = 5;
```

```
константы
    // стили кисти
    грскПусто = 0;
    грскЦвет  = 1;
константы
    // стили шрифта
    грсшЖирный  = 1; // жирный шрифт
    грсшКурсив  = 2; // курсив
    грсшПодчерк = 4; // одинарное подчёркивание
    грсшЗачерк  = 8; // зачёркнутый текст
```

Функция грОкно()

```
функция грОкно(вх заголовок: строка): объект;
```

Создаёт новое графическое окно — закладку в окне отладочной консоли, предназначенную для вывода данных в графическом режиме. Название новой закладки должно быть передано в параметре **заголовок**. Созданное окно можно использовать в качестве холста для функций графического вывода.

Возвращает объект, соответствующий созданному окну — этот объект можно использовать в качестве холста для функций графического вывода. Перед завершением работы программы все графические окна должны быть закрыты вызовом процедуры **грУничтожить()**.

Процедура грУничтожить()

```
процедура грУничтожить(вв холст: объект);
```

Уничтожает переданный **холст** — т.е., закрывает графическое окно, открытое функцией **грОкно()**, или уничтожает объект-изображение. Переданной переменной присваивается значение **пусто**.

Эта функция обязательно должна быть вызвана для каждого графического объекта, созданного программой, иначе при завершении программы возникнет исключение **ОшибкаВыполнения**.

Функция грРазмер()

```
функция грРазмер(вх го: объект): Размер;
функция грРазмер(вх холст: объект; вх горз, верт: целое): Размер;
```

Возвращает размеры переданного графического объекта в точках. В параметре **го** может быть передано графическое окно или изображение любого поддерживаемого формата.

Если указаны параметры **горз** и **верт**, функция установит для переданного объекта новые размеры и вернёт прежние размеры. В этом случае первым параметром должен быть передан объект, который может служить холстом — т.е., графическое окно или изображение в формате **image/bmp**.

Процедура грНачать()

```
процедура грНачать(вх холст: объект);
```

Начинает серию операций графического вывода.

Все последующие вызовы функций графического вывода не будут приводить к выводу изображения на экран до тех пор, пока серия не будет закончена вызовом **грЗакончить()**. Таким образом можно заметно увеличить быстродействие программы и избежать ненужных мерцаний на экране.

Процедуры **грНачать()** и **грЗакончить()** поддерживают счётчик вложенности вызовов. Это означает, что серия будет закончена только тогда, когда процедура **грЗакончить()** будет вызвана ровно столько раз, сколько до этого была вызвана процедура **грНачать()**. Поэтому рекомендуется помещать вызов **грЗакончить()** в секцию **напоследок** составного блока инструкций, например:

```
начало
    грНачать(окно);
    начало
        грКруг(окно, 50, 50, 40);
        грПрямоугольник(окно, 0, 10, 40, 20);
        грОтрезок(окно, 0, 10, 40, 20);
    напоследок
        // в этот момент все нарисованные фигуры
        // будут показаны на экране одновременно
        грЗакончить(окно);
    конец;
окончание.
```

Процедура грЗакончить()

```
процедура грЗакончить(вх холст: объект);
```

Заканчивает серию операций графического вывода, начатую вызовом **грНачать()** и выводит на экран все внесённые изменения.

Процедуры **грНачать()** и **грЗакончить()** поддерживают счётчик вложенности вызовов. Это означает, что серия будет закончена только тогда, когда процедура **грЗакончить()** будет вызвана ровно столько раз, сколько до этого была вызвана процедура **грНачать()**. Поэтому рекомендуется помещать вызов **грЗакончить()** в секцию **напоследок** составного блока инструкций.

Процедура грПеро()

```
процедура грПеро(вх холст: объект; вх цвет: целое);
процедура грПеро(вх холст: объект; вх цвет, толщина: целое);
процедура грПеро(вх холст: объект; вх цвет, толщина, стиль: целое);
```

Устанавливает свойства пера — цвет, толщину и стиль. Свойства пера используются для прорисовки линий и контуров геометрических фигур на переданном холсте.

Цвет должен быть указан в 24-битном формате RGB, где первый (младший) байт соответствует красному цвету, второй байт зелёному, а третий синему.

Толщина означает толщину линии в точках.

В параметре **стиль** можно передать одну из констант **грспXXXX**.

Процедура грКисть()

```
процедура грКисть(вх холст: объект; вх цвет: целое);
процедура грКисть(вх холст: объект; вх цвет, стиль: целое);
```

Устанавливает свойства кисти — цвет и стиль. Свойства кисти используются для заливки геометрических фигур и в качестве фона при выводе текста на переданном холсте.

Цвет должен быть указан в 24-битном формате RGB, где первый (младший) байт соответствует красному цвету, второй байт зелёному, а третий синему.

В параметре **стиль** можно передать одну из констант **грскXXXX**.

Процедура грШрифт()

```
процедура грШрифт(вх холст: объект; вх цвет: целое);
процедура грШрифт(вх холст: объект; вх цвет, стиль: целое);
процедура грШрифт(вх холст: объект; вх цвет, стиль, размер: целое);
процедура грШрифт(вх холст: объект; вх цвет, стиль, размер: целое; вх имя: строка);
```

Устанавливает свойства шрифта — имя, размер, стиль и цвет. Свойства шрифта используются для вывода текста на холст.

В параметре **имя** должно быть указано наименование шрифта. Если здесь передана пустая строка, то используется такой же шрифт, какой был указан в настройках среды для вывода текста в отладочной консоли.

Параметр **размер** означает размер шрифта в пунктах. Если здесь передан нуль, то используется такой же размер шрифта, какой был указан в настройках среды для вывода текста в отладочной консоли.

В параметре **стиль** можно передать нуль — это соответствует обычному тексту, — либо одно или несколько значений констант **грсшXXXX**, скомбинированных с помощью побитового ИЛИ. Например, выражение **грсшЖирный | грсшКурсив** означает жирный курсив, а выражение **грсшЗачерк | грсшПодчерк | грсшКурсив** означает зачёркнутый и подчёркнутый курсив.

Цвет должен быть указан в 24-битном формате RGB, где первый (младший) байт соответствует красному цвету, второй байт зелёному, а третий синему.

Процедура грОбрезка()

```
процедура грОбрезка(вх холст: объект; вх г1, в1, г2, в2: целое);
процедура грОбрезка(вх холст: объект; вх вкл: логическое);
```

Устанавливает для переданного холста прямоугольную область обрезки. Параметры **г1, в1** определяют координаты верхнего левого угла прямоугольника, а параметры **г2, в2** — правого нижнего.

Если установлена область обрезки, то все функции графического вывода будут иметь эффект только в пределах этой области, а любой графический вывод, выходящий за её пределы, будет обрезан по границам области.

Если в параметре **вкл** передано **нет**, то установленная область обрезки будет отменена, так что вновь станет возможен вывод в любом месте холста. Если передано **да**, то будет вновь применена та область обрезки, которая была установлена ранее.

Функция грТочка()

```
функция грТочка(вх холст: объект; вх г, в: целое): целое;
функция грТочка(вх холст: объект; вх г, в, цвет: целое): целое;
```

Возвращает цвет точки с указанными координатами на переданном холсте. Если передан **цвет**, то для точки будет установлен новый цвет; при этом функция вернёт прежний цвет точки.

Цвет должен быть указан в 24-битном формате RGB, где первый (младший) байт соответствует красному цвету, второй байт зелёному, а третий синему.

Процедура грКруг()

```
процедура грКруг(вх холст: объект; вх г, в, радиус: целое);
```

Выводит на **холст** круг с указанным радиусом. В параметрах **г** и **в** должны быть переданы горизонтальная и вертикальная координаты центра круга.

Для прорисовки контура окружности используются установленные свойства пера, область внутри окружности заполняется в соответствии с установленными свойствами кисти.

Процедура грЭллипс()

```
процедура грЭллипс(вх холст: объект; вх г1, в1, г2, в2: целое);
```

Выводит на **холст** эллипс, вписанный в прямоугольник с указанными координатами. В параметрах **г1** и **в1** должны быть переданы горизонтальная и вертикальная координаты верхнего левого угла прямоугольника, а в параметрах **г2** и **в2** — правого нижнего.

Для прорисовки контура эллипса используются установленные свойства пера, область внутри эллипса заполняется в соответствии с установленными свойствами кисти.

Процедура грДуга()

```
процедура грДуга(
    вх холст: объект; вх г1, в1, г2, в2: целое; вх начало, размер: дробное);
```

Выводит на **холст** дугу эллипса, вписанного в прямоугольник с указанными координатами. В параметрах **г1** и **в1** должны быть переданы горизонтальная и вертикальная координаты верхнего левого угла прямоугольника, а в параметрах **г2** и **в2** — правого нижнего.

Параметр **начало** определяет угол, с которого начинается дуга, а параметр **размер** — угловой размер дуги. Эти параметры должны быть указаны в радианах, отсчёт ведётся от крайней правой точки эллипса против часовой стрелки.

Для прорисовки дуги используются установленные свойства пера.

Процедура грСегмент()

```
процедура грСегмент(
    вх холст: объект; вх г1, в1, г2, в2: целое; вх начало, размер: дробное);
```

Выводит на **холст** сегмент эллипса, вписанного в прямоугольник с указанными координатами. В параметрах **г1** и **в1** должны быть переданы горизонтальная и вертикальная координаты верхнего левого угла прямоугольника, а в параметрах **г2** и **в2** — правого нижнего.

Параметр **начало** определяет угол, с которого начинается сегмент, а параметр **размер** — угловой размер сегмента. Эти параметры должны быть указаны в радианах, отсчёт ведётся от крайней правой точки эллипса против часовой стрелки.

Для прорисовки контура сегмента используются установленные свойства пера, область внутри сегмента заполняется в соответствии с установленными свойствами кисти.

Процедура грСектор()

```
процедура грСектор(
    вх холст: объект; вх г1, в1, г2, в2: целое; вх начало, размер: дробное);
```

Выводит на **холст** сектор эллипса, вписанного в прямоугольник с указанными координатами. В параметрах **г1** и **в1** должны быть переданы горизонтальная и вертикальная координаты верхнего левого угла прямоугольника, а в параметрах **г2** и **в2** — правого нижнего.

Параметр **начало** определяет угол, с которого начинается сектор, а параметр **размер** — угловой размер сектора. Эти параметры должны быть указаны в радианах, отсчёт ведётся от крайней правой точки эллипса против часовой стрелки.

Для прорисовки контура сектора используются установленные свойства пера, область внутри сектора заполняется в соответствии с установленными свойствами кисти.

Процедура грОтрезок()

```
процедура грОтрезок(вх холст: объект; вх г1, в1, г2, в2: целое);
```

Выводит на **холст** отрезок от точки с координатами **г1, в1** до точки с координатами **г2, в2**. Для прорисовки отрезка используются установленные свойства пера.

Процедура грЛоманая()

```
процедура грЛоманая(вх холст: объект; вх точки: МассивТочек);
```

Выводит на **холст** ломаную линию. В параметре **точки** должен быть передан массив пар координат для каждой вершины ломаной линии.

Для прорисовки звеньев ломаной используются установленные свойства пера.

Процедура грПрямоугольник()

```
процедура грПрямоугольник(вх холст: объект; вх г1, в1, г2, в2: целое);
процедура грПрямоугольник(вх холст: объект; вх г1, в1, г2, в2, р: целое);
процедура грПрямоугольник(вх холст: объект; вх г1, в1, г2, в2, рг, рв: целое);
```

Выводит на **холст** прямоугольник. Параметры **г1, в1** определяют координаты верхнего левого угла прямоугольника, а параметры **г2, в2** — правого нижнего.

В параметре **р** можно передать радиус закругления углов прямоугольника. Если переданы **рг** и **рв**, то **рг** определяет радиус закругления по горизонтали, а **рв** — по вертикали.

Для прорисовки контура прямоугольника используются установленные свойства пера, область внутри прямоугольника заполняется в соответствии с установленными свойствами кисти.

Процедура грМногоугольник()

```
процедура грМногоугольник(вх холст: объект; вх точки: МассивТочек);
```

Выводит на **холст** многоугольник. В параметре **точки** должен быть передан массив пар координат для каждой вершины многоугольника.

Для прорисовки сторон многоугольника используются установленные свойства пера, область внутри многоугольника заполняется в соответствии с установленными свойствами кисти.

Функция грРазмерТекста()

```
функция грРазмерТекста(вх холст: объект; вх текст: строка): Размер;
```

Возвращает размер прямоугольной области, требуемой для вывода переданной строки на переданный **холст**.

Для вычисления размера используются свойства шрифта, установленные вызовом **грШрифт()**.

Функция грТекст()

```
функция грТекст(вх холст: объект; вх г, в: целое; вх текст: строка): Размер;
```

Выводит на **холст** переданный **текст**. Координаты верхнего левого угла прямоугольника с текстом должны быть переданы в параметрах **г** и **в**. Для вывода текста используются свойства шрифта, установленные вызовом **грШрифт()**, фон прямоугольника заполняется в соответствии с текущими свойствами кисти.

Возвращает вертикальный и горизонтальный размер прямоугольника, содержащего выведенный текст.

Функция грИзоСоздать()

```
функция грИзоСоздать(вх источник: объект; вх г1, в1, г2, в2: целое): объект;
```

Создаёт изображение путём копирования прямоугольной области из переданного источника. Источником может служить графическое окно или другое изображение, либо значение **пусто**. В последнем случае создаётся чёрный прямоугольник шириной **г2-г1** и высотой **в2-в1**.

В параметрах **г1, в1** нужно передать координаты верхнего левого угла копируемого прямоугольника, а в параметрах **г2, в2** — правого нижнего.

Изображение всегда создаётся в формате **image/bmp**, канал прозрачности не поддерживается. Созданное изображение можно выводить на холсты, а также использовать в качестве холста для функций графического вывода.

Функция возвращает объект, соответствующий созданному изображению. Этот объект должен быть уничтожен вызовом **грУничтожить()**.

Функция грИзоЗагрузить()

```
функция грИзоЗагрузить(вх имяФайла: строка): объект;
```

Загружает изображение из файла с переданным именем и возвращает объект, соответствующий созданному изображению. Созданное изображение можно выводить на холсты; кроме того, если изображение имеет формат **image/bmp**, то его можно использовать в качестве холста для функций графического вывода.

Поддерживаются следующие форматы изображений: **image/bmp**, **image/x-xpixmap**, **image/png**, **image/jpeg**, **image/tiff**, **image/gif**, **image/x-portable-bitmap**, а также форматы **cur**, **ico** и **icns**.

Каждое успешно загруженное изображение должно быть уничтожено вызовом **грУничтожить()**.

Процедура грИзоСохранить()

```
процедура грИзоСохранить(вх изо: объект; вх имяФайла: строка);
```

Записывает переданное изображение в файл с указанным именем. В параметре **изо** может быть передано изображение в любом поддерживаемом формате, а также графическое окно. В последнем случае будет создан файл в формате **image/bmp**, в котором будет сохранено содержимое окна.

Процедура грИзоВывести()

```
процедура грИзоВывести(вх холст: объект; вх г, в: целое; вх изо: объект);
```

Выводит на **холст** переданное изображение. В параметрах **г, в** должны быть указаны координаты верхнего левого угла прямоугольника, куда будет выведено изображение, а в параметре **изо** — объект-изображение в любом поддерживаемом формате или любой другой холст.

События — обработка событий

Модуль **События** содержит библиотеку для обработки событий клавиатуры и мыши в окнах графического интерфейса пользователя.

Все функции этой библиотеки работают только в том случае, если программа запущена в отладочной среде. При запуске программы из системного терминала модуль **События** недоступен и вызывает ошибку при компиляции программы.

Обработка событий клавиатуры и мыши в графических окнах реализована в виде очереди событий. Вызовом **сбтЗаказать()** программа инициализирует очередь событий, функции **сбтЕсть()** и **сбтЗабрать()** предназначены для выборки событий из очереди, а функции **сбтСколько()** и **сбтСмотреть()** — для просмотра событий в очереди.

Типы

```
тип
    Событие/Событий = структура
        что: целое;  // тип события - одна из констант сбтXXXX
        код: целое;  // номер клавиши или кнопки, код символа
        инфо: целое; // состояние специальных клавиш
        г, в: целое; // координаты указателя мыши
    окончание;
```

Структура **Событие** содержит информацию о событии клавиатуры или мыши, возвращаемую функциями **сбтЗабрать()** и **сбтСмотреть()**.

Поле **что** содержит тип события — значение одной из констант **сбтXXXX**.

Поле **код** содержит, в зависимости от типа события:

- **сбтКлНаж** и **сбтКлОтп** — виртуальный код клавиши на клавиатуре;
- **сбтКлСмв** — кодпойнт Юникод введённого символа;
- **сбтМшНаж** и **сбтМшОтп** — номер кнопки мыши, одна из констант **кнмXXXX**;
- **сбтМшКлс** — дистанцию вращения колеса, отрицательное число соответствует вращению на себя;
- **сбтМшВх**, **сбтМшВых** и **сбтМшДвг** — поле не используется и равно нулю.

Поле **инфо** содержит одно или несколько значений констант **сскXXXX**, скомбинированных с помощью побитового ИЛИ. Для событий **сбтКлСмв**, **сбтМшВх и сбтМшВых** это поле не используется и равно нулю.

Биты, связанные с состоянием мыши, действительны только для событий мыши; для событий клавиатуры эти биты всегда равны нулю.

Поля **г** и **в** для событий мыши содержат горизонтальную и вертикальную координаты указателя мыши в системе координат графического окна. Для событий клавиатуры, а также **сбтМшВх** и **сбтМшВых** эти поля не используются и равны нулю.

Примечание: события нажатий клавиш и ввода символов — **сбтКлНаж** и **сбтКлСмв** — автоматически повторяются, если нажать и удерживать клавишу на клавиатуре. При этом событие отпускания клавиши **сбтКлОтп** возникает однократно в тот момент, когда клавиша отпущена.

Константы

```
константы
    // типы событий
    сбтКлНаж =   1; // Нажатие клавиши на клавиатуре
    сбтКлОтп =   2; // Отпускание клавиши на клавиатуре
    сбтКлСмв =   4; // Ввод символа с клавиатуры
    сбтМшНаж =   8; // Нажатие кнопки мыши
    сбтМшОтп =  16; // Отпускание кнопки мыши
    сбтМшКлс =  32; // Вращение колеса мыши
    сбтМшВх  =  64; // Вход указателя мыши в границы окна
    сбтМшВых = 128; // Выход указателя мыши за границы окна
    сбтМшДвг = 256; // Перемещение указателя мыши в окне

константы
    // состояние специальных клавиш
    сскШифт   =  1; // Нажата клавиша Shift
    сскКтрл   =  2; // Нажата клавиша Ctrl
    сскАльт   =  4; // Нажата клавиша Alt
    сскЛКМ    =  8; // Нажата левая кнопка мыши
    сскПКМ    = 16; // Нажата правая кнопка мыши
    сскСКМ    = 32; // Нажата средняя кнопка мыши
    сскДвКлик = 64; // Щелчок мыши следует считать двойным

константы
    // кнопки мыши
    кнмЛевая   =   8;
    кнмПравая  =  16;
    кнмСредняя =  32;
```

Процедура сбтЗаказать()

```
процедура сбтЗаказать(окно: объект; что: целое);
```

Создаёт очередь событий для графического окна. В параметре **что** необходимо указать типы событий, которые будут помещаться в очередь. Здесь можно передать одну или несколько констант **сбтXXXX**, скомбинированных с помощью логического ИЛИ.

Если в параметре **что** передано ненулевое значение, то для указанного окна создаётся очередь событий, в которую окно будет помещать все происходящие в нём события указанных типов. Программа должна извлекать эти события из очереди вызовом **сбтЗабрать()**.

Если в параметре **что** передан нуль, очередь событий для указанного окна уничтожается.

Функция сбтЕсть()

```
функция сбтЕсть(окно: объект): логическое;
```

Возвращает истинное значение, если для указанного окна была создана очередь событий, и в ней есть хотя бы одно событие. В этом случае программа может вызвать **`сбтЗабрать()`** для извлечения события из очереди.

Функция сбтЗабрать()

```
функция сбтЗабрать(окно: объект): Событие;
```

Удаляет из очереди переданного окна ближайшее (самое старое) событие и возвращает структуру с информацией об этом событии.

Если для переданного окна не была создана очередь событий, либо если очередь пуста, возникает исключение **`ОшибкаВыполнения`**.

Функция сбтСколько()

```
функция сбтСколько(окно: объект): целое;
```

Возвращает количество событий, имеющихся в очереди переданного окна. Если очередь событий не была создана, функция вернёт нуль.

Функция сбтСмотреть()

```
функция сбтСмотреть(окно: объект; идкс: целое): Событие;
```

Возвращает структуру с информацией о событии с указанным индексом из очереди переданного окна, при этом событие не удаляется из очереди.

В параметре **`идкс`** должен быть передан порядковый номер события, начиная с нуля. События в очереди расположены в порядке их возникновения, т.е. чем новее событие, тем больше индекс.

Если для переданного окна не была создана очередь событий, либо если очередь пуста, возникает исключение **`НеверныйИндекс`**.

Исполнители

Мышка

Исполнитель представляет собой мышку, которая бегает по лабиринту. Мышка может делать шаг в указанном направлении, поворачиваться, проверять наличие стен вокруг текущей клетки лабиринта, считывать числа и стрелки в текущей клетке, а также закрашивать и очищать текущую клетку.

Предусмотрено два варианта указания направления движения для Мышки:

- Относительное — вправо, влево, вперёд или назад по отношению к текущему направлению Мышки;
- Абсолютное — на север, на юг, на запад или на восток, т.е. по направлению к верхней, нижней, левой или правой границе поля, вне зависимости от направления Мышки.

Константы

```
константы
    // направления относительно текущего положения Мышки
    влево/слева = 1;
    вправо/справа = 2;
    вперед/вперёд/впереди = 3;
    назад/сзади = 4;
константы
    // абсолютные значения направлений
    наЗапад/наЗападе = 5;
    наСевер/наСевере = 6;
    наВосток/наВостоке = 7;
    наЮг/наЮге = 8;
```

Процедура загрузитьОбстановку()

```
процедура загрузитьОбстановку(вх файл: строка; вх индекс: целое);
```

Загружает обстановку исполнителя Мышка из файла с переданным именем. Если файл содержит несколько обстановок, в параметре **индекс** можно передать номер нужной обстановки (нумерация обстановок начинается с нуля).

Файл обстановки может быть создан путём экспорта из редактора учебных курсов.

При решении задач из Практикума не нужно явно загружать обстановки, т.к. они являются составной частью решаемой задачи и загружаются автоматически при запуске программы. Однако, если программа не является решением задачи, но использует модуль **Мышка**, то обстановку необходимо загружать с помощью этой функции.

Процедура шаг()

```
процедура шаг();
процедура шаг(вх куда: целое);
```

Перемещает Мышку на соседнюю клетку в указанном направлении. Параметр **куда** может принимать значения от 1 до 8 (см. **Константы**).

Если процедура вызвана без параметров, будет сделан шаг в том направлении, куда Мышка смотрит в настоящий момент.

Если в указанном направлении расположена стена, возникнет исключение **ОшибкаВыполнения**.

Процедура повернуть()

```
процедура повернуть(вх куда: целое);
```

Поворачивает Мышку в указанном направлении. Параметр **куда** может принимать значения от 1 до 8 (см. **Константы**).

Процедура закрасить()

```
процедура закрасить();
```

Закрашивает клетку, в которой находится Мышка.

Процедура очистить()

```
процедура очистить();
```

Возвращает первоначальный цвет клетке, в которой находится Мышка.

Функция стена()

```
функция стена(): логическое;
функция стена(вх где: целое): логическое;
```

Возвращает **да**, если между текущей и соседней клеткой в указанном направлении расположена стена. Параметр **где** может принимать значения от 1 до 8 (см. **Константы**). Если параметр не передан, применяется значение **впереди**.

Функция закрашено()

```
функция закрашено(): логическое;
```

Возвращает **да**, если текущая клетка закрашена.

Функция стрелка()

```
функция стрелка(): целое;
```

Возвращает направление стрелки (одно из значений: **наЗапад**, **наСевер**, **наЮг** или **наВосток**), если в текущей клетке есть стрелка. При отсутствии стрелки возвращает нуль.

Функция естьЧисло()

```
функция естьЧисло(): логическое;
```

Возвращает **да**, если в текущей клетке есть число.

Функция число()

```
функция число(): целое;
```

Возвращает значение числа, если в текущей клетке есть число, в противном случае возвращает нуль.

Функция температура()

```
функция температура(): дробное;
```

Возвращает значение температуры для текущей клетки.

Функция радиация()

```
функция радиация(): дробное;
```

Возвращает значение радиации для текущей клетки.