

Язык программирования Тривиль

Версия языка: 0.9.7

Алексей Недоря

26.03.2026

Содержание

| | | |
|----------|--|-----------|
| 1 | Назначение | 5 |
| 2 | Обзор языка | 6 |
| 3 | Лексика | 7 |
| 3.1 | Комментарии | 7 |
| 3.2 | Разделители синтаксических конструкций | 7 |
| 3.3 | Идентификаторы | 7 |
| 3.4 | Служебные слова | 8 |
| 3.5 | Знаки операций и знаки препинания | 8 |
| 3.6 | Литералы | 8 |
| 3.6.1 | Целочисленные литералы | 8 |
| 3.6.2 | Вещественные литералы | 8 |
| 3.6.3 | Строковые литералы | 9 |
| 3.6.4 | Многострочные литералы | 9 |
| 3.6.5 | Символьные литералы | 9 |
| 3.6.6 | Управляющие коды | 10 |
| 3.7 | Модификаторы | 10 |
| 4 | Описания и области действия | 11 |
| 4.1 | Области действия | 11 |
| 4.2 | Предопределенные идентификаторы | 11 |
| 4.3 | Описание констант | 12 |
| 4.4 | Описание переменных | 12 |
| 4.5 | Описание функций | 13 |
| 4.5.1 | Параметры | 13 |
| 4.5.2 | Полиморфные параметры | 14 |
| 4.5.3 | Внешние функции | 14 |
| 4.6 | Описание методов | 14 |
| 5 | Типы | 15 |
| 5.1 | Предопределенные типы | 15 |
| 5.1.1 | Тип Строка | 15 |
| 5.1.2 | Тип Строка8 | 15 |
| 5.2 | Указание типа | 16 |
| 5.3 | Описание типов | 16 |
| 5.4 | Тип вектора | 16 |
| 5.5 | Тип класса | 17 |
| 5.5.1 | Инициализация полей | 17 |
| 5.5.2 | Экспорт полей | 17 |
| 5.5.3 | Наследование | 17 |
| 5.6 | Может быть тип | 18 |
| 5.7 | Тип протокола | 18 |
| 5.7.1 | Использование функций протокола | 19 |
| 5.8 | Тип функции | 20 |
| 6 | Выражения | 21 |
| 6.1 | Первичные выражения | 21 |
| 6.2 | Операнды | 21 |
| 6.3 | Доступ к полям, методам и импортируемым объектам | 22 |
| 6.4 | Индексация | 22 |
| 6.5 | Преобразование типа | 22 |
| 6.6 | Вызов функции или метода | 24 |
| 6.6.1 | Развернутое выражение | 24 |
| 6.7 | Конструктор вектора | 24 |
| 6.8 | Конструктор экземпляра класса | 26 |
| 6.9 | Подтверждение типа | 26 |
| 6.10 | Приоритеты операции | 26 |
| 6.11 | Унарные операции | 27 |
| 6.12 | Проверка типа | 27 |

| | | |
|-----------|--|-----------|
| 6.13 | Описание попутной переменной | 27 |
| 6.14 | Бинарные операции | 28 |
| 6.14.1 | Арифметические операции | 29 |
| 6.14.2 | Операции сравнения | 29 |
| 6.14.3 | Логические операции | 29 |
| 6.14.4 | Битовые операции | 29 |
| 6.14.5 | Операции сдвига | 29 |
| 6.15 | Константные выражения | 29 |
| 7 | Операторы | 30 |
| 7.1 | Списки операторов и Блоки | 30 |
| 7.2 | Локальные описания | 30 |
| 7.3 | Выражение, как оператор | 30 |
| 7.4 | Оператор присваивания | 30 |
| 7.5 | Инкремент и декремент | 31 |
| 7.6 | Оператор 'если' | 31 |
| 7.6.1 | Попутная переменная в операторе 'если' | 31 |
| 7.7 | Оператор 'надо' | 32 |
| 7.7.1 | Попутная переменная в операторе 'надо' | 32 |
| 7.8 | Оператор 'выбор' | 32 |
| 7.8.1 | Выбор по выражению | 33 |
| 7.8.2 | Выбор по предикатам | 33 |
| 7.8.3 | Выбор по типу | 34 |
| 7.9 | Оператор 'пока' | 35 |
| 7.10 | Оператор цикл | 35 |
| 7.11 | Оператор 'прервать' | 35 |
| 7.12 | Оператор 'вернуть' | 36 |
| 7.13 | Оператор 'авария' и аварийная ситуация | 36 |
| 8 | Стандартные функции и методы | 37 |
| 8.1 | Стандартные функции | 37 |
| 8.1.1 | Функция 'длина' | 37 |
| 8.1.2 | Функции для полиморфных параметров | 37 |
| 8.1.3 | Функция 'сигнатура' | 37 |
| 8.2 | Встроенные методы для векторов | 37 |
| 8.2.1 | Метод 'добавить' | 38 |
| 8.2.2 | Метод 'сократить' | 38 |
| 9 | Модули | 39 |
| 9.1 | Заголовок модуля | 39 |
| 9.2 | Импорт | 39 |
| 9.3 | Вход или инициализация модуля | 40 |
| 9.4 | Инициализация и исполнение программы | 40 |
| 10 | Обобщенные модули | 41 |
| 10.1 | Последствия принятого решения | 42 |
| 10.2 | Уточненный синтаксис модуля | 42 |
| 11 | Правила совместимости | 43 |
| 11.1 | Эквивалентность типов | 43 |
| 11.2 | Совместимость по присваиванию | 43 |
| 12 | Применять с осторожностью | 44 |
| 12.1 | Осторожное преобразование типа | 44 |
| A | Компиляторы и версии языка | 45 |
| A.1 | Компиляторы | 45 |
| A.2 | Версия 0.9.0 от 07.09.2023 | 45 |
| A.3 | Версия 0.9.1 от 08.09.2023 | 45 |
| A.4 | Версия 0.9.2 от 08.10.2023 | 45 |
| A.5 | Версия 0.9.3 от 29.04.2024 | 45 |
| A.6 | Версия 0.9.4 от 15.06.2024 | 45 |

| | | |
|------|----------------------------|----|
| A.7 | Версия 0.9.5 от 22.03.2025 | 45 |
| A.8 | Версия 0.9.6 от 26.03.2026 | 45 |
| A.9 | Версия 0.9.7 от 26.03.2026 | 45 |
| A.10 | Планируемые доработки | 46 |

1. Назначение

Язык программирования Тривиль разработан в рамках работы на семейством языков программирования [Языки выходного дня \(ЯВД\)](#) проекта [Интенсивное программирование](#).

Тривиль является предварительным языком семейства ЯВД, предназначенным для реализации компиляторов и экосистемы следующих языков семейства. В рамках классификации языков, принятом в проекте Интенсивное программирование, это язык L2.

Основными требованиями к языку были поставлены:

- язык должен быть минимально достаточным, то есть в него должны быть включены только те типы и конструкции, которые необходимы для реализации компиляторов
- язык должен быть современным с точки зрения набора конструкций
- язык должен поддерживать надежное программирование (автоматическое управление памятью, отсутствие неопределенного поведения, безопасность указателей, минимизация неявных конструкций)
- язык должен обеспечивать легкость чтения и понимания (readability) и легкость разработки
- как следствие: язык должен быть русскоязычным. Лексика и синтаксис языка должны минимизировать переключение на латиницу и обратно в процессе разработки программ

Название языка происходит от слова *тривиальный*, что означает, что при разработке языка практически везде использовались решения, проверенные в других современных языках программирования, в первую очередь *донорами* являются Go, Swift, Kotlin и Oberon.

В итоге разработки, несмотря на первоочередную направленность на разработку компиляторов, Тривиль является языком программирования общего назначения, пригодным для решения широкого круга задач.

Получившийся язык (и экосистема) обладает существенными достоинствами для использования в качестве полигона для обучения студентов разработке компиляторов, библиотек, алгоритмов оптимизации и так далее, в первую очередь это:

- Простота языка
- Современный вид и набор конструкций языка
- Простота компилятора
- Открытая лицензия

2. Обзор языка

Тривиль - это модульный язык с явным экспортом и импортом, с поддержкой ООП и автоматическим управлением памятью (сборка мусора).

Программа на языке Тривиль состоит из модулей (единиц компиляции), исходный текст каждого модуля расположен в одном или нескольких исходных файлах.

Пример программы:

```
1  модуль x
2
3  импорт "std/вывод"
4
5  вход {
6      вывод.ф("Привет!\n")
7  }
```

Для описания языка используется EBNF в формате, близком к формату ANTLR4. Операции:

| | |
|-------|--------------------------------|
| () | группировка |
| X* | повторение 0 и более раз |
| X+ | повторение 1 и более раз |
| X? | опциональность X (0 или 1 раз) |
| X Y | X или Y |

Пример:

Список-операторов: `Оператор (Разделитель Оператор)*`

3. Лексика

Исходный текст есть последовательность лексем: идентификаторов (§3.3), служебных слов (§3.4), знаков операций и знаков препинания (§3.5), литералов (§3.6) и модификаторов (§3.7). Каждая лексема состоит из последовательности Unicode символов (unicode code point) в кодировке UTF-8.

Пробелы (U+0020), символы табуляции (U+0009) и символы завершения строки (U+000D, U+000A) разделяют лексемы, и, игнорируются, кроме следующих случаев:

- Символы завершения строк могут использоваться как разделители синтаксических конструкций (§3.2).
- Пробелы являются значащими символами в идентификаторах, состоящих из нескольких слов (§3.3).
- Пробелы являются значащими в строковых и символьных литералах (§3.3).

Несколько разделителей трактуются, как один.

Исходный текст может содержать *комментарии* (§3.1).

3.1. Комментарии

Есть две формы комментариев:

- Строчный комментарий начинается с последовательности символов `//` и заканчивается в конце строки.
- Блочный комментарий начинается с последовательности символов `/*` и заканчивается последовательностью символов `*/`. Блочные комментарии могут быть вложенные.

Комментарий

```
: '//' (любой символ, кроме завершения строки) *  
| '/*' (любой символ) * '*/'
```

3.2. Разделители синтаксических конструкций

Некоторые синтаксические правила используют нетерминал *Разделитель* для разделения двух подряд идущих синтаксических конструкций, например:

Список-операторов: `Оператор (Разделитель Оператор) *`

В качестве разделителя может использоваться символ `;` или символ завершения строки.

Разделитель: `';` | Символ-завершения-строки

Пример:

```
1 а := 1; б := 2  
2 в := 1
```

В строке 1 операторы разделены символом `;`, а оператор в строке 2 отделен от операторов строки 1 символом завершения строки.

Ошибка компиляции - нет разделителя:

```
1 а := 1 б := 2
```

3.3. Идентификаторы

Идентификатор - это последовательность *Слов*, разделенных пробелами или символами дефис `'-'` с опциональным суффиксом.

Каждое слово состоит из *Букв* и *Цифр*, и начинается с Буквы. Буквой считается любой Unicode символ, имеющий признак *Letter*, и, дополнительно, символы `'№'` и `'_'`.

Суффиксом может быть символ `'?'` или произвольное количество символов *одиночная кавычка*.

Идентификатор: `Слово ((' ' | '-') Слово) * Суффикс?`

Слово: `Буква (Буква | Цифра) *`

Буква: `Unicode-letter | '_' | '№'`

Цифра: '0' .. '9'
Суффикс: '?' | '*'

Примеры идентификаторов:

```
1 буква
2 буква-или-цифра
3 №-символа
4 это цифра?
5 ОписаниеТипа
6 v'
```

3.4. Служебные слова

Следующие служебные слова зарезервированы и не могут быть использованы, как идентификаторы:

| | | | | |
|---------|--------|-----------|----------|-------|
| авария | если | конст | позже | среди |
| вернуть | иначе | мб | пока | тип |
| вход | импорт | модуль | прервать | типа |
| выбор | класс | надо | протокол | фн |
| другое | когда | осторожно | пусть | цикл |

3.5. Знаки операций и знаки препинания

Следующие последовательности символов обозначают знаки операций и знаки препинания:

```
+ - * / %
= # < <= > >=
& | ~ =>
|& || |^ |~ |< |>
:= ++ --
( ) [ ] { }
(: . ^ , : ;
```

3.6. Литералы

Литерал

- : Целочисленный-литерал
- | Вещественный-литерал
- | Строковый-литерал
- | Символьный-литерал
- | Многострочный-литерал

3.6.1. Целочисленные литералы

Целочисленный-литерал:

Десятичный-литерал | Шестнадцатеричный-литерал

Десятичный-литерал: Цифра+

Шестнадцатеричный-литерал: '0x' Цифра16+

Цифра16: '0'..'9' | 'a'..'f' | 'A'..'F'

Тип десятичного литерала: Цел64, тип шестнадцатеричного литерала: Слово64 (§5.1).

3.6.2. Вещественные литералы

В текущей реализации есть только одна форма записи вещественных литералов, без экспоненты.

Вещественный-литерал: Цифра+ '.' Цифра*

Тип вещественного литерала: Вещ64 (§5.1).

3.6.3. Строковые литералы

Строковый литерал - это последовательность символов, определяющая константу типа Строка (§5.1). Два вида строковых литералов:

- Однострочный литерал - последовательность символов, заключенных в двойные кавычки: ". Закрывающая кавычка должен быть расположена на той же строка текста, что и открывающая.
- Многострочный литерал (§3.6.4) - это последовательность символов, заключенные в три двойные кавычки: """.

Строковый-литерал: Однострочный-литерал | Многострочный-литерал

Однострочный-литерал

: '...' (~('"' | Символ-завершения-строки) | Управляющий-код) * ''

Строковый литерал может содержать символы, закодированные с помощью управляющего кода (§3.6.6), которая начинается с символа '\ '.

Примеры однострочных литералов:

```
1 "Привет, мир!"
2 "Двойная кавычка \" в строке"
```

3.6.4. Многострочные литералы

Многострочный литерал позволяет задать текст любой длины, которые может быть разбит на строки. Многострочный литерал может содержать символы, закодированные с помощью управляющего кода (§3.6.6).

Многострочный-литерал

: '"""' Символ-завершения-строки

Строка-литерала*

пробел* '"""'

Строка-литерала

: (~'"""' | Управляющий-код) * '\'? Символ-завершения-строки

Правила формирования многострочного литерала:

- За открывающая последовательностью должен стоять символ завершения строки.
- Если строка литерала завершается символом '\ ', то символ завершения строки не попадает в литерал. Если этого символа нет, то в литерал попадает символ LF (U+000A).
- В строке перед завершающей последовательностью могут стоять только пробелы, задающие отступ. Число пробелов, равное отступу до закрывающей кавычки, исключается из начала каждой строки литерала.
- В литерал не попадают первый и последний символы завершения строки, а именно, стоящий сразу за открывающей последовательностью и стоящий перед строкой с закрывающей последовательность.

Пример многострочного литерала, задан отступ и исключение одного символа завершения строки:

```
1 ""
2     Привычка свыше \
3     нам дана:
4     Замена счастью она.
5 ""
```

Вывод литерала:

```
Привычка свыше нам дана:
Замена счастью она.
```

3.6.5. Символьные литералы

Символьный литерал задает значение для Unicode code point. Он записывается как один или несколько символов, заключенных в одинарные кавычки. Символьный литерал может быть закодирован с помощью управляющего кода (§3.6.6).

Символьный-литерал

```
: ""  
~ ("'" | Символ-завершения-строки) | Управляющий-код  
""
```

Тип символьного литерала: Символ (§5.1).

3.6.6. Управляющие коды

Управляющим кодом (escape-sequence) называется последовательность символов, используемая для задания непечатных или сложных для ввода символов, или символов, которые не могут напрямую присутствовать в литерале, например: символ (") в литерале: "Язык \"Тривиль\"".

Управляющий-код:

```
: '\'  
( 'u' Цифра16 Цифра16 Цифра16 Цифра16  
| 'u' '{' Цифра16+ '}'  
| 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v'  
| ""  
| ""  
| '\'  
)
```

Примеры управляющих кодов:

```
1 \n // ASCII LF (line-feed)  
2 \" // символ "  
3 \\ // символ \  
4 \u0020 // символ пробел в шестнадцатеричной кодировке  
5 \u{42E} // буква Ю
```

3.7. Модификаторы

Модификаторы используются в исходном тексте, чтобы внести изменение в семантику и/или синтаксис конструкции языка, см., например, §4.5.3.

Модификатор: '@' Буква+ Список-атрибутов?

Список-атрибутов: '(' (Атрибут (',' Атрибут)*)? ')'

Атрибут: Строковый-литерал ':' Строковый-литерал

```
1 @внеш("имя":"print_string")
```

4. Описания и области действия

Каждый идентификатор, встречающийся в программе, должен быть описан, если только это не предопределенный идентификатор (§4.2). Идентификатор может быть описан как тип, константа, переменная, функция или метод, а так же как параметр в сигнатуре функции или метода, и как поле в типе класса.

Описание

- : Описание-типа
 - | Описание-констант
 - | Описание-переменной
 - | Описание-функции
 - | Описание-метода
-

4.1. Области действия

Описанный идентификатор используется для ссылки на связанный объект, в тех частях программы, которые попадают в *область действия* описания. Идентификатор не может обозначать более одного объекта в пределах заданной области действия. Область действия может содержать внутри себя другие области действия, в которых идентификатор может быть переопределен.

Область действия, которая содержит в себе все исходные тексты на языке Тривиль называется *Универсум*.

Области действия:

- Областью действия предопределенного идентификатора является Универсум.
- Областью действия идентификатора, описанного на верхнем уровне (вне какой-либо функции), является весь модуль (§9).
- Областью действия идентификатора импортируемого модуля является файл (часть модуля), содержащего импорт (§9.2).
- Областью действия идентификатора, обозначающего параметр функции, является тело функции (§4.5).
- Областью действия идентификатора, описанного в *списке операторов* (§7.1) является часть списка операторов от точки завершения описания и до завершения этого списка операторов.
- Область действия попутной переменной в операторе **если** описана в §7.6.1, а в операторе **надо** в §7.7.1.

В описании верхнего уровня за идентификатором может следовать признак экспорта ' * ', указывающий, что идентификатор *экспортирован* и может использоваться в другом модуле, *импортирующем* данный (§9.2).

Правило для опционально экспортированного идентификатора:

Идент-оп: Идентификатор ' * ' ?

4.2. Предопределенные идентификаторы

Следующие идентификаторы неявно описаны в области действия *Универсум*.

Типы (§5.1):

Байт Цел64 Слово64 Вещ64 Лог Символ Строка Строка8 Пусто

Константы типа Лог (§5.1):

ложь истина

Литерал, обозначающий отсутствие значения для *может быть* типа (§5.6):

пусто

Стандартные функции (§8.1):

длина тег нечто сигнатура

Кроме того, для векторных типов определен набор встроенных методов (§8.2).

4.3. Описание констант

Описание константы связывает идентификатор с постоянным значением. Значение константы может быть задано явно или неявно, в случае группового описания констант.

Описание-констант: 'конст' (Константа | Группа-констант)
Константа: Идент-оп (':' Указ-типа)? '=' Выражение

Если тип константы не указан, то он устанавливается равным типу выражения (§6). Выражение для константы должно быть константным выражением (§6.15).

```
1 конст к1: Цел64 = 1 // тип Цел64
2 конст к2: Байт = 2 // тип Байт
3 конст к3 = 3 // тип Цел64
4 конст к4 = "Привет" // тип Строка
```

Групповое описание констант позволяет опускать тип и выражение для всех констант, кроме первой константы в группе и указать признак экспорт для всех констант группы.

Группа-констант:
'*'? '('
Константа (Разделитель След-константа)*
)'

След-константа: Идент-оп (':' Указ-типа)? '=' Выражение)?

Пример группового экспорта:

```
1 конст * (  
2     Счетчик = 1  
3     Имя = "Вася"  
4 )
```

Пример неявного задания значения для констант:

```
1 конст (  
2     // операции  
3     ПЛЮС = 1 // тип Цел64, значение = 1  
4     МИНУС // тип Цел64, значение = 2  
5     ОСТАТОК // тип Цел64, значение = 3  
6     // служебные слова  
7     ЕСЛИ = 21 // тип Цел64, значение = 21  
8     ИНАЧЕ // тип Цел64, значение = 22  
9     ПОКА // тип Цел64, значение = 23  
10 )
```

4.4. Описание переменных

Описание переменной создает переменную, привязывает к ней идентификатор и указывает её тип и начальное значение через *Инициализацию*.

Описание-переменной:
'пусть' Идент-оп (':' Указ-типа)? Инициализация
Инициализация: (':=' | '=') Выражение

Если тип переменной не указан, то он устанавливается равным типу выражения (§6).

Каждая переменная должна быть явно проинициализирована, начальным значение переменной является значение выражения. Если инициализация задана через лексему '=', то значение переменной не может быть изменено (*переменная с единственным присваиванием*), если же в инициализации используется лексема ':=', то значение может быть изменено (§7.4, §7.5), а переменная называется *изменяемой*.

Если тип переменной явно задан, то выражение должно быть *совместимо по присваиванию* с типом переменной (§11.2).

Явное указание типа:

```
1 пусть ц1: Цел64 = 1  
2 пусть ц2: Цел64 := 2
```

Неявное указание типа:

```
1 пусть ц3 = 3 // неявное указание типа
2 пусть ц4 := 4
3 пусть имя = Имя языка ()
```

Компилятор выдает ошибку, при попытке изменить значение переменной с единственным присваиванием:

```
1 ц1 := 2
2 ц3++
3 имя := "C++"
```

4.5. Описание функций

Описание функции состоит из идентификатора, сигнатуры и тела функции. Сигнатура функции определяет формальные параметры и тип результата (если таковой имеется). Особым видом функции является метод, см. §4.6.

Описание-функции: 'фн' Идент-оп Сигнатура-функции Тело
Сигнатура-функции: '(' Список-параметров? ')' Тип-результата?
Тип-результата: ':' ('*' | Указ-типа)
Тело: (Блок | Модификатор)?

Вместо тела функции может стоять модификтор (§3.7) @внеш, см. §4.5.3.

Если у функции задан тип результата, тело функции должно завершаться операторами **вернуть** (§7.12) или **авария** (§7.13) на каждом пути управления.

Если в типе результата вместо указания типа использован символ '*', то функция возвращает полиморфный результат. Такая функция может возвращать значение любого типа. Для работы с полиморфным значением определены специальные стандартные функции (§8.1.2).

```
1 фн Факториал(ц: Цел64): Цел64 {
2     если ц <= 1 { вернуть 1 }
3     вернуть ц * Факториал(ц - 1)
4 }
```

4.5.1. Параметры

Формальные параметры - это идентификаторы, которые обозначают *аргументы* (фактические параметры), указанные при вызове функции (§6.6).

Список-параметров: Параметр (',' Параметр)* ',' '?'
Параметр:
Идентификатор (':' | ':=') '...' ? ('*' | Указ-типа)

Параметры бывают двух видов: *входные* и *выходные* параметры. Вид параметра задается символом после идентификатора, задающего имя параметра. Для входных параметров это символ ':', для выходных - символ ':='. Выходной параметр, в более привычной терминологии, является *in out* параметром.

Аргументом для входного параметра должно быть выражение, тип которого совместим по присваиванию с типом параметра. Аргументом для выходного параметра должно быть *изменяемое выражение* (§7.4), тип которого эквивалентен типу параметра.

В теле функции, входной параметр имеет значение, которое передается в функцию, изменяется функцией и передается обратно из функции для замены исходного значения.

```
1 фн извлечь число(стр: Строка, число:= Цел64): Лог {...}
2
3 пусть рез := 0
4 если ~ извлечь число(с, рез) {
5     вывод.ф("не удалось извлечь число")
6 }
```

Последний параметр функции может иметь тип с префиксом `...` - это *вариативный параметр*. Функция с таким параметром может быть вызвана с нулем или более аргументов для этого параметра. Если тип вариативного параметра указан как *T*, то в теле функции типом параметра является `[] T`.

```

1 фн сумма (числа: ...Цел64) : Цел64 {...}
2
3 // В вызове можно указать любое число аргументов
4 ответ := сумма() // без аргументов
5 ответ := сумма(1, 2, 3) // три аргумента

```

4.5.2. Полиморфные параметры

Если вместо типа параметра указан символ '*', то параметр называется полиморфным. Аргумент, соответствующий этому параметру, может быть выражением любого типа. Для вариативного полиморфного параметра каждый аргумент может быть выражением любого типа. Для работы с полиморфным параметром определены специальные стандартные функции (§8.1.2).

```

1 фн печать по формату (формат: Строка, аргументы: ...*) {
2   ...
3 }

```

4.5.3. Внешние функции

Функция, в которой вместо тела стоит модификатор @внеш, является *внешней функцией*, то есть реализованной каким-то внешним способом. Атрибут "имя" модификатора задает внешнее имя функции.

```

1 фн строка (с: Строка) @внеш ("имя": "print_string")

```

Если атрибут "имя" не задан, внешнее имя совпадает с идентификатором функции.

4.6. Описание методов

Метод - это функция, связанная с типом классом. Для вызова метода (§6.6) должен быть указан экземпляр этого класса или расширенного класса. В описание метода класс, с которым связан метод указывается с помощью *Привязки*.

Описание-метода: 'фн' Привязка Идент-оп Сигнатура-функции Блок
Привязка: '(' Идентификатор ':' Указ-типа ')'

Привязка определяет идентификатор и тип, который должен быть типом класса. В теле функции идентификатор привязки является параметром указанного типа.

```

1 тип К = класс {}
2
3 фн (к: К) метод() {}

```

Идентификатор метода должен быть уникальным в классе среди идентификатором полей и методов. В расширенном классе может быть определен метод с таким же идентификатором как в одном из базовых классов.

переопределен, то есть опеределен другой метод. При этом сигнатура переопределенного метода должна совпадать с сигнатуром переопределяемого метода, а именно:

- Число параметров должно совпадать
- Типы результата должны быть равны §11.1
- Типы параметров должны быть равны §11.1
- Признаки вариативности и полиморфности для каждого параметра должны совпадать

Пример ошибки при переопределении метода (разное число параметров):

```

1 тип К1 = класс (К) {}
2
3 фн (к: К1) метод(с: Строка) {}

```

5. Типы

Тип определяет набор значений, которые могут принимать переменные этого типа, и набор операций, применимых к значениям такого типа.

5.1. Предопределенные типы

Следующие типы обозначаются предопределенными идентификаторами, значениями данных типов являются:

| Тип | Множество значений |
|---------|--|
| Байт | множество целых чисел от 0 до 255 |
| Цел64 | множество всех 64-битных знаковых целых чисел |
| Слово64 | множество всех 64-битных беззнаковых целых чисел |
| Вещ64 | множество всех 64-разрядных чисел с плавающей запятой стандарта IEEE-754 |
| Лог | константы ложь и истина, а также нет и да |
| Символ | множество всех Unicode символов |
| Строка | множество всех строковых значений (§5.1.1) |
| Строка8 | множество всех строковых значений (§5.1.2) |
| Пусто | литерал пусто |

Операции над значениями этих типов определены в (§6.11, §6.14).

Замечание: Константы `нет` и `да` добавлены к типу `Лог` для экспериментальной проверки и могут быть использованы одновременно с константами `ложь` и `истина`.

5.1.1. Тип Строка

Строковый тип представляет собой набор строковых значений. Строковое значение - это (возможно, пустая) последовательность Unicode символов в кодировке UTF-8. Количество символов называется длиной строки и никогда не бывает отрицательным. Строки неизменяемы: после создания изменить содержимое строки невозможно. Количество символов строки можно определить с помощью встроенной функции `длина` (§8.1.1).

Операция индексации для строк не определена:

```
1  пусть с = "привет"  
2  пусть сим = с[0] // ошибка
```

Для работы с байтами строки можно использовать тип `Строка8` (§5.1.2).

5.1.2. Тип Строка8

Тип `Строка8` введен для удобной и быстрой работы со строками в стандартных библиотеках. Единственным способом получения значения типа `Строка8` является преобразование из строки. Это преобразование выполняется только во время компиляции.

```
1  пусть с8 = "Привет" (:Строка8)
```

Значение типа `Строка8` - это неизменяемая, индексируемая последовательность байтов. Стандартная функция `длина` примененная к значению типа `Строка8` возвращает число байтов.

Сравнение типов:

| | Строка | Строка8 |
|--------------|-----------------------------|---------------------------|
| Определение | последовательность символов | последовательность байтов |
| Индексация | нет | да |
| длина(с) | число символов | число байтов |
| можно менять | нет | нет |

```

1 пусть с = "Привет"
2 пусть с8 = с(:Строка8)
3
4 вывод.ф("%v\n", длина(с)) // выведет: 6
5 вывод.ф("%v\n", длина(с8)) // выведет: 12
6 вывод.ф("%v\n", с8[0]) // выведет: 0xD0

```

Значение типа Строка8 неизменяемое, менять байты в нем нельзя:

```

1 пусть с8 = "Привет"(:Строка8)
2 с8[0] := 1 // ошибка

```

5.2. Указание типа

Тривиль является языком со статической типизацией, что означает, что тип любого объекта языка явно или неявно указывается во время описания объекта. Неявное указание типа может быть использовано в описании констант (§4.3), переменных (§4.4) и полей класса (§5.5).

Указ-типа

```

: 'мб' Указ-типа
| '[' ']' Указ-типа
| Имя-типа

```

Имя-типа: (Идентификатор '.')? Идентификатор

Если указание типа начинается с **мб**, то определяется *может быть* тип (§5.6). Множество значений объекта с типом **мб** Т состоит из значения, обозначенного предопределенным идентификатором **пусто** и значений типа Т.

Если указание типа начинается с '[' ']', то определяется тип вектора (§5.4).

Если в качестве *имени типа* используется пара идентификаторов, то первый идентификатор должен обозначать имя импортированного модуля (§9.2), а второй идентификатор должен быть идентификатором экспортированного из этого модуля объекта, в данном контексте - идентификатором типа.

5.3. Описание типов

Описание типа связывает идентификатор с новым или существующим типом.

Описание-типа:

```

'тип' Идент-оп '='
( Тип-класса
| Тип-протокола
| Тип-функции
| Указ-типа
)

```

Если в описание типа указан тип класса (§5.5), тип протокола (§5.7) или тип функции (§5.8), то создается новый тип и определяется структура данных этого типа, и, как следствие, набор операций над данными этого типа.

Если в описании типа стоит указание на тип (§5.2), то это определение еще одного имени для уже существующего типа или задание имени для *типа вектора* (§5.4) или *может быть типа* (§5.6).

```

1 тип Цел = Цел64
2 тип Текст= []Строка
3 тип Строка? = мб Строка

```

Так как указание типа не создает новый тип, использование типа Цел из примера идентично использованию типа Цел64.

5.4. Тип вектора

Вектор - это пронумерованная последовательность элементов одного типа, называемая типом элемента. Количество элементов называется длиной вектора, длина не может быть отрицательной. Элементы вектора доступны через операцию индексации (§6.4). Для индексации вектора используются целочисленные индексы от 0 до длина-1. Векторы всегда одномерны, но типом элемента может быть вектор, там самым формируя многомерную структуру.

Тип-вектора: '[' ']' Указ-типа

```
1 тип Байты = []Байт
2 тип Матрица = [][]Вещ64
```

Длина вектора может изменяться во время выполнения, другими словами, вектор - это *динамический массив*. Для получения текущей длины вектора используется стандартная функция `длина` (§8.1.1). Начальное значение для объекта типа вектор задается с помощью конструктора вектора (§6.7). Далее, длину вектора можно поменять с помощью стандартных методов (§8.2).

Пример инициализации вектора из трех элементов:

```
1 пусть байты = Байты[1, 2, 3]
```

5.5. Тип класса

Тип класса - это структура, состоящая из полей. С типом класса могут быть связаны функции, называемые методами (§4.6).

Описание класса состоит из опционального указания базового класса и списка полей. Если базовый класс указан, то класс *наследует* поля и методы базового класса (§5.5.3).

Для каждого поля в списке задается идентификатор, тип (явно или неявно) и начальное значение. Областью действия идентификаторов полей является само описание класса, но они могут быть доступны в операции доступа к полям и методам (§6.3).

Тип-класса: 'класс' Базовый-класс? '{' Список-полей? '}'
Базовый-класс: '(' Указ-типа ')'
Список-полей: Поле (Разделитель Поле)*
Поле: Идент-оп (':' Указ-типа)? Инициализация

```
1 тип Человек = класс {
2     имя: Строка := ""
3     возраст: Цел64 := 0
4 }
```

Если тип поля не указан, то он устанавливается равным типу выражения (§6).

5.5.1. Инициализация полей

Каждое поле класса должно быть явно проинициализировано. Если инициализация задана через лексему '=', то значение поля не может быть изменено (*поле с единственным присваиванием*), если же в инициализации используется лексема ':=', то значение поля может быть изменено (§7.4, §7.5), а поле называется *изменяемым*.

Если в инициализации задано *Выражение*, то начальным значением поля является значение выражения. Если при этом тип поля явно задан, то выражение должно быть *совместимо по присваиванию* с типом поля (§11.2).

Вместо выражения может быть указано служебное слово **позже**, это *поздняя инициализация*. Такая форма инициализации разрешена только для полей, тип которых явно указан.

Значение поля с поздней инициализацией должно быть задано при создании экземпляра класса в конструкторе экземпляра класса (§6.8).

5.5.2. Экспорт полей

Если тип класса экспортируется, его поля могут быть помечены признаком экспорта, такие поля называются *экспортированными полями*. Поля, которые не экспортированы, доступны только в том модуле, в котором описан тип класса.

5.5.3. Наследование

Наследование позволяет определить новый (*расширенный*) класс на основе существующего (*базового*) класса.

```

1 тип Работник = класс (Человек) {
2     зарплата := 0.0
3 }

```

Базовый класс *Человек*, указанный в описании класса *Работник*, называется *прямым базовым классом*. Так как *Человек* может быть, в свою очередь, расширением другого базового класса, для каждого класса определен список базовых классов, возможно, пустой. Термин *базовый класс* будет использоваться для обозначения любого класса из этого списка. Циклы в списке базовых классов запрещены.

Расширенный класс наследует поля и методы базового класса и может добавлять новые поля и методы. Обращение к полям и методам базового класса ничем не отличается от обращения к полям и методам самого класса. Методы базового класса можно переопределять, сохраняя те же самые типы параметра и тип результата (§11.1).

5.6. Может быть тип

Как правило, современные языки программирования ограничивают работу со объектами ссылочных (*reference*) типов для того, чтобы сделать явными все места в программе, в которых может возникнуть ошибка использования нулевой ссылки (*null pointer exception*). Так как русская терминология в этой области не устоялась, приходится обращаться к англоязычным терминам.

Язык Тривиль следует уже выработанному в современных языках программированию подходу (но не синтаксису):

- Если в указании типа объекта использована нотация *мб T*, где *T* - это некоторый ссылочный тип, то значением этого объекта, кроме значений типа *T*, может быть специальное значение 'пусто'. Тип такого объекта называется *может быть T*. Тип *T* называется *базовым типом*.
- Иначе, если служебное слово *мб* отсутствует в указании типа объекта, значением этого объекта может быть только значений типа *T*.

Ссылочными типами являются Строка, типы вектора и типы класса, к остальным типам *мб* не может применяться.

Для объекта типа *мб T* определены следующие действия:

- Присваивание объекту значения 'пусто' или значения типа *T* (§11.2)
- Сравнение на равно/не равно с 'пусто' или с другим объектом типа *мб T* (§6.14.2)
- Операция подтверждения типа '^' (§6.9), позволяющая перейти от значения типа *мб T* к значению типа *T*

Пример указания типа и использования объекта:

```

1 пусть кличка: мб Строка := пусто
2 ...
3 если кличка # пусто { вывод.ф("%v\n", кличка^}

```

5.7. Тип протокола

Тип протокола задает множество заголовков функций.

```

Тип-протокола: 'протокол' { Заголовок-функции* }
Заголовок-функции: 'фн' Идентификатор Сигнатура-функции

```

Значением переменной типа протокол *П* может быть значение любого класса *K*, который

- содержит все методы с именами функций, заданными в протоколе,
- сигнатура каждого такого метода совпадает с сигнатурой функции протокола с тем же именем.

Если выполнены условия, мы будем говорить, что класс *K реализует* протокол *П*, как в этом примере:

```

1 тип П = протокол { фн мин(а: Цел64, б: Цел64): Цел64 }
2 тип К = класс {}
3
4 фн (к: К) мин(а: Цел64, б: Цел64): Цел64 {
5     надо а <= б иначе вернуть б
6     вернуть а
7 }
8
9 вход{
10     пусть п: П = К{}
11 }

```

Так как в описании класса, реализующего протокол, нет упоминания типа протокола, обычно говорят, что при проверке совместимости протокола и значения класса используется *утиная типизация*.

Неявная проверка на то, что класс значения реализует протокол (другими словами, на совместимость значения класса и протокола), происходит при присваивании во время выполнения программы. Для проверки используется динамический тип значения класса (объекта), а не статический (заданный при описании) тип переменной. Если значение класса не совместимо с протоколом, происходит *авария*.

Для явной проверки совместимости можно использовать проверку типа (§6.12) или преобразование типа (§6.5).

В данном примере, проверка на совместимость проходит (авария не происходит):

```

1 тип П = протокол { фн мин(а: Цел64, б: Цел64): Цел64 }
2 тип К = класс {}
3 тип К1 = класс (К) {}
4
5 фн (к: К1) мин(а: Цел64, б: Цел64): Цел64 {
6     надо а <= б иначе вернуть б
7     вернуть а
8 }
9
10 вход{
11     пусть к: К := К1{}
12     пусть п: П := к
13 }

```

Присваивание `п := к` выполняется, так как `к` указывает на объект класса `К1`, который реализует протокол. Присваивание `п := К{}` приведет к аварии, так как объект класса `К1` не реализует протокол.

При присваивании значения класса переменной типа протокол новый объект не создается. Протокол является прокси-объектом, который хранит присваиваемый объект, и обеспечивающий доступ к части его методов.

Переменной типа протокол, кроме значения класса, может быть присвоен другой протокол. При этом проверяется совместимость протокола и сохраненного в протоколе объекта.

В некоторых случаях, несовместимость протокола при присваивании может быть определена во время компиляции, а именно, если сигнатура метода значения не совпадает с сигнатурой функции протокола:

```

1 тип К = класс {}
2 фн (к: К) сообщить(ошибка: Строка) {}
3
4 тип П = протокол { фн сообщить(ошибка: Цел64) }
5
6 вход{
7     пусть п: П := К{} // несовместимость типов
8 }

```

5.7.1. Использование функций протокола

Использование функции протокола не отличается от использования метода класса, а именно, такая функция может быть использована в вызове или в получении функционального значения (§5.8).

```

1  тип Ввод-Вывод = протокол {
2      фн прочитать () : Символ
3      фн записать (с: Символ)
4  }
5
6  фн прочитать строку (вв: Ввод-Вывод) : Строка {
7      пусть сб = строки.Сборщик {}
8      пока истина {
9          пусть с = вв.прочитать ()
10         надо с # '\n' иначе вернуть сб.строка ()
11         сб.добавить символ (с)
12     }
13     авария ("")
14 }

```

5.8. Тип функции

Тип функции задает набор всех функций с одинаковыми типами параметров и результатов.

Тип-функции: 'фн' [Сигнатура-функции](#)

Переменная типа функции совместима по присваиванию с любой функцией или методом с эквивалентной сигатурой.

Пример корректного использования:

```

1  тип Сообщить = фн (с: Строка)
2
3  фн записать (с: Строка) {}
4
5  тип Консоль = класс {}
6  фн (к: Консоль) напечатать (с: Строка) {}
7
8  вход {
9      пусть сообщить: Сообщить := записать
10     сообщить ("Привет")
11     сообщить := Консоль {}.напечатать
12     сообщить ("Еще привет")
13 }

```

Если переменной функционального типа присваивается метод, то переменная *захватывает* объект, к которому применяется метод. Этот объект будет использоваться во всех последующих вызовах переменной.

6. Выражения

Выражение определяет вычисление значения путем применения операций к операндам.

Выражение

- : Унарное-выражение
- | Проверка-типа
- | Попутная-переменная
- | Выражение Бинарная-операция Выражение

Унарное-выражение

- : Первичное-выражение
- | Унарная-операция Унарное-выражение

Проверка-типа: Выражение 'типа' Указ-типа

Попутная-переменная: Выражение '=>' Идентификатор

6.1. Первичные выражения

Первичные выражения - это выражения, которые являются операндами унарных (§6.11) и бинарных (§6.14) операций .

Первичное-выражение

- : Операнд
- (Доступ
- | Индексация
- | Преобразование
- | Вызов
- | Конструктор-вектора
- | Конструктор-класса
- | Подтверждение-типа
-) *

```
1 1
2 к.поле
3 а[№]
4 Факториал(5)
5 А[1, 2, 3]
6 К{имя: "Вася", возраст: 23}
7 а^
8 объект.вектор[номер].метод()
```

6.2. Операнды

Операнды - это элементарные значения в выражении. Операнд может быть литералом, идентификатором, обозначающим константу, переменную или функцию, выражением в скобках или конструктором вектора анонимного типа (§6.7).

Операнд

- : Литерал
- | Идентификатор
- | '(' Выражение ')'
- | '[' ']' Указ-типа Конструктор-вектора

Тип операнда:

| Операнд | Тип выражения |
|---------------------------------------|--------------------------------|
| Идентификатор | тип заданного объекта |
| Целочисленный-литерал | Цел64 |
| Вещественный-литерал | Вещ64 |
| Строковый-литерал | Строка |
| Многострочный-литерал | Строка |
| Символьный-литерал | Символ |
| '(' Выражение ')' | тип выражения |
| '[' ']' Указ-типа Конструктор-вектора | тип вектора: '[' ']' Указ-типа |

Замечание: если операнд является идентификатором импортированного модуля, то тип его не определен, а выражение корректно только если далее следует доступ к объекту модуля.

6.3. Доступ к полям, методам и импортируемым объектам

Доступ применяется к первичному выражению.

Доступ: `'.' Идентификатор`

В выражении `что-то.имя что-то` может быть:

- Выражением типа T , где T - это тип класса. В этом случае, имя должно быть идентификатором поля или метода типа T или базового класса T .
- Или идентификатором импортированного модуля. В этом случае, имя должно быть идентификатором экспортируемого объекта этого модуля.

Типом выражения доступа является тип объекта, поля или метода.

```

1 чел.возраст := 5
2
3 если чел.возраст < 18 { ... }
4
5 чел.указать возраст(25) // вызов метода

```

6.4. Индексация

Индексация применяется к первичному выражению.

Индексация: `'[' Выражение ']'`

Выражение `что-то[индекс]` обозначает элемент вектора или вариативного параметра, индекс которого определяется выражением `индекс`.

- `что-то` должно быть идентификатором, обозначающим вариативный параметр
- или `что-то` должно быть выражением типа T , где T - это тип вектора
- `индекс` должен быть выражением типа Цел64 или Байт

Если во время исполнения `индекс` выходит за границы вектора или параметра (`индекс < 0` | `индекс >= длина`), происходит *авария* (§7.13).

Типом выражения индексации является тип элемента вектора, если индексируется вектор или тип вариативного параметра, если индексируется вариативный параметр.

Так как индексация и конструктор вектора (§6.7) не всегда различимы синтаксически, выбор между ними происходит на уровне семантики, если выражение `что-то` обозначает тип, то это конструктор вектора, иначе индексация.

6.5. Преобразование типа

Преобразование типа применяется к первичному выражению, и преобразует, если это возможно, значение выражения к *целевому типу*, указанному в операции преобразования.

Преобразование: `'(:' 'осторожно'? Указ-типа ')'`

Если в преобразование есть служебное слово **осторожно**, то это небезопасная операция, которая рассматривается отдельно (§12.1).

Часть преобразований, например, преобразования между числовыми типами и символами, а так же строковые преобразования могут изменить представление значения первичного выражения и повлечь за собой затраты на выполнение.

Преобразование может привести к *аварийному завершению*, если условие преобразования не выполнено, см. столбец **Условие выполнения** в таблице. Если в столбце указано 'нет условия', то преобразование не может привести к *аварийному завершению*.

Разрешенные преобразования для базовых типов и векторов байтов и символов:

| Целевой тип | Тип выражения | Условие выполнения |
|-------------|---|---|
| Байт | Цел64, Слово64, Символ, Строковый литерал длины 1 | значение в диапазоне 0..255 |
| Цел64 | Байт, Символ, Строковый литерал длины 1 | нет условия |
| Цел64 | Слово64 | значение в диапазоне 0..Max(Цел64) |
| Цел64 | Вещ64 | нет условия |
| Слово64 | Байт, Символ, Строковый литерал длины 1 | нет условия |
| Слово64 | Цел64 | не отрицательное значение |
| Вещ64 | Цел64 | нет условия |
| Символ | Байт, Строковый литерал длины 1 | нет условия |
| Символ | Цел64, Слово64 | значение в диапазоне, разрешенном для Unicode символа |
| Строка | Символ, [] Символ, [] Байт | нет условия |
| [] Байт | Строка, Символ | нет условия |
| [] Символ | Строка | всегда |

Пример преобразований:

```

1 тип Байты = []Байт
2 пусть байты = "Привет" (:Байты)
3
4 пусть три = pi (:Цел64) // вещественное к целому

```

Разрешенные преобразования для остальных типов:

| Целевой тип | Тип выражения | Условие выполнения |
|--------------------|----------------------------|---|
| Класс Ц | Класс К | К является базовым классом класса Ц |
| Класс Ц | мб К, где К - класс | значение не равно пусто, и К является базовым классом класса Ц |
| Класс Ц | Протокол | <i>динамический</i> тип объекта, сохраненного в протоколе, равен Ц или является расширением Ц |
| Протокол П | Класс | <i>динамический</i> тип операнда реализует П |
| Протокол П | Протокол Р | <i>динамический</i> тип объекта, сохраненного в протоколе, реализует П |
| произвольный тип Т | полиморфный тип (*) | полиморфное значение содержит значение типа Т |

TBD: раз-

делить статическую и динамическую семантику.

Преобразование полиморфного типа к базовому:

```

1 фн хочу целое (п: *) {
2     пусть к = п (:Цел64)
3 }

```

Особенности преобразований:

- Преобразование к типу класса изменяет только тип (статический тип объекта), но не представление значения.
- Преобразование к типу протокола строит новое значение протокола.

6.6. Вызов функции или метода

Вызов применяется к первичному выражению. Это выражение

- является идентификатором стандартной функции (§8.1)
- или обозначает функцию, тогда это *вызов функции*
- или является выражением типа класс, тогда это *вызов метода*

Вызов: '(' Список-аргументов? ')'
Список-аргументов: Аргумент (',' Аргумент)* ',' '?'
Аргумент: Выражение '...'?

В вызове указывается список аргументов. Число аргументов в списке должно быть равно числу параметров в сигнатуре функции или метода, за исключением случая, когда в сигнатуре есть вариативный параметр. Для вариативного параметра число аргументов может быть от нуля и более.

Каждый аргумент в вызове является *выражением*. Если для вариативного параметра задан единственный аргумент, то после выражения может стоять знак операции `...`, делающий *развернутое выражение* (§6.6.1).

Для всех других аргументов, выражение должно быть *совместимо по присваиванию* с типом параметра (§11.2).

Типом выражения вызова является тип результата. Если в сигнатуре функции или метода тип результата отсутствует, то вызов не может быть операндом для любой операции.

```
1 фн Факториал(ц: Цел64) : Цел64 { /*тело*/ }
2
3 пусть рез = Факториал(5)
```

6.6.1. Развернутое выражение

Развернутое выражение может быть использовано только как аргумент для вариативного параметра и только, если это единственный аргумент для этого параметра. Если развернутое выражение `x...` используется как аргумент для параметра `п: ...T`, то должно выполняться одно из условий:

- `x` является вариативным параметром того же типа `T`
- `x` - это выражение типа вектора, причем тип вектора определен, как `[]T`

Пример использования развернутого вектора в качестве аргумента вариативного параметра:

```
1 фн Соединить(список: ...Строка) { /*тело*/ }
2
3 тип Строки = []Строка
4
5 фн Сохранить(строки: Строки) {
6     пусть текст = Соединить(строки...)
7     /* сохранение текста */
8 }
```

6.7. Конструктор вектора

Конструктор вектора применяется к первичному выражению, которое должно обозначать тип вектора. Результатом выполнения конструктора является вектор.

Конструктор-вектора: '[' (Значения | Элементы)? ']'
Значения: Значение (',' Значение)* ',' '?'
Значение: Выражение
Элементы: Элемент (',' Элемент)* ',' '?'
Элемент:
 ('длина' | 'выделить' | '*' | Индекс) ':' Выражение
Индекс: Выражение

Простой конструктор вектора задает последовательность выражений, каждое из которых должно быть *совместимым по присваиванию* с типом элемента вектора (§11.2). Первое выражение в последовательности определяет значение элемента вектора с индексом 0, и так далее. Если ни одного значения не задано, то длина вектора равно нулю.

```

1 тип Числа = []Цел64
2
3 пусть ч = Число[] // вектор длины 0
4 пусть ч1 = Число[1, 2, 3] // вектор длины 3

```

Пример использования конструктора вектора в вызове:

```

1 тип Строки = []Строка
2 фн Сохранить(строки: Строки) { /*тело*/ }
3
4 Сохранить(Строки["привет", "мир"])

```

Конструктор вектора может быть задан последовательностью *Элементов*, где каждый элемент - это пара, разделенная двоеточием. Если хотя бы один элемент задан парой, то все элементы конструктора должны быть также заданы парами.

Для *Элемента* X: выражение:

| X | выражение |
|----------|---|
| длина | задает длину конструируемого вектора |
| выделить | задает число элементов, выделяемых (резервируемых) для конструируемого вектора |
| * | задает значение по умолчанию для всех элементов, для которых оно не задано явно |
| Индекс | задает значение элемента с индексом <i>Индекс</i> |

Следующие условия должны выполняться для вектора, заданного *Элементами*:

- выражение для длина должно быть *совместимым по присваиванию* с типом Цел64
- выражение для выделить должно быть *совместимым по присваиванию* с типом Цел64. Оно определяет число элементов, выделенных для вектора, но не обязательно использованных. Если значение для выделить меньше или равно длине, то оно игнорируется.
- выражение для * должно быть *совместимым по присваиванию* с типом элемента вектора (§11.2)
- *Индекс* должен быть выражением *совместимым по присваиванию* с типом Цел64
- *Индекс* должен быть константным выражением (§6.15)
- выражение для *Индекса* должно быть *совместимым по присваиванию* с типом элемента вектора (§11.2)
- если длина вектора явно задана, то значение всех индексов должно быть в диапазоне [0..длина-1]. Это ограничение проверяется во время компиляции, если длина задана константным выражением, иначе во время исполнения
- если в конструкторе задаются значения не для всех индексов, задание значения по умолчанию является обязательным
- повторное задание значения для элемента с одним индексом является ошибкой

Если длина конструируемого вектора не задана явно, то она равна максимальному индексу + 1.

Примеры конструкторов:

```

1 тип Числа = []Цел64
2
3 пусть ч1 = Число[длина: 3, *: 0]
4 // вектор [0, 0, 0]
5
6 пусть ч2 = Число[длина: 5, *:0, 1: 1, 3: 3]
7 // вектор [0, 1, 0, 3, 0]
8
9 пусть ч3 = Число[*: 0, 1: 1, 3: 3]
10 // [0, 1, 0, 3], длина - макс индекс + 1

```

Пример ошибки - значение по умолчанию не задано:

```

1 пусть ч3 = Число[1: 1, 3: 3]

```

Вектор является динамическим массивом, длина которого может изменяться во время выполнения. Конструктор вектора выделяет память под $\max(\text{длина, выделить})$ элементов. Если к вектору добавляются элементы (см. §8.2.1) и новое число элементов не превышает число выделенных, то выделения новой памяти не происходит.

```
1 пусть ч1 = Число [выделить: 100]
2 // вектор [], выделено место до 100 элементов
```

TBD: методы выделено, выделить

6.8. Конструктор экземпляра класса

Конструктор класса применяется к первичному выражению, которое должно обозначать тип класса. Результатом выполнения конструктора является экземпляр класса.

Конструктор-класса: '{' Список-значений-полей? '}'
Список-значений-полей: Значение-поля (',' Значение-поля)* ',' '?
Значение-поля: Идентификатор ':' Значение

Для каждого *Значения поля* идентификатор должен обозначать поле конструируемого класса. В результирующем экземпляре значением этого поля будет значение выражения *Значение*. *Значение* должно быть выражением, *совместимым по присваиванию* с типом поля (§11.2).

Указание нескольких значений для одного идентификатора поля является ошибкой.

Конструктор должен задавать значения для каждого поля класса с *поздней инициализацией* (§5.5.1).

Если класс импортирован из другого модуля, в конструкторе не могут использоваться идентификаторы не экспортированных полей. Если при этом у не экспортированного поля задана *поздняя инициализация*, конструктор такого класса может быть использован только в модуле, содержащем описание класса.

```
1 тип Человек = класс {
2     имя: Строка = позже
3     возраст := 0
4 }
5
6 пусть Вася = Человек{имя: "Вася", возраст: 25}
```

Пример ошибки - значение поля "имя" с поздней инициализацией не задано:

```
1 пусть Некто := Человек{возраст: 25}
```

6.9. Подтверждение типа

Подтверждение типа применяется к первичному выражению, которое должно быть *может быть* типа (§5.6).

Подтверждение-типа: '^'

Если значение первичного выражение, тип которого есть **мб** *T* равно 'пусто', то происходит *авария* (§7.13), иначе тип выражения становится равным *T*.

```
1 пусть кличка: мб Строка = пусто
2 ...
3 кличка = "Мурка"
4 ...
5 напечатать (кличка^)
```

6.10. Приоритеты операции

Приоритет операций определяет порядок вычисления выражения. Операции с более высоким приоритетом выполняются до выполнения операций с более низким приоритетом.

Унарные операции (§6.11) имеют высший приоритет, то есть выражение $-x + 1$ выполняется как $(-x) + 1$.

Таблица определяет приоритеты для бинарных операций (§6.14), операции проверки типа (§6.12) и операции определения попутной переменной (§6.13):

| Приоритет | Операция |
|-----------|--------------------|
| 6 | * / % & < > |
| 5 | + - ^ |
| 4 | = # < <= > >= типа |
| 3 | => |
| 2 | & |
| 1 | |

Операции умножения имеют самый высокий приоритет, а операция логическое ИЛИ (|) имеет самый низкий приоритет.

Бинарные операции с одинаковым приоритетом ассоциируются слева направо. Например, $x / y * z$ эквивалентно $(x / y) * z$.

6.11. Унарные операции

Унарная-операция: '-' | '~' | '|~'

| Операция | Действие | Разрешенные типы |
|----------|----------------------|-----------------------------|
| - | отрицание | Байт, Цел64, Слово64, Вещ64 |
| ~ | логическое НЕ | Лог |
| ~ | инвертирование битов | Байт, Цел64, Слово64 |

Тип результата унарных операций равен типу операнда.

6.12. Проверка типа

Операция *проверка типа* проверяет *динамический* тип выражения.

Проверка-типа: Выражение 'типа' Указ-типа

Операция применима к выражению типа класс, мб класс или протокол. Тип, указанный в операции справа называется *целевым типом*. Он должен быть типом класса или протокола. Тип результата - логический.

Для операнда мб типа со значением 'пусто' операция возвращает ложь. Для непустых операндов:

| Целевой тип | Тип операнда | Проверка выдает истину, если |
|-------------|--------------|--|
| Класс Ц | Класс | динамический тип операнда равен Ц или является расширением Ц |
| Класс Ц | Протокол | динамический тип объекта, сохраненного в протоколе, равен Ц или является расширением Ц |
| Протокол П | Класс | динамический тип операнда реализует П |
| Протокол П | Протокол Р | динамический тип объекта, сохраненного в протоколе, реализует П |

Проверка динамического типа объекта класса:

```

1 тип К1 = класс {}
2 тип К2 = класс (К1) {}
3
4 пусть к1: К1 = К1{}
5 пусть к2: К1 = К2{}
6 пусть к3: мб К1 = пусто
7
8 пусть б := к1 типа К2 // ложь
9 б := к2 типа К2 // истина
10 б := к3 типа К2 // ложь

```

6.13. Описание попутной переменной

Операция => (читается *тогда*) описывает новую *попутную* переменную, если значением левого выражения является истина. Если значением выражения является ложь, никакого действия не происходит. Операция выдает результат типа Лог, равный результату выражения.

Ограничения на использование операции:

- Операция может быть использована только в условии оператора **если** (§7.6.1) или оператора **надо** (§7.7.1). Область действия переменной задается отдельно для каждого оператора.
- Если операция стоит в подвыражении, то подвыражения должны быть соединены операцией *логическое И*.

Ограничения на вид выражения и операции определяется таблицей:

| Вид выражения | Тип операнда 'x' | Тип переменной | Значение переменной |
|---------------|--------------------|----------------|---------------------|
| x типа Ц | класс или протокол | Ц | x |
| x типа Ц | мб Т | Ц | x^ |
| x # пусто | мб Т | Т | x^ |
| x | мб Т | Т | x^ |

Попутная переменная является *переменной с единственным присваиванием*, её значение не может быть изменено.

Попутная переменная для проверки типа:

```

1 тип Фигура = класс {}
2 тип Круг = класс (Фигура) { радиус: Вещ64 }
3
4 фн площадь (ф: Фигура): Вещ64 {
5     если фигура типа Круг => круг {
6         // круг здесь типа Круг
7         вернуть круг.радиус * круг.радиус
8     }
9     авария ("площадь неизвестна")
10 }
```

Попутная переменная для проверки *может быть* значения:

```

1 фн обход (узел: мб Узел) {
2     если узел # пусто => узел' {
3         // узел' здесь типа Узел
4         обход (узел'.Левый)
5         обход (узел'.Правый)
6     }
7 }
```

Сокращенная форма:

```

1 если узел => узел' { // # пусто можно убрать
2     // узел' здесь типа Узел
3 }
```

Описание попутной переменной нельзя использовать вместе с логическим ИЛИ:

```

1 если нужна-проверка | узел # пусто => узел' | {
2 }
```

6.14. Бинарные операции

Бинарная-операция

- : Арифметические
- | Сравнения
- | Логические
- | Битовые
- | Сдвиги

Арифметические: '+' | '-' | '*' | '/' | '%'

Сравнения: '=' | '#' | '<' | '<=' | '>' | '>='
 Логические: '|' | '&'
 Битовые: '||' | '|&' | '|^'
 Сдвиги: '|<' | '|>'

Замечание: знаки всех операции, работающих с битовым представлением значений, начиная с версии 0.9.6 (§A.8), содержат символ *вертикальная черта* (|). См. §6.14.4, §6.14.5, §6.11 (инвертирование битов).

6.14.1. Арифметические операции

Арифметические операторы применяются к числовым операндам, выдают результат того же типа, что и тип операндов. Типы левого и правого операнда должны совпадать.

| Операция | Действие | Разрешенные типы |
|----------|--------------------|-----------------------------|
| + | сумма | Байт, Цел64, Слово64, Вещ64 |
| - | разница | Байт, Цел64, Слово64, Вещ64 |
| * | произведение | Байт, Цел64, Слово64, Вещ64 |
| / | деление | Байт, Цел64, Слово64, Вещ64 |
| % | остаток от деления | Байт, Цел64, Слово64 |

6.14.2. Операции сравнения

Операции сравнения выдают результат типа Лог.

Все операции сравнения '=', '#', '<', '<=', '>', '>=' применимы к операндам типов Байт, Цел64, Слово64, Цел64 и Символ. Типы левого и правого операнда, при этом, должны совпадать.

Операции '=', '#' также применимы в следующих случаях:

| Тип левого операнда | Правый операнд |
|---------------------|---|
| Лог | операнд типа Лог |
| Строка | операнд типа Строка |
| класс <i>T</i> | операнд типа <i>T</i> |
| мб <i>T</i> | 'пусто' или операнд типа мб <i>T</i> |

6.14.3. Логические операции

Операции *логическое ИЛИ* (|) и *логическое И* (&) применимы к операндам типа Лог и выдают результат типа Лог.

a | *b* означает если *a*, то истина, иначе *b*
a & *b* означает если *a*, то *b*, иначе ложь

6.14.4. Битовые операции

Операции *битовое ИЛИ* (|), *битовое И* (&) и *битовое исключаящее ИЛИ* (|^) применимы к операндам целых типов (Байт, Цел64, Слово64), оба операнда должны быть одного типа. Тип результата равен типу операндов.

6.14.5. Операции сдвига

Операции *сдвиг влево* (|<) и *сдвиг вправо* (|>) применимы к операндам целых типов и выдают результат того же типа, что и тип первого операнда.

6.15. Константные выражения

Константные выражения - это выражения, которые могут быть вычислены во время компиляции.

Минимальные требования к компилятору языка Тривиль: компилятор должен вычислять выражение, состоящее из одного операнда, который является литералом или идентификатором, обозначающим константу или функцию (не метод).

7. Операторы

Операторы задают действия.

Оператор

- : Локальное-описание
- | Простой-оператор
- | Оператор-если
- | Оператор-надо
- | Оператор-выбор
- | Оператор-пока
- | Оператор-цикл
- | Оператор-вернуть
- | Оператор-прервать
- | Оператор-авария

Простой-оператор

- : Оператор-выражение
 - | Оператор-присваивания
 - | Инкремент
 - | Декремент
-

7.1. Списки операторов и Блоки

Операторы сгруппированы в списки *списки операторов*, которые могут включать локальные описания. *Блок* - это список операторов (возможно, пустой) в фигурных скобках. *Блоки* задают последовательность действий для функций (§4.5), методов (§4.6), входа в модуль (§9.3) и частей составных операторов.

Список-операторов: `Оператор (Разделитель Оператор)*`

Блок: `'{' Список-операторов? '}'`

7.2. Локальные описания

Локальное описание определяет идентификатор, для которого областью действия (§4.1) является часть списка операторов, от точки завершения описания до завершающей списка, исключая вложенные блоки, в которых описан такой же идентификатор.

Локальное-описание: `Описание-переменной`

TBD: Нужны ли локальные константы? Или это задача оптимизации?

7.3. Выражение, как оператор

Выражение, являющееся вызовом функции или метода, кроме вызова стандартных функций (§8.1), может быть использовано как оператор. Результат вызова при этом игнорируется.

Оператор-выражение: `Выражение`

```
1 напечатать ("Привет")
2 Факториал (5)
```

Не могут быть использованы в качестве оператора:

```
1 1 + 1
2 длина (a)
```

7.4. Оператор присваивания

Присваивание заменяет текущее значение, хранящееся в переменной, новым значением, заданным выражением.

Выражение в левой части должно быть *изменяемым*:

| Левое выражение | | Условие |
|-----------------|-----------------|--|
| Имя объекта | пер или мод.пер | <i>пер</i> - это изменяемая переменная (§4.4) |
| Доступ к полю | что-то.поле | <i>что-то</i> - это изменяемое выражение, <i>поле</i> - это изменяемое поле (§5.5.1) |
| Индексация | что-то [индекс] | <i>что-то</i> - это изменяемое выражение |
| Преобразование | что-то (: Тип) | <i>что-то</i> - это изменяемое выражение |

Выражение в правой части должно быть *совместимо по присваиванию* с типом левого выражение (§11.2).

```
1 пусть ц := 0
2 пусть байты := Байты[1, 2, 3]
3 ц := 2
4 байты[ц] = 5
```

Ошибка присваивания в неизменяемое выражение:

```
1 пусть ц = 0
2 ц := 2
```

7.5. Инкремент и декремент

Операторы *Инкремент* ('++') и *Декремент* ('--') увеличивают или уменьшают свои операнды на 1.

Инкремент: Выражение '++'

Декремент: Выражение '--'

Как и в случае присваивания (§7.4), выражение должно быть *изменяемым*. Тип выражения должен быть одного из типов: Байт, Цел64, Слово64.

```
1 пусть ц := 1
2 пусть байты := Байты[1, 2, 3]
3 байты[ц]++
```

7.6. Оператор 'если'

Оператор **если** определяет условное выполнение двух ветвей в соответствии со значением логического выражения. Если выражение принимает значение истина, выполняется Блок первой ветви (*Блок-если*), в противном случае, выполняется ветвь **иначе**, если она задана (*Блок-иначе*).

Оператор-если:

'если' Выражение Блок ('иначе' (Оператор-если | Блок))?

```
1 если ц > макс { ц := макс }
2
3 если Буква?(сим) { Имя() }
4 иначе если Цифра?(сим) { Число() }
5 иначе { Ошибка!() }
```

7.6.1. Попутная переменная в операторе 'если'

В область действия попутной переменной в операторе **если** входит:

- Часть условного выражения от идентификатора переменной и до завершения условия. Замечание: подвыражения в условии могут быть соединены только операцией логическое И (&);

- Список операторов *блока-если*.

```

1  fn тест(усл1: Лог, усл2: Лог, с: мб Строка): Строка {
2      если усл1 & с => с' & (усл2 | с' # "") {
3          вернуть с'
4      }
5      вернуть ""
6  }

```

Список операторов *блока-иначе* не входит в область действия попутной переменной:

```

1  если с => с' {
2      вернуть с' // попутная переменная видна
3  } иначе {
4      вернуть с' // ошибка компиляции: попутная переменная не видна
5  }

```

7.7. Оператор 'надо'

Оператор **надо** используется для завершения исполнения операторов некоторого контекста, если условие, заданное логическим выражением, не выполнено.

Оператор-надо:

'надо' Выражение 'иначе' (Завершающий-оператор | Блок)

Завершающий-оператор

: Оператор-вернуть
| Оператор-прервать
| Оператор-авария

Если в ветви **иначе** стоит *Блок*, то последним оператором блока должен быть завершающий оператор.

| Завершающий оператор | Действие |
|-------------------------|---|
| вернуть (§7.12) | выход из тела функции, метода или входа |
| прервать (§7.11) | выход из ближайшего объемлющего цикла |
| авария (§7.13) | аварийное завершение программы |

```

1  надо делитель # 0 иначе авария("деление на ноль")
2
3  надо число > 1 иначе вернуть 1

```

7.7.1. Попутная переменная в операторе 'надо'

В область действия попутной переменной в операторе **надо** входит:

- Часть условного выражения от идентификатора переменной и до завершения условия. Замечание: подвыражения в условии могут быть соединены только операцией логическое И (&);
- Часть списка операторов, в котором расположен оператор **надо**, от точки завершения оператора **надо** и до завершения этого списка операторов.

```

1  fn тест(усл1: Лог, усл2: Лог, с: мб Строка): Строка {
2      надо усл1 & с => с' & (усл2 | с' # "") иначе вернуть ""
3      вернуть с'
4  }

```

Операторы части *иначе* не входят в область действия попутной переменной:

```

1  надо с => с' иначе вернуть с' // ошибка компиляции

```

7.8. Оператор 'выбор'

Оператор **выбор** выбирает одну ветвь выполнения из нескольких вариантов.

Оператор-выбор

: Выбор-по-выражению
| Выбор-по-предикатам
| Выбор-по-типу

Есть три разновидности оператора выбора

- *выбор по выражению* (§7.8.1)
- *выбор по предикатам* (§7.8.2)
- *выбор по типу* (§7.8.3)

7.8.1. Выбор по выражению

В операторе *выбор по выражению* выбор идет по значению выражения оператора (значение сравнивается со значениями выражений в вариантах).

Выбор-по-выражению:

```
'выбор' Выражение '{'  
  Вариант*  
  ('другое' Список-операторов)?  
'}'
```

Вариант:

```
'когда' Выражение (',' Выражение) * ':'  
Список-операторов?
```

При выполнении вычисляется *Выражение* оператора, затем выражения в *Вариантах* вычисляются слева направо и сверху вниз. Как только значение выражения *Варианта* стало равным значению выражения оператора, выполняется список операторов этого *Варианта*. Остальные варианты игнорируются. Если ни один из вариантов не выполнен, и есть ветка **другое**, выполняется список операторов этой ветки.

Тип каждого выражения в вариантах должен быть равен эквивалентен типу первого выражения (§11.1).

```
1  выбор x {  
2  когда 0: вернуть "ничего"  
3  когда 1: вернуть "один"  
4  когда 2: вернуть "два"  
5  другое вернуть "много"  
6  }
```

7.8.2. Выбор по предикатам

Если выражение после служебного слова **выбор** отсутствует, то это оператор *выбора по предикатам*.

Выбор-по-предикатам:

```
'выбор' '{'  
  Вариант*  
  ('другое' Список-операторов)?  
'}'
```

Вариант:

```
'когда' Выражение (',' Выражение) * ':'  
Список-операторов?
```

В этом случае, каждое выражение в *Вариантах* должно быть выражением логического типа. Эти выражения вычисляются слева направо и сверху вниз до тех пор, когда значение выражения не равно истине. Как только значение выражения стало равно истине, выполняется список операторов этого варианта. Остальные варианты игнорируются. Если ни один из вариантов не выполнен, и есть ветка **другое**, выполняется список операторов этой ветки.

```
1  выбор x {  
2  когда x < 0: вернуть "отрицательное"  
3  когда x > 0: вернуть "положительное"  
4  другое вернуть "ноль"  
5  }
```

7.8.3. Выбор по типу

В операторе *выбор по типу* выбор ветви выполнения происходит по типу выражения.

Выбор-по-типу:

```
'выбор' Переменная-варианта? 'тип' Выражение '{'  
  Вариант-типа*  
  ('другое' Список-операторов?)?  
'}'
```

Переменная-варианта:

```
'пусть' Идентификатор ':'
```

Вариант-типа:

```
'когда' Указ-типа (',' Указ-типа)* ':'  
Список-операторов?
```

Рассмотрим сначала оператор, в котором *переменная варианта* не указана.

При выполнении такого оператора вычисляется *Выражение*, тип которого должен быть типом класса (§5.5). Для дальнейшего сравнения используется тип объекта, то есть тип, с которым объект был создан, иначе говоря, динамический тип.

В следующем примере для сравнения будет использоваться тип `K2`, так как это тип объекта `k`:

```
1 тип K1 = класс {}  
2 тип K2 = класс (K1) {}  
3 вход{  
4   пусть k: K = K2{  
5   выбор тип k { ... }  
6 }
```

Динамический тип выражения сравнивается слева направо и сверху вниз на равенство типам, указанным в вариантах. Как только один из типов *Варианта* оказался равен типу выражения, выполняется список операторов этого *Варианта*. Остальные варианты игнорируются. Если ни один из вариантов не выполнен, и есть ветка **другое**, выполняется список операторов этой ветки.

Сравнение типа выражение оператора выполняется на строгое равенство, наследование не учитывается. Таким образом, в примере работает ветка `когда K2`, независимо от порядка веток в операторе.

```
1 тип K1 = класс {}  
2 тип K2 = класс (K1) {}  
3 вход{  
4   пусть k: K = K2{  
5   выбор тип k {  
6     когда K1:  
7     когда K2: // работает эта ветка  
8   }  
9 }
```

Если в операторе указана *переменная варианта*, то в каждой ветке варианта доступна переменная с именем, заданным идентификатором, и типом, указанным в варианте. При этом в варианте должен быть указан только один тип. Переменная варианта доступна также в ветке **другое**, тип её равен типу выражения оператора.

```
1 тип Человек = класс { имя := "" }  
2 тип Работник = класс (Человек) {  
3   зарплата := 0.0  
4 }  
5 фн зарплата(чел: Человек): Вещ64 {  
6   выбор пусть x: тип чел {  
7     когда Работник: вернуть x.зарплата  
8     другое вернуть 0.0  
9   }  
10 }
```

Переменная варианта является *переменной с единственным присваиванием*, её значение изменить нельзя.

7.9. Оператор 'пока'

Оператор **пока** определяет повторное выполнение блока до тех пор, пока значение логического выражение равно истинна. Выражение вычисляется перед каждой итерацией.

Оператор-пока: 'пока' Выражение Блок

```
1 пока a < б {
2   а := а * 2
3 }
```

7.10. Оператор цикл

Оператор **цикл** определяет выполнение блока для всех элементов вектора или другого *индексируемого* объекта.

Оператор-цикл:

'цикл' ПеременныеЦикла 'среди' Выражение Блок
ПеременныеЦикла
: ПеременнаяЦикла
| '[' ПеременнаяЦикла ']' ПеременнаяЦикла?
ПеременнаяЦикла: Идентификатор

Выражение цикла должно быть *индексируемым*, то есть

- тип выражения должен быть типом вектора
- или выражение должно обозначать вариативный параметр.

Оператор позволяет задать две переменные цикла или только одну из них. В теле цикла, значением *переменной индекса*, заданной в квадратных скобках, является текущий индекс (от 0 до длины-1), а *переменной элемента* - значение элемента по этому индексу.

Типы переменных цикла задаются неявно:

- тип переменной индекса есть Цел64
- тип переменной элемента равен типу элемента индексируемого выражения

TBD: Надо ли задавать типы переменных цикла явно?

```
1 пусть числа = Числа[9, 8, 7]
2 цикл [№]число среди числа {
3   вывод.ф("%v:%v, ", №, число)
4 }
5 // вывод: 0:9, 1:8, 2:7,
```

Каждая переменная цикла является *переменной с единственным присваиванием*, её значение в теле цикла изменить нельзя.

7.11. Оператор 'прервать'

Оператор **прервать** завершает выполнение самого внутреннего оператора цикла в рамках одной и той же функции.

Оператор-прервать: 'прервать'

```
1 пока истина {
2   а := а * 2
3   если а > б { прервать }
4 }
```

7.12. Оператор 'вернуть'

Оператор **вернуть** завершает выполнение тела функции, метода или входа, и, возможно, возвращает значение.

Оператор-вернуть: `'вернуть' (Выражение | Разделитель)`

Если оператор **вернуть** используется в теле входа или в теле функции или метода, в сигнатуре которых не указан тип результата, то выражение в операторе должно отсутствовать.

И наоборот, если в оператор используется в теле функции или метода, в сигнатуре которых указан тип результата, выражение должно быть и оно должно быть *совместимо по присваиванию* с типом результата (§11.2).

```
1 fn Факториал(ц: Цел64): Цел64 {
2     если ц <= 1 { вернуть 1 }
3     вернуть ц * Факториал(ц - 1)
4 }
```

Если выражение в операторе присутствует, то оно должно начинаться на одной строке с служебным словом **вернуть**, иначе компилятор выдает ошибку:

```
1 fn цифра?(сим: Символ): Лог {
2     вернуть // ошибка компиляции
3     '0' <= сим & сим <= '9'
4 }
```

А так верно:

```
1 fn цифра?(сим: Символ): Лог {
2     вернуть '0' <= сим
3     & сим <= '9'
4 }
```

7.13. Оператор 'авария' и аварийная ситуация

Оператор **авария** запускает аварийную ситуацию, которая, как правило, приводит к аварийному завершению программы.

Оператор-авария: `'авария' '(' Выражение ')'`

Тип выражения должен быть Строка. Значение этого выражения используется в сообщении об аварийной ситуации.

Кроме явного запуска аварийной ситуации, аварийная ситуация может быть запущена неявно, в следующих случаях:

- Выход индекса за границы вектора при индексации
- Операция подтверждения типа выполнена над объектом со значением 'пусто'
- Недопустимое преобразование типа
- Недопустимая последовательность байт в кодировке UTF-8
- Невозможность выделения памяти для динамического объекта
- Нереализованная возможность в системе поддержки выполнения

Язык Тривиль не содержит средств перехвата аварийной ситуации или восстановления после него. Такие средства могут быть добавлены на уровне библиотек.

8. Стандартные функции и методы

8.1. Стандартные функции

8.1.1. Функция 'длина'

Стандартная функция `длина` возвращает длину строк, векторов и вариативных параметров, см. также (§5.1.2). Тип результата: `Цел64`.

Длина строки считается в символах. Для остальных типов длина выдает число элементов.

```
1 пусть д1 = длина ("Привет") // равно 6
2
3 пусть д2 = длина ("Привет" (:Байты)) // равно 12
```

8.1.2. Функции для полиморфных параметров

Стандартные функции `тег` и `нечто` предназначены для работы с полиморфными параметрами (§4.5.2)

Аргумент функция `тег` может быть типом или полиморфным параметром или элементом вариативного полиморфного параметра. Для каждого типа определен его *Тег*, который является постоянным объектом времени исполнения. Постоянный, в данном контексте, контексте, означает, что Тег типа не меняется во время исполнения программы. Тег представлен значением типа `Слово64`.

| Вид аргумента | Пример | Результат |
|---|----------------------------|---------------------|
| предопределенный тип | <code>тег(Цел64)</code> | Тег указанного типа |
| описанный тип | <code>тег(Байты)</code> | Тег указанного типа |
| полиморфный параметр | <code>тег(п)</code> | Тег параметра |
| элемент вариативного полиморфного параметра | <code>тег(п[номер])</code> | Тег элемента |

Аргумент функция `нечто` должен быть полиморфным параметром или элементом вариативного полиморфного параметра. Функция выдает результат типа `Слово64`, которые является скрытым представлением значения переданного в полиморфный параметр.

К результату функции `нечто` может применяется преобразование типа (§6.5) или *осторожное* преобразование типа (§12.1).

```
1 фн Что это? (п: *) {
2     если тег(п) = тег(Цел64) {
3         вывод.ф("это число: %v", нечто(п) (:Цел64))
4     }
5     иначе {
6         вывод.ф("это не число")
7     }
8 }
```

8.1.3. Функция 'сигнатура'

Стандартная функция `сигнатура` должна вызываться с одним аргументом, который должен быть типом функции (§5.8). Функция возвращает строку, которая содержит представление сигнатуры функцию в некоторой кодировке, зависящей от реализации. Результат функции используется для динамической проверки типа.

8.2. Встроенные методы для векторов

Для всех типов векторов определены стандартные методы:

- добавить
- сократить

TBD: Уточнить набор и семантику методов.

8.2.1. Метод 'добавить'

Метод `добавить` можно применить к любому вектору. Если вектор типа `[]T`, то метод можно рассматривать, как описанный с заголовком:

```
1 фн (в: []T) добавить (аргументы: ...T)
```

Это означает, что в вызове метода можно указать любое количество аргументов типа `T` или одно *развернутое выражение* (§6.6).

Вызов метода добавляет к вектору указанные аргументы.

```
1 тип Числа = []Цел64
2
3 пусть фиб = Числа[1, 1]
4 фиб.добавить(2, 3, 5) // теперь вектор: [1, 1, 2, 3, 5]
```

8.2.2. Метод 'сократить'

Метод `сократить` позволяет уменьшить длину вектора. Для вектор типа `[]T`, то метод можно рассматривать, как описанный с заголовком:

```
1 фн (в: []T) сократить (новая-длина: Цел64)
```

Если `новая-длина` больше чем текущая длина, то вызов метода не изменяет вектор.

Если `новая-длина` меньше или равна нулю, то длина вектора становится равной 0.

```
1 тип Числа = []Цел64
2
3 пусть а = Числа[1, 2, 3]
4 а.сократить(5) // вектор не изменился, по-прежнему: [1, 2, 3]
5 а.сократить(2) // вектор стал: [1, 2]
6 а.сократить(0) // вектор стал: []
```

Вызов метода не влияет на количество выделенной для вектора памяти (§6.7).

9. Модули

Программа на языке Тривиль состоит из модулей (единиц компиляции), исходный текст каждого модуля расположен в одном или нескольких исходных файлах.

Каждый исходный файл состоит из заголовка модуля, за которым следует, возможно, пустой список импорта, за которым следует, возможно, пустой набор описаний типов, констант, переменных, функций и методов и, возможно, вход в модуль.

Модуль : Исходный-файл+

Исходный-файл:

Заголовок-модуля

Список-импорта

Описание-или-вход*

Заголовок-модуля:

```
'модуль' Идентификатор Разделитель  
( 'осторожно' Разделитель ) ?
```

Описание-или-вход: Описание | Вход

9.1. Заголовок модуля

Каждый исходный файл начинается с заголовка модуля. Заголовок содержит идентификатор, который определяет модуль, к которому принадлежит файл. Идентификатор не принадлежит никакой области действия и не связан ни с каким объектом.

Заголовок модуля может содержать признак **осторожно**, который означает, что в исходном файле можно использовать *небезопасные операции* (§12).

9.2. Импорт

Наличие импорта в исходном файле указывает, что этот исходный файл зависит от функциональности импортируемого модуля. Импорт обеспечивает доступ к экспортированным идентификаторам импортируемого модуля.

Каждый импорт содержит путь импорта, указывающий на размещение импортируемого модуля.

Список-импорта: Импорт*

Импорт: 'импорт' Путь-импорта Разделитель

Путь-импорта: ''' Путь-в-хранилище | Файловый-путь '''

TBD: явное задание имени импорта.

Импорт добавляет описание идентификатора, текстуально равного последнему имени в пути импорта, который будет использоваться для доступа к экспортированным объектам импортируемого модуля.

```
1  модуль x  
2  
3  импорт "std::вывод"  
4  
5  вход {  
6      вывод.ф("Привет!")  
7  }
```

Путь импорта может быть задан как *путь в хранилище* исходных текстов или как *файловый путь*. В любом случае, в итоге, он должен указывать на папку, содержащую исходные файлы импортируемого модуля.

Путь-в-хранилище:

Имя-хранилища '::' Имя-папки ('/' Имя-папки)*

Путь в хранилище состоит из имени хранилища, которое явным или неявным образом (см. README) должно быть привязано к определенной папке файловой системы и пути, относительно этой папки:

```
1 импорт "std:юникод/utf8"  
2  
3 пусть декодер = utf8.декодер(...)
```

Язык не определяет формально набор символов, которые могут использоваться в *имени хранилища* и *имени папки*, надеясь на здравый смысл разработчиков.

Файловый путь может быть абсолютным или относительным (в терминах инструментальной платформы). Не рекомендуется использовать абсолютные пути, так как это приводит к непереносимому коду. Относительный путь всегда трактуется как путь относительно той папки, в которой запущен компилятор (рабочая папка).

Например в случае импорта:

```
1 импорт "трик/лексер"
```

будет импортирован модуль, который расположен в подпапке 'трик/лексер' рабочей папки.

Имена папок должны разделяться символом / (слеш) независимо от инструментальной платформы, на которой используется компилятор.

9.3. Вход или инициализация модуля

Модуль может содержать действия, которые выполняются при инициализации модуля - *вход в модуль*.

```
1 Вход: 'вход' Блок
```

TBD: описать: переменные с поздней инициализацией

Система должна обеспечивать следующие условия инициализации модуля М:

- инициализация модуля выполняется один раз
- инициализации М выполняется **после** инициализации всех модулей, которые импортирует М
- инициализации М выполняется **до** инициализации тех модулей, которые импортируют М

9.4. Инициализация и исполнение программы

Программа состоит из головного модуля и всех прямо или косвенно импортированных модулей.

Исполнение программы состоит из:

- Инициализации всех модулей, импортированных из головного модуля. Это приводит к рекурсивной инициализации всех используемых модулей программы. Для корректной инициализации граф импорта должен быть ациклическим.
- И, затем, выполнение входа головного модуля.

10. Обобщенные модули

Практически все современные языки программирования содержат конструкции для определения обобщенных (generic) типов и функций. Одна из важнейших областей применения обобщенных конструкций - библиотечная реализация полиморфных контейнеров: стек, очередь, деревья, словари (map).

Как правило, семантика обобщенных типов и их реализации весьма нетривиальна, и тем самым не подходит для Тривиль.

Для языка Тривиль выбрано решение предельно простое во всех отношениях. Обобщение в нем сделано на уровне модулей, то есть только модуль целиком может быть обобщенным или, другими словами, параметризованным. Причем, он может быть параметризован не только типом, а также любым языковым объектом, например, константой или функцией.

Обобщенный модуль - это модуль, в котором опущено то определение (или определения), которым он параметризован. Другими словами - обобщенный модуль является недоопределенным.

Рассмотрим, в качестве примера, обобщенный модуль `Стек` (полный текст см. "std/контейнеры/стек"):

```
1  модуль стек
2
3  тип Элементы = []Элемент
4
5  тип Стек* = класс {
6      элементы = Элементы[]
7      верх: Цел64 := 0 // ... [0..верх[
8  }
9
10 фн (с: Стек) положить*(э: Элемент) {
11     если с.верх = длина(с.элементы) {
12         с.элементы.добавить(э)
13     } иначе {
14         с.элементы[с.верх] := э
15     }
16     с.верх++
17 }
18 /* другие методы */
```

Тип элемента стека - это *Элемент*, и этот тип не определен в самом модуле.

Для использования конкретного стека, например, стека целых чисел, необходимо определить *настройку*. Настройка - отдельный модуль, в котором указан путь к обобщенному модулю и заданы параметры обобщенного модуля:

```
1  настройка "std/контейнеры/стек"
2  модуль стек-цел
3
4  тип Элемент = Цел64
```

Настроенный модуль используется обычным образом. Например, если стек-цел размещен в папке "модули/стек-цел", то использование его выглядит так:

```
1  модуль пример
2
3  импорт "модули/стек-цел"
4
5  вход {
6      пусть с = стек-цел.Стек{}
7      с.положить(1)
8  }
```

Несколько настроек можно использовать одновременно, например, если модель стек настроен также на Строку:

```

1  модуль пример
2
3  импорт "стд/вывод"
4  импорт "модули/стек-цел"
5  импорт "модули/стек-строk"
6
7  вход {
8      пусть c1 = стек-цел.Стек{ }
9      пусть c2 = стек-строk.Стек{ }
10
11     c1.положить (777)
12     c2.положить ("привет")

```

10.1. Последствия принятого решения

Сравнение подхода с другими известными подходами выходит за рамки описания языка. Замечу только, что главной причиной выбора такого подхода является тривиальность реализации и простота использования.

Прямым (и несколько непривычным) следствием подхода является то, что обобщенный модуль не может быть использован без настройки. Попытка импорта обобщенного модуля приведет к ошибкам компиляции.

Перед компиляцией компилятор соберет модуль, например, стек-цел, из двух частей:

- из обобщенного модуля, в котором параметры не определены
- и из модуля с настройкой, в котором параметры определены

Такой собранный модуль является полностью определенным, он компилируется обычным образом. Каждый настроенный модуль компилируется отдельно и для него строится отдельный код. Так код для стек-цел никак не связан с кодом для стек-строk.

Говоря в принятых терминах, Тривиль использует *мономорфизацию на уровне исходного текста*, тем самым получая оптимальный по скорости код за счет увеличения размера кода. Вся работа с обобщенными модулями происходит во время компиляции, во время исполнения все модули являются полностью определенными.

Замечу, что никаких синтаксических конструкций для ограничения параметров (constraints) в языке не предусмотрено, они накладываются неявно. Например, если в обобщенном модуле использовать операцию <, то настройка на арифметический тип будет работать, а настройка на тип, для которого эта операция не определена, приведет к ошибкам компиляции.

10.2. Уточненный синтаксис модуля

Уточненный синтаксис модуля с добавлением опциональной настройки (исходное определение см. §9):

```

Модуль : Исходный-файл+
Исходный-файл :
    ('настройка' Путь-импорта) ?
Заголовок-модуля
Список-импорта
Описание-или-вход*

```

11. Правила совместимости

11.1. Эквивалентность типов

Во многих случаях, например, для большинства бинарных операций или при переопределении методов, требуется эквивалентность типов.

Два типа $T1$ и $T2$ эквивалентны, если выполняется одно из условий:

- типы $T1$ и $T2$ это один и тот же тип, возможно, переименованный, так как переименование типа не создает новый тип
- типы $T1$ и $T2$ это типы векторов и типы их элементов эквивалентны
- типы $T1$ и $T2$ это *может быть* типы и их базовые типы эквивалентны

Переименование типа не создает новый тип, таким образом, после описания:

```
1 тип Числа = Цел64
2 пусть ч: Число = 1
3 пусть ц: Цел64 = 2
4 пусть с: Слово64 = 3
```

типы переменных $ч$ и $ц$ эквивалентны, так как это один и тот же тип. В то же время, типы переменных $с$ и $ц$ не эквивалентны, так как это разные типы.

11.2. Совместимость по присваиванию

Совместимость по присваиванию используется в следующих случаях:

- инициализация констант и переменных с явным типом: пусть $x: T = \text{выражение}$, выражение должно быть совместимо с типом T
- оператор присваивания: $x := \text{выражение}$
- оператор вернуть выражение, выражение должно быть совместимо с типом результата функции
- передача фактических параметров при вызове: $\Phi(\text{аргумент1}, \dots, \text{аргументN})$, аргумент должен быть совместим с типом параметра

Значение типа $TВ$ совместимо по присваиванию с целевым типом $TП$ (типом переменной, константы или параметра), если выполняется одно из условий:

- $TП$ и $TВ$ эквивалентны
- $TП$ это целый тип (Байт, Цел64, Слово64), а значение - это литерал целого типа, причем значение литерала входит в допустимый диапазон значений $TП$
- $TП$ и $TВ$ это типы векторов и типы их элементов эквивалентны
- $TП$ это *мб* $T1$, а значение есть 'пусто' или $TП$ это *мб* $T2$ и типы $T1$ и $T2$ эквивалентны
- $TП$ и $TВ$ это типы классов и $TВ$ является расширением $TП$
- переменная является полиморфным параметром

TBD: добавить совместимость для протоколов и функциональных типов.

12. Применять с осторожностью

Низкоуровневые типы и операции.

12.1. Осторожное преобразование типа

Часть преобразований типа являются низкоуровневыми и/или требуют особого внимания разработчика. Для использования таких преобразований операция преобразования должна быть помечена служебным словом **осторожно**, кроме того исходный файл, в котором используется *небезопасное преобразование* тоже должен быть помечен служебным словом **осторожно** (§9).

Преобразование: '(:' 'осторожно'? Указ-типа ')

Разрешенные преобразования:

| Целевой тип | Тип выражения |
|--------------------|--------------------|
| Цел64 | Слово64 |
| Вещ64 | Слово64 |
| Слово64 | Цел64 |
| Слово64 | Вещ64 |
| Слово64 | ссылочный тип |
| Слово64 | функциональный тип |
| ссылочный тип | Слово64 |
| функциональный тип | функциональный тип |

Ссылочными типами являются Строка, типы вектора и типы класса.

Особенности преобразований:

- Все преобразования выполняются без изменения битового представления.
- Для преобразования Слово64 в ссылочный тип выполняется проверка времени исполнения, и запускается *аварийная ситуация* (§7.13) в случае несовпадения целевого типа и динамического типа выражения.
- Преобразование функционального типа в Слово64 разрешено только, если операнд является функцией (но не методом).

А. Компиляторы и версии языка

А.1. Компиляторы

Основной компилятор **трик**, написанный на Тривиле, реализует последнюю описанную версию языка Триவில்.

Компилятор **tric**, написанный на Go, является вспомогательным и предназначен, главным образом для компиляции основного компилятора. Этот компилятор реализует не полный язык, а только то подмножество, которое используется в основном компиляторе и используемыми в нем библиотеками.

Текущая версия **tric** реализует язык версии 0.9.0 и дополнения, сделанные в версии 0.9.6. Дополнения, сделанные в остальных версиях языка, например, 0.9.1 или 0.9.7, не реализованы в этом компиляторе.

А.2. Версия 0.9.0 от 07.09.2023

- Добавлен тип Пусто с единственным значением - литералом `пусто` (§5.1)

А.3. Версия 0.9.1 от 08.09.2023

- Разрешен возврат полиморфного типа из функции (§4.5)
- Разрешено преобразование типа из значения полиморфного типа к произвольному типу (§6.5)

А.4. Версия 0.9.2 от 08.10.2023

- Добавлены многострочные литералы (§3.6.4)

А.5. Версия 0.9.3 от 29.04.2024

В версию добавлены экспериментальные конструкции, в первую очередь, для языка Арவில், но они доступны и в коде на Тривиле.

- Функциональные типы
- Функциональные значения для функций и методов
- Тип протокола

А.6. Версия 0.9.4 от 15.06.2024

- Добавлены преобразования и проверки типов для пар: класс - протокол
- Добавлены осторожные преобразования для функциональных значений

А.7. Версия 0.9.5 от 22.03.2025

- Анонимные типы векторов и конструкторы векторов анонимного типа, см. §5.2, §5.3, §6.2.

А.8. Версия 0.9.6 от 26.03.2026

- Грамматика идентификатора (§3.3):
 - Символ `!` не может больше стоять в конце идентификатора.
 - Одиночная кавычка `'` или несколько могут стоять в конце идентификатора.
- Для типа Лог (§5.1) добавлены:
 - Константа `да` (тоже, что и истина).
 - Константа `нет` (тоже, что и ложь).
- Изменены знаки всех операций, работающих с битовым представлений значений, см. §6.14.4, §6.14.5, инвертирование битов в §6.11.
- Добавлен встроенный метод `'сократить'` для векторов: §8.2.2.
- Переменная варианта доступна в ветке **другое** оператора **выбор**, см. §7.8.3.
- Добавлено описание попутной переменной: §6.13, §7.6.1, §7.7.1.

А.9. Версия 0.9.7 от 26.03.2026

- Доработаны строковые и символьные литералы: §3.6.3, §3.6.4, §3.6.5, §3.6.6.

А.10. Планируемые доработки

- Может-быть протоколы